[MS-TDS]:
Tabular Data Stream Protocol Specification

Release: Monday, January 25, 2010

Revision Summary

Date	Revision History	Revision Class	Comments
03/14/2008	0.1	Major	Initial Availability.
06/20/2008	0.1.1	Editorial	Revised and edited the technical content.
07/25/2008	0.1.2	Editorial	Revised and edited the technical content.
08/29/2008	0.1.3	Editorial	Revised and edited the technical content.
10/24/2008	0.1.4	Editorial	Revised and edited the technical content.
12/05/2008	0.2	Minor	Updated the technical content.
01/16/2009	0.3	Minor	Updated the technical content.
02/27/2009	0.4	Minor	Updated the technical content.
04/10/2009	0.5	Minor	Updated the technical content.
05/22/2009	0.5.1	Editorial	Revised and edited the technical content.
07/02/2009	1.0	Major	Updated and revised the technical content.
08/14/2009	1.1	Minor	Updated the technical content.
09/25/2009	2.0	Major	Updated and revised the technical content.
11/06/2009	3.0	Major	Updated and revised the technical content.
12/18/2009	4.0	Major	Updated and revised the technical content.
01/29/2010	4.1	Minor	Updated the technical content.

Contents

1 Introduction

This document specifies the Microsoft Tabular Data Stream (TDS) Protocol. All references to the term SQL Server refer to the Microsoft® SQL Server® product line. The TDS protocol is an application layer request/response protocol that facilitates interaction with a database server and provides for:

- Authentication and channel encryption negotiation.
- Specification of requests in SQL (including Bulk Insert).
- Invocation of a stored procedure or user-defined function, also known as a **remote procedure call (RPC)**.
- Return of data.
- Transaction manager requests.

1.1 Glossary

The following terms are defined in [MS-GLOS]:

atomic transaction
Active Directory (AD)
big-endian
domain
domain controller (DC)
Globally Unique Identifier (GUID)
interface
interface identifier (IID)
little-endian
nullable column
Security Support Provider Interface (SSPI)
Session Multiplex Protocol (SMUX)
stored procedure
table response
Transaction Manager
Universal Naming Convention (UNC)

The following terms are specific to this document:

bulk insert: A method for efficiently populating the rows of a table from the client to the server.

client: A program that establishes connections for the purpose of sending requests.

column: A set of data composed of the same field from each row in a table.

Common Language Runtime User-Defined Type (CLR UDT): A data type created and defined by the user on SQL Server by using a .NET Framework common language runtime assembly.

data store: A repository for data.

data stream: A stream of data that corresponds to specific TDS semantics. A single data stream can represent an entire TDS message or only a specific, well-defined portion of a TDS message. A TDS data stream can span multiple network data packets.

delete: To remove a row from a table.

Distributed Transaction Coordinator (DTC): A Microsoft Windows service that coordinates transactions across multiple databases. Further information may be obtained from [MSDN-DTC].

final state: The application layer has finished communication and the lower layer connection should be disconnected.

initial state: A prerequisite for application layer communication. A lower layer channel which can provide reliable communication must be established.

insert: To add a row to a table.

Multiple Active Result Sets (MARS): A feature introduced in SQL Server 2005 that allows applications to have more than one pending request per connection. Further information may be obtained from [MSDN-MARS].

out-of-band: A type of event that happens outside of the standard sequence of events. Specifically, the idea that a signal or message can be sent during an unexpected time and will not cause any protocol parsing issues.

query: A character string expression sent to a data store that contains a set of operations that request data from the data store.

query notification: A feature introduced in SQL Server 2005 that allows the client to register for notification on changes to a given query result. Further information may be obtained from [MSDN-QUERYNOTE].

Remote Procedure Call (RPC): The direct invocation of a stored procedure or user-defined function on the server.

server: An application program that accepts connections to service requests by sending back responses. Any program may be capable of being both a client and a server. Use of these terms refers only to the role being performed by the program for a particular connection rather than to the program's capabilities in general.

Service Broker (SSB): A feature introduced in SQL Server 2005 that provides a mechanism for queuing and reliable delivery of messages. Further information may be obtained from [MSDN-ServBroker].

SQL Server User Authentication (SQLAUTH): An authentication mechanism used to support SQL Server user accounts. The username and password of the user account are transmitted as part of the login message that the client sends to the server.

structurally invalid: A data stream that does not follow the header defined, the rule for the specific message type defined in section 2, or both.

TDS session: A successfully established communication over a period of time between a client and a server on which the Tabular Data Stream (TDS) protocol is used for message exchange.

Unicode: The set of characters as defined by [UNICODE] that are encoded in UCS-2.

update: An add, modify, or delete operation of one or more objects or attribute values.

Virtual Interface Architecture (VIA): A high-speed interconnect requiring special hardware and drivers provided by third parties.

MAY, SHOULD, MUST, SHOULD NOT, MUST NOT: These terms (in all caps) are used as described in [RFC2119]. All statements of optional behavior use either MAY, SHOULD, or SHOULD NOT.

1.2 References

1.2.1 Normative References

We conduct frequent surveys of the normative references to assure their continued availability. If you have any issue with finding a normative reference, please contact dochelp@microsoft.com. We will assist you in finding the relevant information. Please check the archive site, http://msdn2.microsoft.com/en-us/library/E4BD6494-06AD-4aed-9823-445E921C9624, as an additional source.

[IANAPORT] Internet Assigned Numbers Authority, "Port Numbers", November 2006, http://www.iana.org/assignments/port-numbers

[IEEE754] Institute of Electrical and Electronics Engineers, "Standard for Binary Floating-Point Arithmetic", IEEE 754-1985, October 1985, http://ieeexplore.ieee.org/servlet/opac?punumber=2355

[MS-DTYP] Microsoft Corporation, "Windows Data Types", March 2007.

[MS-LCID] Microsoft Corporation, "Windows Language Code Identifier (LCID) Reference", March 2007.

[RFC793] Postel, J., "Transmission Control Protocol", STD 7, RFC 793, September 1981, http://www.ietf.org/rfc/rfc0793.txt

[RFC1122] Braden, R., Ed., "Requirements for Internet Hosts -- Communication Layers", STD 3, RFC 1122, October 1989, http://www.ietf.org/rfc/rfc1122.txt

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997, http://www.ietf.org/rfc/rfc2119.txt

[RFC2246] Dierks, T., and Allen, C., "The TLS Protocol Version 1.0", RFC 2246, January 1999, http://www.ietf.org/rfc/rfc2246.txt

[RFC4234] Crocker, D., Ed., and Overell, P., "Augmented BNF for Syntax Specifications: ABNF", RFC 4234, October 2005, http://www.ietf.org/rfc/rfc4234.txt

[SSL3] Netscape, "SSL 3.0 Specification", http://tools.ietf.org/html/draft-ietf-tls-ssl-version3-00

If you have any trouble finding [SSL3], please check here.

[UNICODE] The Unicode Consortium, "Unicode Home Page", 2006, http://www.unicode.org/

[US-ASCII] Columbia University, "The US ASCII Character Set", 1986, http://www.columbia.edu/kermit/ascii.html

[VIA] Intel Corporation, "Intel Virtual Interface (VI) Architecture Developer's Guide", September 1998, ftp://download.intel.com/design/servers/vi/

1.2.2 Informative References

[MS-GLOS] Microsoft Corporation, "Windows Protocols Master Glossary", March 2007.

[MS-SSCLRT] Microsoft Corporation, "SQL Server Common Language Runtime (CLR) Types Serialization Formats", July 2009.

[MSDN-Autocommit] Microsoft Corporation, "SQL Server - Autocommit Transactions", http://msdn.microsoft.com/en-us/library/aa386980(VS.85).aspx

[MSDN-BEGIN] Microsoft Corporation, "BEGIN TRANSACTION (Transact SQL)", http://msdn.microsoft.com/en-us/library/ms188929.aspx

[MSDN-BINXML] Microsoft Corporation, "SQL Server Binary XML Structure Specification", http://msdn.microsoft.com/en-us/library/cc580569.aspx

[MSDN-BOUND] Microsoft Corporation, "Using Bound Sessions", http://msdn.microsoft.com/en-us/library/ms177480.aspx

[MSDN-BROWSE] Microsoft Corporation, "Browse Mode", http://msdn.microsoft.com/en-us/library/aa936959(SQL.80).aspx

[MSDN-ColSets] Microsoft Corporation, "Using Column Sets", http://msdn.microsoft.com/en-us/library/cc280521.aspx

[MSDN-COMMIT] Microsoft Corporation, "COMMIT TRANSACTION (Transact SQL)", http://msdn.microsoft.com/en-us/library/ms190295.aspx

[MSDN-DTC] Microsoft Corporation, "Distributed Transaction Coordinator", http://msdn.microsoft.com/en-us/library/ms684146.aspx

[MSDN-INSERT] Microsoft Corporation, "Microsoft Corporation, "INSERT (Transact-SQL)", http://msdn.microsoft.com/en-us/library/ms174335.aspx

[MSDN-MARS] Microsoft Corporation, "Multiple Active Result Sets (MARS) in SQL Server 2005", http://msdn.microsoft.com/en-us/library/ms345109.aspx

[MSDN-MC-SMP] Microsoft Corporation, "Session Multiplex Protocol Specification", http://msdn.microsoft.com/en-us/library/cc219643.aspx

[MSDN-NamedPipes] Microsoft Corporation, "Creating a Valid Connection String Using Named Pipes", http://msdn.microsoft.com/en-us/library/ms189307.aspx

[MSDN-QUERYNOTE] Microsoft Corporation, "Using Query Notifications", http://msdn.microsoft.com/en-us/library/ms175110.aspx

[MSDN-SETIMTRAN] Microsoft Corporation, "SET IMPLICIT_TRANSACTIONS (Transacts-SQL)", http://msdn.microsoft.com/en-us/library/ms187807.aspx

[MSDN-SQLCollation] Microsoft Corporation, "Selecting a SQL Collation", http://msdn.microsoft.com/en-us/library/ms144250.aspx

MSDN-UPDATETEXT] Microsoft Corporation, "UPDATETEXT (Transact-SQL)", http://msdn.microsoft.com/en-us/library/ms189466.aspx

[MSDN-WRITETEXT] Microsoft Corporation, "WRITETEXT (Transact-SQL)", http://msdn.microsoft.com/en-us/library/ms186838.aspx

[NTLM] Microsoft Corporation, "Microsoft NTLM", http://msdn.microsoft.com/en-us/library/aa378749.aspx

If you have any trouble finding [NTLM], please check here.

[RFC4120] Neuman, C., Yu, T., Hartman, S., and Raeburn, K., "The Kerberos Network Authentication Service (V5)", RFC 4120, July 2005, http://www.ietf.org/rfc/rfc4120.txt

[PIPE] Microsoft Corporation, "Named Pipes", http://msdn.microsoft.com/en-us/library/aa365590.aspx

[RFC4178] Zhu, L., Leach, P., Jaganathan, K., and Ingersoll, W., "The Simple and Protected Generic Security Service Application Program Interface (GSS-API) Negotiation Mechanism", RFC 4178, October 2005, http://www.ietf.org/rfc/rfc4178.txt

[SSPI] Microsoft Corporation, "SSPI", http://msdn.microsoft.com/en-us/library/aa380493.aspx

1.3 Protocol Overview (Synopsis)

The Tabular Data Stream (TDS) Protocol is an application-level protocol used for the transfer of requests and responses between clients and database server systems. In such systems, the client will typically establish a long-lived connection with the server. Once the connection is established using a transport-level protocol, TDS messages are used to communicate between the client and the server. A database server can also act as the client if needed, in which case a separate TDS connection must be established. Note that the **TDS session** is directly tied to the transport-level session, meaning that a TDS session is established when the transport-level connection is established and the server receives a request to establish a TDS connection. It persists until the transport-level connection is terminated (for example, when a TCP socket is closed). In addition, TDS does not make any assumption about the transport protocol used, but it does assume the transport protocol supports reliable, in-order delivery of the data.

TDS includes facilities for authentication and identification, channel encryption negotiation, specification of requests in SQL, stored procedure calls, returning data, and transaction manager requests. Returned data is self-describing and record-oriented. The data streams describe the names, types and optional descriptions of the rows being returned. The following diagram depicts a (simplified) typical flow of communication in the TDS Protocol.

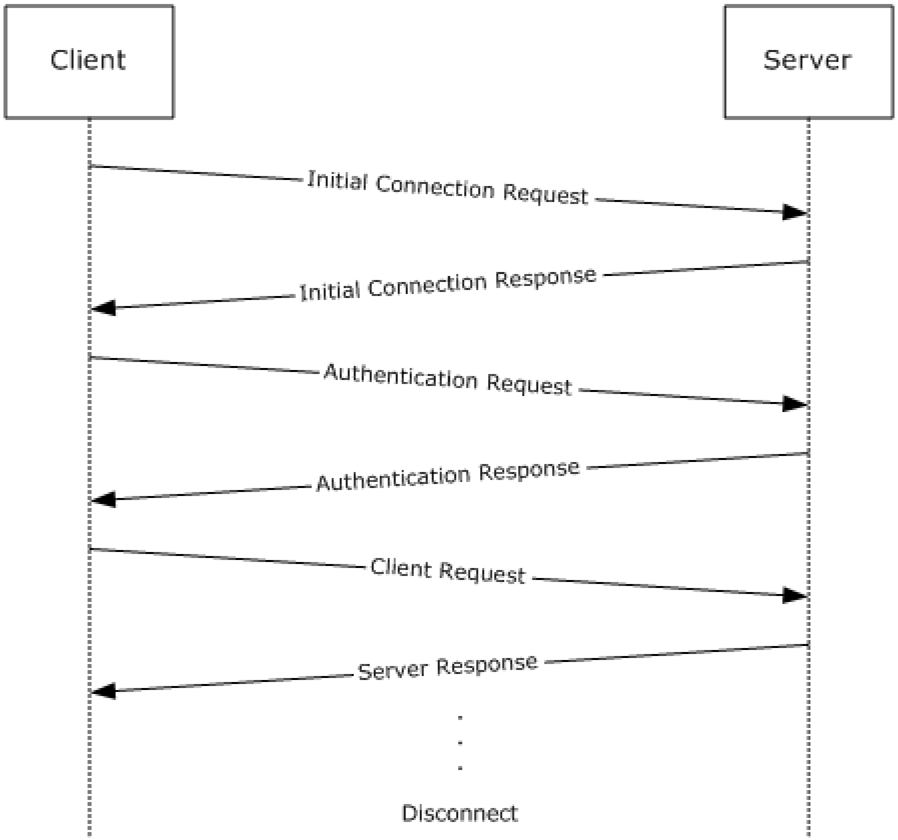

Figure 1: Communication flow in the TDS protocol

The following example is a high-level description of the messages exchanged between the client and the server to execute a simple client request such as the execution of an SQL statement. It is assumed that the client and the server have already established a connection and authentication has succeeded.

```
Client:SQL statement
```

The server executes the statement and then sends back the results to the client. The data columns being returned are first described by the server (represented as column metadata or COLMETADATA) and then the rows follow. A completion message is sent after all the row data has been transferred.

```
Server:COLMETADATAdata stream
ROWdata stream
.
.
ROWdata stream
DONEdata stream
```

See section 2.2.4 for additional information on the correlation between data stream and TDS buffer. The following table outlines the Microsoft® SQL Server® version and the corresponding TDS version.

SQL Server version	TDS version
SQL Server 7.0	7.0
SQL Server 2000	7.1
SQL Server 2000 SP1	7.1 Revision 1
SQL Server 2005	7.2
SQL Server 2008	7.3.A, 7.3.B

Additional details about which SQL Server version corresponds to which TDS version number is defined in LOGINACK (section 2.2.7.11).

TDS version 7.3.A does not include support for NBCROW and fSparseColumnSet.

1.4 Relationship to Other Protocols

The Tabular Data Stream (TDS) protocol depends upon a network transport connection being established prior to a TDS conversation occurring (the choice of transport protocol is not important to TDS). TDS depends on Transport Layer Security (TLS)/Secure Socket Layer (SSL) for network channel encryption. Although the TDS protocol depends on TLS/SSL to encrypt data transmission, the negotiation of the encryption setting between the client and server and the initial TLS/SSL handshake are handled in the TDS layer.

If the Multiple Active Result Set (MARS) feature [MSDN-MARS] is enabled, then the Session Multiplexing protocol (SMUX)[MSDN-MC-SMP] is required.

This relationship is illustrated in the following figure.

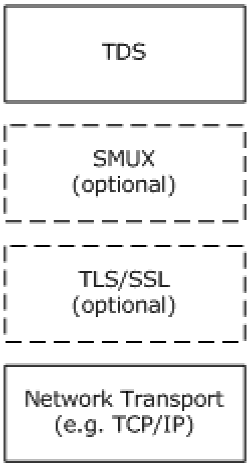

Figure 2: Protocol relationship

1.5 Prerequisites/Preconditions

Throughout this document, it is assumed that the client has already discovered the server and established a network transport connection for use with TDS.

No security association is assumed to have been established at the lower layer before TDS begins functioning. For [SSPI] authentication to be used, [SSPI] support must be available on both the client and server machines. If channel encryption is to be used, then TLS/SSL support must be present on both the client and server machines, and a certificate suitable for encryption must be deployed on the server machine.

1.6 Applicability Statement

The TDS protocol is appropriate for use to facilitate request/response communications between an application and a database server in all scenarios where network or local connectivity is available.

1.7 Versioning and Capability Negotiation

This document covers versioning issues in the following areas.

- **Supported Transports:** This protocol can be implemented on top of any network transport protocol as discussed in section 2.1.
- **Protocol Versions:** The TDS protocol supports the "TDS 7.1", "TDS 7.2", and "TDS 7.3" explicit dialects. The dialect version is negotiated as part of the LOGIN7 message data stream, which is defined in section 2.2.6.3. (Note that once a protocol feature has been introduced, it is supported in subsequent versions of the TDS protocol until explicitly removed.)
- **Security and Authentication Methods:** The TDS protocol supports **SQLAUTH**. It also supports SSPI authentication and indirectly supports any authentication mechanism that SSPI supports. The use of SSPI in TDS is defined in sections 2.2.6.3 and 3.2.5.1.
- **Localization:** Localization-dependent protocol behavior is specified in sections 2.2.5.1.2 and 2.2.5.6.
- **Capability Negotiation:** This protocol does explicit capability negotiation as specified in this section.

In general, the TDS protocol does not provide facilities for capability negotiation as the complete set of supported features is fixed for each version of the protocol. Certain features such as authentication type are not negotiated but rather requested by the client. However, one feature that is negotiated is channel encryption. The encryption behavior used for the TDS session is negotiated in the initial messages exchanged by the client and server. See the PRELOGIN description in section 2.2.6.4 for further details.

Note that the cipher suite for TLS/SSL and the authentication mechanism for SSPI are negotiated outside the influence of TDS [RFC2246] [SSL3].

1.8 Vendor-Extensible Fields

None.

1.9 Standards Assignments

Parameter	TCP port value	Reference
Default SQL Server instance TCP port	1433	[IANAPORT]

2 Messages

The formal syntax of all messages is provided in Augmented Backus-Naur Form (ABNF), as specified in [RFC4234].

2.1 Transport

The TDS protocol does not prescribe a specific underlying transport protocol to use on the Internet or on other networks. TDS only presumes a reliable transport that guarantees in-sequence delivery of data.

The chosen transport may be either stream-oriented or message-oriented. If a message-oriented transport is used, then any TDS packet sent from a TDS client to a TDS server MUST be contained within a single transport data unit. Any additional mapping of TDS data onto the transport data units of the protocol in question is outside the scope of this specification.

The current version of the TDS protocol has implementations over the following transports:

- TCP [RFC793].
- Microsoft Windows Named Pipes in message mode [PIPE]. Please see [MSDN-NamedPipes] for additional information related to Microsoft-specific implementations.
- A reliable transport over the VIA interface [VIA].
- Optionally, the TDS protocol has implementations for the following two protocols on top of the preceding transports:
    - Transport Layer Security (TLS) [RFC2246]/Secure Socket Layer (SSL), in case TLS/SSL encryption is negotiated.
    - Session Multiplex Protocol [MSDN-MC-SMP], in case the Multiple Active Results Set (MARS) feature [MSDN-MARS] is requested.

2.2 Message Syntax

Character data, such as T-SQL statements, within a TDS message is in Unicode, unless the character data represents the data value of an ASCII data type, such as a non-Unicode data column. Character counts within TDS are a count of characters, rather than bytes, except when explicitly specified as byte counts.

2.2.1 Client Messages

Messages sent from the client to the server are as follows:

- A pre-login record
- A login record
- A SQL command (in any language that the server will accept)
- A SQL command followed by its associated binary data (for example, the data for a bulk copy command)
- An RPC

- An attention signal

These are briefly described later; detailed descriptions of message contents are in section 2.2.6.

2.2.1.1 Pre-Login

Before a login occurs, a handshake denominated pre-login occurs between client and server, setting up contexts such as encryption and MARS-enabled. See section 2.2.6.4 for additional details.

2.2.1.2 Login

When the client wishes to establish a TDS protocol connection with the server side, the client sends a login message data stream to the server. The client may have more than one connection to the server but each one is established separately in the same way. See section 2.2.6.3 for additional details.

Once the server has received the login record from the client, it will notify the client that it has either accepted or rejected the connection request. See section 3.3.5.1 for additional details.

2.2.1.3 SQL Command

To send an SQL command or batch of SQL commands, the SQL command, represented by a Unicode string, is copied into the data section of a TDS packet and then sent to the SQL server. An SQL batch may span more than one TDS packet. See section 2.2.6.6 for additional details.

2.2.1.4 SQL Command with Binary Data

The bulk insert/bulk load operation is a case of an SQL command that consists of a Unicode string followed by binary data. The client sends the bulk insert request and then a COLMETADATA token that describes the raw data is sent. Multiple rows of binary data are then sent to the server. The data is not formatted in storage row format but the format described by the COLMETADATA token. The stream is the same as if the data was being selected from the server rather than being sent to the server. See section 2.2.6.1 for additional details.

2.2.1.5 Remote Procedure Call

To execute a RPC on the server, the client sends an RPC message *data stream* to the server. This is a binary stream that contains the RPC name or numeric identifier, options, and parameters. RPCs must be in a separate TDS message and not intermixed with SQL commands. There can be several RPCs in one message. See section 2.2.6.5 for additional details.

2.2.1.6 Attention

The client can interrupt and cancel the current command by sending an **Attention** message. This is also known as **out-of-band** data, but any TDS packet that is currently being sent MUST be finished before sending the **Attention** message. After the client sends an **Attention** message, the client MUST read until it receives an **Attention** acknowledgment.

If a complete request has been sent to the server then sending a cancel requires sending an **Attention** packet. An example of this behavior is if the client has already sent a request, which has the last packet with EOM bit (0x01) set in status. The **Attention** packet is the only way to interrupt a complete request that has already been sent to the server. See section 2.2.4.4.2 for additional details.

If a complete request has not been sent to the server, the client MUST send the next packet with both ignore bit (0x02) and EOM bit (0x01) set in the status to cancel the request. An example of this behavior is if one or more packets have been sent but the last packet with EOM bit (0x01) set in status has not been sent. Setting the ignore and EOM bit terminates the current request, and the server MUST ignore the current request. When the ignore and EOM bit is set the server will not send an attention acknowledgment but instead return a table response with a single DONE token with a status of DONE_ERROR to indicate the incoming request was ignored. See section 2.2.3.1.2 for additional details about the buffer header status code.

2.2.1.7 Transaction Manager Request

The client can request that the connection enlist in an [MSDN-DTC] transaction.

2.2.2 Server Messages

Messages sent from the server to the client are:

- A pre-login response
- A login response
- row data
- The return status of an RPC
- return parameters of an RPC
- request completion information
- error and information
- An attention signal

These are briefly described below; detailed descriptions of message contents are in section 2.2.6.

2.2.2.1 Pre-Login Response

The pre-login response is a tokenless packet data stream. The data stream consists of the response to the information requested by the client pre-login message. For a detailed description of this stream, see section 2.2.6.4.

2.2.2.2 Login Response

The login response is a token stream consisting of information about the server's characteristics, optional information and error messages, followed by a completion message.

The LOGINACK token **data stream** includes information about the SQL interface and the server's product code and name. For a detailed description of the login response data stream, see section 2.2.7.11.

If there are any messages in the login response, an ERROR or INFO token data stream is returned from the server to the client. Please refer to section 2.2.7.9 or 2.2.7.10 respectively for addition information.

The server may send, as part of the login response, one or more ENVCHANGE token data streams if the login changed the environment and the associated notification flag was set. An example of an environment change includes the current database context and language setting. Details on the different environment changes are described in section 2.2.7.8.

If the database specified for connection in the login packet is participating in real-time log shipping the corresponding ENVCHANGE is included as part of the response.

A done packet MUST be present as the final part of the login response, and a DONE token data stream is the last thing sent in response to a client login request. See section 2.2.7.5 for additional information on the DONE token data stream.

2.2.2.3 Row Data

If the client request results in data being returned, the data will precede any other data streams returned from the server except warnings. Row data MUST be preceded by a description of the column names and data types. See section COLMETADATA for additional information on how the column names and data types are described.

2.2.2.4 Return Status

When an RPC is executed by the server, the server MUST return a status value. This is a 4-byte integer and is sent via the RETURNSTATUS token. The server also returns a RETURNSTATUS token when a query is executed using the T-SQL EXEC syntax. See section 2.2.7.15 for additional information.

2.2.2.5 Return Parameters

The response format for execution of a stored procedure is identical regardless of whether the request was sent in T-SQL query or RPC format. It is always a "table response to command" type message.

If the procedure explicitly sends any data, then the message starts with l token stream of rows, informational messages, and error messages. This data is sent in the usual way.

When the RPC is invoked, some or all of its parameters are designated as output parameters. All output parameters will have values returned from the server. For each output parameter, there is a corresponding return value, sent via the RETURNVALUE token. The RETURNVALUE token data stream is also used for sending back the value returned by a user-defined function (UDF), if it is called as an RPC. See section 2.2.7.16 for additional details on the RETURNVALUE token.

2.2.2.6 Response Completion ("DONE")

The client reads results in logical units and can tell when all results have been received by examining the DONE token data stream.

When executing a batch of SQL commands, the server MUST return a DONE token data stream for each set of results. All but the last DONE will have the DONE_MORE bit set in the **Status** field of the DONE token data stream. Therefore, the client can always tell after reading a DONE whether or not there are more results. See section 2.2.7.5 for additional details on the DONE token.

For stored procedures, completion of statements in the stored procedure is indicated by a DONEINPROC token data stream for each statement and a DONEPROC token data stream for each completed stored procedure. For example, if a stored procedure executes two other stored procedures, a DONEPROC token data stream will signal the completion of each stored procedure. See section 2.2.7.6 and 2.2.7.7 respectively for additional details on DONEINPROC and DONEPROC tokens.

When executing SQL commands as a result of a trigger event, the completion of each of the statements inside the trigger is indicated by a DONEINPROC token data stream.

2.2.2.7 Error and Info Messages

Besides returning description of Row data and the data itself, TDS provides a token data stream type for the server to send error or informational messages to the client. These are the INFO token data stream and the ERROR token data stream.

2.2.2.8 Attention Acknowledgment

Once a client has sent an interrupt signal to the server, the client MUST read returning data until the interrupt has been acknowledged. Attentions are acknowledged in the DONE token data stream.

2.2.3 Packets

A packet is the unit written or read at one time. A message may consist of one or more packets. A packet always includes a packet header and is usually followed by packet data that contains the message. Each new message starts in a new packet.

In practice, both the client and server will try to read a packet full of data. They will pick out the header to see how much more (or less) data there is in the communication.

At login time, clients MAY specify a requested "packet" size as part of the LOGIN7 message stream. This identifies the size used to break large messages into different "packets". Server acknowledgment of changes in the negotiated packet size is transmitted back to the client via ENVCHANGE token stream. The negotiated packet size is the maximum value that can be specified in the **Length** packet header field described in section 2.2.3.1.3.

For requests sent to the server larger than the current negotiated "packet" size, the client MUST send all but the last packet with a total number of bytes equal to the negotiated size. Only the last packet in the request can contain an actual number of bytes smaller than the negotiated packet size. If any of the preceding packets are sent with a length less than the negotiated packet size the server SHOULD disconnect the client when the next network payload arrives. In certain implementations the server will send an error message back to the client.

2.2.3.1 Packet Header

To implement messages on top of existing, arbitrary transport layers, a packet header is included as part of the packet. The packet header precedes all data within the packet. It is always 8 bytes in length. Most importantly, the buffer header states the **Type** and **Length** of the entire packet.

The following is a detailed description of each item within the packet header.

2.2.3.1.1 Type

Type defines the type of message. **Type** is a 1 byte unsigned char. Types are as follows:

Value	Description	Buffer data?
1	SQL batch. This can be any language that the server understands.	Yes
2	Pre-TDS7 login (only used by legacy clients older than Microsoft® SQL Server® 7.0).	Yes

Value	Description	Buffer data?
3	RPC.	Yes
4	Table response to a command. This indicates a stream that contains the server response to a client request.	Yes
5	Unused.	-
6	Attention signal.	No
7	Bulk load data. This type is used to send binary data to the server.	Yes
8-13	Unused.	-
14	Transaction manager request	Yes
15	Unused	-
16	TDS7 login (MUST be used by all clients that support SQL Server 7.0 or later).	Yes
17	SSPI message.	Yes
18	Pre-login message.	Yes

If an unknown **Type** is specified, the message receiver SHOULD disconnect the connection. If a valid **Type** is specified, but is unexpected (per section 3), the message receiver SHOULD disconnect the connection. For example, the server could disconnect the connection if the server receives a message with **Type** equal 16 when the connection is already logged in. This applies to both the client and the server.

The following table highlights which messages, as described previously in sections 2.2.1 and 2.2.2, correspond to which packet header type.

Message type	Client or server message	Buffer header type
Pre-Login	Client	2 or 18 depending on whether the client supports TDS v7.0+
Login	Client	16 + 17 (if Integrated authentication)
SQL Command	Client	1
SQL Command with Binary Data	Client	7
RPC	Client	3
Attention	Client	6
Transaction Manager Request	Client	14
Pre-Login Response	Server	4
Login Response	Server	4

Message type	Client or server message	Buffer header type
Row Data	Server	4
Return Status	Server	4
Return Parameters	Server	4
Request Completion	Server	4
Error and Info Messages	Server	4
Attention Acknowledgement	Server	4

2.2.3.1.2 Status

Status is a bit field used to indicate the message state. **Status** is a 1-byte unsigned char. The following Status bit flags are defined.

Value	Description
0x00	"Normal" message.
0x01	End of message (EOM). The last byte in the packet is EOM.
0x02	(From client to server) Ignore this event (0x01 MUST also be set).
0x04	Not used.
0x08	RESETCONNECTION (Introduced in TDS 7.1) (From client to server) Reset this connection before processing event. Only set for event types Batch, RPC, or DTC Request. If clients want to set this bit, it MUST be part of the first packet of the message. This signals the server to clean up the environment state of the connection back to the default environment setting, effectively simulating a logout and a subsequent login, and provides server support for connection pooling. This bit SHOULD be ignored if it is set in a packet that is not the first packet of the message. This status bit MUST NOT be set in conjunction with the RESETCONNECTIONSKIPTRAN bit. Distributed transactions and isolation levels will not be reset.
0x10	(Introduced in TDS 7.3) RESETCONNECTIONSKIPTRAN (From client to server) Reset the connection before processing event but do not modify the transaction state (the state will remain the same before and after the reset). The transaction in the session may be a local transaction that is started from the session or it may be a distributed transaction in which the session is enlisted. This status bit MUST NOT be set in conjunction with the RESETCONNECTION bit. Otherwise identical to RESETCONNECTION.

All other bits are not used and MUST be ignored.

2.2.3.1.3 Length

Length is the size of the packet including the 8 bytes in the packet header. It is the number of bytes from the start of this header to the start of the next packet header. Length is a 2-byte, unsigned short int and is represented in network byte order (big-endian).

2.2.3.1.4 SPID

Spid is the process ID on the server, corresponding to the current connection. This information is sent by the server to the client and is useful for identifying which thread on the server sent the TDS packet. It is provided for debugging purposes. The client MAY send the SPID value to the server. If the client does not, then a value of 0x0000 SHOULD be sent to the server. This is a 2-byte value and is represented in network byte order (big-endian).

2.2.3.1.5 PacketID

PacketID is used for numbering message packets that contain data in addition to the packet header. PacketID is a 1-byte, unsigned char. Each time packet data is sent, the value of **PacketID** is incremented by 1, modulo 256. This allows the receiver to track the sequence of TDS packets for a given message. This value is currently ignored.

2.2.3.1.6 Window

This 1 byte is currently not used. This byte SHOULD be set to 0x00 and SHOULD be ignored by the receiver.

2.2.3.2 Packet Data

Packet data for a given message follows the packet header (see **Type** in section 2.2.3.1.1 for messages that contain packet data). As previously stated, a message can span more than one packet. Because each new message MUST always begin within a new packet, a message that spans more than one packet only occurs if the data to be sent exceeds the maximum packet data size, which is computed as (negotiated packet size - 8 bytes), where the 8 bytes represents the size of the packet header.

2.2.4 Packet Data Token and Tokenless Datastreams

The messages contained in packet data that pass between the client and the server maybe one of two types: a "token stream" or a "tokenless stream". A token stream consists of one or more "tokens" each followed by some token-specific data. A "token" is a single byte identifier used to describe the data that follows it (for example contains token data type, token data length, and so on). Tokenless streams are typically used for simple messages. Messages that may require a more detailed description of the data within it are sent as a token stream. The following table highlights which messages, as described previously in sections 2.2.1 and 2.2.2, use token streams and which do not.

Message type	Client or server message	Token stream?
Pre-Login	Client	No
Login	Client	No
SQL Command	Client	No
SQL Command with Binary Data	Client	Yes
Remote Procedure Call (RPC)	Client	Yes
Attention	Client	No
Transaction Manager Request	Client	No

Message type	Client or server message	Token stream?
Pre-Login Response	Server	No
Login Response	Server	Yes
Row Data	Server	Yes
Return Status	Server	Yes
Return Parameters	Server	Yes
Request Completion	Server	Yes
Error and Info Messages	Server	Yes
Attention Acknowledgement	Server	No

If a stream spans more than one packet, then the EOM bit of the packet header **Status** code MUST be set to 0 for every packet header. The EOM bit MUST be set to 1 in the last packet to signal that the stream ends. In addition, the **PacketID** field of subsequent packets MUST be incremented as defined in section 2.2.3.1.5.

2.2.4.1 Tokenless Stream

As shown in the previous section, some messages do not use tokens to describe the data portion of the data stream. In these cases, all the information required to describe the packet data is contained in the packet header. This is referred to as a tokenless stream and is essentially just a collection of packets and data.

2.2.4.2 Token Stream

More complex messages (for example, row data) are constructed using tokens. As previously described, a token consists of a single byte identifier, followed by token-specific data.

2.2.4.2.1 Token Definition

There are four classes of token definitions:

- Zero Length Token(xx01xxxx)
- Fixed Length Token(xx11xxxx)
- Variable Length Tokens(xx10xxxx)
- Variable Count Tokens(xx00xxxx)

The following sections specify the bit pattern of each token class, various extensions to this bit pattern for a given token class, and a description of its function(s).

2.2.4.2.1.1 Zero Length Token(xx01xxxx)

This class of token is not followed by a length specification. There is no data associated with the token. A zero length token always has the following bit sequence:

0	1	2	3	4	5	6	7
x	x	0	1	x	x	x	x

In the diagram above, x denotes a bit position which can contain the bit value 0 or 1.

2.2.4.2.1.2 Fixed Length Token(xx11xxxx)

This class of token is followed by 1, 2, 4 or 8 bytes of data. No length specification follows this token since the length of its associated data is encoded in the token itself. The different fixed data-length token definitions take the form of one of the following bit sequences, depending on whether the token is followed by 1, 2, 4 or 8 bytes of data.

0	1	2	3	4	5	6	7	Description
x	x	1	1	0	0	x	x	Token is followed by 1 byte of data.
x	x	1	1	0	1	x	x	Token is followed by 2 bytes of data.
x	x	1	1	1	0	x	x	Token is followed by 4 bytes of data.
x	x	1	1	1	1	x	x	Token is followed by 8 bytes of data.

In the diagram above, x denotes a bit position which can contain the bit value 0 or 1.

Fixed-length tokens are use by the following data types: *bigint, int*, *smallint*, *tinyint, float, real, money, smallmoney, datetime, smalldatetime,* and *bit*. The type definition is always represented in COLMETADATA and ALTMETADATA data streams as a single byte Type. Additional details are specified in section 2.2.5.3.1.

2.2.4.2.1.3 Variable Length Tokens(xx10xxxx)

This class of token definition is followed by a length specification. The length (in bytes) of this length is included in the token itself as a "Length" value (see the "Length" rule of the COLINFO token stream). The various different variable length token definitions have the following bit sequence:

0	1	2	3	4	5	6	7	Description
0	0	1	0	0	1	x	x	Length of data represented by 1 byte
0	0	1	0	1	0	x	x	Length of data represented by 1 byte
0	0	1	0	1	1	x	x	Length of data represented by 1 byte
0	1	1	0	0	1	x	x	Length of data represented by 1 byte
0	1	1	0	1	0	x	x	Length of data represented by 1 byte
0	1	1	0	1	1	x	x	Length of data represented by 1 byte
1	0	1	0	x	x	x	x	Length of data represented by 2 bytes
1	1	1	0	x	x	x	x	Length of data represented by 2 bytes
0	0	1	0	0	0	x	x	Length of data represented by 4 bytes

0	1	2	3	4	5	6	7	Description
0	1	1	0	0	0	x	x	Length of data represented by 4 bytes

In the preceding diagram, x denotes a bit position which can contain the bit value 0 or 1.

There are two data types that are of variable length. These are "real" variable length data types like *char and binary* and "nullable" data types, which are either their normal fixed length corresponding to their type_info, or a special length if *null.*

Char and binary data types have values that are either null or 0 to 65534 (0x0000 to 0xFFFE) bytes in length. *Null* is represented by a length of 65535 (0xFFFF). A *char or binary*, which cannot be *null* , can still have a length of zero (for example an empty value). A program that MUST pad a value to a fixed length will typically add blanks to the end of a *char* and binary zeros to the end of a *binary*.

Text and image data types have values that are either *null*, or 0 to 2 gigabytes (0x00000000 to 0x7FFFFFFF bytes) in length. *Null* is represented by a length of -1 (0xFFFFFFFF). No other length specification is supported.

Other nullable data types have a length of 0 if they are *null*.

2.2.4.2.1.4 Variable Count Tokens(xx00xxxx)

This class of token definition is followed by a count of the number of fields that follow the token. Each field length is dependent on the token type. The total length of the token can be determined only by walking the fields. A variable count token always has its third and fourth bits set to 0.

0	1	2	3	4	5	6	7
x	x	0	0	x	x	x	x

In the diagram above, x denotes a bit position which can contain the bit value 0 or 1.

Currently there are two variable count tokens. COLMETADATA and ALTMETADATA both use a 2-byte count.

2.2.4.3 Done and Attention Tokens

The server generates a DONE token for most TRANSACT-SQL [MS-TSQL] statements, including any statements that might get executed due to triggers and/or stored procedures. It is very common for an application user to submit a simple update to the server and cause more than one SQL statement to be executed due to triggers on the target table. This is the main reason that trigger writers are advised not to use *select* statements to return messages but rather to use the TRANSACT-SQL print statement.

The attention signal is sent using the out-of-band write provided by the network library. An out-of-band write is the ability to send the attention signal no matter if the sender is in the middle of sending or processing a message or simply sitting idle. If that function is not supported, the client must simply read and discard all of the data from the server until the final DONE token is read. Not all pre-Microsoft® SQL Server® 7.0 servers support the attention signal using the message header. The older implementation was for the client to send a 1 byte message (no header) containing "A" using the out-of-band write.

2.2.4.4 Token Stream Examples

The following two examples highlight token stream communication. The packaging of these token streams into packets is not shown in this section. Actual TDS network data samples are available in section 4.

2.2.4.4.1 Sending an SQL Batch

In this example, an SQL statement is sent to the server and the results are sent to the client. The SQL statement is as follows:

```
SQLStatement =  select name, empid from employees
                update employees set salary = salary * 1.1
                select name from employees where department = 'HR'

Client:    SQLStatement

Server:    COLMETADATA   data stream
           ROW           data stream
           .
           .
           ROW           data stream
           DONE          data stream (with DONE_COUNT & DONE_MORE
                                      bits set)
           DONE          data stream (for UPDATE, with DONE_COUNT &
                                      DONE_MORE bits set)
           COLMETADATA   data stream
           ROW           data stream
           .
           .
           ROW           data stream
           DONE          data stream (with DONE_COUNT bit set)
```

2.2.4.4.2 Out-of-Band Attention Signal

In this example, an SQL command is sent to the server, yet before all the data has been returned an interrupt or "Attention Signal" is sent to the server. The client reads and discards any data received between the time the interrupt was sent and the interrupt acknowledgment was received. The interrupt acknowledgment from the server is a bit set in the status field of the DONE token.

```
Client:    select name, empid from employees

Server:    COLMETADATA   data stream
           ROW           data stream
           .
           .
           ROW           data stream

Client:    ATTENTION SENT
```

[The client reads and discards any data already buffered by the server until the acknowledgment is found. There may be or may not be a DONE token with the DONE_MORE bit clear prior to the DONE token with the DONE_ATTN bit set.]

```
Server:    DONE          data stream (with DONE_ATTN bit set)
```

2.2.5 Grammar Definition for Token Description

The Tabular Data Stream consists of a variety of messages. Each message consists of a set of bytes transmitted in a predefined order. This predefined order or grammar can be specified using Augmented Backus-Naur Form [RFC4234]. Details can be found in the following subsections.

2.2.5.1 General Rules

Data structure encodings in TDS are defined in terms of the following fundamental definitions:

BIT: A single bit value of either 0 or 1.

```
BIT = %b0 / %b1
```

BYTE: An unsigned single byte (8-bit) value. The range is 0 to 255.

```
BYTE = 8BIT
```

BYTELEN: An unsigned single byte (8-bit) value representing the length of the associated data. The range is 0 to 255.

```
BYTELEN = BYTE
```

USHORT: An unsigned 2-byte (16-bit) value. The range is 0 to 65535.

```
USHORT = 2BYTE
```

LONG: A signed 4-byte (32-bit) value. The range is -(2^31) to (2^31)-1.

```
LONG = 4BYTE
```

ULONG: An unsigned 4-byte (32-bit) value. The range is 0 to (2^32)-1

```
ULONG = 4BYTE
```

DWORD: An unsigned 4-byte (32-bit) value. The range when used as a numeric value is 0 to (2^32)-1.

```
DWORD = 32BIT
```

ULONGLONG: An unsigned 8 byte (64-bit) value. The range is 0 to (2^64)-1.

```
ULONGLONG = 8BYTE
```

UCHAR: An unsigned single byte (8-bit) value representing a character. The range is 0 to 255.

```
UCHAR = BYTE
```

USHORTLEN: An unsigned 2-byte (16-bit) value representing the length of the associated data. The range is 0 to 65535.

```
USHORTLEN = 2BYTE
```

USHORTCHARBINLEN: An unsigned 2-byte (16-bit) value representing the length of the associated character or binary data. The range is 0 to 8000.

```
USHORTCHARBINLEN = 2BYTE
```

LONGLEN: A signed 4-byte (32-bit) value representing the length of the associated data. The range is -(2^31) to (2^31)-1.

```
LONGLEN = 4BYTE
```

ULONGLONGLEN: An unsigned 8 byte (64-bit) value representing the length of the associated data. The range is 0 to (2^64)-1.

```
ULONGLONGLEN = 8BYTE
```

PRECISION: An unsigned single byte (8-bit) value representing the precision of a numeric number.

```
PRECISION = 8BIT
```

SCALE: An unsigned single byte (8-bit) value representing the scale of a numeric number.

```
SCALE = 8BIT
```

GEN_NULL: A single byte (8-bit) value representing a T-SQL NULL value.

```
GEN_NULL = %x00
```

CHARBIN_NULL: A 2-byte (16-bit) or 4-byte (32-bit) value representing a T-SQL NULL value for a character or binary data type. Please refer to TYPE_VARBYTE for additional details.

```
CHARBIN_NULL = (%xFF %xFF) / (%xFF %xFF %xFF %xFF)
```

FRESERVEDBIT: A FRESERVEDBIT is a BIT value used for padding that does not transmit information. FRESERVEDBIT fields SHOULD be set to %b0 and MUST be ignored on receipt.

```
FRESERVEDBIT = %b0
```

FRESERVEDBYTE: A FRESERVEDBYTE is a BYTE value used for padding that does not transmit information. FRESERVEDBYTE fields SHOULD be set to %x00 and MUST be ignored on receipt.

```
FRESERVEDBYTE = %x00
```

UNICODECHAR: A single Unicode character in UCS-2 encoding, as specified in UNICODE [UNICODE].

```
UNICODECHAR = 2BYTE
```

Notes:

- All integer types are represented in reverse byte order (little-endian) unless otherwise specified.
- FRESERVEDBIT and FRESERVEDBYTE are often used to pad unused parts of a byte or bytes. The value of these reserved bits should be ignored. These elements are generally set to 0.

2.2.5.1.1 Least Significant Bit Order

Certain tokens will possess rules that are comprised of an array of independent bits. These are typically "flag" rules in which each bit is a flag indicating that a specific feature or option is enabled/requested. Normally, the bit array will be arranged in least significant bit order (or typical array index order) meaning that the first listed flag is placed in the least significant bit position (identifying the least significant bit as one would in an integer variable). For example, if *Fn* is the *nth* flag, then the following rule definition:

```
FLAGRULE = F0 F1 F2 F3 F4 F5 F6 F7
```

would be observed on the wire in the natural value order F7F6F5F4F3F2F1F0.

If the rule contains 16 bits, then the order of the bits observed on the wire will follow the little-endian byte ordering. For example:

```
FLAGRULE = F0 F1 F2 F3 F4 F5 F6 F7 F8 F9 F10 F11 F12 F13 F14 F15
```

will have the following order on the wire: F7F6F5F4F3F2F1F0 F15F14F13F12F11F10F9F8.

2.2.5.1.2 Collation Rule Definition

The collation rule is used to specify collation information for character data or metadata describing character data. This is typically specified as part of the LOGIN7 message or part of a column definition in server results containing character data. See COLMETADATA for additional information on column definition.

```
LCID              =    20BIT

fIgnoreCase       =    BIT
fIgnoreAccent     =    BIT
fIgnoreWidth      =    BIT
fIgnoreKana       =    BIT
fBinary           =    BIT
ColFlags          =    fIgnoreCase fIgnoreAccent fIgnoreKana
                       fIgnoreWidth fBinary FRESERVEDBIT FRESERVEDBIT
                       FRESERVEDBIT
Version           =    4BIT
SortId            =    BYTE

COLLATION         =    LCID ColFlags Version SortId
```

COLLATION represents either a Microsoft® SQL Server® Collation or a Microsoft Windows® Collation.

Notes:

- Version can be of value 0, 1, or 2. or 1. A value of "0" denotes collations in SQL Server 2000. A value of "1" denotes collations introduced in Microsoft® SQL Server® 2005. A value of "2" denotes collations introduced in SQL Server 2008.
- If COLLATION specifies 0x00 00 00 00 00 then it indicates request for use of raw collation.
- ColFlags is represented in least significant bit order.

A SQL Server collation is one of a predefined set of sort orders. It is identified by having SortId with values as defined by [MSDN-SQLCollation].

For a windows collation (SortId==0), the LCID bits correspond to a LocaleId as defined by the National Language Support (NLS) functions in Windows. See [MS-LCID] for additional details.

The GetLocaleInfo Windows API can be used to retrieve information about the locale. In particular, querying for the LOCALE_IDEFAULTANSICODEPAGE retrieves the code page information for the given locale.

For either collation type, the different comparison flags map to those defined as valid comparison flags for the CompareString Windows API.

However, for SQL collations with non-Unicode data, the SortId should be used to derive comparison information flags, such as whether for a given SortId a lower "a" equals an upper "A".

2.2.5.2 Data Stream Types

2.2.5.2.1 Unknown Length Data Streams

Unknown length data streams are used by some tokenless data streams. It is a stream of bytes. The number of bytes within the data stream is defined in the packet header as specified in section 2.2.3.1.

```
BYTESTREAM        =    *BYTE
UNICODESTREAM     =    *(2BYTE)
```

2.2.5.2.2 Variable-Length Data Streams

Variable-length data streams consist of a stream of characters or a stream of bytes. The two types are similar, in that they both have a length rule and a data rule.

Characters

Variable-length character streams are defined by a length field followed by the data itself. There are three types of variable-length character streams, each dependent on the size of the length field (for example, a BYTE, USHORT, or LONG). In this section, "value" refers to the value of the variable, not the size of the variable type (for example, not LONGLEN but the value stored in the variable "x" of type LONGLEN). If the length field is zero, then no data follows the length field.

```
B_VARCHAR         =    BYTELEN *CHAR
US_VARCHAR        =    USHORTLEN *CHAR
L_VARCHAR         =    LONGLEN *CHAR
```

Note that the lengths of B_VARCHAR, US_VARCHAR, L_VARCHAR are given in Unicode characters.

Generic Bytes

Similar to the variable-length character stream, variable-length byte streams are defined by a length field followed by the data itself. There are three types of variable-length byte streams, each dependent on the size of the length field (for example, a BYTE, USHORT, or LONG). If the value of the length field is zero, then no data follows the length field.

```
B_VARBYTE         =    BYTELEN *BYTE
US_VARBYTE        =    USHORTLEN *BYTE
L_VARBYTE         =    LONGLEN *BYTE
```

2.2.5.2.3 Data Type Dependent Data Streams

Some messages contain variable data types. The actual type of a given variable data type is dependent on the type of the data being sent within the message as defined in the TYPE_INFO rule.

For example, the RPCRequest message contains the TYPE_INFO and TYPE_VARBYTE rules. These two rules contain data of a type that is dependent on the actual type used in the value of the FIXEDLENTYPE or VARLENTYPE rules of the TYPE_INFO rule.

Data type-dependent data streams occur in three forms: integers, fixed and variable bytes, and partially length-prefixed bytes.

Integers

Data type-dependent integers may be either a BYTELEN, USHORTCHARBINLEN, or LONGLEN in length. This length is dependent on the TYPE_INFO associated with the message. If the data type (for example, FIXEDLENTYPE or VARLENTYPE rule of the TYPE_INFO rule) is of type SSVARIANTTYPE, TEXTTYPE, NTEXTTYPE, IMAGETYPE the integer length is LONGLEN. If the data type is BIGCHARTYPE, BIGVARCHARTYPE, NCHARTYPE, NVARCHARTYPE, BIGBINARYTYPE, or BIGVARBINARYTYPE, the integer length is USHORTCHARBINLEN. For all other data types, the integer length is BYTELEN.

```
TYPE_VARLEN      =   BYTELEN
                     /
                     USHORTCHARBINLEN
                     /
                     LONGLEN
```

Fixed and Variable Bytes

The data type to be used in a data type-dependent byte stream is defined by the TYPE_INFO rule associated with the message.

For variable-length types, with the exception of PLP (see Partially Length-prefixed Bytes below), the TYPE_VARLEN value defines the length of the data to follow. As described above, the TYPE_INFO rule defines the type of TYPE_VARLEN (for example BYTELEN, USHORTCHARBINLEN or LONGLEN).

For fixed-length types, the TYPE_VARLEN rule is not present. In these cases the number of bytes to be read is determined by the TYPE_INFO rule (for example, if "INT2TYPE" is specified as the value for the FIXEDLENTYPE rule of the TYPE_INFO rule, 2 bytes should be read, as "INT2TYPE" is always 2 bytes in length. See Data Types Definitions for more details).

The data following this may be a stream of bytes or a NULL value. The 2-byte CHARBIN_NULL rule is used for BIGCHARTYPE, BIGVARCHARTYPE, NCHARTYPE, NVARCHARTYPE, BIGBINARYTYPE and BIGVARBINARYTYPE types and the 4-byte CHARBIN_NULL rule is used for TEXTTYPE, NTEXTTYPE, and IMAGETYPE. The GEN_NULL rule applies to all other types aside from PLP:

```
TYPE_VARBYTE = GEN_NULL / CHARBIN_NULL / PLP_BODY
             / ([TYPE_VARLEN] *BYTE)
```

Partially Length-prefixed Bytes

Unlike fixed or variable byte stream formats, Partially length-prefixed bytes (PARTLENTYPE) do not require the full data length to be specified before the actual data is streamed out. [1] Thus, it is ideal for those applications where the data length may not be known upfront (that is, xml serialization). A value sent as PLP can be either NULL, a length followed by chunks (as defined by PLP_CHUNK) or an unknown length token followed by chunks, which MUST end with a PLP_TERMINATOR. The rule below describes the stream format (for example, the format of a singleton PLP value):

```
PLP_BODY=    PLP_NULL
```

```
                    /
                    ((ULONGLONGLEN / UNKNOWN_PLP_LEN)
                    *PLP_CHUNK PLP_TERMINATOR)

PLP_NULL         =  %xFFFFFFFFFFFFFFFF

UNKNOWN_PLP_LEN  =  %xFFFFFFFFFFFFFFFE

PLP_CHUNK        =  ULONGLEN 1*BYTE

PLP_TERMINATOR   =  %x00000000
```

Comments:

- TYPE_INFO rule specifies a Partially Length-prefixed Data type (PARTLENTYPE, see 2.2.5.4.3).
- In the UNKNOWN_PLP_LEN case, the data is represented as a series of zero or more chunks, each consisting of the length field followed by length bytes of data (see the PLP_CHUNK rule). The data is terminated by PLP_TERMINATOR (which is essentially a zero-length chunk).
- In the actual data length case, the ULONGLONGLEN specifies the length of the data and is followed by any number of PLP_CHUNKs containing the data. The length of the data specified by ULONGLONGLEN is used as a hint for the receiver. The receiver SHOULD validate that the length value specified by ULONGLONGLEN matches the actual data length.

2.2.5.3 Packet Data Stream Headers - ALL_HEADERS Rule Definition

Some message streams are preceded by a variable number of headers as specified by the ALL_HEADERS rule. The ALL_HEADERS rule is supported only by certain versions of TDS. Other versions of TDS do not include these headers.<2>

The list of headers that are applicable to the different types of messages are described in the following table.

Stream headers MUST be present only in the first packet of requests that span more than one packet. The ALL_HEADERS rule applies only to the three client request types defined in the table below and MUST NOT be included for other request types. For the applicable request types, each header MUST appear at most once in the stream or packet's ALL_HEADERS field.

Header	Value	SQLBatch	RPCRequest	TransactionManagerRequest
Query Notifications	0x00 01	Optional	Optional	Disallowed
Transaction Descriptor	0x00 02	Required	Required	Required

Stream-Specific Rules:

```
TotalLength      =  DWORD    ;including itself
HeaderLength     =  DWORD    ;including itself
HeaderType       =  USHORT;
HeaderData       =  *BYTE
Header           =  HeaderLength HeaderType HeaderData
```

Stream Definition:

```
ALL_HEADERS       =    TotalLength 1*Header
```

Parameter	Description
TotalLength	Total length of ALL_HEADERS stream.
HeaderLength	Total length of an individual header.
HeaderType	The type of header, as defined by the value field in the preceding table.
HeaderData	The data stream for the header. See header definitions in the following subsections.
Header	A structure containing a single header.

2.2.5.3.1 Query Notifications Header

This packet data stream header allows the client to specify that a notification is desired on the results of the request. The contents of the header specify the information necessary for delivery of the notification. See [MSDN-QUERYNOTE] for additional details on SQL Server Query Notifications functionality.

Stream Specific Rules:

```
NotifyId          =    USHORT UNICODESTREAM ; user specified value
                                              when subscribing to the
                                              query notification
SSBDeployment     =    USHORT UNICODESTREAM ;
NotifyTimeout     =    ULONG                ; duration in which the query
                                              notification subscription
                                              is valid
```

The USHORT field defined within the NotifyId and SSBDeployment rules specifies the length, in bytes, of the actual data value, defined by the UNICODESTREAM, that follows it.

Stream Definition:

```
Header Data       =    NotifyId
                       SSBDeployment
                       [NotifyTimeout]
```

2.2.5.3.2 Transaction Descriptor Header

This packet data stream contains information regarding transaction descriptor and number of outstanding requests as they apply to [MSDN-MARS].

The TransactionDescriptor MUST be 0, and OutstandingRequestCount MUST be 1 if the connection is operating in AutoCommit mode. See [MSDN-Autocommit] for more information on autocommit transactions.

Stream-Specific Rules:

```
OutstandingRequestCount =   DWORD     ; number of requests currently
                                        active on the connection
TransactionDescriptor   =   ULONGLONG ; For each connection, a number that
                                        uniquely
                                        identifies the transaction
                                        the request is associated
                                        with.
                                        Initially generated by
                                        the server when a new transaction
                                        is created and returned to
                                        the client as part of the
                                        ENVCHANGE token stream.
```

Stream Definition:

```
Header Data       =    TransactionDescriptor
                       OutstandingRequestCount
```

2.2.5.4 Data Type Definitions

The subsections within this section describe the different sets of data types and how they are categorized. Specifically, data values are interpreted and represented in association with their data type. Details about each data type categorization are described in the following sections.

2.2.5.4.1 Fixed-Length Data Types

Note that these fixed-length data types are all 1 byte in length, as specified in section 2.2.4.2.1.2.

```
NULLTYPE          =    %x1F  ; Null (no data associated with this type)
INT1TYPE          =    %x30  ; TinyInt (1 byte data representation)
BITTYPE           =    %x32  ; Bit (1 byte data representation)
INT2TYPE          =    %x34  ; SmallInt (2 byte data representation)
INT4TYPE          =    %x38  ; Int (4 byte data representation)
DATETIM4TYPE      =    %x3A  ; SmallDateTime (4 byte data
                               representation)
FLT4TYPE          =    %x3B  ; Real (4 byte data representation)
MONEYTYPE         =    %x3C  ; Money (8 byte data representation)
DATETIMETYPE      =    %x3D  ; DateTime (8 byte data representation)
FLT8TYPE          =    %x3E  ; Float (8 byte data representation)
MONEY4TYPE        =    %x7A  ; SmallMoney (4 byte data representation)
INT8TYPE          =    %x7F  ; BigInt (8 byte data representation)

FIXEDLENTYPE      =    NULLTYPE
                       \
                       INT1TYPE
                       \
                       BITTYPE
                       \
                       INT2TYPE
                       \
                       INT4TYPE
                       \
                       DATETIM4TYPE
                       \
                       FLT4TYPE
                       \
                       MONEYTYPE
                       \
                       DATETIMETYPE
                       \
                       FLT8TYPE
                       \
                       MONEY4TYPE
                       \
                       INT8TYPE
```

2.2.5.4.2 Variable-Length Data Types

The data type token values defined in this section have a length value associated with the data type, because the data value corresponding to these data types are represented by a variable number of bytes. The token values defined in this section follow the rule defined in section 2.2.4.2.1.3. <3>

```
GUIDTYPE          =   %x24  ; UniqueIdentifier
INTNTYPE          =   %x26  ; (see below)
DECIMALTYPE       =   %x37  ; Decimal (legacy support)
NUMERICTYPE       =   %x3F  ; Numeric (legacy support)
BITNTYPE          =   %x68  ; (see below)
DECIMALNTYPE          =   %x6A  ; Decimal
NUMERICNTYPE      =   %x6C  ; Numeric
FLTNTYPE          =   %x6D  ; (see below)
MONEYNTYPE        =   %x6E  ; (see below)
DATETIMNTYPE      =   %x6F  ; (see below)
DATENTYPE         =   %x28  ; (introduced in TDS 7.3)
TIMENTYPE         =   %x29  ; (introduced in TDS 7.3)
DATETIME2NTYPE    =   %x2a  ; (introduced in TDS 7.3)
DATETIMEOFFSETNTYPE = %x2b  ; (introduced in TDS 7.3)
CHARTYPE          =   %x2F  ; Char (legacy support)
VARCHARTYPE       =   %x27  ; VarChar (legacy support)
BINARYTYPE        =   %x2D  ; Binary (legacy support)
VARBINARYTYPE     =   %x25  ; VarBinary (legacy support)

BIGVARBINTYPE     =   %xA5  ; VarBinary
BIGVARCHRTYPE     =   %xA7  ; VarChar
BIGBINARYTYPE     =   %xAD  ; Binary
BIGCHARTYPE       =   %xAF  ; Char
NVARCHARTYPE      =   %xE7  ; NVarChar
NCHARTYPE         =   %xEf  ; NChar
XMLTYPE           =   %xF1  ; XML
UDTTYPE           =   %xF0  ; CLR-UDT

TEXTTYPE          =   %x23  ; Text
IMAGETYPE         =   %x22  ; Image
NTEXTTYPE         =   %x63  ; NText
SSVARIANTTYPE     =   %x62  ; Sql_Variant

BYTELEN_TYPE      =   GUIDTYPE
                      /
                      INTNTYPE
                      /
                      DECIMALTYPE
                      /
```

```
                        NUMERICTYPE
                        /
                        BITNTYPE
                        /
                        DECIMALNTYPE
                        /
                        NUMERICNTYPE
                        /
                        FLTNTYPE
                        /
                        MONEYNTYPE
                        /
                        DATETIMNTYPE
                        /
                        DATENTYPE
                        /
                        TIMENTYPE
                        /
                        DATETIME2NTYPE
                        /
                        DATETIMEOFFSETNTYPE
                        /
                        CHARTYPE
                        /
                        VARCHARTYPE
                        /
                        BINARYTYPE
                        /
                        VARBINARYTYPE  ; the length value associated
                                         with these data types is
                                         specified within a BYTE
```

For MONEYNTYPE, the only valid lengths are 0x04 and 0x08, which map to smallmoney and money SQL Server data types respectively.

For DATETIMNTYPE, the only valid lengths are 0x04 and 0x08, which map to smalldatetime and datetime SQL Server data types respectively.

For INTNTYPE, the only valid lengths are 0x01, 0x02, 0x04, and 0x08, which map to tinyint, smallint, int and bigint SQL Server data types respectively.

For FLTNTYPE, the only valid lengths are 0x04 and 0x08, which map to 7-digit precision float and 15-digit precision float SQL Server data types respectively.

For GUIDTYPE, the only valid lengths are 0x10 for non-null instances and 0x00 for NULL instances.

For BITNTYPE, the only valid lengths are 0x01 for non-null instances and 0x00 for NULL instances.

For DATENTYPE, the only valid lengths are 0x03 for non-NULL instances and 0x00 for NULL instances.

For TIMENTYPE, the only valid lengths (along with the associated scale value) are:

SCALE	**1**	**2**	**3**	**4**	**5**	**6**	**7**
LENGTH	0x03	0x03	0x04	0x04	0x05	0x05	0x05

For DATETIME2NTYPE, the only valid lengths (along with the associated scale value) are:

SCALE	**1**	**2**	**3**	**4**	**5**	**6**	**7**
LENGTH	0x06	0x06	0x07	0x07	0x08	0x08	0x08

For DATETIMEOFFSETNTYPE, the only valid lengths (along with the associated scale value) are:

SCALE	**1**	**2**	**3**	**4**	**5**	**6**	**7**
LENGTH	0x08	0x08	0x09	0x09	0x0A	0x0A	0x0A

Exceptions are thrown when invalid lengths are presented to the server during BulkLoadBCP and RPC requests.

```
USHORTLEN_TYPE    =    BIGVARBINTYPE
                       /
                       BIGVARCHRTYPE
                       /
                       BIGBINARYTYPE
                       /
                       BIGCHARTYPE
                       /
                       NVARCHARTYPE
                       /
                       NCHARTYPE   ; the length value associated with
                                     these data types is specified
                                     within a USHORT

LONGLEN_TYPE      =    IMAGETYPE
/
NTEXTTYPE
/
SSVARIANTTYPE
/
TEXTTYPE
                       /
                       XMLTYPE     ; the length value associated with
                                     these data types is specified
                                     within a LONG
```

Notes:

- MaxLength for an SSVARIANTTYPE is 8009 (8000 for strings). For additional details, refer to section 2.2.5.5.4.
- XMLTYPE is only a valid LONGLEN_TYPE for BulkLoadBCP.

MaxLength for an SSVARIANTTYPE is 8009 (string of 8000 bytes).

```
VARLENTYPE        =    BYTELEN_TYPE
                       /
                       USHORTLEN_TYPE
                        /
```

```
                      LONGLEN_TYPE
```

Nullable values are returned using the INTNTYPE, BITNTYPE, FLTNTYPE, GUIDTYPE, MONEYNTYPE, and DATETIMNTYPE tokens which will use the length byte to specify the length of the value or GEN_NULL as appropriate.

2.2.5.4.3 Partially Length-Prefixed Data Types

The data value corresponding to the set of data types defined in this section follows the rule defined in the partially length-prefixed stream definition (section 2.2.5.2.3).

```
PARTLENTYPE      =    XMLTYPE
                      /
                      BIGVARCHRTYPE
                      /
                      BIGVARBINTYPE
                      /
                      NVARCHARTYPE

                      /
                      UDTTYPE
```

BIGVARCHRTYPE, BIGVARBINTYPE, and NVARCHARTYPE can represent two types each:

- The regular type with a known maximum size range from 0 to 8000, defined by USHORTLEN_TYPE.
- A type with unlimited max size, known as varchar(max), varbinary(max) and nvarchar(max), which has a max size of 0xFFFF, defined by PARTLENTYPE. <4>

2.2.5.5 Data Type Details

The subsections within this section specify the formats in which values of system data types are serialized in TDS.

2.2.5.5.1 System Data Type Values

The subsections within this section specify the formats in which values of various common system data types are serialized in TDS.

2.2.5.5.1.1 Integers

All integer types are represented in reverse byte order (little-endian) unless otherwise specified. Each integer takes a whole number of bytes as follows:

bit: 1 byte

tinyint: 1 byte

smallint: 2 bytes

int: 4 bytes

bigint: 8 bytes

2.2.5.5.1.2 Timestamp

timestamp/rowversion is represented as an 8-byte binary sequence with no particular interpretation.

2.2.5.5.1.3 Character and Binary Strings

See Variable-Length Data Types (section 2.2.5.4.2) and Partially Length-Prefixed Data Types (section 2.2.5.4.3).

2.2.5.5.1.4 Fixed-Point Numbers

smallmoney is represented as a 4-byte signed integer. The TDS value is the **smallmoney** value multiplied by 10^4.

money is represented as an 8-byte signed integer. The TDS value is the **money** value multiplied by 10^4. The 8-byte signed integer itself is represented in the following sequence:

- One 4-byte integer that represents the more significant half.
- One 4-byte integer that represents the less significant half.

2.2.5.5.1.5 Floating-Point Numbers

float(*n*) follows the 32-bit [IEEE754] binary specification when *n* <= 24 and the 64-bit [IEEE754] binary specification when 25 <= *n* <= 53.

2.2.5.5.1.6 Decimal/Numeric

decimal(*p*, *s*) is represented in the following sequence:

- One 1-byte unsigned integer that represents the sign of the decimal value as follows:
  - 1 means negative.
  - 0 means nonnegative.
- One 4-, 8-, 12-, or 16-byte signed integer that represents the decimal value multiplied by 10^s. The exact size of this integer is determined based on p as follows:
  - 4 bytes if 1 <= *p* <= 9.
  - 8 bytes if 10 <= *p* <= 19.
  - 12 bytes if 20 <= *p* <= 28.
  - 16 bytes if 29 <= *p* <= 38.

2.2.5.5.1.7 GUID

uniqueidentifier is represented as a 16-byte binary sequence with no specific interpretation.

2.2.5.5.1.8 Date/Times

smalldatetime is represented in the following sequence:

- One 2-byte signed integer that represents the number of days since January 1, 1900.

- One 2-byte unsigned integer that represents the number of minutes elapsed since 12 AM that day.

datetime is represented in the following sequence:

- One 4-byte signed integer that represents the number of days since January 1, 1900. Negative numbers are allowed to represents dates since January 1, 1753.
- One 4-byte unsigned integer that represents the number of one three-hundredths of a second (300 counts per second) elapsed since 12 AM that day.

date is represented as one 3-byte unsigned integer that represents the number of days since January 1, year 1.

time(*n*) is represented as one unsigned integer that represents the number of 10^{-n} second increments since 12 AM within a day. The length of that integer in bytes depends on the precision *n* as follows:

- 3 bytes if 0 <= *n* < = 2.
- 4 bytes if 3 <= *n* < = 4.
- 5 bytes if 5 <= *n* < = 7.

datetime2(*n*) is represented as a concatenation of **time**(*n*) followed by **date** as specified above.

datetimeoffset(*n*) is represented as a concatenation of **datetime2**(*n*) followed by one 2-byte signed integer that represents the time zone offset as the number of minutes from UTC. The time zone offset MUST be between -840 and 840.

2.2.5.5.2 Common Language Runtime (CLR) Instances

The following data type definition stream is used for UDT_INFO in TYPE_INFO.[<5>]

```
DB_NAME                  =   B_VARCHAR  ; database name of the UDT
SCHEMA_NAME              =   B_VARCHAR  ; schema name of the UDT
TYPE_NAME                =   B_VARCHAR  ; type name of the UDT

MAX_BYTE_SIZE            =   USHORT     ; max length in bytes
ASSEMBLY_QUALIFIED_NAME =   US_VARCHAR ; name of the CLR assembly

UDT_METADATA             =   ASSEMBLY_QUALIFIED_NAME

UDT_INFO_IN_COLMETADATA =   MAX_BYTE_SIZE
                             DB_NAME
                             SCHEMA_NAME
                             TYPE_NAME
                             UDT_METADATA

UDT_INFO_IN_RPC    =   DB_NAME       ; database name of the UDT
                       SCHEMA_NAME   ; schema name of the UDT
                       TYPE_NAME     ; type name of the UDT

UDT_INFO             = UDT_INFO_IN_COLMETADATA        ;when sent as part of COLMETADATA
                      /
                      UDT_INFO_IN_RPC                 ;when sent as part of RPC call
```

MAX_BYTE_SIZE is only sent from the server to the client in COLMETADATA and is an unsigned short with a value within the range 1 to 8000 or 0xFFFF. The value 0xFFFF signifies the maximum LOB size indicating a UDT with a maximum size greater than 8000 bytes (also referred to as a Large UDT; introduced in TDS 7.3). MAX_BYTE_SIZE is not sent to the server as part of RPC calls.

Note UserType in the COLMETADATA stream is always 0x0000 for UDTs.

Note The actual data value format associated with a UDT data type definition stream is specified in [MS-SSCLRT].

2.2.5.5.3 XML Values

This section defines the XML data type definition stream. <6>

```
SCHEMA_PRESENT= BYTE;
DBNAME                =   B_VARCHAR
OWNING_SCHEMA         =   B_VARCHAR
XML_SCHEMA_COLLECTION =   US_VARCHAR

XML_INFO              =   SCHEMA_PRESENT
                          [DBNAME OWNING_SCHEMA
                          XML_SCHEMA_COLLECTION]
```

SCHEMA_PRESENT specifies "0x01" if the type has an associated schema collection and DBNAME, OWNING_SCHEMA and XML_SCHEMA_COLLECTION MUST be included in the stream, or '0x00' otherwise.

DBNAME specifies the name of the database where the schema collection is defined.

OWNING_SCHEMA specifies the name of the relational schema containing the schema collection.

XML_SCHEMA_COLLECTION specifies the name of the XML schema collection to which the type is bound.

Note The actual data value format associated with a XML data type definition stream uses the [MS-BINXML] format. See [MS-BINXML] for additional details.

2.2.5.5.4 SQL_VARIANT Values

The SSVARIANTTYPE is a special data type that acts as a place holder for other data types. When a SSVARIANTTYPE is filled with a data value, it takes on properties of the base data type that represents the data value. To support this dynamic change, for those that are not NULL (GEN_NULL) the SSVARIANTTYPE instance has an SSVARIANT_INSTANCE internal structure according to the following definition.

```
VARIANT_BASETYPE    =   BYTE      ; data type definition
VARIANT_PROPBYTES   =   BYTE      ; see below
VARIANT_PROPERTIES  =   *BYTE     ; see below
VARIANT_DATAVAL     =   1*BYTE    ; actual data value

SSVARIANT_INSTANCE  =   VARIANT_BASETYPE
                        VARIANT_PROPBYTES
                        VARIANT_PROPERTIES
                        VARIANT_DATAVAL
```

VARIANT_BASETYPE is the TDS token of the base type.

VARIANT_BASETYPE	VARIANT_PROPBYTES	VARIANT_PROPERTIES
GUIDTYPE, BITTYPE, INT1TYPE, INT2TYPE, INT4TYPE, INT8TYPE, DATETIMETYPE, DATETIM4TYPE, FLT4TYPE, FLT8TYPE, MONEYTYPE, MONEY4TYPE, DATENTYPE	0	<not specified>
TIMENTYPE, DATETIME2NTYPE, DATETIMEOFFSETNTYPE	1	1 byte specifying scale
BIGVARBINTYPE, BIGBINARYTYPE	2	2 bytes specifying max length
NUMERICNTYPE, DECIMALNTYPE	2	1 byte for precision followed by 1 byte for scale
BIGVARCHRTYPE, BIGCHARTYPE, NVARCHARTYPE, NCHARTYPE	7	5-byte COLLATION, followed by a 2-byte max length

Note Data types cannot be NULL when inside a sql_variant. If the value is NULL, then the sql_variant itself should be NULL, but it is not allowed to specify a non-null sql_variant instance and have a NULL value wrapped inside it. A raw collation MAY NOT be specified within a sql_variant.

2.2.5.5.5 Table Valued Parameter (TVP) Values

Table Valued Parameters (or User Defined Table Type as the type is known on the server) encapsulate an entire table of data with 1 to 1024 columns and an arbitrary number of rows. At the present time, TVPs are only permitted to be used as input parameters and do not appear in output parameters or in result set columns.

TVPs MUST only be sent by a TDS client reporting itself as a TDS major version 7.3 or later. If a client reporting itself less than TDS version 7.3 attempts to send a TVP, this MUST result in the server rejecting the request with a TDS protocol error.

2.2.5.5.5.1 Metadata

```
TVPTYPE         =   0xF3
TVP_TYPE_INFO   =   TVPTYPE
                    TVP_TYPENAME
                    TVP_COLMETADATA
                    [TVP_ORDER_UNIQUE]
                    [TVP_COLUMN_ORDERING]
                    TVP_END_TOKEN
                    *TVP_ROW
                    TVP_END_TOKEN
```

Parameter	Description
TVPTYPE	0xF3
TVP_TYPENAME	Type name of the TVP
TVP_COLMETADATA	Column-specific metadata

Parameter	Description
[TVP_ORDER_UNIQUE]	Optional metadata token
[TVP_COLUMN_ORDERING]	Optional metadata token
TVP_END_TOKEN	End optional metadata
*TVP_ROW	0..N TVP_ROW tokens
TVP_END_TOKEN	End of rows

TVP_TYPENAME definition

```
DBNAME        =   B_VARCHAR ;Database where TVP type resides
OwningSchema  =   B_VARCHAR ; Schema where TVP type resides
TypeName      =   B_VARCHAR ; TVP type name
TVP_TYPENAME  =   DbName
                  OwningSchema
                  TypeName
```

TVP_COLMETADATA definition

```
DbName              =   B_VARCHAR ; Database where TVP type resides
fNullable           =   BIT       ; Column is nullable - 0x01
fCaseSen            =   BIT       ; Column is case-sensitive - 0x02
usUpdateable        =   2BIT      ; 2-bit value, one of:
                                  ; 0 = ReadOnly  - 0x00
                                  ; 1 = ReadWrite - 0x04
                                  ; 2 = Unknown   - 0x08
fIdentity           =   BIT       ; Column is identity column - 0x10
fComputed           =   BIT       ; Column is computed - 0x20
usReservedODBC      =   2BIT      ; Reserved bits for ODBC - 0x40+80
fFixedLenCLRType    =   BIT       ; Fixed length CLR type - 0x100
fDefault            =   BIT       ; Column is default value - 0x200
usReserved          =   6BIT      ; Six leftover reserved bits.

Flags               =   fNullable
                        fCaseSen
                        usUpdateable
                        fIdentity
                        fComputed
                        usReservedODBC
                        fFixedLenCLRType
                        fDefault
                        usReserved
Count               =   USHORT    ; Column count up to 1024 max
ColName             =   B_VARCHAR ; Name of column
UserType            =   ULONG     ; UserType of column

TypeColumnMetaData  =   UserType
                        Flags
                        TYPE_INFO
                        ColName   ; Column metadata instance


TVP_NULL_TOKEN       =   0xFFFF

TVP_COLMETADATA      =   TVP_NULL_TOKEN / (Count <Count>TvpColumnMetaData)
```

DbName, OwningSchema, and TypeName are limited to 128 Unicode characters max identifier length.

DbName is required to be zero-length, only OwningSchema and TypeName can be specified. DbName, OwningSchema, and TypeName are all optional fields and may ALL contain zero length strings. Client SHOULD follow these two rules:

- If the TVP is a parameter to a stored procedure or function where parameter metadata is available on the server side, then the client can send all zero length strings for TVP_TYPENAME.
- If the TVP is a parameter to an ad-hoc SQL statement, parameter metadata information is not available on a stored procedure or function on the server. In this case the client is responsible to send sufficient type information with the TVP to allow the server to resolve the TVP type from sys.types. Failure to send needed type information in this case will result in complete failure of RPC call prior to execution.

Only one new flag, fDefault, is added here from existing COLMETADATA.

Additional details about input TVPs and usage of flags

- For an input TVP, if the fDefault flag is set on a column, then the client MUST not emit the corresponding TvpColumnData data for the associated column when sending each TVP_ROW.
- For an input TVP, the fCaseSen, usUpdateable, and fFixedLenCLRType flags are ignored.
- usUpdateable is ignored by server on input, it is "calculated" metadata.
- The fFixedLenCLRType flag is not used by the server.
- Output TVPs are not currently supported.

TVP Flags Usage Chart

Flag	Input behavior
fNullable	Allowed
fCaseSen	Ignored
usUpdateable	Ignored
fIdentity	Allowed
fComputed	Allowed
usReservedODBC	Ignored

Flag	Input behavior
fFixedLenCLRType	Ignored
fDefault	Allowed (if set, data not sent in TvpColumnData)
usReserved	Ignored

2.2.5.5.5.2 Optional Metadata Tokens

TVP_ORDER_UNIQUE definition

```
TVP_ORDER_UNIQUE_TOKEN =   0x10
Count                  =   USHORT ; Count of ColNums to follow
ColNum                 =   USHORT ; A single-column ordinal
fOrderAsc              =   BIT    ; Column-ordered ascending - 0x01
fOrderDesc             =   BIT    ; Column-ordered descending - 0x02
fUnique                =   BIT    ; Column is in unique set - 0x04
Reserved1              =   5BIT   ; Five reserved bits

OrderUniqueFlags       =   fOrderAsc
                           fOrderDesc
                           fUnique
                           Reserved1

TVP_ORDER_UNIQUE       =   TVP_ORDER_UNIQUE_TOKEN
                           (Count <Count>(ColNum OrderUniqueFlags))
```

TVP_ORDER_UNIQUE is similar to the ORDER token that is currently used in TDS responses from the server.

TVP_ORDER_UNIQUE is optional.

ColNum ordinals are 1..N, where 1 is the first column in TVP_COLMETADATA. That is, ordinals start with 1.

Each TVP_ORDER_UNIQUE token can describe a set of columns for ordering and/or a set of columns for uniqueness.

The first column ordinal with an ordering bit set is the primary sort column, the second column ordinal with an ordering bit set is the secondary sort column, and so on.

The client can send 0 or 1 TVP_ORDER_UNIQUE tokens in a single TVP.

The TVP_ORDER_UNIQUE token must always be sent after TVP_COLMETADATA and before the first TVP_ROW token.

When a TVP is sent to the server, each ColNum ordinal inside a TVP_ORDER_UNIQUE token MUST refer to a client generated column. Ordinals that refer to columns with fDefault set will be rejected by the server.

OrderUniqueFlags Possible Combinations And Meaning

fOrderAsc	fOrderDesc	fUnique	Meaning
FALSE	FALSE	FALSE	Invalid flag state, rejected by server
FALSE	FALSE	TRUE	Column is in unique set
FALSE	TRUE	FALSE	Column is ordered descending
FALSE	TRUE	TRUE	Column is ordered descending and in unique set
TRUE	FALSE	FALSE	Column is ordered ascending
TRUE	FALSE	TRUE	Column is ordered ascending and in unique set
TRUE	TRUE	FALSE	Invalid flag state, rejected by server
TRUE	TRUE	TRUE	Invalid flag state, rejected by server

TVP_COLUMN_ORDERING

TVP_COLUMN_ORDERING is an optional TVP metadata token used to allow the TDS client to send a different ordering of the columns in a TVP from the default ordering.

ColNum ordinals are 1..N where 1 is first column in the TVP (ordinals start with 1 in other words). These are the same ordinals used with the TDS ORDER token for example to refer to column ordinal as the columns appear in left to right order.

```
TVP_COLUMN_ORDERING_TOKEN =   %x11
Count                     =   USHORT ; Count of ColNums to follow
ColNum                    =   USHORT ; A single-column ordinal

TVP_COLUMN_ORDERING       =   TVP_COLUMN_ORDERING_TOKEN
                              (Count <Count>ColNum)
```

The client can send 0 or 1 TVP_COLUMN_ORDERING tokens in a single TVP.

The TVP_COLUMN_ORDERING token MUST always be sent after TVP_COLMETADATA and before the first TVP_ROW token.

Additional details about TVP_COLUMN_ORDERING

TVP_COLUMN_ORDERING is used to re-order the columns in a TVP. For example if a TVP is defined as:

```
create type myTvpe as table (f1 int, f2 varchar(max), f3 datetime)
```

The TDS client may want to send the f2 field last inside the TVP as an optimization (streaming the large value last). So the client can send TVP_COLUMN_ORDERING with order 1,3,2 to indicate that inside the TVP_ROW section the column f1 is sent first, f3 is sent second, and f2 is send third.

So the TVP_COLUMN_ORDERING token on the wire for this example would be:

```
11    ; TVP_COLUMN_ORDERING_TOKEN
03 00 ; Count  - Number of ColNums to follow.
01 00 ; ColNum - TVP column ordinal 1 is sent first in TVP_COLMETADATA.
03 00 ; ColNum - TVP column ordinal 3 is sent second in TVP_COLMETADATA.
02 00 ; ColNum - TVP column ordinal 2 is sent third in TVP_COLMETADATA.
```

Duplicate ColNum values are considered an error condition. The ordinal values of the columns in the actual TVP type are ordered starting with 1 for the first column and adding one for each column from left to right. The client MUST send one ColNum for each column described in the TVP_COLMETADATA (so Count MUST match number of columns in TVP_COLMETADATA).

TVP_ROW definition

```
TVP_ROW_TOKEN  =   %x01           ;  A row as defined by TVP_COLMETADATA follows
TvpColumnData  =   TYPE_VARBYTE   ;  Actual value must match metadata for the column
AllColumnData  =   *TvpColumnData ;  Chunks of data, one per non-default column defined in
TVP_COLMETADATA.
TVP_ROW        =   TVP_ROW_TOKEN
                   AllColumnData
TVP_END_TOKEN  =   %x00           ;  Terminator tag for TVP type meaning no moreTVP_ROWs to
follow and end of successful transmission of a single TVP.
```

TvpColumnData is repeated once for each non-default column of data defined in TVP_COLMETADATA.

Each row will contain one data "cell" per column specified in TVP_COLMETADATA. On input, columns with the fDefault flag set in TVP_COLMETADATA will be skipped to avoid sending redundant data.

Column data is ordered in same order as the order of items defined in TVP_COLMETADATA unless a TVP_COLUMN_ORDERING token has been sent to indicate a change in the ordering of the row values.

2.2.5.5.5.3 TDS Type Restrictions

Within a TVP, the following legacy TDS types are not supported:

TDS type	Replacement type
Binary	BigBinary
VarBinary	BigVarBinary
Char	BigChar
VarChar	BigVarChar
Bit	BitN
Int1	IntN
Int2	IntN

TDS type	Replacement type
Int4	IntN
Int8	IntN
Float4	FloatN
Float8	FloatN
Money	MoneyN
Decimal	DecimalN
Numeric	NumericN
DateTime	DatetimeN
DateTime4	DatetimeN
Money4	MoneyN

Additional types not allowed in TVP:

- Null type (NULLTYPE:=‘0x1f’) is not allowed in a TVP.
- TVP type is not allowed in a TVP (no nesting of TVP in a TVP).
- TDS types should not be confused with SQL Server data types.

2.2.5.6 Type Info Rule Definition

The TYPE_INFO rule applies to several messages used to describe column information. For columns of fixed data length, the type is all that is required to determine the data length. For columns of a variable-length type, TYPE_VARLEN defines the length of the data contained within the column, with the following exceptions introduced in TDS 7.3:

DATE MUST NOT have a TYPE_VARLEN. The value is either 3 bytes or 0 bytes (null).

TIME, DATETIME2, and DATETIMEOFFSET MUST NOT have a TYPE_VARLEN. The lengths are determined by the SCALE as indicated in section 2.2.5.4.2.

PRECISION and SCALE MUST occur if the type is NUMERIC, NUMERICN, DECIMAL or DECIMALN.

SCALE (without PRECISION) MUST occur if the type is TIME, DATETIME2, or DATETIMEOFFSET (introduced in TDS 7.3). PRECISION MUST be less than or equal to decimal 38 and SCALE MUST be less than or equal to the precision value.

COLLATION only occurs if the type is BIGCHARTYPE, BIGVARCHRTYPE, TEXTTYPE, NTEXTTYPE, NCHARTYPE or NVARCHARTYPE.

UDT_INFO always occurs if the type is UDTTYPE.

XML_INFO always occurs if the type is XMLTYPE.

USHORTMAXLEN' does not occur if PARTLENTYPE is XMLTYPE or UDTTYPE.

```
USHORTMAXLEN      =    %xFFFF
```

```
TYPE_INFO         =   FIXEDLENTYPE
                      /
                      (VARLENTYPE TYPE_VARLEN [COLLATION])
                      /
                      (VARLENTYPE TYPE_VARLEN [PRECISION SCALE])
                      /
                      (VARLENTYPE SCALE) ; (introduced in TDS 7.3)
                      /
                      VARLENTYPE         ; (introduced in TDS 7.3)
                      /
                      (PARTLENTYPE
                      [USHORTMAXLEN]
                      [COLLATION]
                      [XML_INFO]
                      [UDT_INFO])
```

2.2.5.7 Data Buffer Stream Tokens

The tokens defined as follows are used as part of the token-based data stream. Details on how each token is used inside the data stream are in section 2.2.6.

```
ALTMETADATA_TOKEN  =   %x88
ALTROW_TOKEN       =   %xD3
COLMETADATA_TOKEN  =   %x81
COLINFO_TOKEN      =   %xA5
DONE_TOKEN         =   %xFD
DONEPROC_TOKEN     =   %xFE
DONEINPROC_TOKEN   =   %xFF
ENVCHANGE_TOKEN    =   %xE3
ERROR_TOKEN        =   %xAA
INFO_TOKEN         =   %xAB
LOGINACK_TOKEN     =   %xAD
NBCROW_TOKEN       =   %xD2 ; (introduced in TDS 7.3)
OFFSET_TOKEN       =   %x78
ORDER_TOKEN        =   %xA9
RETURNSTATUS_TOKEN =   %x79
RETURNVALUE_TOKEN  =   %xAC
ROW_TOKEN          =   %xD1
SSPI_TOKEN         =   %xED
TABNAME_TOKEN      =   %xA4
```

2.2.6 Packet Header Message Type Stream Definition

2.2.6.1 Bulk Load BCP

Stream Name:

```
BulkLoadBCP
```

Stream Function:

Describes the format of bulk-loaded data through the **"INSERT BULK"** T-SQL statement. The format is a COLMETADATA token describing the data being sent, followed by multiple ROW tokens, ending with a DONE token. The stream is equivalent to that produced by the server if it were sending the same rowset on output.

Stream Comments:

- Packet header type is 0x07.
- This message sent to the server contains bulk data to be inserted. The client MUST have previously notified the server where this data is to be inserted. See [MSDN-INSERT] for additional details on the INSERT BULK syntax.
- A sample BulkLoadBCP message is in section 4.10.

Stream-Specific Rules:

```
BulkLoad_METADATA    =    COLMETADATA_TOKEN
BulkLoad_ROW         =    ROW_TOKEN
BulkLoad_DONE        =    DONE_TOKEN
```

Submessage Definition:

```
BulkLoadBCP          =    BulkLoad_METADATA
                          *BulkLoad_ROW
                          BulkLoad_DONE
```

Note that for INSERT BULK operations, XMLTYPE is to be sent as NVARCHAR(N) or NVARCHAR(MAX) data type. An error is produced if XMLTYPE is specified.

INSERT BULK operations for data type UDTTYPE is not supported. Use VARBINARYTYPE to insert instances of User Defined Types.

INSERT BULK operations do not support type specifications of DECIMALTYPE and NUMERICTYPE. To insert these data types, use DECIMALN and NUMERICNTYPE.

2.2.6.2 Bulk Load Update Text/Write Text

Stream Name:

```
BulkLoadUTWT
```

Stream Function:

Describes the format of bulk-loaded data with UpdateText or WriteText. The format is the length of the data followed by the data itself.

Stream Comments:

- Packet header type 0x07.
- This message sent to the server contains bulk data to be inserted. The client MUST have previously issued a **"WRITETEXT BULK"** or **"UPDATETEXT BULK"** T-SQL statement to the

server. See [MSDN-WRITETEXT] and [MSDN-UPDATETEXT] for details on the WRITETEXT BULK and UPDATETEXT BULK syntax.

- The server returns a RETURNVALUE token containing the new timestamp for this column.

Stream-Specific Rules:

```
BulkData =L_VARBYTE
```

Sub Message Definition:

```
BulkLoadUTWT      =    BulkData
```

Stream Parameter Details

Parameter	Description
BulkData	Contains the BulkData length and BulkData data within the L_VARBYTE.

2.2.6.3 LOGIN7

Stream Name:

```
LOGIN7
```

Stream Function:

Defines the authentication rules for use between client and server.<7>

Stream Comments:

- Packet header type 0x10.
- The length of a LOGIN7 stream MUST NOT be longer than 128K-1(byte) bytes.
- The OffsetLength and Data rules define the variable-length portions of this data stream. The OffsetLength rule lists the offset from the start of the structure, and the length for each parameter. If the parameter is not used, the parameter length field MUST be zero. The data itself (for example, the Data rule) follows these parameters.
- The first parameter of the OffsetLength rule (ibHostName) indicates the start of the variable length portion of this data stream. As such it MUST NOT be zero. This is required for forward compatibility (for example, later versions of TDS, with additional parameters, can be successfully skipped by down-level servers).
- A sample LOGIN7 message is in section 4.2.

Stream-Specific Rules:

```
Length            =    DWORD
TDSVersion        =    DWORD
```

```
PacketSize       =   DWORD
ClientProgVer    =   DWORD
ClientPID        =   DWORD
ConnectionID     =   DWORD

fByteorder       =   BIT
fChar            =   BIT
fFloat           =   2BIT
fDumpLoad        =   BIT
fUseDB           =   BIT
fDatabase        =   BIT
fSetLang         =   BIT

OptionFlags1     =   fByteorder
                     fChar
                     fFloat
                     fDumpLoad
                     fUseDB
                     fDatabase
                     fSetLang

fLanguage        =   BIT
fODBC            =   BIT
fTranBoundary    =   BIT            ; (removed in TDS 7.2)
fCacheConnect    =   BIT            ; (removed in TDS 7.2)
fUserType        =   3BIT
fIntSecurity     =   BIT

OptionFlags2     =   fLanguage
                     fODBC
                     (fTransBoundary / FRESERVEDBIT)
                     (fCacheConnect / FRESERVEDBIT)
                     fUserType
                     fIntSecurity

fSQLType         =   4BIT
fOLEDB           =   BIT             ; (introduced in TDS 7.2)

TypeFlags        =   fSQLType
                     (FRESERVEDBIT / fOLEDB)
                     3FRESERVEDBIT

fChangePassword  =   BIT             ; (introduced in TDS 7.2)
fUserInstance    =   BIT             ; (introduced in TDS 7.2)
fSendYukonBinaryXML =  BIT           ; (introduced in TDS 7.2)
fUnknownCollationHandling =  BIT     ; (introduced in TDS 7.2)

OptionFlags3     =   (FRESERVEDBIT / fChangePassword)
                     (FRESERVEDBIT / fSendYukonBinaryXML)
                     (FRESERVEDBIT / fUserInstance)
                     (FRESERVEDBIT / fUnknownCollationHandling)
                     4FRESERVEDBIT

ClientTimZone    =   LONG;
ClientLCID       =   LCID
                     ColFlags
                     Version

ibHostName       =   USHORT
```

```
cchHostName      =   USHORT
ibUserName       =   USHORT
cchUserName      =   USHORT
ibPassword       =   USHORT
cchPassword      =   USHORT
ibAppName        =   USHORT
cchAppName       =   USHORT
ibServerName     =   USHORT
cchServerName    =   USHORT
ibUnused         =   USHORT
cbUnused         =   USHORT
ibCltIntName     =   USHORT
cchCltIntName    =   USHORT
ibLanguage       =   USHORT
cchLanguage      =   USHORT
ibDatabase       =   USHORT
cchDatabase      =   USHORT
ClientID         =   6BYTE
ibSSPI           =   USHORT
cbSSPI           =   USHORT
ibAtchDBFile     =   USHORT
cchAtchDBFile    =   USHORT
ibChangePassword =   USHORT    ; (introduced in TDS 7.2)
cchChangePassword =  USHORT    ; (introduced in TDS 7.2)
cbSSPILong       =   DWORD     ; (introduced in TDS 7.2)

OffsetLength     =   ibHostName
                     cchHostName
                     ibUserName
                     cchUserName
                     ibPassword
                     cchPassword
                     ibAppName
                     cchAppName
                     ibServerName
                     cchServerName
                     ibUnused
                     cchUnused
                     ibCltIntName
                     cchCltIntName
                     ibLanguage
                     cchLanguage
                     ibDatabase
                     cchDatabase
                     ClientID
                     ibSSPI
                     cbSSPI
                     ibAtchDBFile
                     cchAtchDBFile
                     [ibChangePassword]
                     [cchChangePassword]
                     [cbSSPILong]
```

All variable-length fields in the login record are optional. This means that the length of the field may be specified as zero. If the length is specified as zero, then the offset MUST be ignored. The only exception is ibHostName, which MUST always point to the beginning of the variable-length data in the login record even in the case where no variable-length data is included.

```
Data               =    *BYTE
```

Stream Definition:

```
LOGIN7             =    Length
                        TDSVersion
                        PacketSize
                        ClientProgVer
                        ClientPID
                        ConnectionID
                        OptionFlags1
                        OptionFlags2
                        TypeFlags
                        (FRESERVEDBYTE / OptionFlags3)
                        ClientTimZone
                        ClientLCID
                        OffsetLength
                        Data
```

Stream Parameter Details

Parameter	Description
Length	The total length of the LOGIN7 structure.
TDSVersion	The highest TDS version being used by the client (for example, 0x00000071 for TDS 7.1. If the TDSVersion value sent by the client is greater than the value that the server recognizes, the server MUST use the highest TDS version that it can use. This provides a mechanism for clients to discover the server TDS by sending a standard Login7 message. If the TDSVersion value sent by the client is lower than the highest TDS version the server recognizes, the server MUST use the TDS version sent by the client.<8> For information on what the server sends to the client, see the LOGINACK token.
PacketSize	The desired packet size being requested by the client.
ClientProgVer	The version of the interface library (for example, ODBC or OLEDB) being used by the client.
ClientPID	The process ID of the client application.
ConnectionID	The connection ID of the primary Server. Used when connecting to an "Always Up" backup server.
OptionFlags1	▪ Represented in least significant bit order. ▪ fByteOrder: The byte order used by client for numeric and datetime data types.

Parameter	Description
	▪ 0 = ORDER_X86 ▪ 1 = ORDER_68000 ▪ fChar: The character set used on the client. ▪ 0 = CHARSET_ASCII ▪ 1 = CHARSET_EBDDIC ▪ fFloat: The type of floating point representation used by the client. ▪ 0 = FLOAT_IEEE_754 ▪ 1 = FLOAT_VAX ▪ 2 = ND5000 ▪ fDumpLoad: Set is dump/load or BCP capabilities are needed by the client. ▪ 0 = DUMPLOAD_ON ▪ 1 = DUMPLOAD_OFF ▪ fUseDB: Set if the client desires warning messages on execution of the USE statement. If this flag is not set, the server MUST NOT inform the client when the database changes, and therefore the client will be unaware of any accompanying collation changes. ▪ 0 = USE_DB_ON ▪ 1 = USE_DB_OFF ▪ fDatabase: Set if the change to initial database must succeed if the connection is to succeed. ▪ 0 = INIT_DB_WARN ▪ 1 = INIT_DB_FATAL ▪ fSetLang: Set if the client desires warning messages on execution of a language change statement. ▪ 0 = SET_LANG_OFF ▪ 1 = SET_LANG_ON
OptionFlags2	▪ Represented in least significant bit order. ▪ fLanguage: Set if the change to initial language must succeed if the connect is to succeed. ▪ 0 = INIT_LANG_WARN ▪ 1 = INIT_LANG_FATAL ▪ fODBC: Set if the client is the ODBC driver. This causes the server to set ANSI_DEFAULTS to ON, IMPLICIT_TRANSACTIONS to OFF, TEXTSIZE to 0x7FFFFFFF (2GB) (TDS 7.2 and below), TEXTSIZE to infinite (introduced in TDS 7.3), and

Parameter	Description
	ROWCOUNT to infinite. - 0 = ODBC_OFF - 1 = ODBC_ON - fTranBoundary - fCacheConnect - fUserType: The type of user connecting to the server. - 0 = USER_NORMAL—regular logins - 1 = USER_SERVER—reserved - 2 = USER_REMUSER—DQ login - 3 = USER_SQLREPL—replication login - fIntSecurity: The type of security required by the client. - 0 = INTEGRATED_SECURTY_OFF - 1 = INTEGRATED_SECURITY_ON
OptionFlags3	- Represented in least significant bit order. - fChangePassword: Specifies whether the login request should change. - 0 = No change request. ibChangePassword MUST be 0. - 1 = Request to change login's password. - fSendYukonBinaryXML: 1 if XML data type instances are returned as binary XML. - fUserInstance: 1 if client is requesting separate process to be spawned as user instance. - fUnknownCollationHandling: This bit is used by the server to determine if a client is able to properly handle collations introduced after TDS 7.2. TDS 7.2 and lower clients are encouraged to use this login packet bit. Servers MUST ignore this bit when it is sent by TDS 7.3 or higher clients. See [MSDN-SQLCollation] and [MS-LCID] document for the complete list of SQL Server Collations and Windows LCIDs. - 0 = The server MUST restrict the collations sent to a specific set of collations. It MAY disconnect or send an error if some other value is outside the specific collation set. The client MUST properly support all collations within the collation set. - 1 = The server MAY send any collation that fits in the storage space. The client MUST be able to both properly support collations and gracefully fail for those it does not support.
TypeFlags	- Represented in least significant bit order. - fSQLType: The type of SQL the client sends to the server.

Parameter	Description
	- 0 = SQL_DFLT - 1 = SQL_TSQL - fOLEDB: Set if the client is the OLEDB driver. This causes the server to set ANSI_DEFAULTS to ON, IMPLICIT_TRANSACTIONS to OFF, TEXTSIZE to 0x7FFFFFFF (2GB) (TDS 7.2 and below), TEXTSIZE to infinite (introduced in TDS 7.3), and ROWCOUNT to infinite. - 0 = OLEDB_OFF - 0 = OLEDB_OFF
ClientTimeZone	The time zone of the client machine.
ClientLCID	The language code identifier (LCID) value for the client collation. If ClientLCID is specified, the specified collation is set as the session collation. Note that the total ClientLCID is 4 bytes, which implies that there is no support for SQL Sort orders.
OffsetLength	The variable portion of this message. A stream of bytes in the order shown, indicates the offset (from the start of the message) and length of various parameters: - IbHostname & cchHostName: The client machine name. - IbUserName & cchUserName: The client user ID. - IbPassword & cchPassword: The password supplied by the client. - IbAppName & cchAppName: The client application name. - IbServerName & cchServerName: The server name. - ibUnused & cbUnused: reserved for future use. - ibCltIntName & cchCltIntName: The interface library name (ODBC or OLEDB). - ibLanguage & cchLanguage: The initial language (overrides the user ID's default language). - ibDatabase & cchDatabase: The initial database (overrides the user ID's default database). - ClientID: The unique client ID (created used NIC address). - ibSSPI & cbSSPI: SSPI data. If cbSSPI < USHRT_MAX, then this length MUST be used for SSPI and cbSSPILong MUST be ignored. If cbSSPI == USHRT_MAX, then cbSSPILong MUST be checked. If cbSSPILong > 0, then that value MUST be used. If cbSSPILong ==0, then cbSSPI (USHRT_MAX) MUST be used. - ibAtchDBFile & cchAtchDBFile: The file name for a database that is to be attached during the connection process. - ibChangePassword & cchChangePassword: New password for the specified login. - cbSSPILong: Used for large SSPI data when cbSSPI==USHRT_MAX.

Parameter	Description
Data	The actual variable-length data portion referred to by OffsetLength.

Login Data Validation Rules

cchHostName MUST specify at most 128 Unicode characters.

cchUserName MUST specify at most 128 Unicode characters.

cchPassword MUST specify at most 128 Unicode characters.

cchAppName MUST specify at most 128 Unicode characters.

cchServerName MUST specify at most 128 Unicode characters.

cchCltIntName MUST specify at most 128 Unicode characters.

cchLanguage MUST specify at most 128 Unicode characters.

cchDatabase MUST specify at most 128 Unicode characters.

cchAtchDBFile MUST specify at most 260 Unicode characters.

cchChangePassword MUST specify at most 128 Unicode characters.

The value at ibUserName—if specified—is semantically enclosed in brackets ([]) and MUST conform to the rules for valid delimited object identifiers. Login MUST fail otherwise.

The value at ibDatabase—if specified—is semantically enclosed in brackets ([]) and MUST conform to the rules for valid delimited object identifiers. Login MUST fail otherwise.

Before submitting a password from the client to the server, starting with the position pointed to by IbPassword, the client SHOULD take each byte and swap the four high bits with the four low bits. Then the client SHOULD do a bit-XOR with 0xA5 (10100101). The client should do this for every byte in the password buffer.Before reading a submitted password, starting with the position pointed to by IbPassword, the server SHOULD do a bit-XOR with 0xA5 (10100101). Then it should take each byte and swap the four high bits with the four low bits. The server should do this for every byte in the password buffer.

2.2.6.4 PRELOGIN

Stream Name:

```
PRELOGIN
```

Stream Function:

A message sent by the client to set up context for login. The server responds to a client PRELOGIN message with a message of packet header type 0x04 and the packet data containing a PRELOGIN structure.

This message stream is also used to wrap SSL handshake payload, should encryption be needed. In this scenario, where PRELOGIN message is transporting the SSL handshake payload, the packet data is simply the raw bytes of the SSL handshake payload.

Stream Comments:

- Packet header type 0x12.
- A sample PRELOGIN message is in section 4.1.

Stream-Specific Rules:

```
UL_VERSION        =   ULONG          ; version of the sender

US_SUBBUILD       =   USHORT         ; sub-build number of the sender

B_FENCRYPTION     =   BYTE
B_INSTVALIDITY    =   *BYTE %x00     ; name of SQL Server instance
                                     ; or just %x00
UL_THREADID       =   ULONG          ; client application thread id
                                     ; used for debugging purposes
B_MARS            =   BYTE           ; sender requests MARS support
TERMINATOR        =   %xFF           ; signals end of PRELOGIN message

PL_OPTION_DATA    =   *BYTE          ; actual data for the option
PL_OFFSET         =   USHORT         ; big endian
PL_OPTION_LENGTH  =   USHORT         ; big endian
PL_OPTION_TOKEN   =   BYTE           ; token value representing the option

PRELOGIN_OPTION   =   (PL_OPTION_TOKEN
                      PL_OFFSET
                      PL_OPTION_LENGTH)
                      /
                      TERMINATOR

SSL_PAYLOAD       =   *BYTE           ; SSL handshake raw payload
```

Stream Definition:

```
PRELOGIN          =   (*PRELOGIN_OPTION
                      *PL_OPTION_DATA)
                      /
                      SSL_PAYLOAD
```

PL_OPTION_TOKEN is described in the following table.

PL_OPTION_TOKEN	Value	Description
VERSION	0x00	The server MAY use the VERSION sent by the client to the server for debugging purposes or MAY ignore the value. The client MAY use the version returned from the server to determine which features should be enabled or disabled. The client MAY do this only if they know this feature

Description for VERSION:

```
PL_OPTION_DATA = UL_VERSION
                 US_SUBBUILD
```

PL_OPTION_TOKEN	Value	Description
		is supported by that version of the database.
ENCRYPTION	0x01	`PL_OPTION_DATA = B_FENCRYPTION`
INSTOPT	0x02	`PL_OPTION_DATA = B_INSTVALIDITY`
THREADID	0x03	`PL_OPTION_DATA = UL_THREADID` This value SHOULD be empty when being sent from the server to the client.
MARS	0x04	`PL_OPTION_DATA = B_MARS` ▪ 0x00 = Off ▪ 0x01 = On
TERMINATOR	0xFF	Termination token.

Notes:

- PL_OPTION_TOKEN VERSION MUST be the first token sent as part of PRELOGIN. If this is not the case, the connection is closed by the server.
- TERMINATOR does not include length and offset specifiers.
- If encryption is agreed upon during pre-login, SSL negotiation between client and server happens immediately after the PRELOGIN packet. Then login proceeds. Please refer to section 3.3.5.1 for additional information.
- A PRELOGIN message wrapping the SSL_PAYLOAD will only occur after the initial PRELOGIN message containing the PRELOGIN_OPTION and PL_OPTION_DATA information.

Encryption

During the pre-login handshake, the client and the server will negotiate the wire encryption to be used. The possible encryption option values are as follows.

Setting	Value	Description
ENCRYPT_OFF	0x00	Encryption is available but off.
ENCRYPT_ON	0x01	Encryption is available and on.
ENCRYPT_NOT_SUP	0x02	Encryption is not available.
ENCRYPT_REQ	0x03	Encryption is required.

The client sends the server the value ENCRYPT_OFF, ENCRYPT_NOT_SUP, or ENCRYPT_ON. Depending upon whether the server has encryption available and enabled, the server will respond with an ENCRYPTION value in the response according to the following table.

Client	Server ENCRYPT_OFF	Server ENCRYPT_ON	Server ENCRYPT_NOT_SUP
ENCRYPT_OFF	ENCRYPT_OFF	ENCRYPT_REQ	ENCRYPT_NOT_SUP
ENCRYPT_ON	ENCRYPT_ON	ENCRYPT_ON	ENCRYPT_NOT_SUP (connection terminated)
ENCRYPT_NOT_SUP	ENCRYPT_NOT_SUP	ENCRYPT_REQ (connection terminated)	ENCRYPT_NOT_SUP

Assuming that the client is capable of encryption, the server will require the client to behave in the following manner.

Client	Value returned from server when ENCRYPT_OFF	Value returned from server when ENCRYPT_ON	Value returned from server when ENCRYPT_REQ	Value returned from server when ENCRYPT_NOT_SUP
ENCRYPT_OFF	Encrypt login packet only	Encrypt entire connection	Encrypt entire connection	No encryption
ENCRYPT_ON	Error (connection terminated)	Encrypt entire connection	Encrypt entire connection	Error (connection terminated)

If the client and server negotiate to enable encryption, an SSL handshake will take place immediately after the initial PRELOGIN/table response message exchange. The SSL payloads MUST be transported as data in TDS buffers with the message type set to 0x12 in the packet header. For example:

```
0x 12 01 00 4e 00 00 00 00// Buffer Header
0x 16 03 01 00 &// SSL payload
```

This applies to SSL traffic. <9> Upon successful completion of the SSL handshake, the client will proceed to send the LOGIN7 stream to the server to initiate authentication.

Instance Name

If available, the client may send the server the name of the instance to which it is connecting as a NULL-terminated multi-byte character set (MBCS) string in the INSTOPT option. If the string is non-empty, then the server will compare it to its instance name (in the server's locale) and if there is a mismatch, the server will return an INSTOPT option containing a byte with the value of 1 in the pre-login table response message. Otherwise, the server will return an INSTOPT containing a byte with the value of 0. The client can then use this information for verification purposes and could terminate the connection if the instance name is incorrect.

2.2.6.5 RPC Request

Stream Name:

```
RPCRequest
```

Stream Function:

Request to execute an RPC.

Stream Comments:

- Packet header type 0x03.
- To execute an RPC on the server, the client sends a RPCRequest data stream to the server. This is a binary stream that contains the RPC Name (or ProcID), Options, and Parameters. Each RPC MUST be contained within a separate message and not mixed with other SQL commands.
- A sample RPCRequest message is in section 4.6.

Stream-Specific Rules:

```
ProcID           =    USHORT
ProcIDSwitch     =    %xFF %xFF
ProcName         =    US_VARCHAR
NameLenProcID    =    ProcName
                      /
                      (ProcIDSwitch ProcID)

fWithRecomp      =    BIT
fNoMetaData      =    BIT
fReuseMetaData   =    BIT
OptionFlags      =    fWithRecomp
                      fNoMetaData
                      fReuseMetaData
                      13FRESERVEDBIT

fByRefValue      =    BIT
fDefaultValue    =    BIT
StatusFlags      =    fByRefValue
                      fDefaultValue
                      6FRESERVEDBIT

ParamMetaData    =    B_VARCHAR
                      StatusFlags
                      (TYPE_INFO / TVP_TYPE_INFO)    ; (TVP_TYPE_INFO introduced in TDS 7.3)
ParamLenData     =    TYPE_VARBYTE

ParameterData    =    ParamMetaData
                      ParamLenData;

BatchFlag        =    %x80 / %xFF   ; (TDS 7.1: %x80 / TDS 7.2: %xFF)
NoExecFlag       =    %xFE          ; (introduced in TDS 7.2)

RPCReqBatch      =    NameLenProcID
                      OptionFlags
                      *ParameterData
```

The length for the instance value of UDTs is specified as a ULONGLONG. Also note that ParameterData is repeated once for each parameter in the request.

A StatusFlags of fDefaultValue bit MUST be zero for TVP_TYPE_INFO.

fByRefValue MUST be zero for TVP_TYPE_INFO.

Stream Definition:

```
RPCRequest       =   ALL_HEADERS
                     RPCReqBatch
                     *((BatchFlag / NoExecFlag) RPCReqBatch)
                     [BatchFlag / NoExecFlag]
```

Note that RpcReqBatch is repeated once for each RPC in the batch.

Stream Parameter Details:

Parameter	Description
ProcID	The number identifying the special stored procedure to be executed. The valid numbers with associated special stored procedure are as follows: ▪ Sp_Cursor = 1 ▪ Sp_CursorOpen = 2 ▪ Sp_CursorPrepare = 3 ▪ Sp_CursorExecute = 4 ▪ Sp_CursorPrepExec = 5 ▪ Sp_CursorUnprepare = 6 ▪ Sp_CursorFetch = 7 ▪ Sp_CursorOption = 8 ▪ Sp_CursorClose = 9 ▪ Sp_ExecuteSql = 10 ▪ Sp_Prepare = 11 ▪ Sp_Execute = 12 ▪ Sp_PrepExec = 13 ▪ Sp_PrepExecRpc = 14 ▪ Sp_Unprepare = 15
ProcIDSwitch	ProcIDSwitch may occur as part of NameLenProcID (see below).
ProcName	The procedure name length (within US_VARCHAR), which MUST be no more than 1046 bytes.

Parameter	Description
NameLenProcID	If the first USHORT contains 0xFFFF the following USHORT contains the PROCID. Otherwise, NameLenProcID contains the parameter name length and parameter name.
OptionFlags	Bit flags in least significant bit order: ▪ fWithRecomp: 1 if RPC is sent with the "with recompile" option. ▪ fNoMetaData: 1 if no metadata is to be returned for the result set. ▪ fReuseMetaData: 1 if the metadata has not changed from the previous call and the server should reuse its cached metadata (the metadata must still be sent).
StatusFlags	Bit flags in least significant bit order: ▪ fByRefValue: 1 if the parameter is passed by reference (OUTPUT parameter) OR 0 if parameter is passed by value. ▪ fDefaultValue: 1 if the parameter being passed is to be the default value.
ParameterData	The parameter name length and parameter name (within B_VARCHAR), the TYPE_INFO of the RPC data and the type-dependent data for the RPC (within TYPE_VARBYTE).
BatchFlag	Distinguishes the start of the next RPC from another parameter within the current RPC. <10> If the version of TDS in use supports these flags, either the BatchFlag element or the NoExecFlag element MUST be present when another RPC request is in the current batch. BatchFlag SHOULD NOT be sent after the last RPCReqBatch. If BatchFlag is received after the last RPCReqBatch is received, the server MUST ignore it.
NoExecFlag	Indicates that the preceding RPC will not be executed. If this separator is found, the previous RPC will not be executed. Instead, an error message will be returned, followed by the DONEPROC marking that the RPC in the batch has finished, and then execution proceeds to the next RPC in the batch. The tabular data set returned will be very similar to what happens if the RPC does not exist—never execute the RPC, just return an error message, followed by DONEPROC, and then execute the next RPC.<11>

2.2.6.6 SQLBatch

Stream Name:

```
SQLBatch
```

Stream Function:

Describes the format of the SQL Batch message.

Stream Comments:

- Packet header type 0x01.
- A sample SQLBatch message is in section 4.4.

Stream-Specific Rules:

```
SQLText           =   UNICODESTREAM
```

Stream Definition:

```
SQLBatch          =   ALL_HEADERS
                      SQLText
```

The Unicode stream contains the query text within the batch. The following is an example of a valid value for SQLText as follows.

```
Select author_id from Authors
```

2.2.6.7 SSPI Message

Stream Name:

```
SSPIMessage
```

Stream Function:

A request to supply data for Security Support Provider Interface (SSPI) security. Note that SSPI uses the Simple and Protected GSS-API Negotiation Mechanism (SPNEGO) [RFC4178] negotiation.

Stream Comments:

- Packet header type 0x11.
- The initial SSPI data block (the initial SPNEGO security token) is sent from the client to the server in the LOGIN7 message. The server MUST respond with an SSPI token that is the SPNEGO security token response from the server. The client MUST respond with another SSPIMessage, after calling the SPNEGO interface with the server's response.
- This continues until completion or an error.
- The server completes the SSPI validation and returns the last SPNEGO security token as an SSPI token within a LOGINACK token.
- A sample SSPIMessage message is in section 4.9.

Stream-Specific Rules:

```
SSPIData          =   BYTESTREAM
```

Stream Definition:

```
SSPIMessage      =   SSPIData
```

Stream Parameter Details

Parameter	Description
SSPIData	The SSPIData length and SSPIData data using US_VARCHAR format.

2.2.6.8 Transaction Manager Request

Stream Name:

```
TransMgrReq
```

Stream Function:

Query and control operations pertaining to the lifecycle and state of local and distributed transaction objects. Note that distributed transaction operations are coordinated through a Distributed Transaction Coordinator (DTC) implemented to the Microsoft DTC Interface Specification through a Distributed Transaction Coordinator (DTC) implemented to the Microsoft DTC Interface Specification [MSDN-DTC].

Stream Comments:

- Packet header type 0x0E.
- A sample Transaction Manager Request message is given in section 4.11.

Stream-Specific Rules:

```
RequestType      =   USHORT
```

Stream Definition:

```
TransMgrReq      =   ALL_Headers
                     RequestType
                     [RequestPayload]
```

RequestPayload details are as specified in the following table.

Stream Parameter Details

Parameter	Description
RequestType	The types of transaction manager operations desired by the client are specified as follows. If an unknown Type is specified, the message receiver SHOULD disconnect the

Parameter	Description
	connection. ▪ 0 = TM_GET_DTC_ADDRESS. Returns DTC network address as result set with a single-column, single-row binary value. ▪ 1 = TM_PROPAGATE_XACT. Imports DTC transaction into the server and returns a local transaction descriptor as a varbinary result set. ▪ 5 = TM_BEGIN_XACT. Begins a transaction and returns the descriptor in an ENVCHANGE type 8.<12> ▪ 6 = TM_PROMOTE_XACT. Converts an active local transaction into a distributed transaction and returns an opaque buffer in an ENVCHANGE type 15.<13> ▪ 7 = TM_COMMIT_XACT. Commits a transaction. Depending on the payload of the request, it can additionally request that another local transaction be started.<14> ▪ 8 = TM_ROLLBACK_XACT. Rolls back a transaction. Depending on the payload of the request it can indicate that after the rollback, a local transaction is to be started.<15> ▪ 9 = TM_SAVE_XACT. Sets a savepoint within the active transaction. This request MUST specify a nonempty name for the savepoint.<16>

RequestPayload	▪ For RequestType TM_GET_DTC_ADDRESS: The RequestPayload SHOULD be a zero-length US_VARBYTE
	`RequestPayload    =    US_VARBYTE`
	▪ For RequestType TM_PROPAGATE_XACT: Data contains an opaque buffer used by the server to enlist in a DTC transaction [MSDN-ITrans].
	`RequestPayload    =    US_VARBYTE`
	▪ For RequestType TM_BEGIN_XACT:
	`ISOLATION_LEVEL   =    BYTE`
	`BEGIN_XACT_NAME   =    B_VARBYTE`
	`RequestPayload    =    ISOLATION_LEVEL`
	`BEGIN_XACT_NAME`
	This request begins a new transaction, or increments trancount if already in a transaction. If BEGIN_XACT_NAME is nonempty, a transaction is started with the specified name. See the definition for isolation level at the end of this table.
	▪ For RequestType TM_PROMOTE_XACT – No payload.
	This message promotes the transaction of the current request (specified in the Transaction Descriptor header). The current transaction must be part

Parameter	Description
	of the specified header. Note that TM_PROMOTE_XACT is supported only for transactions initiated via TM_BEGIN_XACT, or via piggy back operation on TM_COMMIT/TM_ROLLBACK. An error is returned if TM_PROMOTE_XACT is invoked for a TSQL initiated transaction.

- For RequestType TM_COMMIT_XACT:

```
fBeginXact       =   BIT

XACT_FLAGS       =   fBeginXact
                     7FRESERVEDBIT

ISOLATION_LEVEL  =   BYTE

XACT_NAME        =   B_VARBYTE
BEGIN_XACT_NAME  =   B_VARBYTE

RequestPayload   =   XACT_NAME
                     XACT_FLAGS
                     [ISOLATION_LEVEL
                     BEGIN_XACT_NAME]
```

Without additional flags specified, this command is semantically equivalent to issuing a TSQL COMMIT statement.

The flags in XACT_FLAGS are represented in least significant bit order.

If fBeginXact is 1, then a new local transaction is started after the commit operation is done.

If fBeginXact is 1, then ISOLATION_LEVEL can specify the isolation level to use to start the new transaction, according to the definition at the end of this table. If fBeginXact is 0, then ISOLATION_LEVEL SHOULD NOT be present.

Specifying ISOLATION_LEVEL allows the isolation level to remain in effect for the session, once the xact ends.

If fBeginXact is 0, BEGIN_XACT_NAME SHOULD NOT be present. If fBeginXact is 1, BEGIN_XACT_NAME is nonempty.

If fBeginXact is 1, a transaction MUST be started with the specified name.

See the definition for isolation level at the end of this table.

- For RequestType TM_ROLLBACK_XACT:

```
fBeginXact       =   BIT

XACT_FLAGS       =   fBeginXact
                     7FRESERVEDBIT
```

Parameter	Description

	```
	ISOLATION_LEVEL  =   BYTE

	XACT_NAME        =   B_VARBYTE
	BEGIN_XACT_NAME  =   B_VARBYTE

	RequestPayload   =   XACT_NAME
	                     XACT_FLAGS
	                     [ISOLATION_LEVEL
	                     BEGIN_XACT_NAME]
	```
	The flags in XACT_FLAGS are represented in least significant bit order.
	If XACT_NAME is nonempty, this request rolls back the named transaction. This implies that if XACT_NAME specifies a savepoint name, the rollback only goes back until the specified savepoint.
	Without additional flags specified, this command is semantically equivalent to issuing a TSQL ROLLBACK statement under the current transaction.
	If fBeginXact is 1, then a new local transaction is started after the commit operation is done.
	If fBeginXact is 1, then ISOLATION_LEVEL can specify the isolation level to use to start the new transaction, according to the definition at the end of this table. If fBeginXact is 0, then ISOLATION_LEVELSHOULD NOT be present.
	Specifying ISOLATION_LEVEL allows the isolation level to remain in effect for the session, once the xact ends.
	If fBeginXact is 0, BEGIN_XACT_NAME SHOULD NOT be present. If fBeginXact is 1, BEGIN_XACT_NAME MAY be nonempty.
	If fBeginXact is 1, a transaction MUST be started with the specified name.
	If fBeginXact is 1, and the ROLLBACK only rolled back to a savepoint, the Begin_Xact operation is ignored and trancount remains unchanged.
	See the definition for isolation level at the end of this table.
	▪ For RequestType TM_SAVE_XACT:
	```
	XACT_SAVEPOINT_NAME =   B_VARBYTE

	RequestPayload      =   XACT_SAVEPOINT_NAME
	```
	A nonempty name must be specified as part of this request. Otherwise, an error is raised.

ISOLATION_LEVEL MUST have one of the following values.

Value	Description
0x00	No isolation level change requested. Use current.
0x01	Read Uncommitted.
0x02	Read Committed.
0x03	Repeatable Read.
0x04	Serializable.
0x05	Snapshot.

2.2.7 Packet Data Token Stream Definition

This section describes the various tokens supported in a token-based packet data stream, as described in section 2.2.4.2. The corresponding message types that use token-based packet data streams are identified in the table in section 2.2.4.

2.2.7.1 ALTMETADATA

Token Stream Name:

```
ALTMETADATA
```

Token Stream Function:

Describes the data type, length and name of column data that result from a COMPUTE query.

Token Stream Comments:

The token value is 0x88.

This token is used to tell the client the data type and length of the compute column data. It describes the format of the data found in an ALTROW *data stream.*

All ALTMETADATA *data streams* are grouped together.

This stream does not occur without a preceding COLMETADATA, though there might be COLINFO and TABNAME streams in between.

Token Stream-Specific Rules:

```
TokenType         =   BYTE
Count             =   USHORT
Id                =   USHORT
ByCols            =   UCHAR

Op                =   BYTE
Operand           =   USHORT
UserType          =   USHORT/ULONG; (changed to ULONG in TDS 7.2)
```

```
fNullable        =   BIT
fCaseSen         =   BIT
usUpdateable     =   2BIT        ; 0 = ReadOnly
                                 ; 1 = Read/Write
                                 ; 2 = Unused
fIdentity        =   BIT
fComputed        =   BIT         ; (Introduced in TDS 7.2)
usReservedODBC   =   2BIT
fFixedLenCLRType =   BIT         ; (Introduced in TDS 7.2)
usReserved       =   7BIT

Flags            =   fNullable
                     fCaseSen
                     usUpdateable
                     fIdentity
                     (FRESERVEDBIT / fComputed)
                     usReservedODBC
                     (FRESERVEDBIT / fFixedLenCLRType)
                     usReserved

NumParts         =   BYTE                  ; (introduced in TDS 7.2)
PartName         =   US_VARCHAR            ; (introduced in TDS 7.2)

TableName        =   US_VARCHAR            ; (removed in TDS 7.2)
                     /
                     (NumParts
                     1*PartName)           ; (introduced in TDS 7.2)
ColName          =   B_VARCHAR
ColNum           =   USHORT

ComputeData      =   Op
                     Operand
                     UserType
                     Flags
                     TYPE_INFO
                     [TableName]
                     ColName
```

The **TableName** field is specified only if text, ntext or image columns are included in the result set.

Token Stream Definition:

```
ALTMETADATA      =   TokenType
                     Count
                     Id
                     ByCols
                     *<ByCols>ColNum
                     *<Count>ComputeData
```

Token Stream Parameter Details:

Parameter	**Description**
TokenType	ALTMETADATA_TOKEN
Count	The count of columns (number of aggregate operators) in the token stream.
Id	The Id of the compute statement to which the compute column formats apply. This Id lets the client correctly interpret later ALTROW *data streams.*
ByCols	The number of columns in the "by" list of the compute clause. For example, the compute clause *compute count(sales) by year, month, division, department* has 4 "by" columns.
Op	The type of aggregate operator.

```
AOPSTDEV       = %x30   ; Standard deviation (STDEV)
AOPSTDEVP      = %x31   ; Standard deviation of the population (STDEVP)
AOPVAR         = %x32   ; Variance (VAR)
AOPVARP        = %x33   ; Variance of population (VARP)
AOPCNT         = %x4B   ; Count of rows (COUNT)
AOPSUM         = %x4D   ; Sum of the values in the rows (SUM)
AOPAVG         = %x4F   ; Average of the values in the rows (AVG)
AOPMIN         = %x51   ; Minimum value of the rows (MIN)
AOPMAX         = %x52   ; Maximum value of the rows (MAX)
```

Parameter	**Description**
Operand	The column number, starting from 1, in the select list that is the operand to the aggregate operator.
UserType	The user-defined data type of the column. The value will be 0x0000 with the exceptions of TIMESTAMP (0x0050) and alias types (greater than 0x00FF).
Flags	These bit flags are described in least significant bit order. With the exception of **fNullable**, all of these bit flags SHOULD be set to zero. Please refer to section 2.2.7.4 for a description of each bit flag: ▪ fCaseSens ▪ fNullable is a bit flag, 1 if the column is nullable. ▪ usUpdateable ▪ fIdentity ▪ fComputed ▪ usReservedODBC ▪ fFixedLenCLRType
TableName	See section 2.2.7.4 for a description of TableName. This field SHOULD never be sent because **COMPUTE** queries exclude NTEXT/TEXT/IMAGE.
ColName	The column name. Contains the column name length and column name.
ColNum	USHORT specifying the column number as it appears in the COMPUTE clause. ColNum appears ByCols times.

2.2.7.2 ALTROW

Token Stream Name:

```
ALTROW
```

Token Stream Function:

Used to send a complete row of computed data, where the data format is provided by the ALTMETADATA token.

Token Stream Comments:

- The token value is 0xD3.
- The ALTROW token is similar to the ROW_TOKEN, but also contains an Id field. This Id matches an Id given in ALTMETADATA (one Id for each compute clause). This provides the mechanism for matching row data with correct compute statements.

Token Stream-Specific Rules:

```
TokenType        =   BYTE
Id               =   USHORT

Data             =   TYPE_VARBYTE

ComputeData      =   Data
```

Token Stream Definition:

```
ALTMETADATA      =   TokenType
                     Id
                     *<Count>ComputeData
```

The ComputeData element is repeated Count times (where Count is specified in ALTMETADATA_TOKEN).

Token Stream Parameter Details:

Parameter	Description
TokenType	ALTROW_TOKEN
Id	The Id of the compute statement to which the compute column formats apply. This Id lets the client correctly interpret later ALTROW *data streams.*
Data	The actual data for the column. The TYPE_INFO information describing the data type of this data is given in the preceding COLMETADATA_TOKEN, ALTMETDATA_TOKEN or OFFSET_TOKEN.

2.2.7.3 COLINFO

Token Stream Name:

```
COLINFO
```

Token Stream Function:

Describes the column information in browse mode [MSDN-BROWSE], sp_cursoropen, and sp_cursorfetch.

Token Stream Comments

- The token value is 0xA5.
- The TABNAME token contains the actual table name associated with COLINFO.

Token Stream Specific Rules:

```
TokenType           =    BYTE
Length              =    USHORT

ColNum              =    BYTE
TableNum            =    BYTE
Status              =    BYTE
ColName             =    US_VARCHAR

ColProperty         =    ColNum
                         TableNum
                         Status
                         [ColName]
```

The ColInfo element is repeated for each column in the result set.

Token Stream Definition:

```
COLINFO             =    TokenType
                         Length
                         1*CpLProperty
```

Token Stream Parameter Details:

Parameter	Description
TokenType	COLINFO_TOKEN
Length	The actual data length, in bytes, of the ColProperty stream. The length does not include token type and length field.
ColNum	The column number in the result set.
TableNum	The number of the base table that the column was derived from. The value is 0 if the value

Parameter	Description
	of Status is EXPRESSION.
Status	0x4: EXPRESSION (the column was the result of an expression). 0x8: KEY (the column is part of a key for the associated table). 0x10: HIDDEN (the column was not a member of the selected list, but was added because it was part of a key for the associated table). 0x20: DIFFERENT_NAME (the column name is different than the selected list column name in the case of a column alias).
ColName	The base column name. This only occurs if DIFFERENT_NAME is set in Status.

2.2.7.4 COLMETADATA

Token Stream Name:

```
COLMETADATA
```

Token Stream Function:

Describes the result set for interpretation of following ROW datastreams.

Token Stream Comments:

- The token value is 0x81.
- This token is used to tell the client the data type and length of the column data. It describes the format of the data found in a ROW *data stream*.
- All COLMETADATA *data streams* are grouped together.

Token Stream-Specific Rules:

```
TokenType        =   BYTE
Count            =   USHORT

UserType         =   USHORT/ULONG; (Changed to ULONG in TDS 7.2)

fNullable        =   BIT
fCaseSen         =   BIT
usUpdateable     =   2BIT        ; 0 = ReadOnly
                                 ; 1 = Read/Write
                                 ; 2 = Unused
fIdentity        =   BIT
fComputed        =   BIT         ; (introduced in TDS 7.2)
usReservedODBC   =   2BIT        ; (only exists in TDS 7.3.A and below)
fSparseColumnSet =   BIT         ; (introduced in TDS 7.3.B)
usReserved2      =   2BIT        ; (introduced in TDS 7.3.B)
fFixedLenCLRType =   BIT         ; (introduced in TDS 7.2)
usReserved       =   4BIT
fHidden          =   BIT         ; (introduced in TDS 7.2)
fKey             =   BIT         ; (introduced in TDS 7.2)
fNullableUnknown =   BIT         ; (introduced in TDS 7.2)

Flags            =  fNullable
```

```
                    fCaseSen
                    usUpdateable
                    fIdentity
                    (FRESERVEDBIT / fComputed)
                    usReservedODBC
                    (FRESERVEDBIT / fFixedLenCLRType)
                    (usReserved   / (FRESERVEDBIT fSparseColumnSet usReserved2))
                    (FRESERVEDBIT / fHidden)
                    (FRESERVEDBIT / fKey)
                    (FRESERVEDBIT / fNullableUnknown)

NumParts        =   BYTE         ; (introduced in TDS 7.2)
PartName        =   US_VARCHAR   ;(introduced in TDS 7.2)

TableName       =   NumParts
                    1*PartName

ColName         =   B_VARCHAR

ColumnData      =   UserType
                    Flags
                    TYPE_INFO
                    [TableName]
                    ColName

NoMetaData      =   %xFF %xFF    ; notify receiver that no metadata
                                 ; will follow the COLMETADATA token;
                                 ; this saves network bandwidth;
                                 ; the receiver may have already
                                 ; cached the metadata previously
```

The TableName element is specified only if text, ntext, or image columns are included in the result set.

Token Stream Definition:

```
COLMETADATA      =   TokenType
                     NoMetaData / (Count *<Count>ColumnData)
```

Token Stream Parameter Details:

Parameter	Description
TokenType	COLMETADATA_TOKEN
Count	The count of columns (number of aggregate operators) in the token stream. In the event that the client requested no metadata to be returned (see OptionFlags parameter in RPCRequest token), the value of Count will be 0xFFFF. This has the same effect on Count as a zero value (for example, no ColumnData is sent).
UserType	The user-defined data type of the column. The value will be 0x0000 with the exceptions of TIMESTAMP (0x0050) and alias types (greater than 0x00FF).
Flags	Bit flags in least significant bit order:

Parameter	Description
	▪ fCaseSen is a bit flag. Set to 1 for string columns with binary collation and always for the XML data type. Set to 0 otherwise. ▪ fNullable is a bit flag. Its value is 1 if the column is nullable. ▪ usUpdateable is a 2-bit field. Its value is 0 if column is read-only, 1 if column is read/write and 2 if updateable is unknown.<17> ▪ fIdentity is a bit flag. Its value is 1 if the column is an identity column. ▪ fComputed is a bit flag. Its value is 1 if the column is a COMPUTED column. ▪ usReservedODBC is a 2-bit field that is used by ODS gateways supporting the ODBC ODS gateway driver. ▪ fFixedLenCLRType is a bit flag. Its value is 1 if the column is a fixed-length CLR UDT. ▪ fSparseColumnSet is a bit flag. Set to 1 if the column is in the sparse column set XML column [MSDN-ColSets] (introduced in TDS 7.3.B). ▪ fHidden is a bit flag. Its value is 1 if the column is part of a hidden primary key created to support a T-SQL SELECT statement containing FOR BROWSE. ▪ fKey is a bit flag. Its value is 1 if the column is part of a primary key for the row and the T-SQL SELECT statement contains FOR BROWSE. ▪ fNullableUnknown is a bit flag. Its value is 1 if it is unknown whether the column may be nullable.
TableName	The fully qualified base table name for this column. Contains the table name length and table name. This exists only for text, ntext and image columns. Specifies how many parts will be returned and then repeats PartName once for each NumParts.
ColName	The column name. Contains the column name length and column name.

2.2.7.5 DONE

Token Stream Name:

```
DONE
```

Token Stream Function:

Indicates the completion status of a command.

Token Stream Comments

- The token value is 0xFD.
- This token is used to indicate the completion of a command. As multiple commands may be sent to the server in a single batch, multiple DONE tokens may be generated. In this case, all but the final DONE token will have a Status value with DONE_MORE bit set (details follow).
- For Ad-Hoc SQL command batches, one DONE token is returned for each statement in the batch. The Statements element consists of all T-SQL statements except variable declarations.

- For execution of commands within stored procedures, DONEPROC and DONEINPROC tokens are used in place of DONE tokens.

Token Stream-Specific Rules:

```
TokenType         =    BYTE
Status            =    USHORT
CurCmd            =    USHORT
DoneRowCount      =    LONG / ULONGLONG
```

The type of the DoneRowCount element depends on the version of TDS.<18>

Token Stream Definition:

```
DONE              =    TokenType
                       Status
                       CurCmd
                       DoneRowCount
```

Token Stream Parameter Details:

Parameter	Description
TokenType	DONE_TOKEN
Status	The Status field MUST be a bitwise 'OR' of the following: ▪ 0x00: DONE_FINAL (this DONE is the final DONE in the request). ▪ 0x1: DONE_MORE (this DONE message is not the final DONE message in the result batch. Subsequent statements to follow). ▪ 0x2: DONE_ERROR (on error occurred on the current command). ▪ 0x4: DONE_INXACT (a transaction is in progress).<19> ▪ 0x10: DONE_COUNT (the DoneRowCount value is valid. This is used to distinguish between a valid value of 0 for DoneRowCount or just an initialized variable). ▪ 0x20: DONE_ATTN (the DONE message is a server acknowledgement of a client ATTENTION message). ▪ 0x100: DONE_SRVERROR (used in place of DONE_ERROR when an error occurred on the current command, which is severe enough to require the result set, if any, to be discarded).
CurCmd	The token of the current command.
DoneRowCount	The count of rows that were affected by the command. The value of DoneRowCount is valid if the value of Status includes DONE_COUNT.

2.2.7.6 DONEINPROC

Token Stream Name:

```
DONEINPROC
```

Token Stream Function:

Indicates the completion status of commands within a stored procedure.

Token Stream Comments

- The token value is 0xFF.
- A DONEINPROC token is sent for each executed statement within a stored procedure.
- A DONEINPROC token MUST be followed by another DONEPROC token or a DONEINPROC token.

Token Stream-Specific Rules:

```
TokenType        =   BYTE
Status           =   USHORT
CurCmd           =   USHORT
DoneRowCount     =   LONG / ULONGLONG
```

The type of the DoneRowCount element depends on the version of TDS. <20>

Token Stream Definition:

```
DONEINPROC       =   TokenType
                     Status
                     CurCmd
                     DoneRowCount
```

Token Stream Parameter Details:

Parameter	Description
TokenType	DONEINPROC_TOKEN
Status	The Status field MUST be a bitwise 'OR' of the following: ▪ 0x1: DONE_MORE (this DONEINPROC message is not the final DONE/DONEPROC/DONEINPROC message in the result batch. Results for the next command are to follow). ▪ 0x2: DONE_ERROR (an error occurred on the current command or execution of a stored procedure was interrupted.). ▪ 0x4: DONE_INXACT (a transaction is in progress).<21> ▪ 0x10: DONE_COUNT (the DoneRowCount value is valid. This is used to distinguish between a valid value of 0 for DoneRowCount or just an initialized variable).

Parameter	Description
	▪ 0x100: DONE_SRVERROR (used in place of DONE_ERROR when an error occurred on the current command which is severe enough to require the result set, if any, to be discarded).
CurCmd	The token of the current command.
DoneRowCount	The count of rows that were affected by the command. The value of DoneRowCount is valid if the value of Status includes DONE_COUNT.

2.2.7.7 DONEPROC

Token Stream Name:

```
DONEPROC
```

Token Stream Function:

Indicates the completion status of commands within a stored procedure. This is also generated for EXEC statements at the ad hoc T-SQL execution level in a language event.

Token Stream Comments:

- The token value is 0xFE.
- A DONEPROC token is sent when all the statements within a stored procedure have been executed.
- A DONEPROC token may be followed by another DONEPROC token or a DONEINPROC only if the DONE_MORE bit is set in the Status value.
- There is a separate DONEPROC token sent for each stored procedure called.

Token Stream-Specific Rules:

```
TokenType         =   BYTE
Status            =   USHORT
CurCmd            =   USHORT
DoneRowCount      =   LONG / ULONGLONG
```

The type of the DoneRowCount element depends on the version of TDS. <22>

Token Stream Definition:

```
DONEPROC          =   TokenType
                      Status
                      CurCmd
                      DoneRowCount
```

Token Stream Parameter Details:

Parameter	Description
TokenType	DONEPROC_TOKEN
Status	The Status field MUST be a bitwise 'OR' of the following: ▪ 0x00: DONE_FINAL (this DONEPROC is the final DONEPROC in the request). ▪ 0x1: DONE_MORE (this DONEPROC message is not the final DONEPROC message in the result batch. Results for the next command are to follow). ▪ 0x2: DONE_ERROR (an error occurred on the current command). ▪ 0x4: DONE_INXACT (a transaction is in progress).<23> ▪ 0x10: DONE_COUNT (the DoneRowCount value is valid. This is used to distinguish between a valid value of 0 for DoneRowCount or just an initialized variable). ▪ 0x80: DONE_RPCINBATCH (this DONEPROC message is associated with an RPC within a set of batched RPCs. This flag is not set on the last RPC in the batch). ▪ 0x100: DONE_SRVERROR (used in place of DONE_ERROR when an error occurred on the current command, which is severe enough to require the result set, if any, to be discarded).
CurCmd	The token of the current command.
DoneRowCount	The count of rows that were affected by the command. The value of DoneRowCount is valid if the value of Status includes DONE_COUNT.

2.2.7.8 ENVCHANGE

Token Stream Name:

```
ENVCHANGE
```

Token Stream Function:

A notification of an environment change (for example, database, language, and so on).

Token Stream Comments:

- The token value is 0xE3.
- Includes old and new environment values.
- Type 4 (Packet size) is sent in response to a LOGIN7 message. The server MAY send a value different from the packet size requested by the client. That value MUST be greater than or equal to 512 and smaller than or equal to 32767. Both the client and the server MUST start using this value for packet size with the message following the login response message.
- Type 13 (Database Mirroring) is sent in response to a LOGIN7 message whenever connection is requested to a database that it is being served as primary in real-time log shipping. The ENVCHANGE stream reflects the name of the partner node of the database that is being log shipped.

- Type 15 (Promote Transaction) is sent in response to transaction manager requests with requests of type 6 (TM_PROMOTE_XACT).
- Type 16 (Transaction Manager Address) is sent in response to transaction manager requests with requests of type 0 (TM_GET_DTC_ADDRESS).

Token Stream-Specific Rules:

```
TokenType         =   BYTE
Length            =   USHORT

Type              =   BYTE
NewValue          =   B_VARBYTE
OldValue          =   B_VARBYTE

EnvValueData      =   Type
                      NewValue
                      [OldValue]

EnvValueChg       =   *EnvValueData
```

The EnvValueData element is repeated once for each environment value that has changed.

Token Stream Definition:

```
ENVCHANGE         =   TokenType
                      Length
                      EnvValueChg
```

Token Stream Parameter Details

Parameter	Description
TokenType	ENVCHANGE_TOKEN
Length	The total length of the ENVCHANGE datastream (EnvValueChg).
Type	The type of environment change: ▪ 1: Database ▪ 2: Language ▪ 3: Character set ▪ 4: Packet size ▪ 5: Unicode data sorting local id ▪ 6: Unicode data sorting comparison flags ▪ 7: SQL Collation ▪ 8: Begin Transaction [MSDN-BEGIN]<24>

Parameter	Description
	- 9: Commit Transaction [MSDN-COMMIT]<25> - 10: Rollback Transaction<26> - 11: Enlist DTC Transaction<27> - 12: Defect Transaction<28> - 13: Real Time Log Shipping<29> - 15: Promote Transaction<30> - 16: Transaction Manager Address<31> - 17: Transaction ended<32> - 18: RESETCONNECTION/RESETCONNECTIONSKIPTRAN Completion Acknowledgement<33> - 19: Sends back name of user instance started per login request<34>

Type	Old Value	New Value
1: Database	OLDVALUE = B_VARBYTE	NEWVALUE = B_VARBYTE
2: Language	OLDVALUE = B_VARBYTE	NEWVALUE = B_VARBYTE
3: Character Set	OLDVALUE = B_VARBYTE	NEWVALUE = B_VARBYTE
4: Packet Size	OLDVALUE = B_VARBYTE	NEWVALUE = B_VARBYTE
5: Unicode data sorting local id	OLDVALUE = %x00	NEWVALUE = B_VARBYTE
6: Unicode data sorting comparison flags	OLDVALUE = %x00	NEWVALUE = B_VARBYTE
7: SQL Collation	OLDVALUE = B_VARBYTE	NEWVALUE = B_VARBYTE
8: Begin Transaction	OLDVALUE = %x00	NEWVALUE = B_VARBYTE
9: Commit Transaction	OLDVALUE = "0x00"	OLDVALUE = B_VARBYTE
10: Rollback Transaction	OLDVALUE = %x00	OLDVALUE = B_VARBYTE
11: Enlist DTC Transaction	OLDVALUE = %x00	NEWVALUE = B_VARBYTE

Type	Old Value	New Value
12: Defect Transaction	OLDVALUE = %x00	OLDVALUE = B_VARBYTE

13: Database Mirroring Partner	OLDVALUE = %x00	PARTNER_NODE = B_VARBYTE
		NEWVALUE = PARTNER_NODE

15: Promote Transaction	OLDVALUE = %x00	DTC_TOKEN = L_VARBYTE;
		ACTUAL_LENGTH = ULONG
		NEWVALUE = ACTUAL_LENGTH DTC_TOKEN

16: Transaction Manager Address (not used)	OLDVALUE = %x00	XACT_MANAGER_ADDRESS = B_VARBYTE
		NEWVALUE = XACT_MANAGER_ADDRESS

17: Transaction Ended	OLDVALUE = %x00	OLDVALUE = B_VARBYTE
18: Reset Completion Acknowledgement	OLDVALUE = %x00	NEWVALUE = %x00
19: Sends back info of user instance for logins (login7) requesting so.	OLDVALUE = %x00	NEWVALUE = B_VARBYTE

Notes:

- For types 1, 2, 3, 4, 5, 6, and 19, the payload is a Unicode string; the LENGTH will always reflect the number of bytes.
- ENVCHANGE types 3, 5, and 6 are only sent back to clients running TDS 7.0 or earlier.
- For Types 8, 9, 10, 11, 12 the ENVCHANGE event is returned only if the transaction lifetime is controlled by the user for example explicit transaction commands, including transactions started by SET IMPLICIT_TRANSACTIONS ON.
- For transactions started/committed under auto commit, no stream is generated.
- For operations that change only the value of @@trancount, no ENVCHANGE stream is generated.
- For ENVCHANGE types 8, 9, 10, 11, 12, 17 the payload of NEWVALUE is a ULONGLONG.

```
NEWVALUE         =   %x08 ULONGLONG   ; where %x08 is the
                                      ; size of ULONGLONG
```

- ENVCHANGE type 11 is sent by the server to confirm that it has joined a distributed transaction as requested through a TM_PROPAGATE_XACT request from the client.
- ENVCHANGE type 12 is only sent when a batch defects from either a DTC or bound session transaction.
- LENGTH for ENVCHANGE type 15 is sent as 0x01 indicating only the length of the type token. Client drivers are responsible for reading the additional payload if type is == 15.

- ENVCHANGE type 17 is sent when a batch is used that specified a descriptor for a transaction that has ended. This is only sent in the bound session [MSDN-BOUND] case.
- ENVCHANGE type 18 always produces empty (0x00) old and new values. It simply acknowledges completion of execution of a RESETCONNECTION/RESETCONNECTIONSKIPTRAN request.
- ENVCHANGE type 19 is sent after LOGIN and after /RESETCONNECTION/RESETCONNECTIONSKIPTRAN when a client has requested use of user instances. It is sent prior to the LOGINACK token.

2.2.7.9 ERROR

Token Stream Name:

```
ERROR
```

Token Stream Function:

Used to send an error message to the client.

Token Stream Comments:

- The token value is 0xAA.

Token Stream-Specific Rules:

```
TokenType         =    BYTE
Length            =    USHORT
Number            =    LONG
State             =    BYTE
Class             =    BYTE
MsgText           =    US_VARCHAR
ServerName        =    B_VARCHAR
ProcName          =    B_VARCHAR
LineNumber        =    USHORT / LONG
```

The type of the LineNumber element depends on the version of TDS.<35>

Token Stream Definition:

```
ERROR             =    TokenType
                       Length
                       Number
                       State
                       Class
                       MsgText
                       ServerName
                       ProcName
                       LineNumber
```

Token Stream Parameter Details

Parameter	Description
TokenType	ERROR_TOKEN
Length	The total length of the ERROR datastream, in bytes.
Number	The error number (numbers less than 20001 are reserved by Microsoft® SQL Server®).
State	The error state, used as a modifier to the error number.
Class	The class (severity) of the error. A class of less than 10 indicates an informational message.
MsgText	The message text length and message text using US_VARCHAR format.
ServerName	The server name length and server name using B_VARCHAR format.
ProcName	The stored procedure name length and the stored procedure name using B_VARCHAR format.
LineNumber	The line number in the SQL command batch or stored procedure that caused the error. Line numbers begin at 1; therefore, if the line number is not applicable to the message, the value of LineNumber will be 0.

Class level	Description
0-9	Informational messages that return status information or report errors that are not severe. <36>
10	Informational messages that return status information or report errors that are not severe. <37>
11-16	Errors that can be corrected by the user.
11	The given object or entity does not exist.
12	A special severity for queries that do not use locking because of special query hints. In some cases, read operations performed by these statements could result in inconsistent data, since locks are not taken to guarantee consistency.
13	Transaction deadlock errors.
14	Security-related errors, such as permission denied.
15	Syntax errors in the Transact-SQL command.
16	General errors that can be corrected by the user.
17-19	Software errors that cannot be corrected by the user. These errors require system administrator action.
17	The statement caused the database server to run out of resources (such as memory, locks, or disk space for the database) or to exceed some limit set by the system administrator.
18	There is a problem in the Database Engine software, but the statement completes execution, and the connection to the instance of the Database Engine is maintained. System administrator

Class level	Description
	action is required.
19	A non-configurable Database Engine limit has been exceeded and the current batch process has been terminated. Error messages with a severity level of 19 or higher stop the execution of the current batch. Severity level 19 errors are rare and must be corrected by the system administrator. Error messages with a severity level from 19 through 25 are written to the error log.
20-25	System problems have occurred. These are fatal errors, which means that the Database Engine task that is executing a statement or batch is no longer running. The task records information about what occurred and then terminates. In most cases, the application connection to the instance of the Database Engine may also terminate. If this happens, depending on the problem, the application might not be able to reconnect. Error messages in this range can affect all of the processes accessing data in the same database and may indicate that a database or object is damaged. Error messages with a severity level from 19 through 25 are written to the error log.
20	Indicates that a statement has encountered a problem. Because the problem has affected only the current task, it is unlikely that the database itself has been damaged.
21	Indicates that a problem has been encountered that affects all tasks in the current database, but it is unlikely that the database itself has been damaged.
22	Indicates that the table or index specified in the message has been damaged by a software or hardware problem. Severity level 22 errors occur rarely. If one occurs, run DBCC CHECKDB to determine whether other objects in the database are also damaged. The problem might be in the buffer cache only and not on the disk itself. If so, restarting the instance of the Database Engine corrects the problem. To continue working, reconnect to the instance of the Database Engine; otherwise, use DBCC to repair the problem. In some cases, restoration of the database might be required. If restarting the instance of the Database Engine does not correct the problem, then the problem is on the disk. Sometimes destroying the object specified in the error message can solve the problem. For example, if the message reports that the instance of the Database Engine has found a row with a length of 0 in a non-clustered index, delete the index and rebuild it.
23	Indicates that the integrity of the entire database is in question because of a hardware or software problem. Severity level 23 errors occur rarely. If one occurs, run DBCC CHECKDB to determine the extent of the damage. The problem might be in the cache only and not on the disk itself. If so, restarting the instance of the Database Engine corrects the problem. To continue working, reconnect to the instance of the Database Engine; otherwise, use DBCC to repair the problem. In some cases, restoration of the database might be required.
24	Indicates a media failure. The system administrator may have to restore the database or resolve a hardware issue.

If an error is produced within a result set, the ERROR token is sent before the DONE token for the statement, and such DONE token is sent with the error bit set.

2.2.7.10 INFO

Token Stream Name:

```
INFO
```

Token Stream Function:

Used to send an information message to the client.

Token Stream Comments

- The token value is 0xAB.

Token Stream-Specific Rules:

```
TokenType           =    BYTE
Length              =    USHORT
Number              =    LONG
State               =    BYTE
Class               =    BYTE
MsgText             =    US_VARCHAR
ServerName          =    B_VARCHAR
ProcName            =    B_VARCHAR
LineNumber          =    USHORT / ULONG
```

The type of the LineNumber element depends on the version of TDS.[<38>]

Token Stream Definition:

```
INFO                =    TokenType
                         Length
                         Number
                         State
                         Class
                         MsgText
                         ServerName
                         ProcName
                         LineNumber
```

Token Stream Parameter Details

Parameter	Description
TokenType	INFO_TOKEN
Length	The total length of the INFO datastream, in bytes.
Number	The info number.[<39>]
State	The error state, used as a modifier to the info Number.
Class	The class (severity) of the error. A class of less than 10 indicates an informational message.
MsgText	The message text length and message text using US_VARCHAR format.
ServerName	The server name length and server name using B_VARCHAR format.
ProcName	The stored procedure name length and the stored procedure name using B_VARCHAR format.

Parameter	Description
LineNumber	The line number in the SQL command batch or stored procedure that caused the error. Line numbers begin at 1; therefore, if the line number is not applicable to the message as determined by the upper layer, the value of LineNumber will be 0.

2.2.7.11 LOGINACK

Token Stream Name:

```
LOGINACK
```

Token Stream Function:

Used to send a response to a login request (LOGIN7) to the client.

Token Stream Comments

- The token value is 0xAD.
- If a LOGINACK is not received by the client as part of the login procedure, the logon to the server is unsuccessful.

Token Stream-Specific Rules:

```
TokenType        =   BYTE
Length           =   USHORT
Interface        =   BYTE
TDSVersion       =   DWORD
ProgName         =   B_VARCHAR

MajorVer         =   BYTE
MinorVer         =   BYTE
BuildNumHi       =   BYTE
BuildNumLow      =   BYTE

ProgVersion      =   MajorVer
                     MinorVer
                     BuildNumHi
                     BuildNumLow
```

Token Stream Definition:

```
LOGINACK         =   TokenType
                     Length
                     Interface
                     TDSVersion
                     ProgName
                     ProgVersion
```

Token Stream Parameter Details

Parameter	Description
TokenType	LOGINACK_TOKEN
Length	The total length, in bytes, of the following fields: Interface, TDSVersion, Progname, and ProgVersion.
Interface	The type of interface with which the server will accept client requests: 0: LDEFSQL (server confirms that whatever is sent by the client is acceptable). 1: LXSQL (TSQL is accepted). 2: LSQL (ANSI SQL version 1 is accepted). 3: LSQL2_1 (ANSI SQL version 2, level 1 is accepted). 4: LSQL2_2 (ANSI SQL version 2, level 2 is accepted).
TDSVersion	The TDS version being used by the server (for example, 0x07000000 for a 7.0 server).<40>
ProgName	The name of the server (for example, "Microsoft® SQL Server® ").
MajorVer	The major version number (0-255).
MinorVer	The minor version number (0-255).
BuildNumHi	The high byte of the build number (0-255).
BuildNumLow	The low byte of the build number (0-255).

2.2.7.12 NBCROW

Token Stream Name:

```
NBCROW
```

Token Stream Function:

NBCROW, introduced in TDS 7.3.B, is used to send a row as defined by the COLMETADATA token to the client with null bitmap compression. Null bitmap compression is implemented by using a single bit to specify if the column is null or not null and also by removing all null column values from the row. Removing the null column values (which can be up to 8 bytes per null instance) from the row provides the compression. The null bitmap contains one bit for each column defined in COLMETADATA. In the null bitmap, a bit value of 1 means the column is null and therefore not present in the row, 0 means the column is not null and is present in the row. The null bitmap is always rounded up to the nearest multiple of 8 bits, so there may be 1 to 7 leftover reserved bits at the end of the null bitmap in the last byte of the null bitmap. NBCROW is only used by TDS result set streams from server to client. NBCROW MUST NOT be used in BulkLoadBCP streams. NBCROW is MUST NOT be used in TVP row streams.

Token Stream Comments

- The token value is 0xD2/210.

Token Stream-Specific Rules:

```
TokenType        =   BYTE
TextPointer      =   B_VARBYTE
Timestamp        =   8BYTE
Data             =   TYPE_VARBYTE
NullBitmap       =   <NullBitmapByteCount>BYTE;    see note on NullBitmapByteCount
ColumnData       =   [TextPointer Timestamp] Data
AllColumnData    =   *ColumnData
```

ColumnData is repeated once for each non-null column of data.

NullBitmapBitCount is equal to the number of columns in COLMETADATA.

NullBitmapByteCount is equal to the smallest number of bytes needed to hold 'NullBitmapBitCount' bits.

The server can decide to send either a NBCROW token or a ROW token. For example, server MAY choose to send a ROW token if there will be no byte savings if the result set has no nullable columns, or if a particular row in a result set has no null values. This implies that NBCROW and ROW tokens can be intermixed in the same result set.

When determining whether or not a specific column is null, consider all the columns from left to right ordered using a zero-based index from 0 to 65534 as they occur in the ColumnData section of the COLMETADATA token. The null bitmap indicates that a column is null using a zero bit at the following byte and bit layout:

```
Byte 1                  Byte 2                  Byte 3
----------------------- ----------------------- -----------------------
07 06 05 04 03 02 01 00 15 14 13 12 11 10 09 08 23 22 21 20 19 18 17 16
```

Hence the first byte will contain flags for columns 0 through 7, with the least significant (or rightmost) bit within the byte indicating the zeroth column and the most significant (or leftmost) bit within the byte indicating the seventh column. Column index 8 would be in the second byte as the least significant bit, etc... If the null bitmap bit is set, the column is null and no null token value for the column will follow in the row. If the null bitmap bit is clear, the column is not null and the value for the column follows in the row.

Token Stream Definition:

```
NBCROW             = TokenType
                     NullBitmap
                     AllColumnData
```

Token Stream Parameter Details

Parameter	Description
TokenType	NBCROW_TOKEN (0xD2)
TextPointer	The length of the text pointer and the text pointer for Data.

Parameter	Description
Timestamp	The timestamp of a text/image column.
Data	The actual data for the column. The TYPE_INFO information describing the data type of this data is given in the preceding COLMETADATA_TOKEN.

2.2.7.13 OFFSET

Token Stream Name:

```
OFFSET                   ; (removed in TDS 7.2)
```

Token Stream Function:

Used to inform the client where in the client's command buffer a particular keyword occurs.

Token Stream Comments:

- The token value is 0x78.
- The token has been removed in TDS 7.2.

Token Stream-Specific Rules:

```
TokenType        =   BYTE
Identifier       =   USHORT
OffSetLen        =   USHORT
```

Token Stream Definition:

```
OFFSET           =   TokenType
                     Identifier
                     OffSetLen
```

Token Stream Parameter Details

Parameter	Description
TokenType	OFFSET_TOKEN
Identifier	The keyword to which OffSetLen refers.
OffsetLen	The offset in the command buffer received by the server of the identifier. The command buffer begin with an OffSetLen value of 0 (MOD 64 kilobytes if value of OffSet is larger than 64 kilobytes).

2.2.7.14 ORDER

Token Stream Name:

```
ORDER
```

Token Stream Function:

Used to inform the client by which columns the data is ordered.

Token Stream Comments

- The token value is 0xA9.
- This token is sent only in the event that an ORDER BY clause is executed.

Token Stream-Specific Rules:

```
TokenType         =   BYTE
Length            =   USHORT
ColNum            =   *USHORT
```

The ColNum element is repeated once for each column within the ORDER BY clause.

Token Stream Definition:

```
ORDER             =   TokenType
                      Length
                      ColNum
```

Token Stream Parameter Details

Parameter	Description
TokenType	ORDER_TOKEN
Length	The total length of the ORDER data stream.
ColNum	The column number in the result set.

2.2.7.15 RETURNSTATUS

Token Stream Name:

```
RETURNSTATUS
```

Token Stream Function:

Used to send the status value of an RPC to the client. The server also uses this token to send the result status value of a T-SQL EXEC query.

Token Stream Comments:

- The token value is 0x79.
- This token MUST be returned to the client when an RPC is executed by the server.

Token Stream-Specific Rules:

```
TokenType         =   BYTE
Value             =   LONG
```

Token Stream Definition:

```
RETURNSTATUS      =   TokenType
                      Value
```

Token Stream Parameter Details

Parameter	Description
TokenType	RETURNSTATUS_TOKEN
Value	The return status value determined by the remote procedure. Return status MUST NOT be NULL.

2.2.7.16 RETURNVALUE

Token Stream Name:

```
RETURNVALUE
```

Token Stream Function:

Used to send the return value of an RPC to the client. When an RPC is executed the associated parameters may be defined as input or output (or "return") parameters. This token is used to send a description of the return parameter to the client. This token is also used to describe the value returned by a UDF when executed as an RPC.

Token Stream Comments:

- The token value is 0xAC.
- Multiple return values may exist per RPC. There is a separate RETURNVALUE token sent for each parameter returned.
- Return parameters are sent in the order in which they are defined in the procedure.
- A UDF cannot have return parameters. As such, if a UDF is executed as an RPC there is exactly one RETURNVALUE token sent to the client.

Token Stream-Specific Rules:

```
TokenType        =   BYTE
ParamName        =   B_VARCHAR
ParamOrdinal     =   USHORT
Status           =   BYTE
UserType         =   USHORT/ULONG; (Changed to ULONG in TDS 7.2)


fNullable        =   BIT
fCaseSen         =   BIT
usUpdateable     =   2BIT            ; 0 = ReadOnly
                                     ; 1 = Read/Write
                                     ; 2 = Unused

fIdentity        =   BIT
fComputed        =   BIT             ; (introduced in TDS 7.2)
usReservedODBC   =   2BIT
fFixedLenCLRType =   BIT             ; (introduced in TDS 7.2)
usReserved       =   7BIT

Flags            =   fNullable
                     fCaseSen
                     usUpdateable
                     fIdentity
                     (FRESERVEDBIT / fComputed)
                     usReservedODBC
                     (FRESERVEDBIT / fFixedLenCLRType)
                     usReserved

TypeInfo         =   TYPE_INFO
Value            =   TYPE_VARBYTE
```

Token Stream Definition:

```
RETURNVALUE      =   TokenType
                     ParamOrdinal
                     ParamName
                     Status
                     UserType
                     Flags
                     TypeInfo
                     Value
```

Token Stream Parameter Details:

Parameter	Description
TokenType	RETURNVALUE_TOKEN
ParamOrdinal	Indicates the ordinal position of the output parameter in the original RPC call. Large Object output parameters are reordered to appear at the end of the stream. First the

Parameter	Description
	group of small parameters is sent, followed by the group of large output parameters. There is no reordering within the groups.
ParamName	The parameter name length and parameter name (within B_VARCHAR).
Status	0x01: If ReturnValue corresponds to OUTPUT parameter of a stored procedure invocation. 0x02: If ReturnValue corresponds to return value of User Defined Function.
UserType	The user-defined data type of the column. The value will be 0x00 00 with the exceptions of TIMESTAMP (0x00 50) and alias types (> 0x00 FF).
Flags	These bit flags are described in least significant bit order. All of these bit flags SHOULD be set to zero. Please refer to section 2.2.7.4 for a description of each bit flag. ▪ fCaseSen ▪ fNullable ▪ usUpdateable<41> ▪ fIdentity ▪ fComputed ▪ usReservedODBC ▪ fFixedLengthCLRType
TypeInfo	The TYPE_INFO for the message.
Value	The type-dependent data for the parameter (within TYPE_VARBYTE).

2.2.7.17 ROW

Token Stream Name:

```
ROW
```

Token Stream Function:

Used to send a complete row, as defined by the COLMETADATA token, to the client.

Token Stream Comments:

- The token value is 0xD1.

Token Stream-Specific Rules:

```
TokenType          =    BYTE

TextPointer        =    B_VARBYTE
Timestamp          =    8BYTE
Data               =    TYPE_VARBYTE
```

```
ColumnData        =   [TextPointer Timestamp]
                      Data

AllColumnData     =   *ColumnData
```

The ColumnData element is repeated once for each column of data.

TextPointer and Timestamp MUST NOT be specified if the instance of type text/ntext/image is a NULL instance (GEN_NULL).

Token Stream Definition:

```
ROW               =   TokenType
                      AllColumnData
```

Token Stream Parameter Details:

Parameter	Description
TokenType	ROW_TOKEN
TextPointer	The length of the text pointer and the text pointer for data.
Timestamp	The timestamp of a text/image column. This is not present if the value of data is CHARBIN_NULL or GEN_NULL.
Data	The actual data for the column. The TYPE_INFO information describing the data type of this data is given in the preceding COLMETADATA_TOKEN, ALTMETDATA_TOKEN or OFFSET_TOKEN.

2.2.7.18 SSPI

Token Stream Name:

```
SSPI
```

Token Stream Function:

The SSPI token returned during the login process.

Token Stream Comments:

- The token value is 0xED.

Token Stream-Specific Rules:

```
TokenType         =   BYTE
SSPIBuffer        =   US_VARBYTE
```

Token Stream Definition:

```
SSPI              =   TokenType
                      SSPIBuffer
```

Token Stream Parameter Details:

Parameter	Description
TokenType	SSPI_TOKEN
SSPIBuffer	The length of the SSPIBuffer and the SSPI buffer using B_VARBYTE format.

2.2.7.19 TABNAME

Token Stream Name:

```
TABNAME
```

Token Stream Function:

Used to send the table name to the client only when in browser mode or from sp_cursoropen.

Token Stream Comments:

- The token value is 0xA4.

Token Stream-Specific Rules:

```
TokenType         =   BYTE
Length            =   USHORT

NumParts          =   BYTE              ; (introduced in TDS 7.1 Revision 1)
PartName          =   US_VARCHAR        ; (introduced in TDS 7.1 Revision 1)

TableName         =   US_VARCHAR        ;(removed in TDS 7.1 Revision 1)
                      /
                      (NumParts
                      1*PartName)       ; (introduced in TDS 7.1 Revision 1)

AllTableNames     =   TableName
```

The TableName element is repeated once for each table name in the query.

Token Stream Definition:

```
TABNAME           =   TokenType
```

```
                                Length
                                AllTableNames
```

Token Stream Parameter Details

Parameter	Description
TokenType	TABNAME_TOKEN
Length	The actual data length, in bytes, of the TABNAME token stream. The length does not include token type and length field.
TableName	The name of the base table referenced in the query statement.

2.2.7.20 TVP ROW

Token Stream Name:

```
TVP ROW
```

Token Stream Function:

Used to send a complete table valued parameter (TVP) row, as defined by the TVP_COLMETADATA token from client to server.

Token Stream Comments:

- The token value is 0x01/1.

Token Stream-Specific Rules:

```
TokenType         =   BYTE
TvpColumnData     =   TYPE_VARBYTE
AllColumnData     =   *TvpColumnData
```

TvpColumnData is repeated once for each column of data with a few exceptions. Please refer to the Flags description of the TVP_COLMETADATA section for details on when certain TvpColumnData items are required to be omitted.

Note that unlike the ROW token, TVP_ROW does not use TextPointer + TimeStamp prefix with TEXT, NTEXT and IMAGE types.

Token Stream Definition:

```
TVP ROW               =   TokenType
                          AllColumnData
```

Token Stream Parameter Details:

Parameter	Description
TokenType	TVP_ROW_TOKEN
TvpColumnData	The actual data for the TVP column. The TYPE_INFO information describing the data type of this data is given in the preceding TVP_COLMETADATA token.

3 Protocol Details

This section describes the important elements of the client software and the server software necessary to support the TDS protocol.

3.1 Common Details

As described in section 1.3, TDS is an application-level protocol that is used for the transfer of requests and responses between clients and database server systems. The protocol defines a limited set of messages through which the client may make a request to the server. The TDS server is message-oriented. Once a connection has been established between the client and server, a complete message is sent from client to server. Following this, a complete response is sent from server to client (with the possible exception of when the client aborts the request), and the server then waits for the next request. Other than this Post-Login state, the other states defined by the TDS protocol are (i) pre-authentication (Pre-Login), (ii) authentication (Login), and (iii) when the client sends an attention message (Attention). These will be expanded upon in subsequent sections.

3.1.1 Abstract Data Model

See sections 3.2.1 and 3.3.1 for the abstract data model of the client and server respectively.

3.1.2 Timers

See section 3.2.2 for a description of the client timer used and section 3.3.2 for a description of the server timer used.

3.1.3 Initialization

None.

3.1.4 Higher-Layer Triggered Events

See section 3.2.4 for information on higher-layer triggered events for a TDS client and section 3.3.4 for a TDS server.

3.1.5 Message Processing Events and Sequencing Rules

The following series of sequence diagrams illustrate the possible messages that may be exchanged between client and server. See sections 3.2.5 and 3.3.5 for specific client and server details regarding message processing events and sequencing rules.

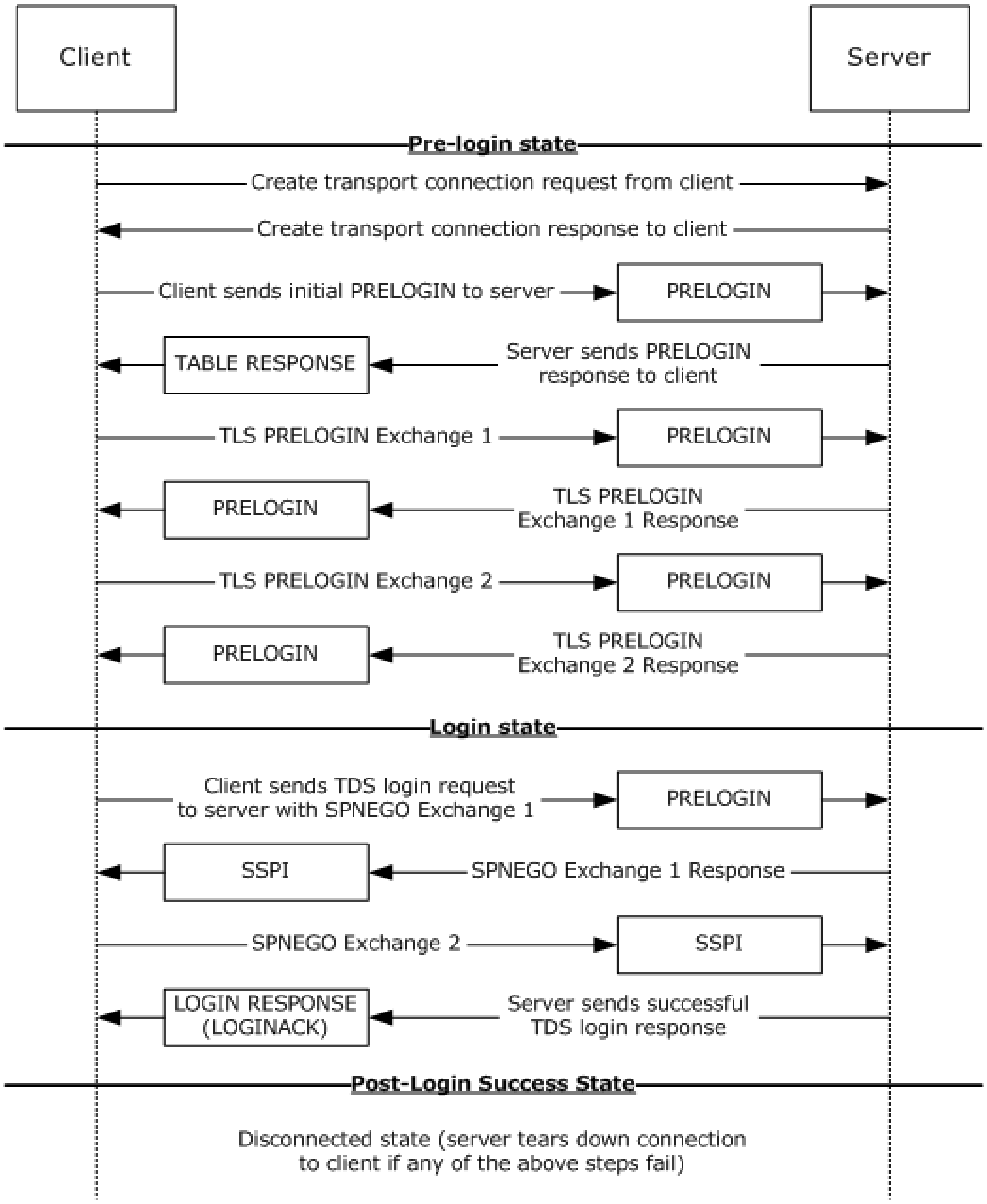
Client
Server
Pre-login state
Create transport connection request from client
Create transport connection response to client
Client sends initial PRELOGIN to server
PRELOGIN
TABLE RESPONSE
Server sends PRELOGIN response to client
TLS PRELOGIN Exchange 1
PRELOGIN
PRELOGIN
TLS PRELOGIN Exchange 1 Response
TLS PRELOGIN Exchange 2
PRELOGIN
PRELOGIN
TLS PRELOGIN Exchange 2 Response
Login state
Client sends TDS login request to server with SPNEGO Exchange 1
PRELOGIN
SSPI
SPNEGO Exchange 1 Response
SPNEGO Exchange 2
SSPI
LOGIN RESPONSE (LOGINACK)
Server sends successful TDS login response
Post-Login Success State
Disconnected state (server tears down connection to client if any of the above steps fail)

Figure 3: Pre-login to post-login sequence

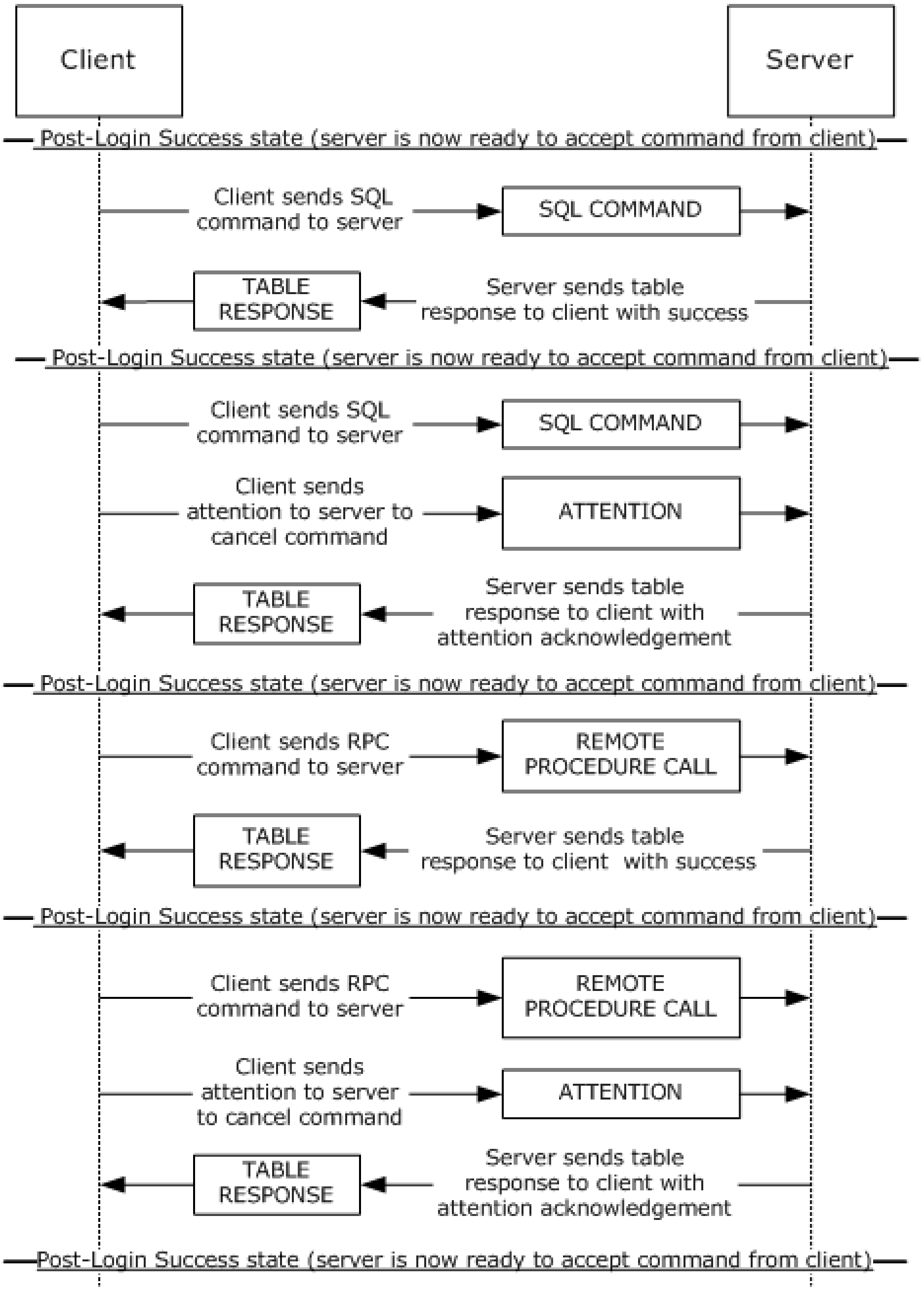
Client
Server
Post-Login Success state (server is now ready to accept command from client)
Client sends SQL command to server
SQL COMMAND
TABLE RESPONSE
Server sends table response to client with success
Post-Login Success state (server is now ready to accept command from client)
Client sends SQL command to server
SQL COMMAND
Client sends attention to server to cancel command
ATTENTION
TABLE RESPONSE
Server sends table response to client with attention acknowledgement
Post-Login Success state (server is now ready to accept command from client)
Client sends RPC command to server
REMOTE PROCEDURE CALL
TABLE RESPONSE
Server sends table response to client with success
Post-Login Success state (server is now ready to accept command from client)
Client sends RPC command to server
REMOTE PROCEDURE CALL
Client sends attention to server to cancel command
ATTENTION
TABLE RESPONSE
Server sends table response to client with attention acknowledgement
Post-Login Success state (server is now ready to accept command from client)

Figure 4: SQL command and RPC sequence

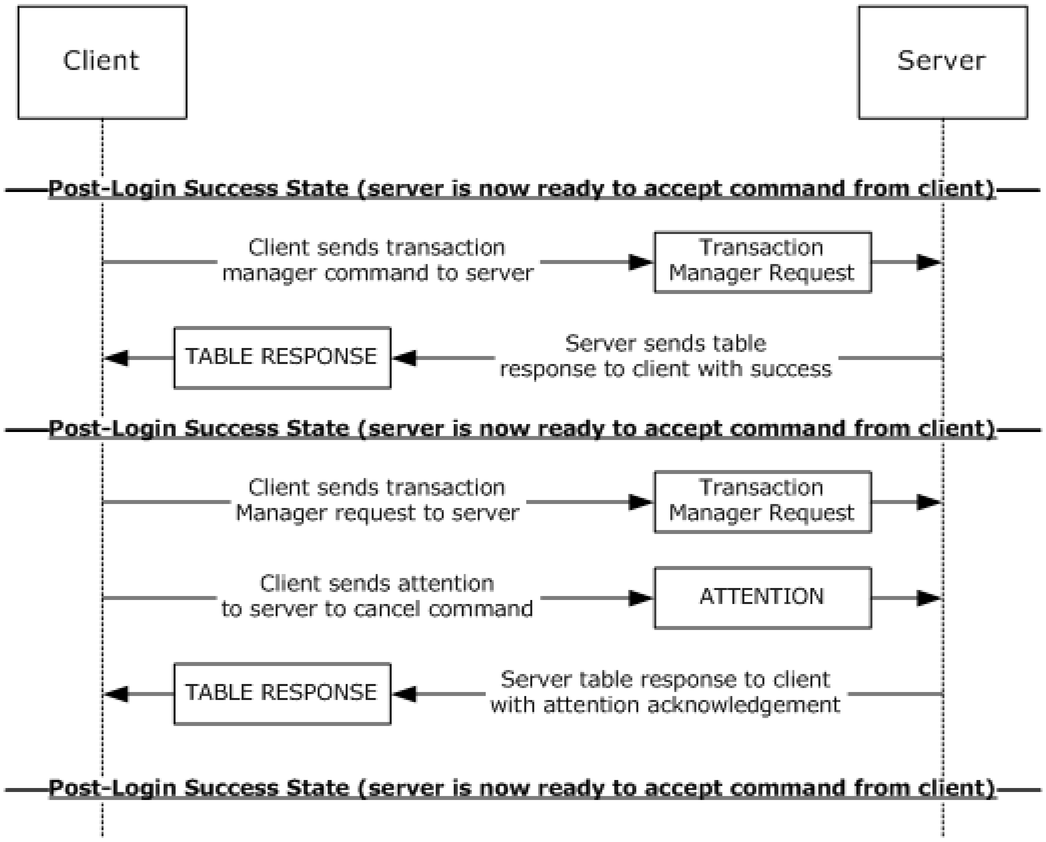
Client
Server
Post-Login Success State (server is now ready to accept command from client)
Client sends transaction manager command to server
Transaction Manager Request
TABLE RESPONSE
Server sends table response to client with success
Post-Login Success State (server is now ready to accept command from client)
Client sends transaction Manager request to server
Transaction Manager Request
Client sends attention to server to cancel command
ATTENTION
TABLE RESPONSE
Server table response to client with attention acknowledgement
Post-Login Success State (server is now ready to accept command from client)

Figure 5: Transaction manager request sequence

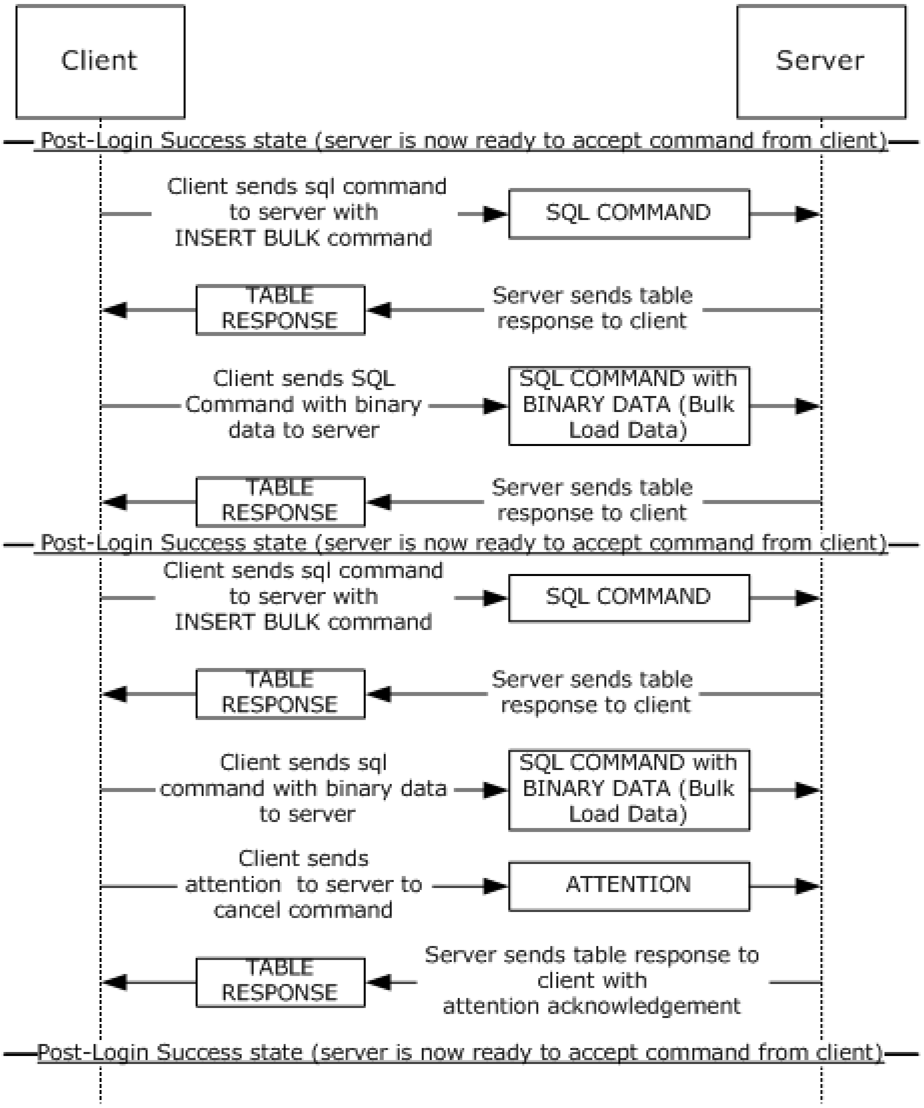
Client
Server
Post-Login Success state (server is now ready to accept command from client)
Client sends sql command to server with INSERT BULK command
SQL COMMAND
TABLE RESPONSE
Server sends table response to client
Client sends SQL Command with binary data to server
SQL COMMAND with BINARY DATA (Bulk Load Data)
TABLE RESPONSE
Server sends table response to client
Post-Login Success state (server is now ready to accept command from client)
Client sends sql command to server with INSERT BULK command
SQL COMMAND
TABLE RESPONSE
Server sends table response to client
Client sends sql command with binary data to server
SQL COMMAND with BINARY DATA (Bulk Load Data)
Client sends attention to server to cancel command
ATTENTION
TABLE RESPONSE
Server sends table response to client with attention acknowledgement
Post-Login Success state (server is now ready to accept command from client)

Figure 6: Bulk insert sequence

3.1.6 Timer Events

See sections 3.2.6 and 3.3.6 for the timer events of the client and server respectively.

3.1.7 Other Local Events

A TDS session is tied to the underlying established network protocol session. As such, loss or termination of a network connection is equivalent to immediate termination of a TDS session.

See sections 3.2.7 and 3.3.7 for the other local events of the client and server respectively.

3.2 Client Details

The following state machine diagram describes TDS on the client side.

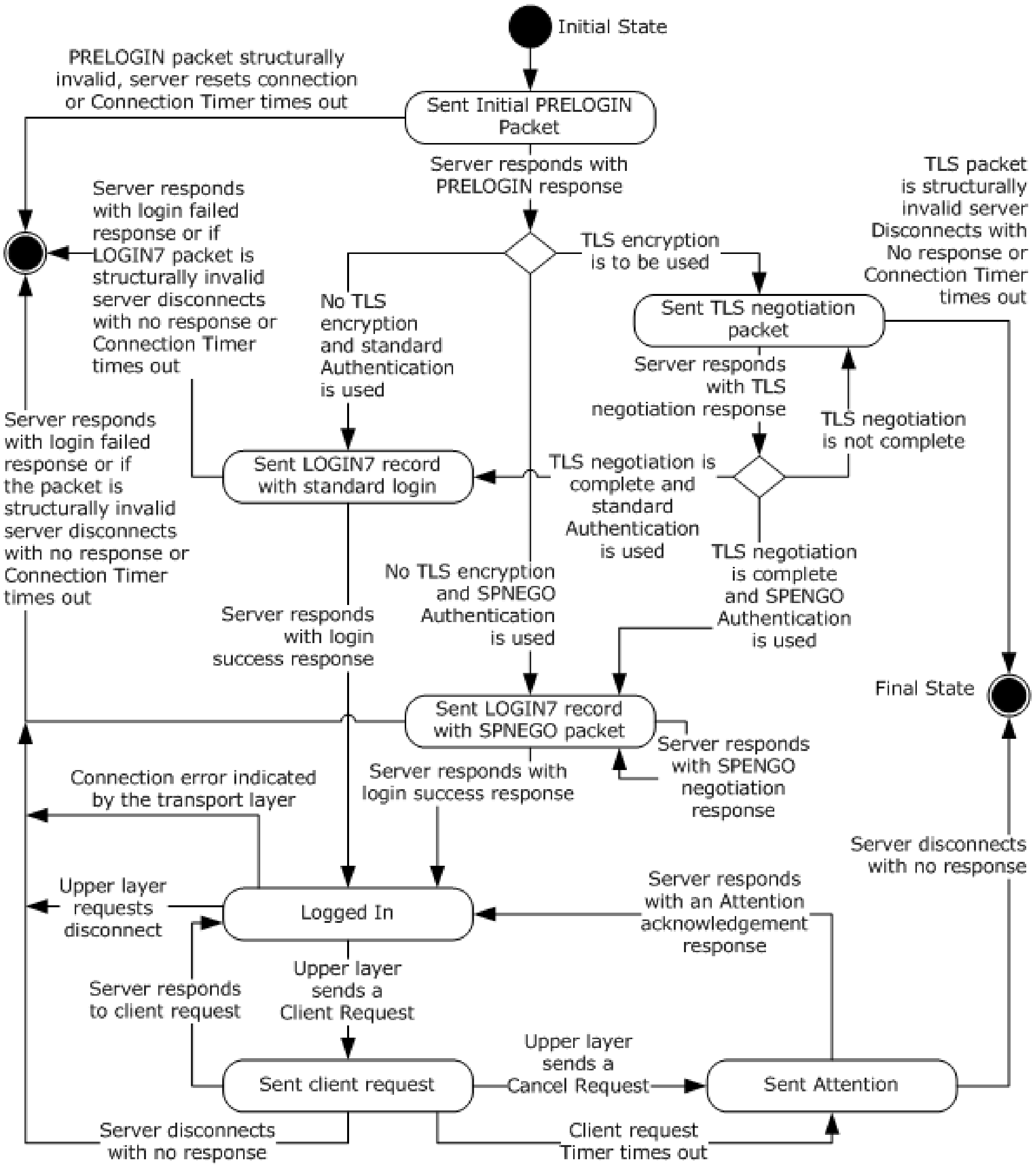
Initial State
PRELOGIN packet structurally invalid, server resets connection or Connection Timer times out
Sent Initial PRELOGIN Packet
Server responds with PRELOGIN response
TLS packet is structurally invalid server Disconnects with No response or Connection Timer times out
Server responds with login failed response or if LOGIN7 packet is structurally invalid server disconnects with no response or Connection Timer times out
TLS encryption is to be used
No TLS encryption and standard Authentication is used
Sent TLS negotiation packet
Server responds with TLS negotiation response
TLS negotiation is not complete
Server responds with login failed response or if the packet is structurally invalid server disconnects with no response or Connection Timer times out
Sent LOGIN7 record with standard login
TLS negotiation is complete and standard Authentication is used
No TLS encryption and SPNEGO Authentication is used
TLS negotiation is complete and SPENGO Authentication is used
Server responds with login success response
Final State
Sent LOGIN7 record with SPNEGO packet
Server responds with SPENGO negotiation response
Connection error indicated by the transport layer
Server responds with login success response
Server disconnects with no response
Upper layer requests disconnect
Logged In
Server responds with an Attention acknowledgement response
Server responds to client request
Upper layer sends a Client Request
Upper layer sends a Cancel Request
Sent client request
Sent Attention
Server disconnects with no response
Client request Timer times out

Figure 7: TDS client state machine

3.2.1 Abstract Data Model

This section describes a conceptual model of possible data organization that an implementation maintains to participate in this protocol. The described organization is provided to facilitate the explanation of how the protocol behaves. This document does not mandate that implementations adhere to this model as long as their external behavior is consistent with that described in this document.

A TDS client may maintain the following data:

- Encryption option. Possible values: no encryption, login-only encryption, full encryption.

- Authentication scheme. Possible values: standard authentication, SSPI authentication.
- Connection time-out value.
- Client request time-out value.
- Cancel time-out value.
- Transaction descriptor value.

3.2.2 Timers

A TDS client SHOULD implement the following three timers:

- Connection Timer. Controls the maximum time spent during the establishment of a TDS connection. The default value SHOULD be 15 seconds. The implementation SHOULD allow the upper layer to specify a nondefault value, including an infinite value (for example no time-out).
- Client Request Timer. Controls the maximum time spent waiting for a query response from the server for a client request sent after the connection has been established. The default value is implementation-dependent. The implementation SHOULD allow the upper layer to specify a non-default value, including an infinite value (for example, no time-out<42>).
- Cancel Timer. Controls the maximum time spent waiting for a query cancellation acknowledgement after an Attention request is sent to the server. The default value is implementation-dependent. The implementation SHOULD allow the upper layer to specify a nondefault value, including an infinite value (for example, no time-out<43>).

For all three timers, a client can implement a minimum timeout value that is as short as desired. If a TDS client implementation implements any of the timers, it MUST implement their behavior according to this specification.

A TDS client SHOULD request the transport to detect and indicate a broken connection if the transport provides such mechanism. If the transport used is TCP, it SHOULD use the TCP Keep-Alives [RFC1122] in order to detect a nonresponding server in case infinite connection time-out or infinite client request time-out is used. The default values of the TCP Keep-Alive values set by a TDS client are 30 seconds of no activity until the first keep-alive packet is sent and 1 second between when successive keep-alive packets are sent if no acknowledgement is received. The implementation SHOULD allow the upper layer to specify other TCP keep-alive values.

3.2.3 Initialization

None.

3.2.4 Higher-Layer Triggered Events

A TDS client MUST support the following events from the upper layer:

- Connection Open Request to establish a new TDS connection to a TDS server.
- Client Request to send a query to a TDS server on an already established TDS connection. The Client Request is a request for one of four types of queries to be sent: SQL Command, SQL Command with Binary Data, Transaction Manager Request, or an RPC.

In addition, it SHOULD support the following event from the upper layer:

- Cancel Request to cancel a client request while waiting for a server response. For example, this enables the upper layer to cancel a long-running client request if the user/upper layer is no longer seeking the result, thus freeing up thus client and server resources. If a client implementation of the TDS protocol supports the Cancel Request event it MUST handle it as described in this specification.

The processing and actions triggered by these events is described in the remaining parts of this section.

When a TDS client receives a Connection Open Request from the upper layer in the "Initial" state of a TDS connection, it performs the following actions:

- If the TDS client implements the Connection Timer, it MUST start the Connection Timer if the connection time-out value is not infinite.
- If there is upper-layer request MARS support, it MUST set the B_MARS byte in the Pre-Login message to 0x01.
- It should send a Pre-Login message to the server using the underlying transport protocol, and
- if the transport does not report an error, it MUST enter the Sent Initial Pre-Login Message state.

When a TDS client receives a Connection Open Request from the upper layer in any state other than the Initial state of a TDS connection, it MUST indicate an error to the upper layer.

When a TDS client receives a Client Request from the upper layer in the "Logged In" state it MUST perform the following actions:

- If the TDS client implements the Query Timer, it MUST start the Client Request Timer if the client request time-out value is not infinite.
- If MARS is enabled, the client MUST keep track whether there is an outstanding active request. If this is the case, then the client MUST initiate a new SMUX session, else an existing SMUX session MAY be used.
- Send either SQL Command, SQL Command with Binary Data, Transaction Manager Request, or a RPC message to the server. The message and its content must match the requested message from the Client Request. If MARS is enabled, the TDS message MUST be passed through to the SMUX layer.
- If the transport does not report an error, then enter the "Sent Client Request" state.

When a TDS client supporting the Cancel Request receives a Cancel Request from the upper layer in the "Sent Client Request" state, it MUST perform the following actions:

- If the TDS client implements the Cancel Timer, it MUST start the Cancel Timer if the Attention request time-out value is not infinite.
- Send an Attention message to the server. This indicates to the server that the currently executing request should be aborted. If MARS is enabled, the Attention message MUST be passed through to the SMUX layer.
- Enter the "Sent Attention" state.

3.2.5 Message Processing Events and Sequencing Rules

The processing of messages received from a TDS server depends on the message type and the state the TDS client is in. The rest of this section describes the processing and actions to take on them. The message type is determined from the TDS packet type and the token stream inside the TDS packet payload, as described in section 2.2.3.

Whenever the TDS client enters either the "Logged In" state or the "**Final State**" state, it MUST stop the Connection Timer (if implemented and running), the Client Request Timer (if implemented and running), and the Cancel Timer (if implemented and running).

Whenever a TDS client receives a **structurally invalid** TDS message, it MUST close the underlying transport connection, indicate an error to the upper layer, and enter the "Final State" state.

When a TDS client receives a Table Response (TDS packet type %x04) from the server, it MUST behave as follows, according to the state of the TDS client.

3.2.5.1 Sent Initial PRELOGIN Packet State

If the response contains a structurally valid PRELOGIN response indicating a success, the TDS client MUST take action according to the Encryption option and Authentication scheme:

- The Encryption option MUST be handled as described in section 2.2.6.4 in the PRELOGIN message description.
- If encryption was negotiated, the TDS client MUST initiate a TLS/SSL handshake, send to the server a TLS/SSL message obtained from the TLS/SSL layer encapsulated in TDS packet(s) of type PRELOGIN (0x12), and enter the "Sent TLS/SSL negotiation packet" state.
- If encryption was not negotiated and the upper layer did not request full encryption, the TDS client MUST send to the server a Login message with the authentication scheme specified by the user, and enter either state "Sent LOGIN7 record with standard login" or "Sent LOGIN7 record with SPNEGO packet" accordingly. The TDS specification does not prescribe the authentication protocol if SSPI authentication is used. The current implementation supports NTLM [NTLM] and Kerberos [RFC4120].
- If encryption was not negotiated and the upper layer requested full encryption, then the TDS client MUST close the underlying transport connection, indicate an error to the upper layer, and enter the "Final State" state.
- If the response received from the server does not contain a structurally valid PRELOGIN response or it contains a structurally valid PRELOGIN response indicating an error, the TDS client MUST close the underlying transport connection, indicate an error to the upper layer, and enter the "Final State" state.

3.2.5.2 Sent TLS/SSL Negotiation Packet State

If the response contains a structurally valid TLS/SSL response message (TDS packet **Type** 0x12), the TDS client MUST pass the TLS/SSL message contained in it to the TLS/SSL layer and MUST proceed as follows:

- If the TLS/SSL layer indicates that further handshake is needed, the TDS client MUST send to the server the TLS/SSL message obtained from the TLS/SSL layer encapsulated in TDS packet(s) of **Type** PRELOGIN (0x12).

- If the TLS/SSL layer indicates successful completion of the TLS/SSL handshake, the TDS client MUST send a Login message to the server with the authentication scheme specified by the user. The TDS client will then enter one of two states, either the "Sent LOGIN7 record with standard login" or "Sent LOGIN7 record with SSPI negotiation packet." The TDS specification does not prescribe the authentication protocol if SSPI authentication is used. The current implementation supports NTLM [NTLM] and Kerberos [RFC4120].
- If login-only encryption was negotiated as described in section Message Syntax in the PreLogin message description, then the first and only the first TDS packet of the Login message MUST be encrypted using TLS/SSL and encapsulated in a TLS/SSL message. All other TDS packets sent or received MUST be in plaintext.
- If full encryption was negotiated as described in section Message Syntax in the PreLogin message description, then all subsequent TDS packets sent or received from this point on MUST be encrypted using TLS/SSL and encapsulated in a TLS/SSL message.
- If the TLS/SSL layer indicates an error, the TDS client MUST close the underlying transport connection, indicate an error to the upper layer, and enter the "Final State" state.

If the response received from the server does not contain a structurally valid TLS/SSL response or it contains a structurally valid response indicating an error, the TDS client MUST close the underlying transport connection, indicate an error to the upper layer, and enter the "Final State" state.

3.2.5.3 Sent LOGIN7 Record with Standard Login State

If the response received from the server contains a structurally valid Login response indicating a successful login, the TDS client MUST indicate successful Login completion to the upper layer and enter the "Logged In" state.

If the response received from the server does not contain a structurally valid Login response or it contains a structurally valid Login response indicating login failure, the TDS client MUST close the underlying transport connection, indicate an error to the upper layer, and enter the "Final State" state.

3.2.5.4 Sent LOGIN7 Record with SPNEGO Packet State

If the response received from the server contains a structurally valid Login response indicating a successful login, the TDS client MUST indicate successful Login completion to the upper layer and enter the "Logged In" state.

If the response received from the server contains a structurally valid SSPI response message, the TDS client MUST send to the server a SSPI message (TDS packet type %x11) containing the data obtained from the applicable SSPI layer.

If the response received from the server does not contain a structurally valid Login response or SSPI response, or if it contains a structurally valid Login response indicating login failure, the TDS client MUST close the underlying transport connection, indicate an error to the upper layer, and enter the "Final State" state.

3.2.5.5 Logged In State

The TDS client waits for notification from the upper layer. If the upper layer requests a query to be sent to the server, the TDS client MUST send the appropriate request to the server and enter the "Sent Client Request" state. If MARS is enabled, the TDS client MUST send the appropriate request to the SMUX layer. If the upper layer requests a termination of the connection, the TDS client MUST disconnect from the server and enter the "Final State" state. If the TDS client detects a connection error from the transport layer, the TDS client MUST disconnect from the server and enter the "Final State" state.

3.2.5.6 Sent Client Request State

If the response received from the server contains a structurally valid response, the TDS client MUST indicate the result of the request to the upper layer and enter the "Logged In" state.

The client has the ability to return data/control to the upper layers while remaining in the Sent Client Request state while the complete response has not been received or processed.

If the TDS client supports Cancel Request and the upper layer requests a Cancel Request to be sent to the server, the TDS client will send an Attention message to the server, start the Cancel Timer, and enter the "Sent Attention" state.

If the response received from the server does not contain a structurally valid response, the TDS client MUST close the underlying transport connection, indicate an error to the upper layer, and enter the "Final State" state.

3.2.5.7 Sent Attention State

If the response is structurally valid and it does not acknowledge the Attention as described in section 2.2.1.6, then the TDS client MUST discard any data contained in the response and remain in the "Sent Attention" state.

If the response is structurally valid and it acknowledges the Attention as described in section 2.2.1.6, then the TDS client MUST discard any data contained in the response, indicate the completion of the query to the upper layer together with the cause of the Attention (either an upper-layer cancellation as described in section 3.2.4 or query time-out as described in section 3.2.2), and enter the "Logged In" state.

If the response received from the server is not structurally valid, then the TDS client MUST close the underlying transport connection, indicate an error to the upper layer, and enter the "Final State" state.

3.2.5.8 Final State

The connection is disconnected. All resources for this connection will be recycled by the TDS server.

3.2.6 Timer Events

If a TDS client implements the Connection Timer and the timer times out, then the TDS client MUST close the underlying connection, indicate the error to the upper layer, and enter the "Final State" state.

If a TDS client implements the Client Request Timer and the timer times out, then the TDS client MUST send an Attention message to the server and enter the "Sent Attention" state.

If a TDS client implements the Cancel Timer and the timer times out, then the TDS client MUST close the underlying connection, indicate the error to the upper layer, and enter the "Final State" state.

3.2.7 Other Local Events

Whenever an indication of a connection error is received from the underlying transport, the TDS client MUST close the transport connection, indicate an error to the upper layer, stop any timers if running, and enter the "Final State" state. If TCP is used as the underlying transport, examples of events that may trigger such action—dependent on the actual TCP implementation—may be media sense loss, a TCP connection going down in the middle of communication, or a TCP keep-alive failure.

3.3 Server Details

The following state machine diagram describes TDS on the server side.

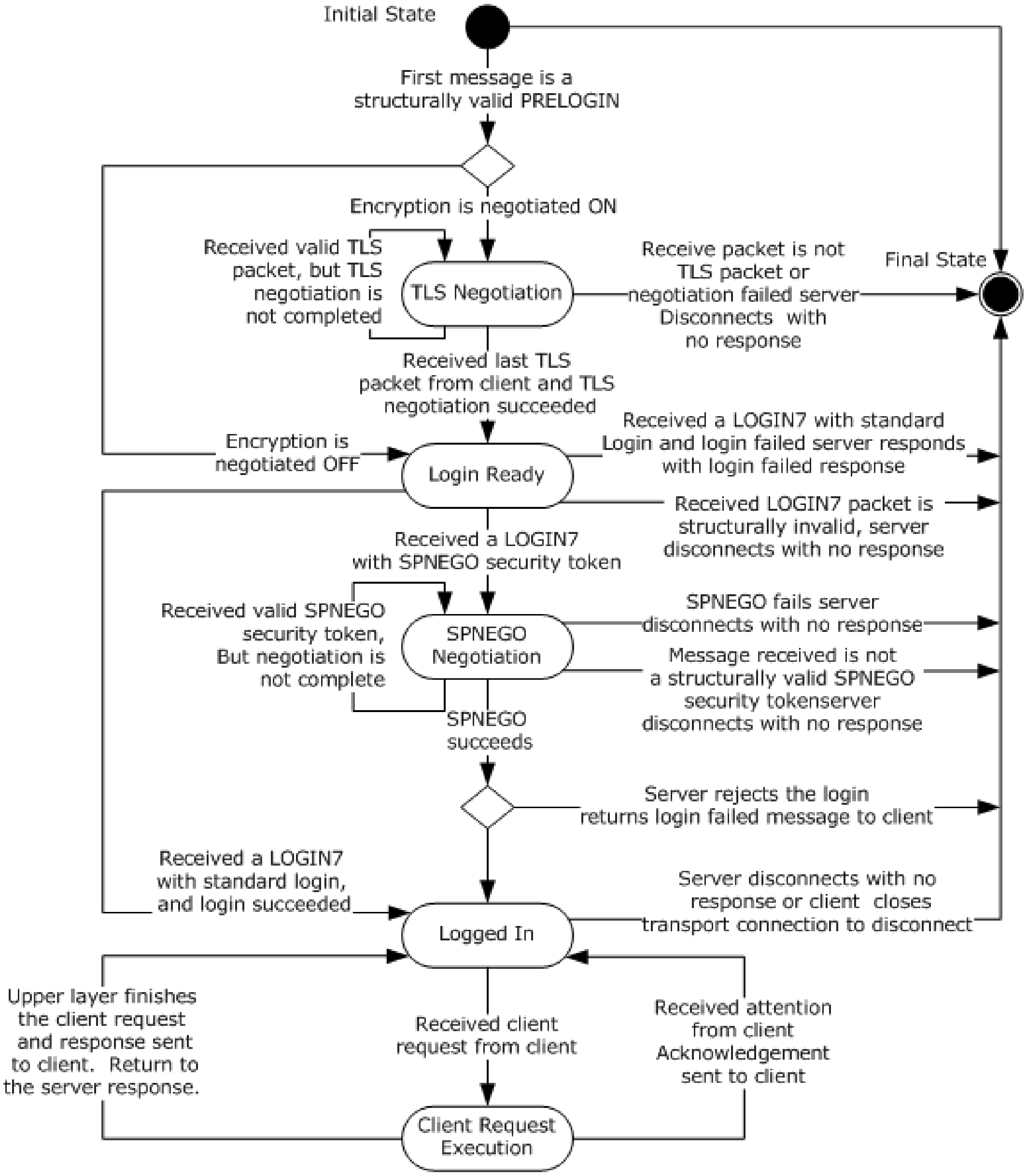
Initial State
First message is a structurally valid PRELOGIN
Encryption is negotiated ON
Received valid TLS packet, but TLS negotiation is not completed
TLS Negotiation
Receive packet is not TLS packet or negotiation failed server Disconnects with no response
Final State
Received last TLS packet from client and TLS negotiation succeeded
Encryption is negotiated OFF
Login Ready
Received a LOGIN7 with standard Login and login failed server responds with login failed response
Received LOGIN7 packet is structurally invalid, server disconnects with no response
Received a LOGIN7 with SPNEGO security token
Received valid SPNEGO security token, But negotiation is not complete
SPNEGO Negotiation
SPNEGO fails server disconnects with no response
Message received is not a structurally valid SPNEGO security tokenserver disconnects with no response
SPNEGO succeeds
Server rejects the login returns login failed message to client
Received a LOGIN7 with standard login, and login succeeded
Server disconnects with no response or client closes transport connection to disconnect
Logged In
Upper layer finishes the client request and response sent to client. Return to the server response.
Received client request from client
Received attention from client Acknowledgement sent to client
Client Request Execution

Figure 8: TDS server state machine

3.3.1 Abstract Data Model

This section describes a conceptual model of possible data organization that an implementation maintains to participate in this protocol. The organization is provided to explain how the protocol behaves. This document does not mandate that implementations adhere to this model as long as their external behavior is consistent with what is described in this document.

3.3.2 Timers

The TDS protocol does not regulate any timer on a data stream. The TDS server MAY implement a timer on any message found in section 2.

3.3.3 Initialization

The server MUST establish a listening endpoint based on one of the transport protocols described in section 2.1. The server may establish additional listening endpoints.

When a client makes a connection request, the transport layer listening endpoint will initialize all resources required for this connection. The server will be ready to receive a pre-login message.

3.3.4 Higher-Layer Triggered Events

A higher layer may choose to terminate a TDS connection at any time. In the current TDS implementation, the upper layer can kill a connection. When this happens, the server MUST terminate the connection and recycle all resources for this connection. No response will be sent to the client.

3.3.5 Message Processing Events and Sequencing Rules

The processing of messages received from a TDS client depends on the message type and the state the TDS server is in. The rest of this section describes the processing and actions to take on them. The message type is determined from the TDS packet type and the token stream inside the TDS packet payload, as described in section 2.2.

The corresponding action will be taken when the server in the following states.

3.3.5.1 Initial State

The TDS server receives the first packet from the client. The packet SHOULD be a PRELOGIN packet to set up context for login. A pre-login message is indicated by the PRELOGIN (0x12) message type described in section 2. The TDS server SHOULD close the underlying transport connection, indicate an error to the upper layer, and enter the "Final State" state, if the first packet is not a structurally correct PRELOGIN packet, the PRELOGIN packet does not contain the client version as the first option token. Otherwise, the TDS server MUST do one of the following:

- Return to the client a PRELOGIN structure wrapped in a table response (0x04) packet with Encryption and enter "TLS/SSL Negotiation" state if encryption is negotiated.
- Return to the client a PRELOGIN structure wrapped in a table response (0x04) packet without Encryption and enter unencrypted "Login Ready" state if encryption is not negotiated.

3.3.5.2 TLS/SSL Negotiation

If the next packet from the TDS client is not a TLS/SSL negotiation packet or the packet is not structurally correct, the TDS server SHOULD close the underlying transport connection, indicate an error to the upper layer, and enter the "Final State" state. A TLS/SSL negotiation packet is a PRELOGIN (0x12) packet header encapsulated with TLS/SSL payload. The TDS server MUST exchange TLS/SSL negotiation packet with the client and reenter this state until the TLS/SSL negotiation is successfully completed. In this case, TDS server enters the "Login Ready" state.

3.3.5.3 Login Ready

Depending on the type of packet received, the server MUST take one of the following actions:

- If a valid LOGIN7 packet with standard login is received, the TDS server MUST respond to the TDS client with a LOGINACK (0xAD) described in section 2 indicating login succeed. The TDS server MUST enter "Logged in" state.
- If a LOGIN7 packet with SSPI Negotiation packet is received, the TDS server MUST enter "SPNEGO Negotiation" state.
- If a LOGIN7 packet with standard login packet is received, but the login is invalid, the TDS server MUST send an ERROR packet, described in section 2, to the client. The TDS server MUST close the underlying transport connection, indicate an error to the upper layer, and enter the "Final State" state.
- If the packet received is not a structurally valid LOGIN7 packet, the TDS server will not send any response to the client. The TDS server MUST close the underlying transport connection, indicate an error to the upper layer, and enter the "Final State" state.

3.3.5.4 SPNEGO Negotiation

This state is used to negotiate the security scheme between the client and server. The TDS server processes the packet received according to the following rules.

- If the packet received is a structurally valid SPNEGO [RFC4178] negotiation packet, the TDS server delegates processing of the security token embedded in the packet to the SPNEGO layer. The SPNEGO layer responds with one of three results, and the TDS server continues processing according to the response as follows:
  - Complete: The TDS server then sends the security token to the upper layer (typically a DBMS) for authorization. If the upper layer approves the security token, the TDS server returns the security token to the client within a LoginACK message and immediately enters the "Logged In" state. If the upper layer rejects the security token, then a "Login failed" ERROR token is sent back to the client, the TDS server closes the connection, and the TDS server enters the "Final State" state.
  - Continue: The TDS server sends a SPNEGO [RFC4178] negotiation response to the client, embedding the new security token returned by SPNEGO as part of the Continue response. The server then waits for a message from the client and re-renters the SPNEGO negotiation state when such a packet is received.
  - Error: The server then MUST close the underlying transport connection, indicate an error to the upper layer, and enter the "Final State" state.
- If the packet received is not a structurally valid SPNEGO [RFC4178] negotiation packet, the TDS server will send no response the client. The TDS server MUST close the underlying transport connection, indicate an error to the upper layer, and enter the "Final State" state.

3.3.5.5 Logged In

If a TDS of type 1, 3, 7, or 14 (see section 2.2.3.1.1) arrives, then the TDS server begins processing by raising an event to the upper layer containing the data of the client request and entering the Client Request Execution state. If any other TDS types arrive, then the server MUST enter the Final State. The TDS server MUST continue to listen for messages from the client while awaiting notification of client request completion from the upper layer.

3.3.5.6 Client Request Execution

The TDS server MUST continue to listen for messages from the client while awaiting notification of client request for completion from the upper layer. The TDS server MUST also do one of the following:

- If the upper layer notifies TDS that the client request has finished successfully, the TDS server MUST send the results in the formats described in section 2 to the TDS client and enter "Logged In" state.
- If the upper layer notifies TDS that an error has been encountered during client request, the TDS server MUST send an ERROR message (described in section 2) to the TDS client and enter "Logged In" state.
- If an attention packet (described in section 2) is received during the execution of the current client request, it MUST deliver a cancel indication to the upper layer. If an attention packet (described in section 2) is received after the execution of the current client request, it SHOULD NOT deliver a cancel indication to the upper layer because there is no existing execution to cancel. The TDS server MUST send an attention acknowledgment to the TDS client and enter the "Logged In" state.
- If another client request packet is received during the execution of the current client request, the TDS server SHOULD queue the new client request, and continue processing the client request already in progress according to the preceding rules. When this operation is complete, the TDS server re-enters the "client request execution" state and processes the newly arrived message.
- If MARS is enabled, all TDS server responses to client request messages MUST be passed through to the SMUX layer.
- If any other message type arrives, the server MUST close the connection and enter the "Final State" state.

3.3.5.7 Final State

The connection is disconnected. All resources for this connection will be recycled by the TDS server.

3.3.6 Timer Events

None.

3.3.7 Other Local Events

When there is a failure in under-layers, the server SHOULD terminate the TDS session without sending any response to the client. The under-layer failure could be triggered by network failure. It can also be triggered by the termination action from the client, which could be communicated to the server stack by under-layers.

4 Protocol Examples

The following sections describe several operations as used in common scenarios to illustrate the function of the TDS protocol. For each example, the binary TDS message is provided followed by the decomposition displayed in XML.

4.1 Pre-Login Request

Pre-Login request sent from the client to the server:

```
12 01 00 2F 00 00 01 00 00 00 1A 00 06 01 00 20
00 01 02 00 21 00 01 03 00 22 00 04 04 00 26 00
01 FF 09 00 00 00 00 00 01 00 B8 0D 00 00 01
```

```
<PacketHeader>
    <Type>
      <BYTE>12 </BYTE>
    </Type>
    <Status>
      <BYTE>01 </BYTE>
    </Status>
    <Length>
      <BYTE>00 </BYTE>
      <BYTE>2F </BYTE>
    </Length>
    <SPID>
      <BYTE>00 </BYTE>
      <BYTE>00 </BYTE>
    </SPID>
    <Packet>
      <BYTE>01 </BYTE>
    </Packet>
    <Window>
      <BYTE>00 </BYTE>
    </Window>
  </PacketHeader>
  <PacketData>
    <Prelogin>
      <TokenType>
         <BYTE>00 </BYTE>
      </TokenType>
      <TokenPosition>
         <USHORT>00 1A</USHORT>
      </TokenPosition>
      <TokenLeng>
         <USHORT>00 06</USHORT>
      </TokenLeng>
      <TokenType>
         <BYTE>01 </BYTE>
      </TokenType>
      <TokenPosition>
         <USHORT>00 20</USHORT>
      </TokenPosition>
      <TokenLeng>
         <USHORT>00 01</USHORT>
      </TokenLeng>
      <TokenType>
```

```
         <BYTE>02 </BYTE>
      </TokenType>
      <TokenPosition>
         <USHORT>00 21</USHORT>
      </TokenPosition>
      <TokenLeng>
         <USHORT>00 01</USHORT>
      </TokenLeng>
      <TokenType>
         <BYTE>03 </BYTE>
      </TokenType>
      <TokenPosition>
         <USHORT>00 22</USHORT>
      </TokenPosition>
      <TokenLeng>
         <USHORT>00 04</USHORT>
      </TokenLeng>
      <TokenType>
         <BYTE>04 </BYTE>
      </TokenType>
      <TokenPosition>
         <USHORT>00 26</USHORT>
      </TokenPosition>
      <TokenLeng>
         <USHORT>00 01</USHORT>
      </TokenLeng>
      <TokenType>
         <BYTE>FF </BYTE>
      </TokenType>
      <PreloginData>
         <BYTES>09 00 00 00 00 00 01 00 B8 0D 00 00 01</BYTE>
      </PreloginData>
    </Prelogin>
  </PacketData>
```

4.2 Login Request

Login request sent from the client to the server:

```
10 01 00 90 00 00 01 00 88 00 00 00 02 00 09 72
00 10 00 00 00 00 00 07 00 01 00 00 00 00 00 00
E0 03 00 00 E0 01 00 00 09 04 00 00 5E 00 08 00
6E 00 02 00 72 00 00 00 72 00 07 00 80 00 00 00
80 00 00 00 80 00 04 00 88 00 00 00 88 00 00 00
00 50 8B E2 B7 8F 88 00 00 00 88 00 00 00 88 00
00 00 00 00 00 00 73 00 6B 00 6F 00 73 00 74 00
6F 00 76 00 31 00 73 00 61 00 4F 00 53 00 51 00
4C 00 2D 00 33 00 32 00 4F 00 44 00 42 00 43 00
```

```
  <PacketHeader>
    <Type>
      <BYTE>10 </BYTE>
    </Type>
    <Status>
      <BYTE>01 </BYTE>
    </Status>
```

```
      <Length>
        <BYTE>00 </BYTE>
        <BYTE>90 </BYTE>
      </Length>
      <SPID>
        <BYTE>00 </BYTE>
        <BYTE>00 </BYTE>
      </SPID>
      <Packet>
        <BYTE>01 </BYTE>
      </Packet>
      <Window>
        <BYTE>00 </BYTE>
      </Window>
    </PacketHeader>
    <PacketData>
      <Login7>
        <Length>
          <DWORD>88 00 00 00 </DWORD>
        </Length>
        <TDSVersion>
          <DWORD>02 00 09 72 </DWORD>
        </TDSVersion>
        <PacketSize>
          <DWORD>00 10 00 00 </DWORD>
        </PacketSize>
        <ClientProgVer>
          <DWORD>00 00 00 07 </DWORD>
        </ClientProgVer>
        <ClientPID>
          <DWORD>00 01 00 00 </DWORD>
        </ClientPID>
        <ConnectionID>
          <DWORD>00 00 00 00 </DWORD>
        </ConnectionID>
        <OptionFlags1>
          <BYTE>E0 </BYTE>
        </OptionFlags1>
        <OptionFlags2>
          <BYTE>03 </BYTE>
        </OptionFlags2>
        <TypeFlags>
          <BYTE>00 </BYTE>
        </TypeFlags>
        <OptionFlags3>
          <BYTE>00 </BYTE>
        </OptionFlags3>
        <ClientTimZone>
          <LONG>E0 01 00 00 </LONG>
        </ClientTimZone>
        <ClientLCID>
          <DWORD>09 04 00 00 </DWORD>
        </ClientLCID>
        <OffsetLength>
          <ibHostName>
            <USHORT>5E 00 </USHORT>
          </ibHostName>
          <cchHostName>
            <USHORT>08 00 </USHORT>
          </cchHostName>
          <ibUserName>
            <USHORT>6E 00 </USHORT>
          </ibUserName>
          <cchUserName>
            <USHORT>02 00 </USHORT>
          </cchUserName>
          <ibPassword>
            <USHORT>72 00 </USHORT>
          </ibPassword>
          <cchPassword>
            <USHORT>00 00 </USHORT>
          </cchPassword>
          <ibAppName>
            <USHORT>72 00 </USHORT>
          </ibAppName>
          <cchAppName>
            <USHORT>07 00 </USHORT>
          </cchAppName>
          <ibServerName>
            <USHORT>80 00 </USHORT>
          </ibServerName>
          <cchServerName>
            <USHORT>00 00 </USHORT>
          </cchServerName>
          <ibUnused>
            <USHORT>80 00 </USHORT>
          </ibUnused>
          <cbUnused>
            <USHORT>00 00 </USHORT>
          </cbUnused>
          <ibCltIntName>
            <USHORT>80 00 </USHORT>
          </ibCltIntName>
          <cchCltIntName>
            <USHORT>04 00 </USHORT>
          </cchCltIntName>
          <ibLanguage>
            <USHORT>88 00 </USHORT>
          </ibLanguage>
          <cchLanguage>
            <USHORT>00 00 </USHORT>
          </cchLanguage>
          <ibDatabase>
            <USHORT>88 00 </USHORT>
          </ibDatabase>
          <cchDatabase>
            <USHORT>00 00 </USHORT>
          </cchDatabase>
          <ClientID>
            <BYTES>00 50 8B E2 B7 8F </BYTES>
          </ClientID>
          <ibSSPI>
            <USHORT>88 00 </USHORT>
          </ibSSPI>
          <cbSSPI>
            <USHORT>00 00 </USHORT>
          </cbSSPI>
          <ibAtchDBFile>
```

```
     <USHORT>88 00 </USHORT>
    </ibAtchDBFile>
    <cchAtchDBFile>
     <USHORT>00 00 </USHORT>
    </cchAtchDBFile>
    <ibChangePassword>
     <USHORT>88 00 </USHORT>
    </ibChangePassword>
    <cchChangePassword>
     <USHORT>00 00 </USHORT>
    </cchChangePassword>
    <cbSSPILong>
     <LONG>00 00 00 00 </LONG>
    </cbSSPILong>
   </OffsetLength>
   <Data>
    <BYTES>73 00 6B 00 6F 00 73 00 74 00 6F 00 76 00 31 00 73 00 61 00
4F 00 53 00 51 00 4C 00 2D 00 33 00 32 00 4F 00 44 00 42 00 43 00 </BYTES>
   </Data>
  </Login7>
 </PacketData>
```

4.3 Login Response

Login response from the server to the client:

```
04 01 01 61 00 00 01 00 E3 1B 00 01 06 6D 00 61
00 73 00 74 00 65 00 72 00 06 6D 00 61 00 73 00
74 00 65 00 72 00 AB 58 00 45 16 00 00 02 00 25
00 43 00 68 00 61 00 6E 00 67 00 65 00 64 00 20
00 64 00 61 00 74 00 61 00 62 00 61 00 73 00 65
00 20 00 63 00 6F 00 6E 00 74 00 65 00 78 00 74
00 20 00 74 00 6F 00 20 00 27 00 6D 00 61 00 73
00 74 00 65 00 72 00 27 00 2E 00 00 00 00 00 00
00 E3 08 00 07 05 09 04 D0 00 34 00 E3 17 00 02
0A 75 00 73 00 5F 00 65 00 6E 00 67 00 6C 00 69
00 73 00 68 00 00 E3 13 00 04 04 34 00 30 00 39
00 36 00 04 34 00 30 00 39 00 36 00 AB 5C 00 47
16 00 00 01 00 27 00 43 00 68 00 61 00 6E 00 67
00 65 00 64 00 20 00 6C 00 61 00 6E 00 67 00 75
00 61 00 67 00 65 00 20 00 73 00 65 00 74 00 74
00 69 00 6E 00 67 00 20 00 74 00 6F 00 20 00 75
00 73 00 5F 00 65 00 6E 00 67 00 6C 00 69 00 73
00 68 00 2E 00 00 00 00 00 00 00 AD 36 00 01 72
09 00 02 16 4D 00 69 00 63 00 72 00 6F 00 73 00
6F 00 66 00 74 00 20 00 53 00 51 00 4C 00 20 00
53 00 65 00 72 00 76 00 65 00 72 00 00 00 00 00
00 00 00 00 FD 00 00 00 00 00 00 00 00 00 00 00
00

 <PacketHeader>
  <Type>
   <BYTE>04 </BYTE>
  </Type>
  <Status>
   <BYTE>01 </BYTE>
  </Status>
```

```
    <Length>
      <BYTE>01 </BYTE>
      <BYTE>61 </BYTE>
    </Length>
    <SPID>
      <BYTE>00 </BYTE>
      <BYTE>00 </BYTE>
    </SPID>
    <Packet>
      <BYTE>01 </BYTE>
    </Packet>
    <Window>
      <BYTE>00 </BYTE>
    </Window>
  </PacketHeader>
  <PacketData>
    <TableResponse>
      <ENVCHANGE>
        <TokenType>
          <BYTE>E3 </BYTE>
        </TokenType>
        <Length>
          <USHORT>1B 00 </USHORT>
        </Length>
        <EnvChangeData>
          <BYTES>01 06 6D 00 61 00 73 00 74 00 65 00 72 00 06 6D 00 61 00
73 00 74 00 65 00 72 00 </BYTES>
        </EnvChangeData>
      </ENVCHANGE>
      <INFO>
        <TokenType>
          <BYTE>AB </BYTE>
        </TokenType>
        <Length>
          <USHORT>58 00 </USHORT>
        </Length>
        <Number>
          <LONG>45 16 00 00 </LONG>
        </Number>
        <State>
          <BYTE>02 </BYTE>
        </State>
        <Class>
          <BYTE>00 </BYTE>
        </Class>
        <MsgText>
          <US_UNICODE>
            <USHORTLEN>
              <USHORT>25 00 </USHORT>
            </USHORTLEN>
            <BYTES ascii="C.h.a.n.g.e.d. .d.a.t.a.b.a.s.e. .c.o.n.t.e.x.t.
.t.o. .'.m.a.s.t.e.r.'...">43 00 68 00 61 00 6E 00 67 00 65 00 64 00 20 00
64 00 61 00 74 00 61 00 62 00 61 00 73 00 65 00 20 00 63 00 6F 00 6E 00 74
00 65 00 78 00 74 00 20 00 74 00 6F 00 20 00 27 00 6D 00 61 00 73 00 74 00
65 00 72 00 27 00 2E 00 </BYTES>
          </US_UNICODE>
        </MsgText>
        <ServerName>
          <B_UNICODE>
```

```
            <BYTELEN>
              <BYTE>00 </BYTE>
            </BYTELEN>
            <BYTES ascii="">
            </BYTES>
          </B_UNICODE>
        </ServerName>
        <ProcName>
          <B_UNICODE>
            <BYTELEN>
              <BYTE>00 </BYTE>
            </BYTELEN>
            <BYTES ascii="">
            </BYTES>
          </B_UNICODE>
        </ProcName>
        <LineNumber>
          <LONG>00 00 00 00 </LONG>
        </LineNumber>
      </INFO>
      <ENVCHANGE>
        <TokenType>
          <BYTE>E3 </BYTE>
        </TokenType>
        <Length>
          <USHORT>08 00 </USHORT>
        </Length>
        <EnvChangeData>
          <BYTES>07 05 09 04 D0 00 34 00 </BYTES>
        </EnvChangeData>
      </ENVCHANGE>
      <ENVCHANGE>
        <TokenType>
          <BYTE>E3 </BYTE>
        </TokenType>
        <Length>
          <USHORT>17 00 </USHORT>
        </Length>
        <EnvChangeData>
          <BYTES>02 0A 75 00 73 00 5F 00 65 00 6E 00 67 00 6C 00 69 00 73
00 68 00 00 </BYTES>
        </EnvChangeData>
      </ENVCHANGE>
      <INFO>
        <TokenType>
          <BYTE>AB </BYTE>
        </TokenType>
        <Length>
          <USHORT>5C 00 </USHORT>
        </Length>
        <Number>
          <LONG>47 16 00 00 </LONG>
        </Number>
        <State>
          <BYTE>01 </BYTE>
        </State>
        <Class>
          <BYTE>00 </BYTE>
        </Class>
        <MsgText>
          <US_UNICODE>
            <USHORTLEN>
              <USHORT>27 00 </USHORT>
            </USHORTLEN>
            <BYTES ascii="C.h.a.n.g.e.d. .l.a.n.g.u.a.g.e. .s.e.t.t.i.n.g.
.t.o. .u.s._.e.n.g.l.i.s.h...">43 00 68 00 61 00 6E 00 67 00 65 00 64 00 20
00 6C 00 61 00 6E 00 67 00 75 00 61 00 67 00 65 00 20 00 73 00 65 00 74 00
74 00 69 00 6E 00 67 00 20 00 74 00 6F 00 20 00 75 00 73 00 5F 00 65 00 6E
00 67 00 6C 00 69 00 73 00 68 00 2E 00 </BYTES>
          </US_UNICODE>
        </MsgText>
        <ServerName>
          <B_UNICODE>
            <BYTELEN>
              <BYTE>00 </BYTE>
            </BYTELEN>
            <BYTES ascii="">
            </BYTES>
          </B_UNICODE>
        </ServerName>
        <ProcName>
          <B_UNICODE>
            <BYTELEN>
              <BYTE>00 </BYTE>
            </BYTELEN>
            <BYTES ascii="">
            </BYTES>
          </B_UNICODE>
        </ProcName>
        <LineNumber>
          <LONG>00 00 00 00 </LONG>
        </LineNumber>
      </INFO>
      <LOGINACK>
        <TokenType>
          <BYTE>AD </BYTE>
        </TokenType>
        <Length>
          <USHORT>36 00 </USHORT>
        </Length>
        <Interface>
          <BYTE>01 </BYTE>
        </Interface>
        <TDSVersion>
          <DWORD>72 09 00 02 </DWORD>
        </TDSVersion>
        <ProgName>
          <B_UNICODE>
            <BYTELEN>
              <BYTE>16 </BYTE>
            </BYTELEN>
            <BYTES ascii="M.i.c.r.o.s.o.f.t. .S.Q.L. .S.e.r.v.e.r.....">4D
00 69 00 63 00 72 00 6F 00 73 00 6F 00 66 00 74 00 20 00 53 00 51 00 4C 00
20 00 53 00 65 00 72 00 76 00 65 00 72 00 00 00 00 00 </BYTES>
          </B_UNICODE>
        </ProgName>
        <ProgVersion>
          <DWORD>00 00 00 00 </DWORD>
        </ProgVersion>
      </LOGINACK>
      <ENVCHANGE>
        <TokenType>
          <BYTE>E3 </BYTE>
        </TokenType>
        <Length>
          <USHORT>13 00 </USHORT>
        </Length>
        <EnvChangeData>
          <BYTES>04 04 34 00 30 00 39 00 36 00 04 34 00 30 00 39 00 36 00
</BYTES>
        </EnvChangeData>
      </ENVCHANGE>
      <DONE>
        <TokenType>
          <BYTE>FD </BYTE>
        </TokenType>
        <Status>
          <USHORT>00 00 </USHORT>
        </Status>
        <CurCmd>
          <USHORT>00 00 </USHORT>
        </CurCmd>
        <DoneRowCount>
          <LONGLONG>00 00 00 00 00 00 00 00 </LONGLONG>
        </DoneRowCount>
      </DONE>
    </TableResponse>
  </PacketData>
```

4.4 SQL Batch Client Request

Client request sent from the client to the server:

```
01 01 00 5C 00 00 01 00 16 00 00 00 12 00 00 00
02 00 00 00 00 00 00 00 00 01 00 00 00 00 0A 00
73 00 65 00 6C 00 65 00 63 00 74 00 20 00 27 00
66 00 6F 00 6F 00 27 00 20 00 61 00 73 00 20 00
27 00 62 00 61 00 72 00 27 00 0A 00 20 00 20 00
20 00 20 00 20 00 20 00 20 00 20 00

  <PacketHeader>
    <Type>
      <BYTE>01 </BYTE>
    </Type>
    <Status>
      <BYTE>01 </BYTE>
    </Status>
    <Length>
      <BYTE>00 </BYTE>
      <BYTE>5C </BYTE>
    </Length>
    <SPID>
      <BYTE>00 </BYTE>
      <BYTE>00 </BYTE>
    </SPID>
    <Packet>
      <BYTE>01 </BYTE>
    </Packet>
    <Window>
      <BYTE>00 </BYTE>
    </Window>
  </PacketHeader>
  <PacketData>
    <SQLBatch>
      <All_HEADERS>
        <TotalLength>
          <DWORD>16 00 00 00 </DWORD>
        </TotalLength>
        <Header>
          <HeaderLength>
            <DWORD>12 00 00 00 </DWORD>
          </HeaderLength>
          <HeaderType>
            <USHORT>02 00 </USHORT>
          </HeaderType>
          <HeaderData>
            <MARS>
              <TransactionDescriptor>
                <ULONGLONG>00 00 00 00 00 00 00 01 </ULONGLONG>
              </TransactionDescriptor>
              <OutstandingRequestCount>
                <DWORD>00 00 00 00 </DWORD>
              </OutstandingRequestCount>
            </MARS>
          </HeaderData>
        </Header>
      </All_HEADERS>
      <SQLText>
        <UNICODESTREAM>
          <BYTES>0A 00 73 00 65 00 6C 00 65 00 63 00 74 00 20 00 27 00 66
00 6F 00 6F 00 27 00 20 00 61 00 73 00 20 00 27 00 62 00 61 00 72 00 27 00
0A 00 20 00 20 00 20 00 20 00 20 00 20 00 20 00 20 00 </BYTES>
        </UNICODESTREAM>
      </SQLText>
    </SQLBatch>
  </PacketData>
```

4.5 SQL Batch Server Response

Server response sent from the server to the client:

```
04 01 00 33 00 00 01 00 81 01 00 00 00 00 00 20
00 A7 03 00 09 04 D0 00 34 03 62 00 61 00 72 00
D1 03 00 66 6F 6F FD 10 00 C1 00 01 00 00 00 00
00 00 00

  <PacketHeader>
    <Type>
      <BYTE>04 </BYTE>
    </Type>
    <Status>
      <BYTE>01 </BYTE>
    </Status>
    <Length>
      <BYTE>00 </BYTE>
      <BYTE>33 </BYTE>
    </Length>
    <SPID>
      <BYTE>00 </BYTE>
      <BYTE>00 </BYTE>
    </SPID>
    <Packet>
      <BYTE>01 </BYTE>
    </Packet>
    <Window>
      <BYTE>00 </BYTE>
    </Window>
  </PacketHeader>
  <PacketData>
    <TableResponse>
      <COLMETADATA>
        <TokenType>
          <BYTE>81 </BYTE>
        </TokenType>
        <Count>
          <USHORT>01 00 </USHORT>
        </Count>
        <ColumnData>
          <UserType>
            <ULONG>00 00 00 00 </ULONG>
          </UserType>
          <Flags>
            <USHORT>20 00 </USHORT>
          </Flags>
          <TYPE_INFO>
            <VARLENTYPE>
              <USHORTLEN_TYPE>
                <BYTE>A7 </BYTE>
              </USHORTLEN_TYPE>
            </VARLENTYPE>
            <TYPE_VARLEN>
              <USHORTCHARBINLEN>
                <USHORT>03 00 </USHORT>
              </USHORTCHARBINLEN>
            </TYPE_VARLEN>
            <COLLATION>
              <BYTES>09 04 D0 00 34 </BYTES>
            </COLLATION>
          </TYPE_INFO>
          <ColName>
            <B_UNICODE>
              <BYTELEN>
                <BYTE>03 </BYTE>
              </BYTELEN>
              <BYTES ascii="b.a.r.">62 00 61 00 72 00 </BYTES>
            </B_UNICODE>
          </ColName>
        </ColumnData>
      </COLMETADATA>
      <ROW>
        <TokenType>
          <BYTE>D1 </BYTE>
        </TokenType>
        <TYPE_VARBYTE>
          <TYPE_VARLEN>
            <USHORTCHARBINLEN>
              <USHORT>03 00 </USHORT>
            </USHORTCHARBINLEN>
          </TYPE_VARLEN>
          <BYTES ascii="foo">66 6F 6F </BYTES>
        </TYPE_VARBYTE>
      </ROW>
      <DONE>
        <TokenType>
          <BYTE>FD </BYTE>
        </TokenType>
        <Status>
          <USHORT>10 00 </USHORT>
        </Status>
        <CurCmd>
          <USHORT>C1 00 </USHORT>
        </CurCmd>
        <DoneRowCount>
          <LONGLONG>01 00 00 00 00 00 00 00 </LONGLONG>
        </DoneRowCount>
      </DONE>
    </TableResponse>
  </PacketData>
```

4.6 RPC Client Request

RPC request sent from the client to the server:

```
03 01 00 2F 00 00 01 00 16 00 00 00 12 00 00 00
02 00 00 00 00 00 00 00 00 01 00 00 00 00 04 00
66 00 6F 00 6F 00 33 00 00 00 00 02 26 02 00

  <PacketHeader>
    <Type>
      <BYTE>03 </BYTE>
    </Type>
    <Status>
      <BYTE>01 </BYTE>
    </Status>
    <Length>
      <BYTE>00 </BYTE>
      <BYTE>2F </BYTE>
    </Length>
    <SPID>
      <BYTE>00 </BYTE>
      <BYTE>00 </BYTE>
    </SPID>
    <Packet>
      <BYTE>01 </BYTE>
    </Packet>
    <Window>
      <BYTE>00 </BYTE>
    </Window>
    </PacketHeader>
    <PacketData>
      <RPCRequest>
        <All_HEADERS>
          <TotalLength>
            <DWORD>16 00 00 00 </DWORD>
          </TotalLength>
          <Header>
            <HeaderLength>
              <DWORD>12 00 00 00 </DWORD>
            </HeaderLength>
            <HeaderType>
              <USHORT>02 00 </USHORT>
            </HeaderType>
            <HeaderData>
              <MARS>
                <TransactionDescriptor>
                  <ULONGLONG>00 00 00 00 00 00 00 01 </ULONGLONG>
                </TransactionDescriptor>
                <OutstandingRequestCount>
                  <DWORD>00 00 00 00 </DWORD>
                </OutstandingRequestCount>
              </MARS>
            </HeaderData>
          </Header>
        </All_HEADERS>
        <RPCReqBatch>
          <NameLenProcID>
            <ProcName>
              <US_UNICODE>
                <USHORTLEN>
                  <USHORT>04 00 </USHORT>
                </USHORTLEN>
                <BYTES ascii="f.o.o.3.">66 00 6F 00 6F 00 33 00 </BYTES>
              </US_UNICODE>
            </ProcName>
          </NameLenProcID>
          <OptionFlags>
            <fWithRecomp>
              <BIT>0</BIT>
            </fWithRecomp>
            <fNoMetaData>
              <BIT>0</BIT>
            </fNoMetaData>
          </OptionFlags>
          <ParameterData>
            <ParamMetaData>
              <B_UNICODE>
                <BYTELEN>
                  <BYTE>00 </BYTE>
                </BYTELEN>
                <BYTES ascii="">
                </BYTES>
              </B_UNICODE>
              <StatusFlags>
                <fByRefValue>
                  <BIT>0</BIT>
                </fByRefValue>
                <fDefaultValue>
                <BIT>1</BIT>
              </fDefaultValue>
            </StatusFlags>
            <TYPE_INFO>
              <VARLENTYPE>
                <BYTELEN_TYPE>
                  <BYTE>26 </BYTE>
                </BYTELEN_TYPE>
              </VARLENTYPE>
              <TYPE_VARLEN>
                <BYTELEN>
                  <BYTE>02 </BYTE>
                </BYTELEN>
              </TYPE_VARLEN>
            </TYPE_INFO>
          </ParamMetaData>
          <ParamLenData>
            <TYPE_VARBYTE>
              <TYPE_VARLEN>
                <BYTELEN>
                  <BYTE>00 </BYTE>
                </BYTELEN>
              </TYPE_VARLEN>
              <BYTES>
              </BYTES>
            </TYPE_VARBYTE>
          </ParamLenData>
        </ParameterData>
      </RPCReqBatch>
    </RPCRequest>
  </PacketData>
```

4.7 RPC Server Response

RPC response sent from the server to the client:

```
04 01 00 27 00 00 01 00 FF 11 00 C1 00 01 00 00
00 00 00 00 00 79 00 00 00 00 FE 00 00 E0 00 00
00 00 00 00 00 00 00

  <PacketHeader>
    <Type>
      <BYTE>04 </BYTE>
    </Type>
    <Status>
      <BYTE>01 </BYTE>
    </Status>
    <Length>
      <BYTE>00 </BYTE>
      <BYTE>27 </BYTE>
    </Length>
    <SPID>
      <BYTE>00 </BYTE>
      <BYTE>00 </BYTE>
    </SPID>
    <Packet>
      <BYTE>01 </BYTE>
    </Packet>
    <Window>
      <BYTE>00 </BYTE>
    </Window>
  </PacketHeader>
  <PacketData>
    <TableResponse>
      <DONEINPROC>
        <TokenType>
          <BYTE>FF </BYTE>
        </TokenType>
        <Status>
          <USHORT>11 00 </USHORT>
        </Status>
        <CurCmd>
          <USHORT>C1 00 </USHORT>
        </CurCmd>
        <DoneRowCount>
          <LONGLONG>01 00 00 00 00 00 00 00 </LONGLONG>
        </DoneRowCount>
      </DONEINPROC>
      <RETURNSTATUS>
        <TokenType>
          <BYTE>79 </BYTE>
        </TokenType>
        <VALUE>
          <LONG>00 00 00 00 </LONG>
        </VALUE>
      </RETURNSTATUS>
      <DONEPROC>
        <TokenType>
          <BYTE>FE </BYTE>
        </TokenType>
        <Status>
          <USHORT>00 00 </USHORT>
        </Status>
        <CurCmd>
          <USHORT>E0 00 </USHORT>
        </CurCmd>
        <DoneRowCount>
          <LONGLONG>00 00 00 00 00 00 00 00 </LONGLONG>
        </DoneRowCount>
      </DONEPROC>
    </TableResponse>
  </PacketData>
```

4.8 Attention Request

Attention request sent from client to server:

```
06 01 00 08 00 00 01 00

<PacketHeader>
  <Type>
    <BYTE>06</BYTE>
  </Type>
  <Status>
```

```
    <BYTE>01</BYTE>
  </Status>
  <Length>
    <BYTE>00</BYTE>
    <BYTE>08</BYTE>
  </Length>
  <SPID>
    <BYTE>00</BYTE>
    <BYTE>00</BYTE>
  </SPID>
  <Packet>
    <BYTE>01</BYTE>
  </Packet>
  <Window>
    <BYTE>00</BYTE>
  </Window>
</PacketHeader>
```

4.9 SSPI Message

SSPI message carrying SSPI payload sent from client to server:

```
11 01 00 60 00 00 01 00 4E 54 4C 4D 53 53 50 00
03 00 00 00 00 00 00 00 58 00 00 00 00 00 00 00
58 00 00 00 00 00 00 00 58 00 00 00 00 00 00 00
58 00 00 00 00 00 00 00 58 00 00 00 00 00 00 00
58 00 00 00 15 C2 88 E2 06 00 71 17 00 00 00 0F
30 81 C1 7D 59 5F E9 3E 1A 7C 98 05 01 72 5C 4F
```

```
<PacketHeader>
  <Type>
    <BYTE>11 </BYTE>
  </Type>
  <Status>
    <BYTE>01 </BYTE>
  </Status>
  <Length>
    <BYTE>00 </BYTE>
    <BYTE>60 </BYTE>
  </Length>
  <SPID>
    <BYTE>00 </BYTE>
    <BYTE>00 </BYTE>
  </SPID>
  <Packet>
    <BYTE>01 </BYTE>
  </Packet>
  <Window>
    <BYTE>00 </BYTE>
  </Window>
</PacketHeader>
<PacketData>
  <SSPI>
    <BYTES>4E 54 4C 4D 53 53 50 00 03 00 00 00 00 00 00 00 58 00 00 00 00
00 00 00 58 00 00 00 00 00 00 00 58 00 00 00 00 00 00 00 58 00 00 00 00
```

```
00 00 58 00 00 00 00 00 00 00 58 00 00 00 15 C2 88 E2 06 00 71 17 00 00
0F 30 81 C1 7D 59 5F E9 3E 1A 7C 98 05 01 72 5C 4F </BYTES>
  </SSPI>
</PacketData>
```

4.10 SQL Command with Binary Data

BULKLOADBCP request sent from client to server:

```
07 01 00 26 00 00 01 00 81 01 00 00 00 00 05
00 32 02 63 00 31 00 D1 00 FD 00 00 00 00 00 00
00 00 00 00 00 00
```

```
<PacketHeader>
  <Type>
    <BYTE>07 </BYTE>
  </Type>
  <Status>
    <BYTE>01 </BYTE>
  </Status>
  <Length>
    <BYTE>00 </BYTE>
    <BYTE>26 </BYTE>
  </Length>
  <SPID>
    <BYTE>00 </BYTE>
    <BYTE>00 </BYTE>
  </SPID>
  <Packet>
    <BYTE>01 </BYTE>
  </Packet>
  <Window>
    <BYTE>00 </BYTE>
  </Window>
</PacketHeader>
<PacketData>
  <BulkLoadBCP>
    <COLMETADATA>
      <TokenType>
        <BYTE>81 </BYTE>
      </TokenType>
      <Count>
        <USHORT>01 00 </USHORT>
      </Count>
      <ColumnData>
        <UserType>
          <ULONG>00 00 00 00 </ULONG>
        </UserType>
        <Flags>
          <USHORT>05 00 </USHORT>
        </Flags>
        <TYPE_INFO>
          <FIXEDLENTYPE>
            <BYTE>32 </BYTE>
          </FIXEDLENTYPE>
        </TYPE_INFO>
        <ColName>
          <B_UNICODE>
            <BYTELEN>
              <BYTE>02 </BYTE>
            </BYTELEN>
            <BYTES ascii="c.1.">63 00 31 00 </BYTES>
          </B_UNICODE>
        </ColName>
      </ColumnData>
    </COLMETADATA>
    <ROW>
      <TokenType>
        <BYTE>D1 </BYTE>
      </TokenType>
      <TYPE_VARBYTE>
        <BYTES>00 </BYTES>
      </TYPE_VARBYTE>
    </ROW>
    <DONE>
      <TokenType>
        <BYTE>FD </BYTE>
      </TokenType>
      <Status>
        <USHORT>00 00 </USHORT>
      </Status>
      <CurCmd>
        <USHORT>00 00 </USHORT>
      </CurCmd>
      <DoneRowCount>
        <LONGLONG>00 00 00 00 00 00 00 00 </LONGLONG>
      </DoneRowCount>
    </DONE>
  </BulkLoadBCP>
</PacketData>
```

4.11 Transaction Manager Request

Transaction Manager Request sent from client to server:

```
0E 01 00 20 00 00 01 00 16 00 00 00
12 00 00 00 02 00 00 00 00 00 00 00
00 00 00 00 00 01 06 00
```

```
<PacketHeader>
  <Type>
    <BYTE>0E </BYTE>
  </Type>
  <Status>
    <BYTE>01 </BYTE>
  </Status>
  <Length>
    <BYTE>00 </BYTE>
    <BYTE>20 </BYTE>
  </Length>
  <SPID>
    <BYTE>00 </BYTE>
    <BYTE>00 </BYTE>
  </SPID>
  <Packet>
    <BYTE>01 </BYTE>
  </Packet>
  <Window>
    <BYTE>00 </BYTE>
  </Window>
</PacketHeader>
<PacketData>
  <TransMgrReq>
    <All_HEADERS>
      <TotalLength>
        <DWORD>16 00 00 00 </DWORD>
      </TotalLength>
      <Header>
        <HeaderLength>
          <DWORD>12 00 00 00 </DWORD>
        </HeaderLength>
        <HeaderType>
          <USHORT>02 00 </USHORT>
        </HeaderType>
        <HeaderData>
          <MARS>
            <TransactionDescriptor>
              <ULONGLONG>00 00 00 00 00 00 00 01 </ULONGLONG>
            </TransactionDescriptor>
            <OutstandingRequestCount>
              <DWORD>00 00 00 00 </DWORD>
            </OutstandingRequestCount>
          </MARS>
        </HeaderData>
      </Header>
    </All_HEADERS>
    <RequestType>
      <USHORT>16 00 </USHORT>
    </RequestType>
    <RequestPayload>
      <TM_PROMOTE_XACT>
      </TM_PROMOTE_XACT>
    </RequestPayload>
  </TransMgrReq>
</PacketData>
```

4.12 TVP Insert Statement

TVP insert statement sent from client to server:

```
03 01 00 52 00 00 01 00 16 00 00 00
12 00 00 00 02 00 00 00 00 00 00 00
00 00 00 00 00 01 03 00 66 00 6F 00
```

```
6F 00 00 00 00 00 F3 00 03 64 00 62
00 6F 00 07 74 00 76 00 70 00 74 00
79 00 70 00 65 00 01 00 00 00 00 00
00 00 26 01 00 00 01 01 02 00
```

```
<tds version="katmai">
  <BufferData>
    <RPCRequest>
      <RPCReqBatch>
        <NameLenProcID>
          <ProcName>
            <US_UNICODE>
              <USHORTLEN>
                <USHORT>03 00 </USHORT>
              </USHORTLEN>
              <BYTES ascii="f.o.o.">66 00 6F 00 6F 00 </BYTES>
            </US_UNICODE>
          </ProcName>
        </NameLenProcID>
        <OptionFlags>
          <fWithRecomp>
            <BIT>false</BIT>
          </fWithRecomp>
          <fNoMetaData>
            <BIT>false</BIT>
          </fNoMetaData>
        </OptionFlags>
        <ParameterData>
          <ParamMetaData>
            <B_UNICODE>
              <BYTELEN>
                <BYTE>00 </BYTE>
              </BYTELEN>
              <BYTES ascii="">
              </BYTES>
            </B_UNICODE>
            <StatusFlags>
              <fByRefValue>
                <BIT>false</BIT>
              </fByRefValue>
              <fDefaultValue>
                <BIT>false</BIT>
              </fDefaultValue>
              <fCookie>
                <BIT>false</BIT>
              </fCookie>
            </StatusFlags>
            <TYPE_INFO>
              <TVP_TYPE_INFO>
                <TVP_TYPE>
                  <BYTE>F3 </BYTE>
                </TVP_TYPE>
                <TVP_TYPE_NAME>
                  <DbName>
                    <B_UNICODE></B_UNICODE>
                  </DbName>
                  <OwningSchema>
                    <B_UNICODE>dbo</B_UNICODE>
                  </OwningSchema>
                  <TypeName>
                    <B_UNICODE>tvptype</B_UNICODE>
                  </TypeName>
                </TVP_TYPE_NAME>
                <TVP_COLMETADATA>
                  <Count>
                    <USHORT>01 00 </USHORT>
                  </Count>
                  <TvpColumnMetaData>
                    <UserType>
                      <ULONG>00 00 00 00 </ULONG>
                    </UserType>
                    <Flags>
                      <USHORT>00 00 </USHORT>
                    </Flags>
            <TYPE_INFO>
              <VARLENTYPE>
                <BYTELEN_TYPE>
                  <BYTE>26 </BYTE>
                </BYTELEN_TYPE>
              </VARLENTYPE>
              <TYPE_VARLEN>
                <BYTELEN>
                  <BYTE>01 </BYTE>
                </BYTELEN>
              </TYPE_VARLEN>
            </TYPE_INFO>
                    <ColName>
                      <B_UNICODE>
                        <BYTELEN>
                          <BYTE>00 </BYTE>
                        </BYTELEN>
                        <BYTES ascii="">
                        </BYTES>
                      </B_UNICODE>
                    </ColName>
                  </TvpColumnMetaData>
                </TVP_COLMETADATA>
                <TVP_END_TOKEN>
                  <TokenType>
                    <BYTE>00 </BYTE>
                  </TokenType>
                </TVP_END_TOKEN>
                <TVP_ROW>
                  <TokenType>
                    <BYTE>01 </BYTE>
                  </TokenType>
                  <AllColumnData>
            <TYPE_VARBYTE>
              <TYPE_VARLEN>
                <BYTELEN>
                  <BYTE>01</BYTE>
                </BYTELEN>
               <BYTES>02</BYTES>
              </TYPE_VARLEN>
            </TYPE_VARBYTE>
                  </AllColumnData>
```

```
                </TVP_ROW>
                <TVP_END_TOKEN>
                  <TokenType>
                    <BYTE>00 </BYTE>
                  </TokenType>
                </TVP_END_TOKEN>
              </TVP_TYPE_INFO>
            </TYPE_INFO>
          </ParamMetaData>
          <ParamLenData>
          </ParamLenData>
        </ParameterData>
      </RPCReqBatch>
    </RPCRequest>
  </BufferData>
</tds>
```

4.13 SparseColumn Select Statement

SparseColumn select statement sent from client to server:

```
04 01 01 B9 00 00 01 00 81 02 00 00 00 00 00 09 00
26 04 02 69 00 64 00 00 00 00 00 0B 04 F1 00 11 73
00 70 00 61 00 72 00 73 00 65 00 50 00 72 00 6F 00
70 00 65 00 72 00 74 00 79 00 53 00 65 00 74 00 D1
04 01 00 00 00 FE FF FF FF FF FF FF FF 7A 00 00 00
3C 00 73 00 70 00 61 00 72 00 73 00 65 00 50 00 72
00 6F 00 70 00 31 00 3E 00 31 00 30 00 30 00 30 00
3C 00 2F 00 73 00 70 00 61 00 72 00 73 00 65 00 50
00 72 00 6F 00 70 00 31 00 3E 00 3C 00 73 00 70 00
61 00 72 00 73 00 65 00 50 00 72 00 6F 00 70 00 32
00 3E 00 66 00 6F 00 6F 00 3C 00 2F 00 73 00 70 00
61 00 72 00 73 00 65 00 50 00 72 00 6F 00 70 00 32
00 3E 00 00 00 00 00 D1 04 02 00 00 00 FE FF FF FF
FF FF FF FF 3E 00 00 00 3C 00 73 00 70 00 61 00 72
00 73 00 65 00 50 00 72 00 6F 00 70 00 31 00 3E 00
31 00 30 00 30 00 30 00 3C 00 2F 00 73 00 70 00 61
00 72 00 73 00 65 00 50 00 72 00 6F 00 70 00 31
00 3E 00 00 00 00 00 D1 04 03 00 00 00 FE FF FF
FF FF FF FF FF 3E 00 00 00 3C 00 73 00 70 00 61
00 72 00 73 00 65 00 50 00 72 00 6F 00 70 00 32
00 3E 00 61 00 62 00 63 00 64 00 3C 00 2F 00 73
00 70 00 61 00 72 00 73 00 65 00 50 00 72 00 6F
00 70 00 32 00 3E 00 00 00 00 00 FD 10 00 C1 00
0A 00 00 00 00 00 00 00
```

```
<tds version="katmai">
  <BufferHeader>
    <Type>
      <BYTE>04 </BYTE>
    </Type>
    <Status>
      <BYTE>01 </BYTE>
      </Status>
      <Length>
        <BYTE>01 </BYTE>
        <BYTE>B9 </BYTE>
      </Length>
      <SPID>
        <BYTE>00 </BYTE>
        <BYTE>00 </BYTE>
      </SPID>
      <Packet>
        <BYTE>01 </BYTE>
      </Packet>
      <Window>
        <BYTE>00 </BYTE>
      </Window>
    </BufferHeader>
    <BufferData>
      <TableResponse>
        <COLMETADATA>
          <TokenType>
            <BYTE>81 </BYTE>
          </TokenType>
          <Count>
            <USHORT>02 00 </USHORT>
          </Count>
          <ColumnData>
            <UserType>
              <ULONG>00 00 00 00 </ULONG>
            </UserType>
            <Flags>
              <USHORT>09 00 </USHORT>
            </Flags>
            <TYPE_INFO>
              <VARLENTYPE>
                <BYTELEN_TYPE>
                  <BYTE>26 </BYTE>
                </BYTELEN_TYPE>
              </VARLENTYPE>
              <TYPE_VARLEN>
                <BYTELEN>
                  <BYTE>04 </BYTE>
                </BYTELEN>
              </TYPE_VARLEN>
            </TYPE_INFO>
            <ColName>
              <B_UNICODE>
                <BYTELEN>
                  <BYTE>02 </BYTE>
                </BYTELEN>
                <BYTES ascii="i.d.">69 00 64 00 </BYTES>
              </B_UNICODE>
            </ColName>
          </ColumnData>
          <ColumnData>
            <UserType>
              <ULONG>00 00 00 00 </ULONG>
            </UserType>
            <Flags fSparseColumn="true">
              <USHORT>0B 04 </USHORT>
            </Flags>
            <TYPE_INFO>
              <VARLENTYPE>
                <USHORTLEN_TYPE>
                  <BYTE>F1 </BYTE>
                </USHORTLEN_TYPE>
              </VARLENTYPE>
              <XML_INFO>
                <SCHEMA_PRESENT>
                  <BYTE>00 </BYTE>
                </SCHEMA_PRESENT>
              </XML_INFO>
            </TYPE_INFO>
            <ColName>
              <B_UNICODE>
                <BYTELEN>
                  <BYTE>11 </BYTE>
                </BYTELEN>
                <BYTES ascii="s.p.a.r.s.e.P.r.o.p.e.r.t.y.S.e.t.">73 00 70 00 61 00 72 00 73 00
  65 00 50 00 72 00 6F 00 70 00 65 00 72 00 74 00 79 00 53 00 65 00 74 00 </BYTES>
              </B_UNICODE>
            </ColName>
          </ColumnData>
        </COLMETADATA>
        <ROW>
          <TokenType>
            <BYTE>D1 </BYTE>
          </TokenType>
          <TYPE_VARBYTE>
            <TYPE_VARLEN>
              <BYTELEN>
                <BYTE>04 </BYTE>
              </BYTELEN>
            </TYPE_VARLEN>
            <BYTES>01 00 00 00 </BYTES>
          </TYPE_VARBYTE>
          <TYPE_VARBYTE>
            <BYTES>FE FF FF FF FF FF FF FF 7A 00 00 00 3C 00 73 00 70 00 61 00 72 00 73 00 65
  00 50 00 72 00 6F 00 70 00 31 00 3E 00 31 00 30 00 30 00 30 00 3C 00 2F 00 73 00 70 00 61 00
  72 00 73 00 65 00 50 00 72 00 6F 00 70 00 31 00 3E 00 3C 00 73 00 70 00 61 00 72 00 73 00 65
  00 50 00 72 00 6F 00 70 00 32 00 3E 00 66 00 6F 00 6F 00 3C 00 2F 00 73 00 70 00 61 00 72 00
  73 00 65 00 50 00 72 00 6F 00 70 00 32 00 3E 00 00 00 00 00 </BYTES>
          </TYPE_VARBYTE>
        </ROW>
        <ROW>
          <TokenType>
            <BYTE>D1 </BYTE>
          </TokenType>
          <TYPE_VARBYTE>
            <TYPE_VARLEN>
              <BYTELEN>
                <BYTE>04 </BYTE>
              </BYTELEN>
            </TYPE_VARLEN>
            <BYTES>02 00 00 00 </BYTES>
          </TYPE_VARBYTE>
          <TYPE_VARBYTE>
```

```
          <BYTES>FE FF FF FF FF FF FF FF 3E 00 00 00 3C 00 73 00 70 00 61 00 72 00 73 00 65
00 50 00 72 00 6F 00 70 00 31 00 3E 00 31 00 30 00 30 00 30 00 3C 00 2F 00 73 00 70 00 61 00
72 00 73 00 65 00 50 00 72 00 6F 00 70 00 31 00 3E 00 00 00 00 00 </BYTES>
        </TYPE_VARBYTE>
      </ROW>
      <ROW>
        <TokenType>
          <BYTE>D1 </BYTE>
        </TokenType>
        <TYPE_VARBYTE>
          <TYPE_VARLEN>
            <BYTELEN>
              <BYTE>04 </BYTE>
            </BYTELEN>
          </TYPE_VARLEN>
          <BYTES>03 00 00 00 </BYTES>
        </TYPE_VARBYTE>
        <TYPE_VARBYTE>
          <BYTES>FE FF FF FF FF FF FF FF 3E 00 00 00 3C 00 73 00 70 00 61 00 72 00 73 00 65
00 50 00 72 00 6F 00 70 00 32 00 3E 00 61 00 62 00 63 00 64 00 3C 00 2F 00 73 00 70 00 61 00
72 00 73 00 65 00 50 00 72 00 6F 00 70 00 32 00 3E 00 00 00 00 00 </BYTES>
        </TYPE_VARBYTE>
      </ROW>
      <NBCROW>
        <TokenType>
          <BYTE>D2 </BYTE>
        </TokenType>
        <NBCBitMap>
          <BYTES>02 </BYTES>
        </NBCBitMap>
        <TYPE_VARBYTE>
          <TYPE_VARLEN>
            <BYTELEN>
              <BYTE>04 </BYTE>
            </BYTELEN>
          </TYPE_VARLEN>
          <BYTES>04 00 00 00 </BYTES>
        </TYPE_VARBYTE>
      </NBCROW>
      <NBCROW>
        <TokenType>
          <BYTE>D2 </BYTE>
        </TokenType>
        <NBCBitMap>
          <BYTES>02 </BYTES>
        </NBCBitMap>
        <TYPE_VARBYTE>
          <TYPE_VARLEN>
            <BYTELEN>
              <BYTE>04 </BYTE>
            </BYTELEN>
          </TYPE_VARLEN>
          <BYTES>05 00 00 00 </BYTES>
        </TYPE_VARBYTE>
      </NBCROW>
      <NBCROW>
        <TokenType>
          <BYTE>D2 </BYTE>
        </TokenType>
```

```
  <NBCBitMap>
    <BYTES>02 </BYTES>
  </NBCBitMap>
  <TYPE_VARBYTE>
    <TYPE_VARLEN>
      <BYTELEN>
        <BYTE>04 </BYTE>
      </BYTELEN>
    </TYPE_VARLEN>
    <BYTES>06 00 00 00 </BYTES>
  </TYPE_VARBYTE>
</NBCROW>
<NBCROW>
  <TokenType>
    <BYTE>D2 </BYTE>
  </TokenType>
  <NBCBitMap>
    <BYTES>02 </BYTES>
  </NBCBitMap>
  <TYPE_VARBYTE>
    <TYPE_VARLEN>
      <BYTELEN>
        <BYTE>04 </BYTE>
      </BYTELEN>
    </TYPE_VARLEN>
    <BYTES>07 00 00 00 </BYTES>
  </TYPE_VARBYTE>
</NBCROW>
<NBCROW>
  <TokenType>
    <BYTE>D2 </BYTE>
  </TokenType>
  <NBCBitMap>
    <BYTES>02 </BYTES>
  </NBCBitMap>
  <TYPE_VARBYTE>
    <TYPE_VARLEN>
      <BYTELEN>
        <BYTE>04 </BYTE>
      </BYTELEN>
    </TYPE_VARLEN>
    <BYTES>08 00 00 00 </BYTES>
  </TYPE_VARBYTE>
</NBCROW>
<NBCROW>
  <TokenType>
    <BYTE>D2 </BYTE>
  </TokenType>
  <NBCBitMap>
    <BYTES>02 </BYTES>
  </NBCBitMap>
  <TYPE_VARBYTE>
    <TYPE_VARLEN>
      <BYTELEN>
        <BYTE>04 </BYTE>
      </BYTELEN>
    </TYPE_VARLEN>
    <BYTES>09 00 00 00 </BYTES>
  </TYPE_VARBYTE>
```

```
      </NBCROW>
      <NBCROW>
        <TokenType>
          <BYTE>D2 </BYTE>
        </TokenType>
        <NBCBitMap>
          <BYTES>02 </BYTES>
        </NBCBitMap>
        <TYPE_VARBYTE>
          <TYPE_VARLEN>
            <BYTELEN>
              <BYTE>04 </BYTE>
            </BYTELEN>
          </TYPE_VARLEN>
          <BYTES>0A 00 00 00 </BYTES>
        </TYPE_VARBYTE>
      </NBCROW>
      <DONE>
        <TokenType>
          <BYTE>FD </BYTE>
        </TokenType>
        <Status>
          <USHORT>10 00 </USHORT>
        </Status>
        <CurCmd>
          <USHORT>C1 00 </USHORT>
        </CurCmd>
        <DoneRowCount>
          <LONGLONG>0A 00 00 00 00 00 00 00 </LONGLONG>
        </DoneRowCount>
      </DONE>
    </TableResponse>
  </BufferData>
</tds>
```

5 Security

5.1 Security Considerations for Implementers

As previously described in this document, the TDS protocol provides facilities for authentication and channel encryption negotiation. If SSPI authentication is requested by the client application, then the exact choice of security mechanisms is determined by the SSPI layer. Likewise, while the decision as to whether channel encryption should be used is negotiated in the TDS layer, the exact choice of cipher suite is negotiated by the TLS/SSL layer.

6 Appendix A: Product Behavior

The information in this specification is applicable to the following Microsoft products:

- Microsoft Windows® 2000 operating system
- Windows® XP operating system
- Windows Server® 2003 operating system
- Windows Vista® operating system
- Windows Server® 2008 operating system
- Windows® 7 operating system
- Windows Server® 2008 R2 operating system
- Microsoft® SQL Server® 2000
- Microsoft® SQL Server® 2005
- Microsoft® SQL Server® 2008
- Microsoft® SQL Server® 2008 R2

Exceptions, if any, are noted below. If a service pack number appears with the product version, behavior changed in that service pack. The new behavior also applies to subsequent service packs of the product unless otherwise specified.

Unless otherwise specified, any statement of optional behavior in this specification prescribed using the terms SHOULD or SHOULD NOT implies product behavior in accordance with the SHOULD or SHOULD NOT prescription. Unless otherwise specified, the term MAY implies that product does not follow the prescription.

<1> Section 2.2.5.2.3: Partially Length-prefixed Bytes were introduced in TDS 7.2.

<2> Section 2.2.5.3: The ALL_HEADERS rule and the different packet data stream headers specified in this section and corresponding subsections are introduced in TDS 7.2.

<3> Section 2.2.5.4.2: XMLTYPE and UDTTYPE were introduced in TDS 7.2.

<4> Section 2.2.5.4.3: This class of types was introduced in TDS 7.2.

<5> Section 2.2.5.5.2: UDT Data Type was introduced in TDS 7.2.

<6> Section 2.2.5.5.3: The XML data type definition stream was introduced in TDS 7.2.

<7> Section 2.2.6.3: SQL Server versions earlier than 7.0 reject this message. When a client receives a reject for this message, the client disconnects, formats the correct login message for the server version, and reconnects to the server.

<8> Section 2.2.6.3: The version numbers used by clients up to SQL Server2008 are as follows.

SQL Server Version	Version Sent from Client to Server
7.0	0x00000070

SQL Server Version	Version Sent from Client to Server
2000	0x00000071
2000 SP1	0x01000071
2005	0x02000972
2008	0x03000A73
2008	0x03000B73

<9> Section 2.2.6.4: The client, no matter if it is TDS 7.1 or TDS 7.2, will send the SSL handshake payloads as data in a PRELOGIN message. For TDS 7.1, the server will send the SSL handshake payloads as data in a table response message (0x04). For TDS 7.2, the server will send the SSL handshake payloads as data in a PRELOGIN message.

<10> Section 2.2.6.5: This element has the value 0x80, except in TDS 7.2 where this element has the value 0xFF.

<11> Section 2.2.6.5: This element is available only in TDS 7.2.

<12> Section 2.2.6.8: This request type was introduced in TDS 7.2.

<13> Section 2.2.6.8: This request type was introduced in TDS 7.2.

<14> Section 2.2.6.8: This request type was introduced in TDS 7.2.

<15> Section 2.2.6.8: This request type was introduced in TDS 7.2.

<16> Section 2.2.6.8: This request type was introduced in TDS 7.2.

<17> Section 2.2.7.4: This element is not implemented in SQL Server.

<18> Section 2.2.7.5: The type of DoneRowCount is LONG for TDS 7.1, and ULONGLONG for TDS 7.2.

<19> Section 2.2.7.5: This bit is not set by SQL Server, and should be considered reserved for future use.

<20> Section 2.2.7.6: The type of DoneRowCount is LONG for TDS 7.1, and ULONGLONG for TDS 7.2.

<21> Section 2.2.7.6: This bit is not set by SQL Server, and should be considered reserved for future use.

<22> Section 2.2.7.7: The type of DoneRowCount is LONG for TDS 7.1, and ULONGLONG for TDS 7.2.

<23> Section 2.2.7.7: This bit is not set by SQL Server, and should be considered reserved for future use.

<24> Section 2.2.7.8: This type was introduced in TDS 7.2.

<25> Section 2.2.7.8: This type was introduced in TDS 7.2.

<26> Section 2.2.7.8: This type was introduced in TDS 7.2.

<27> Section 2.2.7.8: This type was introduced in TDS 7.2.

<28> Section 2.2.7.8: This type was introduced in TDS 7.2.

<29> Section 2.2.7.8: This type was introduced in TDS 7.2.

<30> Section 2.2.7.8: This type was introduced in TDS 7.2.

<31> Section 2.2.7.8: This type is not used by SQL Server.

<32> Section 2.2.7.8: This type was introduced in TDS 7.2.

<33> Section 2.2.7.8: This type was introduced in TDS 7.2.

<34> Section 2.2.7.8: This type was introduced in TDS 7.2.

<35> Section 2.2.7.9: The type of LineNumber is USHORT for TDS 7.1, and LONG for TDS 7.2.

<36> Section 2.2.7.9: SQL Server does not raise system errors with severities of 0 through 9.

<37> Section 2.2.7.9: For compatibility reasons, SQL Server converts severity 10 to severity 0 before returning the error information to the calling application.

<38> Section 2.2.7.10: The type of LineNumber is USHORT for TDS 7.1, and ULONG for TDS 7.2.

<39> Section 2.2.7.10: Numbers less than 20001 are reserved by SQL Server.

<40> Section 2.2.7.11: The following table shows the values in network transfer format.

SQL Server	Client to server	Server to client
7.0	0x00000070	0x07000000
2000	0x00000071	0x07010000
2000 SP1	0x01000071	0x71000001
2005	0x02000972	0x72090002
*2008	0x03000A73	0x730A0003
2008	0x03000B73	0x730B0003

*SQL Server 2008 TDS version 0x03000A73 does not include support for NBCROW and fSparseColumnSet.

<41> Section 2.2.7.16: This flag is not implemented in SQL Server.

<42> Section 3.2.2: In a Windows implementation the default value for MDAC/WDAC and SNAC Client Requests is 0, which is interpreted as no timeout. For an SqlClient Client Request the default value is 30 seconds.

<43> Section 3.2.2: In a Windows implementation the default setting for MDAC/WDAC and SNAC Cancel Timer values is 120 seconds. For an SqlClient Cancel Timer the default value is 5 seconds.

7 Change Tracking

This section identifies changes made to [MS-TDS] protocol documentation between December 2009 and January 2010 releases. Changes are classed as major, minor, or editorial.

Major changes affect protocol interoperability or implementation. Examples of major changes are:

- A document revision that incorporates changes to interoperability requirements or functionality.
- An extensive rewrite, addition, or deletion of major portions of content.
- A protocol is deprecated.
- The removal of a document from the documentation set.
- Changes made for template compliance.

Minor changes do not affect protocol interoperability or implementation. Examples are updates to fix technical accuracy or ambiguity at the sentence, paragraph, or table level.

Editorial changes apply to grammatical, formatting, and style issues.

No changes means that the document is identical to its last release.

Major and minor changes can be described further using the following revision types:

- New content added.
- Content update.
- Content removed.
- New product behavior note added.
- Product behavior note updated.
- Product behavior note removed.
- New protocol syntax added.
- Protocol syntax updated.
- Protocol syntax removed.
- New content added due to protocol revision.
- Content updated due to protocol revision.
- Content removed due to protocol revision.
- New protocol syntax added due to protocol revision.
- Protocol syntax updated due to protocol revision.
- Protocol syntax removed due to protocol revision.
- New content added for template compliance.
- Content updated for template compliance.

- Content removed for template compliance.
- Obsolete document removed.

Editorial changes always have the revision type "Editorially updated."

Some important terms used in revision type descriptions are defined as follows:

Protocol syntax refers to data elements (such as packets, structures, enumerations, and methods) as well as interfaces.

Protocol revision refers to changes made to a protocol that affect the bits that are sent over the wire.

Changes are listed in the following table. If you need further information, please contact protocol@microsoft.com.

Section	**Tracking number (if applicable) and description**	**Major change (Y or N)**	**Revision Type**
1.1 Glossary	Removed "distributed queries" glossary term.	N	Content removed.
1.2.1 Normative References	Removed [FIPS127], [ISO/IEC 14977], and [MS-ERREF].	N	Content removed.
1.2.2 Informative References	Changed [MSDN-MS-BINXML] to [MSDN-BINXML], [MSDN-SQLCOLLATION] to [MSDN-SQLCollation], and [MSDN-UPDATETETETEXT] to [MSDN-UPDATETEXT]. Removed references [MSDN-NetwkPacketSz] and [RFC1964].	N	Content update.

8 Index