[MS-SSTDS]:
Tabular Data Stream Protocol
Version 4.2

Intellectual Property Rights Notice for Open Specifications Documentation

- **Technical Documentation.** Microsoft publishes Open Specifications documentation for protocols, file formats, languages, standards as well as overviews of the interaction among each of these technologies.
- **Copyrights.** This documentation is covered by Microsoft copyrights. Regardless of any other terms that are contained in the terms of use for the Microsoft website that hosts this documentation, you may make copies of it in order to develop implementations of the technologies described in the Open Specifications and may distribute portions of it in your implementations using these technologies or your documentation as necessary to properly document the implementation. You may also distribute in your implementation, with or without modification, any schema, IDL's, or code samples that are included in the documentation. This permission also applies to any documents that are referenced in the Open Specifications.
- **No Trade Secrets.** Microsoft does not claim any trade secret rights in this documentation.
- **Patents.** Microsoft has patents that may cover your implementations of the technologies described in the Open Specifications. Neither this notice nor Microsoft's delivery of the documentation grants any licenses under those or any other Microsoft patents. However, a given Open Specification may be covered by Microsoft's Open Specification Promise (available here: http://www.microsoft.com/interop/osp) or the Community Promise (available here: http://www.microsoft.com/interop/cp/default.mspx). If you would prefer a written license, or if the technologies described in the Open Specifications are not covered by the Open Specifications Promise or Community Promise, as applicable, patent licenses are available by contacting iplg@microsoft.com.
- **Trademarks.** The names of companies and products contained in this documentation may be covered by trademarks or similar intellectual property rights. This notice does not grant any licenses under those rights.
- **Fictitious Names.** The example companies, organizations, products, domain names, e-mail addresses, logos, people, places, and events depicted in this documentation are fictitious. No association with any real company, organization, product, domain name, email address, logo, person, place, or event is intended or should be inferred.

Reservation of Rights. All other rights are reserved, and this notice does not grant any rights other than specifically described above, whether by implication, estoppel, or otherwise.

Tools. The Open Specifications do not require the use of Microsoft programming tools or programming environments in order for you to develop an implementation. If you have access to Microsoft programming tools and environments you are free to take advantage of them. Certain Open Specifications are intended for use in conjunction with publicly available standard specifications and network programming art, and assumes that the reader either is familiar with the aforementioned material or has immediate access to it.

Revision Summary

Date	Revision History	Revision Class	Comments
08/07/2009	0.1	Major	First release.
11/06/2009	0.2	Minor	Updated the technical content.
03/05/2010	0.2.1	Editorial	Revised and edited the technical content.
04/21/2010	0.2.2	Editorial	Revised and edited the technical content.
06/04/2010	0.2.3	Editorial	Revised and edited the technical content.
09/03/2010	0.2.3	No change	No changes to the meaning, language, or formatting of the technical content.
02/09/2011	0.3.0	Minor	Clarified the meaning of the technical content.

Contents

1 Introduction

This document specifies the Microsoft Tabular Data Stream Version 4.2 (TDS 4.2), a Microsoft-proprietary protocol. All references to the term SQL Server refer to the Microsoft® SQL Server® product line. The TDS 4.2 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]:

big-endian
little-endian
Security Support Provider Interface (SSPI)
stored procedure
table response
transaction manager

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.

data stream: A stream of data that corresponds to specific TDS 4.2 semantics. A single data stream can represent an entire TDS 4.2 message or only a specific, well-defined portion of a TDS 4.2 message. A TDS 4.2 data stream can span multiple network data packets.

Distributed Transaction Coordinator (DTC): A Windows service that coordinates transactions across multiple databases. For more information, see [MSDN-DTC].

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

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

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.

remote procedure call (RPC): The direct invocation of a **stored procedure** or user-defined function on the **server**.

request: A TDS 4.2 message initiated by a **client** and sent to a **server**.

response: A TDS 4.2 message sent by a **server** to a **client** in response to a previously issued **request**.

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.

SQL batch: A set of **SQL statements**.

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

SQL statement: A character string expression in a language that the **server** understands.

TDS 4.2 session: A successfully established communication session between a **client** and a **server** on which the TDS 4.2 protocol is used for message exchange.

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

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

[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.

1.2.2 Informative References

[MBCS] Microsoft Corporation, "Code Pages Supported by Windows", http://msdn.microsoft.com/en-us/goglobal/bb964654.aspx

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

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

[MSDN-BULKINSERT] Microsoft Corporation, "About Bulk Import and Bulk Export Operations", http://msdn.microsoft.com/en-us/library/ms187042.aspx

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

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

[MSDN-UPDATETEXT] Microsoft Corporation, "UPDATETEXT (Transact-SQL)", http://msdn.microsoft.com/en-us/library/ms189466(SQL.105).aspx

[MSDN-WRITETEXT] Microsoft Corporation, "WRITETEXT (Transact-SQL)", http://msdn.microsoft.com/en-us/library/ms186838(SQL.105).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

[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 TDS 4.2 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 4.2 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 4.2 connection must be established. Note that the **TDS 4.2 session** is directly tied to the transport-level session, meaning that a TDS 4.2 session is established when the transport-level connection is established and the server receives a request to establish a TDS 4.2 connection. It persists until the transport-level connection is terminated (for example, when a TCP socket is closed). In addition, TDS 4.2 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.

The TDS 4.2 protocol includes facilities for authentication and identification, channel encryption negotiation, issuing of **SQL batches**, **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 figure depicts a (simplified) typical flow of communication for TDS 4.2.

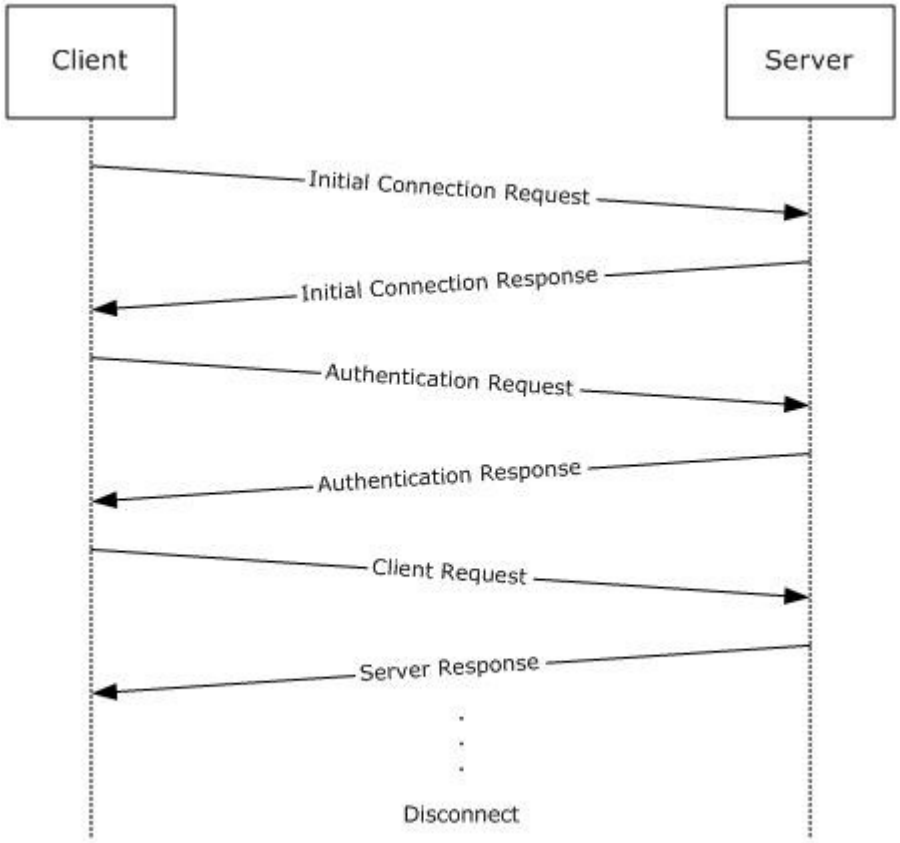

Figure 1: Communication flow in the TDS 4.2 protocol

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 a **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 SQL 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 that contains COLNAME and COLFMT) and then the rows follow. A completion message is sent after all the row data has been transferred.

```
Server:COLNAMEdata stream
COLFMTdata stream
ROWdata stream
.
```

```
.
ROWdata stream
DONEdata stream
```

See section 2.2.4 for additional information on the correlation between the data stream and TDS 4.2 buffer.

1.4 Relationship to Other Protocols

The TDS 4.2 protocol depends upon a network transport connection being established prior to a TDS 4.2 conversation occurring (the choice of transport protocol is not important to TDS 4.2).

This relationship is illustrated in the following figure.

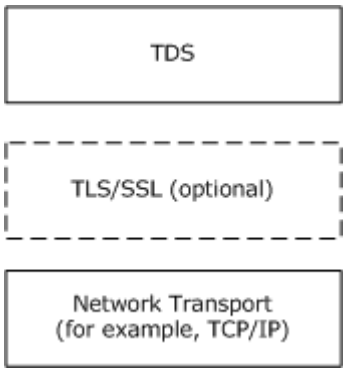

Figure 2: Protocol relationship

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 4.2.

No security association is assumed to have been established at the lower layer before TDS 4.2 begins functioning. For **SSPI** authentication to be used, SSPI support must be available on both the client and server machines (for more information about SSPI, see [SSPI]). If channel encryption is to be used, Transport Layer Security (TLS) /Secure Socket Layer (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. (For more details about TLS, see [RFC2246].)

1.6 Applicability Statement

The TDS 4.2 protocol is appropriate to use for facilitating request/response communications between an application and a database server in all scenarios in which 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:** This protocol supports exactly one version, which is Tabular Data Stream Protocol Version 4.2.
- **Security and Authentication Methods:** The TDS 4.2 protocol supports **SQL Server User Authentication (SQLAUTH)**. SQLAUTH is an authentication mechanism used to support SQL Server user accounts. The user name and password of the user account are transmitted as part of the login message that the client sends to the server. The TDS 4.2 protocol also supports SSPI authentication and indirectly supports any authentication mechanism that SSPI supports. The use of SSPI in TDS 4.2 is defined in sections 2.2.6.7 and 3.2.5.1.
- **Capability Negotiation:** This protocol does explicit capability negotiation as specified in this section.

In general, the TDS 4.2 protocol does not provide facilities for capability negotiation, because the complete set of supported features is fixed for each version of the protocol. Certain features such as authentication type are not negotiated, but instead requested by the client. However, one feature that is negotiated is channel encryption. The encryption behavior used for the TDS 4.2 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 of the influence of TDS 4.2 (for more details, see [RFC2246] and [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 specified in Augmented Backus-Naur Form (ABNF); for more details, see [RFC4234].

2.1 Transport

The TDS 4.2 protocol does not prescribe a specific underlying transport protocol to use on the Internet or on other networks. This protocol 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 4.2 packet sent from a TDS 4.2 client to a TDS 4.2 server MUST be contained within a single transport data unit. Any additional mapping of TDS 4.2 data onto the transport data units of the protocol in question is outside the scope of this specification.

The TDS 4.2 protocol has implementations over the following transports:

- TCP. For more details, see [RFC793].
- Named Pipes in message mode. For more details, see [PIPE].<1>
- Optionally, the TDS 4.2 protocol has implemented TLS (for more details, see [RFC2246]) and SSL on top of the preceding transports, in case TLS/SSL encryption is negotiated.

2.2 Message Syntax

Character data, such as SQL statements, within a TDS 4.2 message is in multi-byte character set (MBCS) format (for more information, see [MBCS]). Character counts within TDS 4.2 messages are 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 batch (in any language that the server will accept)
- A SQL statement followed by its associated binary data (for example, the data for a bulk load SQL statement)
- A remote procedure call (RPC)
- An attention signal
- A transaction manager request

These are briefly described in the subsections under this section; detailed descriptions of message contents are included in section 2.2.6.

2.2.1.1 Pre-Login

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

2.2.1.2 Login

When the client begins to establish a TDS 4.2 protocol connection with the server side, the client MUST send a login message data stream to the server. The client may have more than one connection to the server, but each connection is established separately in the same way. For additional details, see section 2.2.6.3.

After 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. For additional details, see section 3.3.5.1.

2.2.1.3 SQL Batch

To send a SQL statement or a batch of SQL statements, the SQL batch, represented by a multiple-byte character set (MBCS) string, is copied into the data section of a TDS 4.2 packet and then sent to the server. A SQLBatch packet header may span more than one TDS 4.2 packet. For additional details, see section 2.2.6.6.

2.2.1.4 Bulk Load

The bulk insert/bulk load operation is a case of a SQL statement that is followed by binary data. The client first sends the INSERT BULK SQL statement. The server responds with a DONE token. The client then sends a BulkLoadBCP data stream to the server. For additional details, see section 2.2.6.1.

2.2.1.5 Remote Procedure Call

To execute a remote procedure call (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 4.2 message and not intermixed with SQL statements. There can be several RPCs in one message. For additional details, see section 2.2.6.5.

2.2.1.6 Attention

The client can interrupt and cancel the current request by sending an Attention message. This is also known as **out-of-band** data, but any TDS 4.2 packet (request) that is currently being sent MUST be completely sent 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, sending a cancel request requires sending an Attention packet. An example of this behavior is when the client has already sent a request, which has the last packet with the EOM bit (0x01) set in the 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 the ignore bit (0x02) and EOM bit (0x01) set in the status to cancel the request. An example of this behavior is when one or more packets have been sent but the last packet with the EOM bit (0x01) set in the status has not been sent. Setting the ignore and EOM bits terminates the current request, and the server MUST ignore the current request. When the ignore and EOM bits are 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 a **Distributed Transaction Coordinator (DTC)** transaction. For more information, see [MSDN-DTC].

2.2.2 Server Messages

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

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

These are briefly described in the following sections; detailed descriptions of message contents are included 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 server interface and the server's product code and name. For a detailed description of the login response data stream, see section 2.2.7.13.

If there are any messages in the login response, an ERROR or INFO token data stream is returned from the server to the client. For additional information, see sections 2.2.7.11 and 2.2.7.12.

As part of the login response, the server may send 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. For more details about the different environment changes, see section 2.2.7.10.

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 token data stream MUST be the last thing sent in response to a client login request. For additional information about the DONE token data stream, see section 2.2.7.7.

2.2.2.3 Row Data

If the client request results in data being returned, the data precedes any other data streams returned from the server. Row data MUST be preceded by a description of the column names and data types. For additional information about how the column names and data types are described, see sections 2.2.7.6 and 2.2.7.5.

2.2.2.4 Return Status

When a stored procedure 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. A stored procedure execution is requested through either an RPC Batch or SQL Batch message. For additional information, see section 2.2.7.16.

2.2.2.5 Return Parameters

The response format for execution of a stored procedure is identical regardless of whether the request was sent as SQL Batch or RPC Batch. It is always a tabular result-type message.

The procedure can explicitly send any data, including row data, 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 have values returned from the server. For each output parameter, there is a corresponding return value that is 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. For additional details about the RETURNVALUE token, see section 2.2.7.17.

2.2.2.6 Response Completion (DONE)

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

When executing a batch of SQL statements, 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 determine after reading a DONE whether there are more results. For additional details about the DONE token, see section 2.2.7.7.

For stored procedures, completion of SQL statements in a stored procedure is indicated by a DONEINPROC token data stream<2> for each SQL statement and a DONEPROC token data stream for each completed stored procedure. For additional details about DONEINPROC and DONEPROC tokens, see sections 2.2.7.8 and 2.2.7.9, respectively.

2.2.2.7 Error and Info Messages

Besides returning a description of row data and the data itself, TDS 4.2 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, described in section 2.2.7.12 and the ERROR token data stream, described in section 2.2.7.11.

2.2.2.8 Attention Acknowledgment

After 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, described in section 2.2.7.7.

2.2.3 Packets

A packet is the unit written or read at one point in 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, clients may specify a requested packet size as part of the LOGIN 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 the ENVCHANGE token stream, described in section 2.2.7.10. 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.

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** attributes of the entire packet.

The subsections under this section provide 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	Login.	Yes
3	RPC.	Yes
4	Tabular result. 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

Value	Description	Buffer data?
15	Unused.	-
16	Unused.	-
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 (according to section 3.3.5), the message receiver SHOULD disconnect the connection. This applies to both the client and the server. For example, the server could disconnect the connection if the server receives a message with **Type** equal to 2 when the connection is already logged in.

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	18
Login	Client	2+17 (if Integrated authentication)
SQL batch	Client	1
Bulk load	Client	7
RPC	Client	3
Attention	Client	6
Transaction manager request	Client	14
Pre-login response	Server	4
Login response	Server	4
Row data	Server	4
Return status	Server	4
Return parameters	Server	4
Response completion (DONE)	Server	4
Error and info messages	Server	4
Attention acknowledgement	Server	4

2.2.3.1.2 Status

Status is 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). EOM indicates the last packet of the message.
0x02	From client to server. Ignore this event (0x01 MUST also be set).

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 is sent to the TDS 4.2 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, up to 255 (using modulo 256). This allows the receiver to track the sequence of TDS 4.2 packets for a given message. The value is currently ignored by the server.

2.2.3.1.6 Window

This 1-byte item 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 (for messages that contain packet data, see **Type** in section 2.2.3.1.1). 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 occurs only 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 represent the size of the packet header.

If a stream spans more than one packet, the EOM bit of the packet header **Status** code must be set to 0 (zero) 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 Packet Data Token and Tokenless Data Streams

The messages contained in packet data that pass between the client and the server may be 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 1-byte identifier used to describe the data

that follows it (for example, it contains token data type, token data length, and so on). Tokenless streams are typically used for simple messages. Messages that require a more detailed description of the data within them are sent as a token stream. The following table highlights which messages, as described in sections 2.2.1 and 2.2.2, use token streams and which do not.

Message type	Client or server message	Token stream used
Pre-login	Client	No
Login	Client	No
SQL batch	Client	No
Bulk load	Client	Yes
RPC	Client	Yes
Attention	Client	No
Transaction manager request	Client	No
Login response	Server	Yes
Row data	Server	Yes
Return status	Server	Yes
Return parameters	Server	Yes
Response completion (DONE)	Server	Yes
Error and info messages	Server	Yes
Attention acknowledgement	Server	No

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 1-byte identifier, followed by token-specific data.

2.2.4.2.1 Token Definition

There are three classes of token definitions:

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

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 functions.

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 this table, x denotes a bit position that 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, because the length of its associated data is encoded in the token itself. The different fixed - 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 this table, x denotes a bit position that 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 COLFMT and ALTFMT data streams as a single byte type. For additional details, see section 2.2.5.3.1.

2.2.4.2.1.3 Variable-Length Token (xx10xxxx)

This class of token definition is followed by a length specification. The length (in bytes) 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 is represented by 1 byte.
0	0	1	0	1	0	X	X	Length of data is represented by 1 byte.
0	0	1	0	1	1	X	X	Length of data is represented by 1 byte.
0	1	1	0	0	1	X	X	Length of data is represented by 1 byte.
0	1	1	0	1	0	X	X	Length of data is represented by 1 byte.

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

In the preceding table, x denotes a bit position that 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 that are either their normal fixed-length, corresponding to their type_info, or a special length if NULL.

Text and image data types have values that are either NULL or 1 to 2 gigabytes (0x00000000 to 0x7FFFFFFF bytes) in length.

A data type has a length of 0 if it is NULL.

2.2.4.3 DONE and Attention Tokens

The DONE token marks the end of the response for each executed SQL statement. Based on the SQL statement and the context in which it is executed, the server may generate the DONEPROC or DONEINPROC token instead.

The attention signal is sent using the out-of-band write operation provided by the network library. An out-of-band write provides the ability to send the attention signal whether the sender is in the middle of sending or processing a message or simply sitting idle. If the out-of-band operation is not supported, the clients MUST simply read and discard all of the data from the server until the final DONE token is read.

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 4.2 network data samples are available in section 4.

2.2.4.4.1 Sending a SQL Batch

In this example, a 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:     COLNAME   data stream
            COLFMT        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)
COLNAME   data stream
COLFMT        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, a SQL statement is sent to the server; however, 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 that is set in the status field of the DONE token.

```
Client:    select name, empid from employees

Server:    COLNAME   data stream
           COLFMT        datastream
           ROW           data stream
           .
           .
           ROW           data stream

Client:    ATTENTION SENT
```

The client reads and discards any data from the server until a DONE_ATTN acknowledgment is received.

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

2.2.5 Grammar Definition for Token Description

The TDS 4.2 protocol 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 (for more details, see [RFC4234]). Details can be found in the following subsections.

2.2.5.1 General Rules

Data structure encodings in TDS 4.2 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
```

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
```

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 NULL value.

```
GEN_NULL = %x00
```

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
```

Note All integer types are represented in the byte order requested by the client in the lInt2 field of the LOGIN token stream, unless otherwise specified.

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 it 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.

2.2.5.2 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
```

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, no data follows the length field.

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

Note that the lengths of B_VARCHAR and US_VARCHAR are given in bytes.

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 bytes, and variable bytes.

Integers

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

```
TYPE_VARLEN      =   BYTELEN
                     /
                     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, the TYPE_VARLEN value defines the length of the data to follow. As described earlier, the TYPE_INFO rule defines the type of TYPE_VARLEN (for example BYTELEN 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, because "INT2TYPE" is always 2 bytes in length.) See section 2.2.5.3 for more details.

The following data may be a stream of bytes or a NULL value. The GEN_NULL rule applies to all types.

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

2.2.5.3 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.

2.2.5.3.1 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
```

2.2.5.3.2 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 is 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.

```
GUIDTYPE         =   %x24  ; UniqueIdentifier
INTNTYPE         =   %x26  ; (see below)
DECIMALTYPE      =   %x37  ; Decimal
NUMERICTYPE      =   %x3F  ; Numeric
BITNTYPE         =   %x68  ; (see below)
DECIMALN         =   %x6A  ; Decimal
NUMERICNTYPE     =   %x6C  ; Numeric
FLTNTYPE         =   %x6D  ; (see below)
MONEYNTYPE       =   %x6E  ; (see below)
DATETIMNTYPE     =   %x6F  ; (see below)
CHARTYPE         =   %x2F  ; Char
VARCHARTYPE      =   %x27  ; VarChar
BINARYTYPE       =   %x2D  ; Binary
VARBINARYTYPE    =   %x25  ; VarBinary

TEXTTYPE         =   %x23  ; Text
IMAGETYPE        =   %x22  ; Image

BYTELEN_TYPE     =   GUIDTYPE
                     /
                     INTNTYPE
                     /
                     DECIMALTYPE
```

```
                    /
                    NUMERICTYPE
                    /
                    BITNTYPE
                    /
                    DECIMALN
                    /
                    NUMERICNTYPE
                    /
                    FLTNTYPE
                    /
                    MONEYNTYPE
                    /
                    DATETIMNTYPE
                    /
                    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.

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

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

VARLENTYPE       =   BYTELEN_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 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.

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

```
TYPE_INFO        =   FIXEDLENTYPE
                     /
                     (VARLENTYPE, TYPE_VARLEN [PRECISION SCALE])
```

2.2.5.5 Data Buffer Stream Tokens

The tokens defined as follows are used as part of the token-based data stream. For more details about the way each token is used inside the data stream, see section 2.2.6.

```
ALTFMT_TOKEN        =   %xA8
ALTNAME_TOKEN       =   %xA7
ALTROW_TOKEN        =   %xD3
COLFMT_TOKEN        =   %xA1

COLINFO_TOKEN       =   %xA5
COLNAME_TOKEN       =   %xA0
DONE_TOKEN          =   %xFD
DONEPROC_TOKEN      =   %xFE
DONEINPROC_TOKEN    =   %xFF
ENVCHANGE_TOKEN     =   %xE3
ERROR_TOKEN         =   %xAA
INFO_TOKEN          =   %xAB
LOGINACK_TOKEN      =   %xAD
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 with INSERT BULK.

Stream Comments

- The 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. For more information about how to notify the server, see [MSDN-BULKINSERT].
- A sample BulkLoadBCP message is in section 4.10.

Stream-Specific Rules

```
Length              =   USHORT
ImageTextColDim     =   USHORT
TiFlag              =   BYTE
ColId               =   BYTE
Reserved            =   USHORT   ; The server SHOULD ignore
                                   this field.
NumVarCols          =   BYTE
RowNum              =   BYTE
FixedColData        =   *BYTE
Paddings            =   *BYTE
RowLen              =   USHORT
VarColData          =   *BYTE
Adjust              =   1*BYTE
Offset              =   <NumVarCols+1>BYTE
ColData             =   NumVarCols
                        RowNum
                        *FixedColData
                        Paddings
                        RowLen
                        * VarColData
                        Adjust
                        Offset
RowData             =   Length
                        ColData
                        *(ImageTextColLenDim
                        TiFlag
                        ColId
                        Reserved
                        TYPE_VARBYTE)    ;The TYPE_VARBYTE for the
                                          type specified by TiFlag.
```

Stream Definition

```
BulkLoadBCP         =   1*RowData
```

Stream Parameter Details

Stream parameter details are described in the following table.

Parameter	Description
Length	The actual length, in bytes, of the ColData stream. The length does not include the

Parameter	Description
	Length field itself. The value MUST be greater than 0.
ImageTextColDim	The delimiter to mark the beginning of a text or image column. The value MUST be %x00 %x00.
TiFlag	The flag indicates the type of column. It MUST be either TEXTTYPE or IMAGETYPE.
ColId	It contains the Column ID for the text/image column. ColId is associated only with variable-length columns. It starts from the first variable column with ColId = %xFF and decreases by one for each eligible column.
NumVarCols	Number of variable-length columns.
RowNum	The row number of the current row. The TDS 4.2 server SHOULD ignore this field.
RowLen	The length of the current ColData field. The value MUST be identical to the Length field in RowData.
FixedColData	The actual data for a fixed-length column. It repeats for all fixed-length columns in the specific table for the operation, in the order defined in the table.
Paddings	Padding data. The server SHOULD ignore this field. When parsing the data, the server SHOULD get FixedColData from the beginning of ColData and get VarColData based on Offset and Adjust, with the assumption that Offset is the last field in ColData.
VarColData	The actual data for a variable-length column. It repeats for all variable-length columns, excluding text and image columns, in the specific table for the operation. It contains only the data part of TYPE_VARBYTE defined for the corresponding type. The length of VarColData is determined by Adjust and Offset fields as described in the following paragraphs. This field is skipped if the data for a column is NULL; that is, the length for a column is calculated to be 0.
Adjust	The n-th Adjust byte, counted from right backward (n>=1), contains the column number of the first variable length column which starts in the (n+1)-st 256 byte block of the record. The Adjust field has as many bytes as the number of 256 byte blocks in the ColData. Hence, the rightmost Adjust byte contains the column number of the first variable length column whose offset starts in the 2nd block. The next Adjust byte contains the column number of the first variable length column whose offset starts in the 3rd block, and do on. The last (or only) Adjust byte contains NumVarCols +1.
Offset	It contains the starting offset for each variable-length column. These bytes are in reverse order of the column creation order; that is, the last offset byte is for the first variable column. This byte, as adjusted by the Adjust table bytes, is the starting offset for the column's data in ColData. The length is determined by calculating the offset of the next column (previous offset byte) minus the starting offset. If the length is 0, the column is NULL. The leftmost offset byte contains the offset to the end of ColData.

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 length is the length of the data followed by the data itself.

Stream Comments

- The 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 with a WRITETEXT BULK [MSDN-WRITETEXT] or UPDATETEXT BULK [MSDN-UPDATETEXT] SQL statement.
- 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

Stream parameter details are described in the following table.

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

2.2.6.3 LOGIN

Stream Name

```
LOGIN
```

Stream Function

Defines the login record rules for use with SQL Server.

Stream Comments

- The packet header type is 0x02.
- The length of a LOGIN record must be larger than 563 bytes and must be smaller than 573 bytes.

Stream-Specific Rules

```
HostName       =    30BYTE
cbHostName     =    BYTE
UserName       =    30BYTE
cbUserName     =    BYTE
Password       =    30BYTE
cbPassword     =    BYTE
```

```
HostProc       =   8BYTE
cbHostProc     =   BYTE
AppType        =   6BYTE
lInt2          =   BYTE
lInt4          =   BYTE
lChar          =   BYTE
lFloat         =   BYTE
lUseDB         =   BYTE
lDumpLoad      =   BYTE
lInterface     =   BYTE
lType          =   BYTE
lDBLIDFlags    =   BYTE
AppName        =   30BYTE
cbAppName      =   BYTE
ServerName     =   30BYTE
cbServerName   =  BYTE
RemotePassword =  255BYTE
cbRemotePassword = BYTE
TDSVersion     =  DWORD
ProgName       =  10BYTE
cbProgName     =  BYTE
ProgVersion    =  DWORD
Language       =  30BYTE
cbLanguage     =  BYTE
SetLanguage    =  BYTE
PacketSize     =  6BYTE
cbPacketSize   =  BYTE
Padding        =  *8BYTE
```

Stream Definition

```
LOGIN            =   HostName
                     cbHostName
                     UserName
                     cbUserName
                     Password
                     cbPassword
                     HostProc
                     16FRESERVEDBYTE
                     AppType
                     cbHostProc
                     lInt2
                     lInt4
                     lChar
                     lFloat
                     FRESERVEDBYTE
                     lUseDB
                     lDumpLoad
                     lInterface
                     lType
                     6FRESERVEDBYTE
                     lDBLIBFlags
                     AppName
                     cbAppName
                     ServerName
                     cbServerName
                     RemotePassword
                     cbRemotePassword
```

```
TDSVersion
ProgName
cbProgName
ProgVersion
3FRESERVEDBYTE
Language
cbLanguage
SetLang
45FRESERVEDBYTE
PacketSize
cbPacketSize
Padding
```

Stream Parameter Details

Stream parameter details are described in the following table.

Parameter	Description
HostName	Name of the host.
UserName	The user ID used to validate access to the server.
Password	The password used to validate access to the server.
HostProc	The host process identification in hex format.
AppType	The unique client ID, for example: MAC address for the client machine.
lInt2	The byte order for all integer values exchanged between the client and the server unless otherwise specified. ▪ 2 The first byte is the most significant byte (680x0), big-endian ▪ 3 The first byte is the least significant byte (VAX, 80x86), little-endian
lInt4	The type of int4 for the client. The server should ignore this field.
lChar	The character set used on the client: ▪ 6 CHARSET_ASCII ▪ 7 CHARSET_EBCDIC
lFloat	The type of floating point representation used by the client. ▪ 5 FLOAT_VAX ▪ 10 FLOAT_IEEE_754 ▪ 11 ND5000
lUseDB	Set if the client desires warning messages on execution of the USE SQL statement. If this flag is not set, the client is not informed when the database changes. ▪ 0 USE_DB_OFF

Parameter	Description
	▪ 1 USE_DB_ON
lDumpLoad	Set if dump/load or BCP capabilities are needed by the client. ▪ 0 DUMPLOAD_ON ▪ 1 DUMPLOAD_OFF
lInterface	The type of SQL language that the client will send to the server. Only 0 and 1 are supported by SQL Server. ▪ 0 LDEFSQL (Default language. For SQL Server; that is, Transact-SQL) ▪ 1 LXSQL (Explicitly referencing Transact-SQL)
lType	The source of the connection. ▪ 0 Normal user connecting directly. ▪ 2 User login through another server. ▪ 4 Replication login. ▪ 8 Integrated security user login. If this type is used, USERNAME and PASSWORD MUST be ignored.
lDBLIBFlags	Indicates whether SSPI negotiation is required. ▪ 0x01 SSPI negotiation is required.
AppName	The name of the application.
ServerName	The network name for the server the client is connecting to.
RemotePassword	Remote password. The server should ignore the field.
TDSVersion	The TDS version of the client. For TDS 4.2, the value is 0x04020000 and is sent as big-endian. The value should be ignored by the server. If the following two conditions are met, TDS 4.2 should be used for communication between the client and the server: ▪ The packet header of the LOGIN data stream is 0x02. ▪ The lowest byte of ProgVersion is greater than or equal to 0x06.
ProgName	The name of the client program.
ProgVersion	The version of the client program.
Language	The name of the initial language to be used once login is complete.
SetLang	A flag to request notification of language changes. ▪ 0 = SET_LANG_OFF

Parameter	Description
	▪ 1 = SET_LANG_ON
PacketSize	The desired packet size being requested by the client.
Padding	Padding data to the login record. The number of bytes can be any number between 0 and 8. The server should ignore this field.

Login Data Validation Rules

All fields except Padding have a fixed length. Each data field has a corresponding length field that indicates how many bytes of the data field should be used. For example, cbHostName indicates how many bytes of HostName should be used. The remaining bytes of the data field should be ignored.

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 with the packet data containing a PRELOGIN structure.

This message stream is also used to wrap the SSL handshake payload if encryption is 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

- The packet header is type 0x12.
- A sample PRELOGIN message is shown 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
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	PL_OPTION_DATA = UL_VERSION US_SUBBUILD 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 SHOULD do this only if it can determine whether a feature is supported by the current 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.
TERMINATOR	0xFF	Termination token.

Notes

- PL_OPTION_TOKEN VERSION MUST be the first token sent as part of PRELOGIN.
- 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. For additional information, see section 3.3.5.1.
- A PRELOGIN message that wraps the SSL_PAYLOAD occurs only 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 negotiate the wire encryption to be used. The possible encryption option values are described in the following table.

Setting	Value	Description
ENCRYPT_OFF	0x00	Encryption is available but off.

Setting	Value	Description
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 responds 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

The server requires the client to behave in the manner that is described in the following table.

Client	Value returned from server is ENCRYPT_OFF	Value returned from server is ENCRYPT_ON	Value returned from server is ENCRYPT_REQ	Value returned from server is 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)
ENCRYPT_NOT_SUP	Invalid response (connection terminated)	Invalid response (connection terminated)	Error (connection terminated)	No encryption

If the client and server negotiate to enable encryption, an SSL handshake takes place immediately after the initial PRELOGIN/table response message exchange. The SSL payloads MUST be transported as data in TDS 4.2 buffers with the message type set to 0x12 in the packet header. The following is an example.

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

This applies to SSL traffic. Upon successful completion of the SSL handshake, the client proceeds to send the LOGIN 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, the server compares it to its instance name (in the server's locale) and if there is a mismatch, the server returns an INSTOPT option containing a byte with the value of 1 in the pre-login table response message. Otherwise, the server returns an INSTOPT option 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

- The packet header type is 0x03.
- To execute an RPC on the server, the client sends an 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 statements.
- A sample RPCRequest message is shown in section 4.6.

Stream-Specific Rules

```
ProcName          =   B_VARCHAR
fWithRecomp       =   BIT
fNoMetaData       =   BIT
OptionFlags       =   fWithRecomp
                      fNoMetaData
                      14FRESERVEDBIT

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

ParamMetaData     =   B_VARCHAR
                      StatusFlags
                      TYPE_INFO
ParamLenData      =   TYPE_VARBYTE

ParameterData     =   ParamMetaData
                      ParamLenData;

BatchFlag         =   %x80

RPCReqBatch       =   ProcName
                      OptionFlags
                      *ParameterData
```

Stream Definition

```
RPCRequest       =   RPCReqBatch
                     *(BatchFlag RPCReqBatch)
                     [BatchFlag]
```

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

Stream Parameter Details

Stream parameter details are described in the following table.

Parameter	Description
ProcName	The procedure 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 will be returned for the result set.
StatusFlags	Bit flags in least significant bit order: ▪ fByRefValue: 1 if the parameter is passed by reference (OUTPUT parameter) or 0 if the parameter is passed by value. ▪ fDefaultValue: 1 if the parameter being passed will 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. The BatchFlag 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.

2.2.6.6 SQLBatch

Stream Name

```
SQLBatch
```

Stream Function

Describes the format of the SQL batch message.

Stream Comments

- The packet header type is 0x01.
- A sample SQLBatch message is shown in section 4.4.

Stream-Specific Rules

```
SQLText          =    BYTESTREAM
```

Stream Definition

```
SQLBatch         =    SQLText
```

The byte stream contains the text of the SQL batch. The following is an example of a valid value for SQLText.

```
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) negotiation. For more information, see [RFC4178].

Stream Comments

- The packet header type is 0x11.
- If the client requested integrated authentication in the LOGIN message, the server MUST return an SSPI token that contains the SSPI signature<3> of the client driver that the server is supposed to talk to. The client MUST disconnect if the SSPI signature does not match its local value.
- If the SSPI signature matches, the client MUST send the SSPI message, which contains the initial SSPI data block (the initial SPNEGO security token), to the server. 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 occurs.
- The server completes the SSPI validation and returns a LOGINACK token to confirm the login.
- A sample SSPIMessage message is shown in section 4.9.

Stream-Specific Rules

```
SSPIData         =    BYTESTREAM
```

Stream Definition

```
SSPIMessage      =    SSPIData
```

Stream Parameter Details

Stream parameter details are described in the following table.

Parameter	Description
SSPIData	The SSPI data. The length of the data is determined by the Length field in the header.

2.2.6.8 Transaction Manager Request

Stream Name

```
TransMgrReq
```

Stream Function

Request to perform transaction coordination through a Distributed Transaction Coordinator (DTC) implemented to the Microsoft DTC Interface Specification. For more information, see [MSDN-DTC].

Stream Comments

- The packet header type is 0x0E.
- A sample transaction manager request message is shown in section 4.11.

Stream-Specific Rules

```
RequestType      =   USHORT
```

Stream Definition

```
TransMgrReq      =   RequestType
                     RequestPayload
```

RequestPayload details are described in the following table.

Stream Parameter Details

Stream parameter details are described in the following table.

Parameter	Description
RequestType	The types of transaction manager operations requested by the client are specified below. If an unknown Type is specified, the message receiver SHOULD disconnect the connection. ▪ 0 = TM_GET_DTC_ADDRESS. Returns the DTC network address as a result set with a single-column, single-row binary value. ▪ 1 = TM_PROPAGATE_XACT. Imports the DTC transaction into the server and returns a local transaction descriptor as a varbinary result set.
RequestPayload	▪ For RequestType TM_GET_DTC_ADDRESS: The RequestPayload SHOULD be a zero-

Parameter	Description
	length US_VARBYTE. ▪ For RequestType TM_PROPAGATE_XACT: Data contains an opaque buffer used by the server to enlist in a DTC transaction (for more information, see [MSDN-DTC]).

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 ALTFMT

Token Stream Name

```
ALTFMT
```

Token Stream Function

Describes the data type and length of column data that result from a SQL statementthat generates totals.

Token Stream Comments

- The token value is 0xA8.
- This token is used to tell the client the data type and length of the total column data. It describes the format of the data found in an ALTROW data stream.
- ALTNAME and ALTFMT data streams are grouped together. If the SQL statement generates more than one total, there is still exactly one ALTNAME data stream that carries all total columns and one ALTFMT data stream that caries all total formats for each set of totals.
- If the SQL statement generates more than one set of totals, the ALTNAME data streams and ALTFMT data streams arrive in pairs (for example, ALTNAME, ALTFMT, ALTNAME, ALTFMT).
- This stream does not occur without a preceding COLNAME and COLFMT pair, though there might be COLINFO and TABNAME streams in between.

Token Stream-Specific Rules

```
TokenType       =   BYTE
Length          =   USHORT
Id              =   USHORT
CAltCols        =   BYTE

ByCols          =   BYTE
Op              =   BYTE
Operand         =   BYTE
UserType        =   USHORT
fNullable       =   BIT
fCaseSen        =   BIT
usUpdateable    =   2BIT        ; 0 = ReadOnly
```

```
                                ; 1 = Read/Write
                                ; 2 = Unused
  fIdentity         =   BIT
  usReservedODBC    =   2BIT
  Flags             =   fNullable
                        fCaseSen
                        usUpdateable
                        fIdentity
                        FRESERVEDBIT
                        usReservedODBC
                        8FRESERVEDBIT
  TableName         =   B_VARCHAR
  ColNum            =   BYTE
  ComputeData       =   Op
                        Operand
                        UserType
                        Flags
                        TYPE_INFO
                        [TableName]
```

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

Token Stream Definition

```
  ALTFMT       =   TokenType
                       Length
                       Id
                       CAltCols
                       <CAltCols>ComputeData
                       ByCols
                       <ByCols>ColNum
```

Token Stream Parameter Details

Token stream parameter details are described in the following table.

Parameter	Description
TokenType	ALTFMT_TOKEN
Length	The number of bytes in the token stream excluding the **TokenType** and **Length** fields.
Id	The ID of the SQL statement to which the total column formats apply. This ID lets the client correctly interpret later ALTROW data streams.
CAltCols	The number of column data in the data stream.
ByCols	The number of grouping columns in the SQL statement that generates totals. For example, the SQL clause "compute count(sales) by year, month, division, department" has four grouping columns.
Op	The type of aggregate operator. 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)

Parameter	Description
	AOPMIN = %x51 ; Minimum value of the rows (MIN) AOPMAX = %x52 ; Maximum value of the rows (MAX)
Operand	The column number, starting from 1, in the result set that is the operand for the aggregate operator.
UserType	The user type ID of the data type of the column.
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. Refer to section 2.2.7.5 for a description of each bit flag: ▪ fCaseSens ▪ fNullable is a bit flag; set to 1 if the column is nullable ▪ usUpdateable ▪ fIdentity ▪ usReservedODBC
TableName	See section 2.2.7.5 for a description of TableName. This field MUST never be sent, because SQL statements that generate totals exclude TEXT/IMAGE.
ColName	The column name. Contains the column name length and column name.
ColNum	The column number as it appears in the COLFMT data stream. ColNum appears ByCols times.

2.2.7.2 ALTNAME

Token Stream Name

ALTNAME

Token Stream Function

Describes the column names of the SQL statement that generates totals.

Token Stream Comments

- The token value is 0xa7.
- This token is used to tell the client how many total columns are being returned to the client for a particular SQL statement that generates totals. It also indicates the column names for each total column.
- ALTNAME and ALTFMT data streams are grouped together. If the SQL statement generates more than one total, there is still exactly one ALTNAME data stream that carries all total columns and one ALTFMT data stream that carries all total formats for each set of totals. .
- If the SQL statement generates more than one set of totals, the ALTNAME data streams and ALTFMT data streams arrive in pairs (for example, ALTNAME, ALTFMT, ALTNAME, ALTFMT).

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

Token Stream-Specific Rules

```
TokenType        =   BYTE
Length           =   USHORT
Id               =   USHORT
ColNameData      =   B_VARCHAR
```

Token Stream Definition

```
ALTNAME      =       TokenType
                     Length
                     Id
                     1*ColNameData
```

Token Stream Parameter Details

Token stream parameter details are described in the following table.

Parameter	Description
TokenType	ALTNAME_TOKEN
Id	The ID of the SQL statement that generates totals to which the total column formats apply.
ColNameData	The column name for each total column.

2.2.7.3 ALTROW

Token Stream Name

```
ALTROW
```

Token Stream Function

Used to send a complete row of total data, where the data format is provided by the ALTMNAME and ALTFMT tokens.

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 ALTFMT (one Id for each SQL statement). This provides the mechanism for matching row data with correct SQL statements.

Token Stream-Specific Rules

```
TokenType        =   BYTE
Id               =   USHORT
Data             =   TYPE_VARBYTE
```

```
ComputeData      =   Data
```

Token Stream Definition

```
ALTROW      =   TokenType
                Id
                1*ComputeData
```

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

Token Stream Parameter Details

Token stream parameter details are described in the following table.

Parameter	Description
TokenType	ALTROW_TOKEN
Id	The ID of the SQL statement that generates totals to which the total column formats apply.
Data	The actual data for the column. The TYPE_INFO information describing the data type of this data is given in the preceding ALTFMT_TOKEN.

2.2.7.4 COLINFO

Token Stream Name

```
COLINFO
```

Token Stream Function

Describes the column information in browse mode (for more information, see [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          =   B_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

Token stream parameter details are described in the following table.

Parameter	Description
TokenType	COLINFO_TOKEN
Length	The actual data length, in bytes, of the ColProperty stream. The length does not include the token type and the 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 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 requested, but was added because it was part of a key for the associated table). 0x20: DIFFERENT_NAME (the column name is different from the requested column name if there is a column alias).
ColName	The base column name. This occurs only if DIFFERENT_NAME is set in Status.

2.2.7.5 COLFMT

Token Stream Name

```
COLFMT
```

Token Stream Function

Describes the data type and length of the column data for ROWs that follow in the data stream.

Token Stream Comments

- The token value is 0xA1.
- 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 COLFMT data streams are grouped together.

Token Stream-Specific Rules

```
TokenType        =   BYTE
```

```
UserType          =   USHORT
fNullable         =   BIT
fCaseSen          =   BIT
usUpdateable      =   2BIT          ; 0 = ReadOnly
                                    ; 1 = Read/Write
                                    ; 2 = Unused
fIdentity         =   BIT
usReservedODBC    =   2BIT
Flags             =  fNullable
                     fCaseSen
                     usUpdateable
                     fIdentity
                     FRESERVEDBIT
                     usReservedODBC
                     8FRESERVEDBIT
TableName         =   US_VARCHAR
ColFmtData        =   UserType
                     Flags
                     TYPE_INFO
                     [TableName]
```

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

Token Stream Definition

```
COLFMT       =    TokenType
                  Length
                  1*ColFmtData
```

Token Stream Parameter Details

Token stream parameter details are described in the following table.

Parameter	Description
TokenType	COLFMT_TOKEN
Length	The number of bytes in the token stream, excluding the **TokenType** and **Length** fields.
UserType	The user type ID of the column's data type.
Flags	Bit flags in least significant bit order: ▪ fCaseSen is a bit flag. Set to 1 if the column is case sensitive for searches ▪ 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 updateability is unknown.<4> ▪ fIdentity is a bit flag. Its value is 1 if the column is an identity column. ▪ usReservedODBC is a 2-bit field that is used by ODS gateways supporting the ODBC ODS gateway driver.
TableName	The fully qualified base table name for this column. Contains the table name length and

Parameter	Description
	table name. This exists only for text and image columns.

2.2.7.6 COLNAME

Token Stream Name

```
COLNAME
```

Token Stream Function

Describes the column names of the returning rows.

Token Stream Comments

- The token value is 0xA0.
- This token is used to tell the client how many columns of data are being returned to the client. It also indicates the column names for each column of data.
- All COLNAME data streams are grouped together.

Token Stream-Specific Rules

```
TokenType        =   BYTE
Length           =   USHORT
ColNameData      =   B_VARCHAR
```

Token Stream Definition

```
COLNAME          =   TokenType
                     Length
                     1*ColNameData
```

Token Stream Parameter Details

Token stream parameter details are described in the following table.

Parameter	Description
TokenType	COLNAME_TOKEN
Length	The number of bytes in the token stream, excluding the **TokenType** and **Length** fields.
ColNameData	The column name for each column.

2.2.7.7 DONE

Token Stream Name

```
DONE
```

Token Stream Function

Indicates the completion status of a SQL statement.

Token Stream Comments

- The token value is 0xFD.
- This token is used to indicate the completion of a SQL statement. Because multiple SQL statements may be sent to the server in a single SQL batch, multiple DONE tokens may be generated. In this case, all but the final DONE token will have a Status value with the DONE_MORE bit set (details follow).
- A DONE token is returned for each SQL statement in the SQL batch, except for variable declarations.
- For execution of SQL statements 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
```

Token Stream Definition

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

Token Stream Parameter Details

Token stream parameter details are described in the following table.

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 response; subsequent data streams to follow). ▪ 0x2: DONE_ERROR (an error occurred on the current SQL statement). ▪ 0x4: DONE_INXACT (a transaction is in progress).<5> ▪ 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

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

2.2.7.8 DONEINPROC

Token Stream Name

```
DONEINPROC
```

Token Stream Function

Indicates the completion status of a SQL statement within a stored procedure.

Token Stream Comments

- The token value is 0xFF.
- A DONEINPROC token is sent for each executed SQL 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
```

Token Stream Definition

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

Token Stream Parameter Details

Token stream parameter details are described in the following table.

Parameter	Description
TokenType	DONEINPROC_TOKEN
Status	The Status field MUST be a bitwise 'OR' of the following:

Parameter	Description
	▪ 0x1: DONE_MORE (this DONEINPROC message is not the final DONE/DONEPROC/DONEINPROC message in the response; more data streams are to follow). ▪ 0x2: DONE_ERROR (an error occurred on the current SQL statement, or execution of a stored procedure was interrupted.). ▪ 0x4: DONE_INXACT (a transaction is in progress).<6> ▪ 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). ▪ 0x100: DONE_SRVERROR (used in place of DONE_ERROR when an error occurred on the current SQL statement that is severe enough to require the result, if any, to be discarded).
CurCmd	The token of the current SQL statement.
DoneRowCount	The count of rows that were affected by the SQL statement. The value of DoneRowCount is valid if the value of Status includes DONE_COUNT.

2.2.7.9 DONEPROC

Token Stream Name

```
DONEPROC
```

Token Stream Function

Indicates the completion status of a stored procedure. This is also generated for stored procedures executed through SQL statements.

Token Stream Comments

- The token value is 0xFE.
- A DONEPROC token is sent when all the SQL 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 that is called.

Token Stream-Specific Rules

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

Token Stream Definition

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

Token Stream Parameter Details

Token stream parameter details are described in the following table.

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 response; more data streams are to follow). ▪ 0x2: DONE_ERROR (an error occurred on the current stored procedure). ▪ 0x4: DONE_INXACT (a transaction is in progress).<7> ▪ 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 RPC batch). ▪ 0x100: DONE_SRVERROR (used in place of DONE_ERROR when an error occurred on the current stored procedure that is severe enough to require the result set, if any, to be discarded).
CurCmd	The token of the SQL statement for executing stored procedures.
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.

2.2.7.10 ENVCHANGE

Token Stream Name

```
ENVCHANGE
```

Token Stream Function

A notification of an environment change (such as database and language).

Token Stream Comments

- The token value is 0xE3.
- Includes old and new environment values.

Token Stream-Specific Rules

```
TokenType        =   BYTE
Length           =   USHORT
Type             =   BYTE
NewValue         =   B_VARBYTE
OldValue         =   B_VARBYTE
EnvValueData     =   Type
                     NewValue
                     OldValue
```

Token Stream Definition

```
ENVCHANGE        =   TokenType
                     Length
                     EnvValueData
```

Token Stream Parameter Details

Token stream parameter details are described in the following table.

Parameter	Description
TokenType	ENVCHANGE_TOKEN
Length	The total length of the ENVCHANGE data stream (EnvValueData).
Type	The type of environment change: 1: Database 2: Language 3: Character set 4: Packet size

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

Note

For types 1, 2, and 3, the payload is an MBCS string; the LENGTH always reflects the number of bytes.

2.2.7.11 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
```

Token Stream Definition

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

Token Stream Parameter Details

Token stream parameter details are described in the following table.

Parameter	Description
TokenType	ERROR_TOKEN
Length	The total length of the ERROR data stream, in bytes.
Number	The error number.<8>
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 batch or stored procedure that caused the error. Line numbers

Parameter	Description
	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.<9>
10	Informational messages that return status information or report errors that are not severe.<10>
11-16	Errors that can be corrected by the user.
11	The given object or entity does not exist.
12	A special severity for SQL statements that do not use locking because of special options. In some cases, read operations performed by these SQL statements could result in inconsistent data, because locks do not guarantee consistency.
13	Transaction-deadlock errors.
14	Security-related errors, such as permission denied.
15	Syntax errors in the SQL statement.
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 SQL 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 SQL statement completes execution, and the connection to the instance of the database engine is maintained. System administrator action is required.
19	A nonconfigurable database engine limit has been exceeded, and the current SQL batch has been terminated. Error messages with a severity level of 19 or higher stop the execution of the current SQL 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 was executing a SQL 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 will 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	An SQL 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	A problem has been encountered that affects all tasks in the current database, but it is unlikely

Class level	Description
	that the database itself has been damaged.
22	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, 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	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	A media failure occurred. 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 SQL statement, and the DONE token is sent with the error bit set.

2.2.7.12 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
```

Token Stream Definition

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

Token Stream Parameter Details

Token stream parameter details are described in the following table.

Parameter	Description
TokenType	INFO_TOKEN
Length	The total length of the INFO data stream, in bytes.
Number	The info number.<11>
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.
LineNumber	The line number in the SQL 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 is 0.

2.2.7.13 LOGINACK

Token Stream Name

```
LOGINACK
```

Token Stream Function

Used to send a response to a login request 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
VersionMark      =   BYTE
MajorVer         =   BYTE
MinorVer         =   BYTE
BuildNum       =   BYTE
ProgVersion      =   VersionMark
                     MajorVer
                     MinorVer
                     BuildNum
```

Token Stream Definition

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

Token Stream Parameter Details

Token stream parameter details are described in the following table.

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 (The server confirms that whatever is sent by the client is acceptable). 1: LXSQL (T-SQL is accepted).
TDSVersion	The TDS 4.2 version being used by the server. This value is sent as big-endian and MUST be 0x04020000.
ProgName	The name of the server software (for example, "SQL Server").
VersionMark	Always set to 95.
MajorVer	The major version number (0-255).
MinorVer	The minor version number (0-255).
BuildNum	The build number (0-255). If the build number is greater than 255, the server SHOULD send 255.

2.2.7.14 OFFSET

Token Stream Name

```
OFFSET
```

Token Stream Function

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

Token Stream Comments

- The token value is 0x78.

Token Stream-Specific Rules:

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

Token Stream Definition

```
OFFSET           =   TokenType
                     Identifier
                     OffSetLen
```

Token Stream Parameter Details

Token stream parameter details are described in the following table.

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

2.2.7.15 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           =   *BYTE
```

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

Token stream parameter details are described in the following table.

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

2.2.7.16 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 stored procedure executed through SQL Batch.

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

Token stream parameter details are described in the following table.

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

2.2.7.17 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 user-defined function (UDF) when executed as an RPC.

Token Stream Comments

- The token value is 0xAC.
- Multiple return values may exist according to the 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
Length     =   USHORT
Status           =   BYTE
UserType         =   USHORT
fNullable        =   BIT
fCaseSen         =   BIT
usUpdateable     =   2BIT           ; 0 = ReadOnly
                                    ; 1 = Read/Write
                                    ; 2 = Unused
fIdentity        =   BIT
usReservedODBC   =   2BIT
Flags            =   fNullable
                     fCaseSen
                     usUpdateable
```

```
                    fIdentity
                    FRESERVEDBIT
                    usReservedODBC
                    8FRESERVEDBIT
TypeInfo        =   TYPE_INFO
Value           =   TYPE_VARBYTE
```

Token Stream Definition

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

Token Stream Parameter Details

Token stream parameter details are described in the following table.

Parameter	Description
TokenType	RETURNVALUE_TOKEN
Length	The number of bytes in the token stream, excluding the **TokenType** and **Length** fields.
ParamName	The parameter name length and parameter name (within B_VARCHAR).
Status	0x01: If ReturnValue corresponds to the OUTPUT parameter of a stored procedure invocation. 0x02: If ReturnValue corresponds to the return value of the UDF.
UserType	The user type ID of the column's data type.
Flags	These bit flags are described in least significant bit order. All of these bit flags SHOULD be set to zero. For a description of each bit flag, see section 2.2.7.5. ▪ fCaseSen ▪ fNullable ▪ usUpdateable<12> ▪ fIdentity ▪ usReservedODBC
TypeInfo	The TYPE_INFO for the message.
Value	The type-dependent data for the parameter (within TYPE_VARBYTE).

2.2.7.18 ROW

Token Stream Name

```
ROW
```

Token Stream Function

Used to send a complete row, as defined by the COLNAME and COLFMT tokens, 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    =   1*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/image is a NULL instance (GEN_NULL).

Token Stream Definition

```
ROW              =   TokenType
                     AllColumnData
```

Token Stream Parameter Details

Token stream parameter details are described in the following table.

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 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 COLFMT_TOKEN, ALTFMT_TOKEN, or OFFSET_TOKEN.

2.2.7.19 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

Token stream parameter details are described in the following table.

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

2.2.7.20 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
TableName        =   B_VARCHAR
AllTableNames    =   1*TableName
```

The **TableName** element is repeated once for each table name in the SQL statement.

Token Stream Definition

```
TABNAME          =   TokenType
                     Length
```

```
AllTableNames
```

Token Stream Parameter Details

Token stream parameter details are described in the following table.

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

2.3 Directory Service Schema Elements

None.

3 Protocol Details

This section describes the important elements of the client software and the server software necessary to support the TDS 4.2 protocol.

3.1 Common Details

As described in section 1.3, TDS 4.2 is an application-level protocol that is used for the transfer of requests and responses between clients and database server systems. Messages sent by clients or servers must be limited to the set of messages defined in this protocol.

The TDS 4.2 server is message-oriented. After a connection has been established between the client and server, a complete message is sent from the client to the server. Following this, a complete response is sent from the server to the client (with the possible exception of when the client aborts the request), and then the server waits for the next request.

Other than this Post-Login state, the states defined by the TDS 4.2 protocol are as follows: (1) pre-authentication (Pre-Login), (2) authentication (Login), and (3) when the client sends an attention message (Attention). These are described in subsequent sections.

3.1.1 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 what is described in this document.

For information about the abstract data model for the client, see section 3.2.1. For information about the abstract data model for the server, see section 3.3.1.

3.1.2 Timers

For a description of the client timer used, see section 3.2.2. For a description of the server timer used, see section 3.3.2.

3.1.3 Initialization

None.

3.1.4 Higher-Layer Triggered Events

For information about higher-layer triggered events for the client, see section 3.2.4. For information about higher-layer triggered events for the server, see section 3.3.4.

3.1.5 Message Processing Events and Sequencing Rules

The following sequence diagrams illustrate the possible message exchange sequences between client and server. For details about message processing events and sequencing rules for the client, see section 3.2.5. For details about message processing events and sequencing rules for the server, see section 3.3.5.

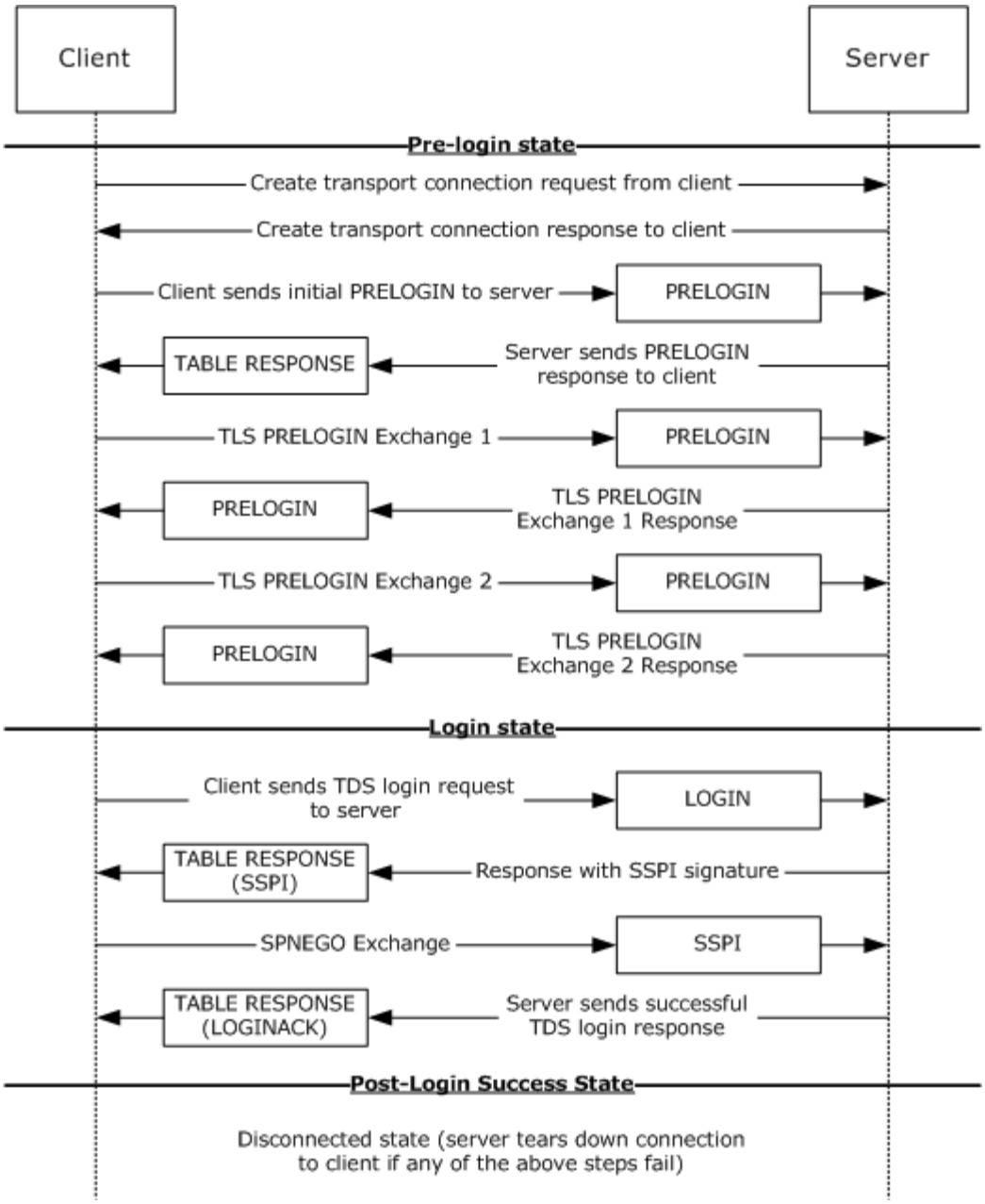

Figure 3: Pre-Login to Post-Login sequence

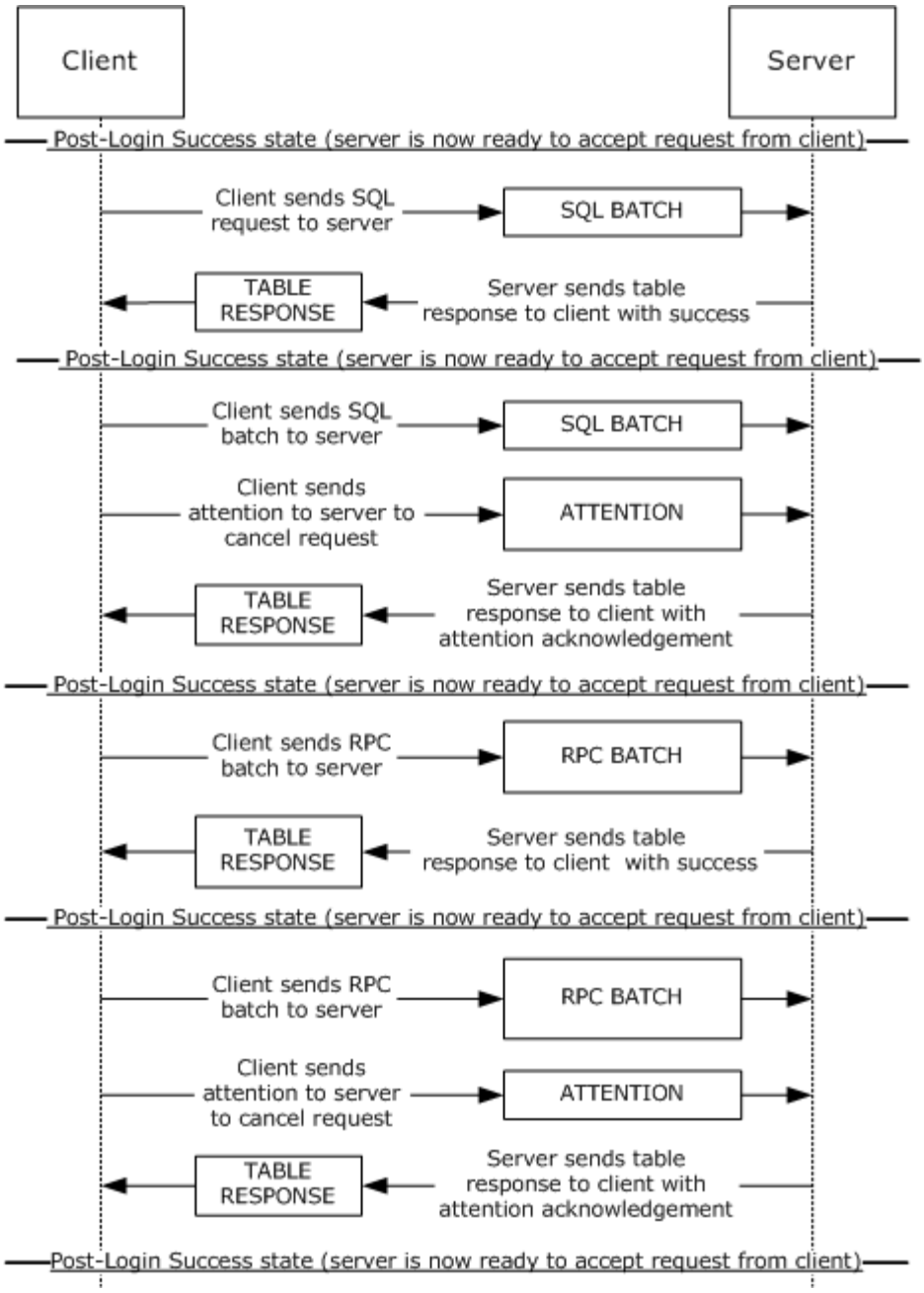

Figure 4: SQL batch and RPC sequence

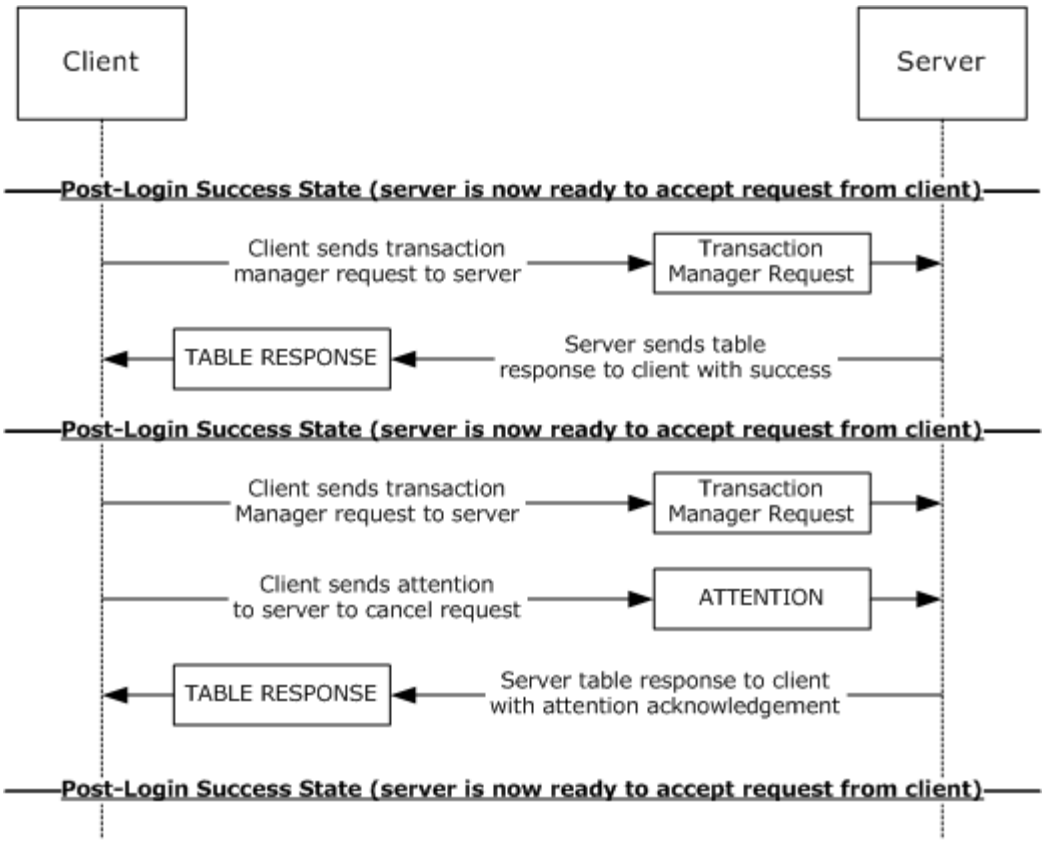

Figure 5: Transaction manager request sequence

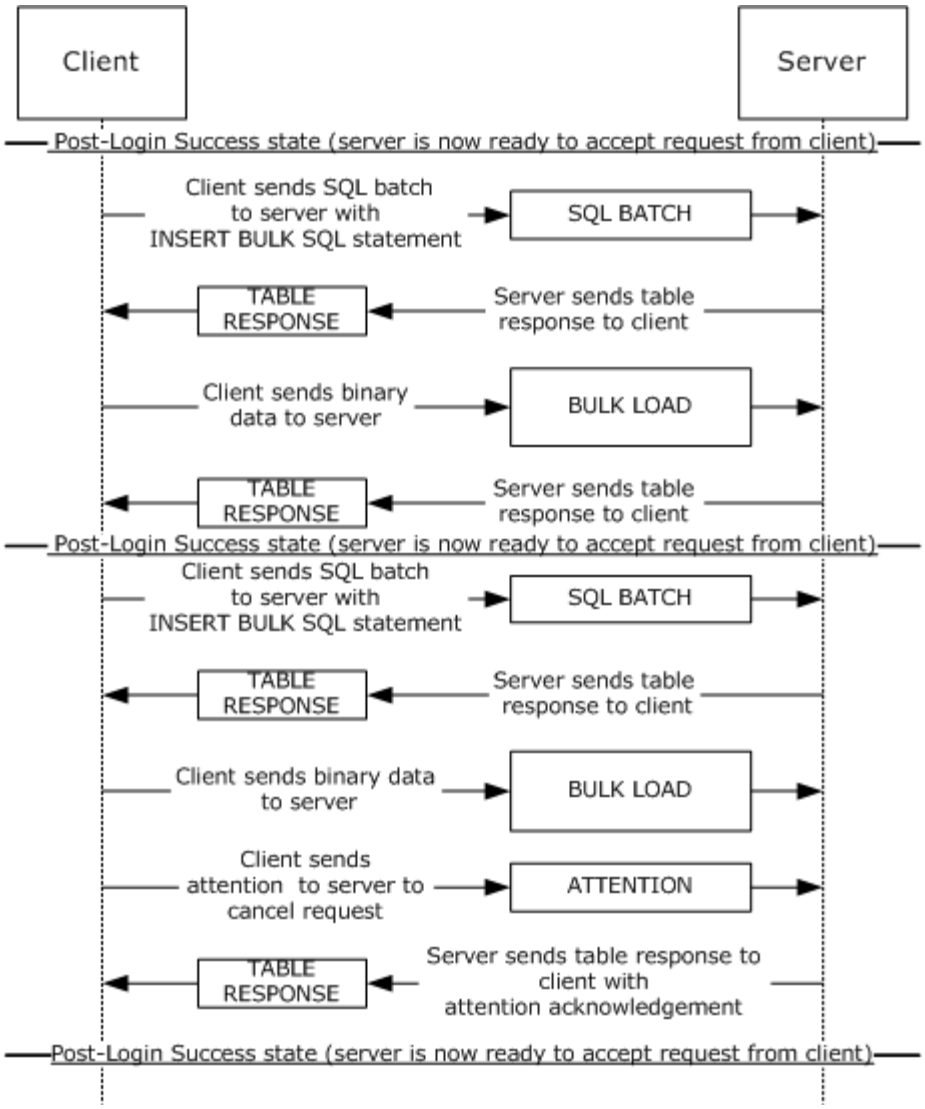

Figure 6: Bulk insert sequence

3.1.6 Timer Events

For the timer events of the client, see section 3.2.6. For the timer events of the server, see section 3.3.6.

3.1.7 Other Local Events

A TDS 4.2 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 4.2 session.

For the other local events of the client, see section 3.2.7. For other local events of the server, see section 3.3.7.

3.2 Client Details

The following state machine diagram describes TDS 4.2 on the client side.

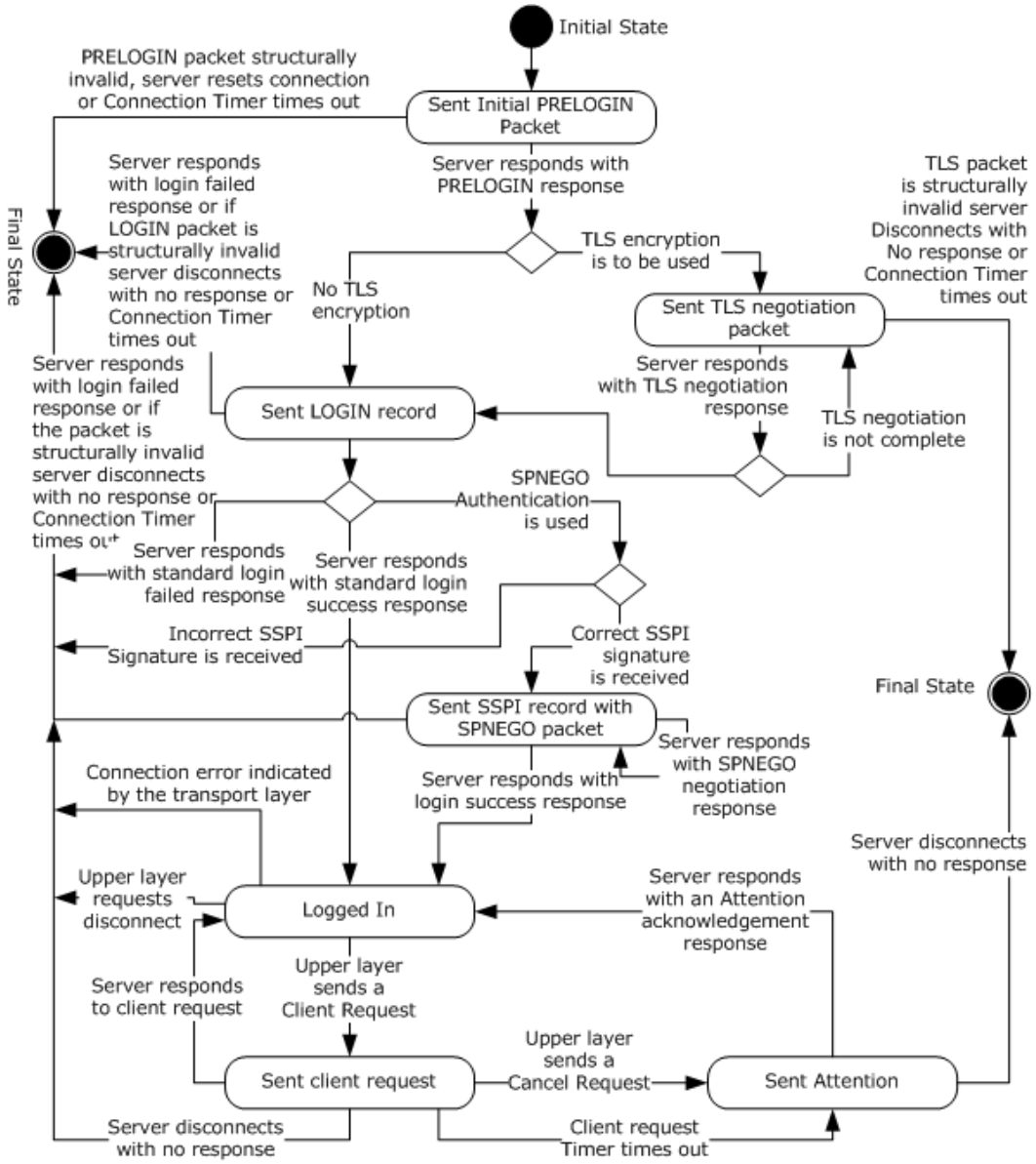

Figure 7: TDS 4.2 client state machine

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 what is described in this document.

A TDS 4.2 client SHOULD maintain the following data:

- **Encryption option**. Possible values are no encryption, login-only encryption, and full encryption. For more details, see the "Encryption" section of PRELOGIN.
- **Authentication scheme**. Possible values are standard authentication and SSPI authentication. For more details, see the "Security and Authentication Methods" section of section 1.7.
- **Connection time-out value**. For more details, see Timers.
- **Client request time-out value**. For more details, see **Timers**.
- **Cancel time-out value**. For more details, see **Timers**.
- **Transaction descriptor value**. For more details, see section 2.2.6.8

3.2.2 Timers

A TDS 4.2 client SHOULD implement the following timers:

- Connection Timer. Controls the maximum time spent during the establishment of a TDS 4.2 connection. The default Connection time-out 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 message response from the server for a client request sent after the connection has been established. The TDS 4.2 protocol does not impose any restriction on the Client request time-out value. The implementation SHOULD allow the upper layer to specify a nondefault value, including an infinite value (for example, no time-out).
- Cancel Timer. Controls the maximum time spent waiting for a message cancellation acknowledgement after an Attention request is sent to the server. The TDS 4.2 protocol does not impose any restriction on the Cancel time-out value. The implementation SHOULD allow the upper layer to specify a nondefault value, including an infinite value (for example, no time-out).

If a TDS 4.2 client implementation implements any of the timers, it MUST implement their behavior according to this specification.

A TDS 4.2 client SHOULD request the transport to detect and indicate a broken connection if the transport provides this mechanism. If the transport used is TCP, it SHOULD use the TCP keep-alives (for more details, see [RFC1122]) in order to detect a nonresponding server if 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 4.2 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 4.2 client MUST support the following events from the upper layer:

- Connection Open Request to establish a new TDS 4.2 connection to a TDS 4.2 server.

- Client Request to send a request to a TDS 4.2 server on an already established TDS 4.2 connection. The Client Request is a message of one of the following four types: SQL Batch, Bulk Load, 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 client and server resources. If a client implementation of the TDS 4.2 protocol supports the Cancel Request event, it MUST handle it as described in this specification.

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

When a TDS 4.2 client receives a Connection Open Request from the upper layer in the **initial state** of a TDS 4.2 connection, it MUST perform the following actions:

- If the TDS 4.2 client implements the Connection Timer, it MUST start the Connection Timer if the connection time-out value is not infinite.
- Send a Pre-Login message to the server, by using the underlying transport protocol.
- If the transport does not report an error, then enter the Sent Initial Pre-Login Message state.

When a TDS 4.2 client receives a Connection Open Request from the upper layer in any state other than the initial state of a TDS 4.2 connection, it MUST indicate an error to the upper layer.

When a TDS 4.2 client receives a Client Request from the upper layer in the Logged In state it MUST perform the following actions:

- If the TDS 4.2 client implements the Client Request Timer, it MUST start the Client Request Timer if the client request time-out value is not infinite.
- Send either the SQL Batch, Bulk Load, transaction manager request, or RPC message to the server. The message and its content MUST match the requested message from the Client Request.
- If the transport does not report an error, then enter the Sent Client Request state.

When a TDS 4.2 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 4.2 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.
- Enter the Sent Attention state.

3.2.5 Message Processing Events and Sequencing Rules

The processing of messages received from a TDS 4.2 server depends on the message type and the state the TDS 4.2 client is in. The message type is determined from the TDS 4.2 packet type and the token stream inside the TDS 4.2 packet payload, as described in section 2.2.3. The rest of this section describes message processing and actions that can be taken on messages.

When the TDS 4.2 client enters either the Logged In state or the **final 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).

When a TDS 4.2 client receives a structurally invalid TDS 4.2 message, it MUST close the underlying transport connection, indicate an error to the upper layer, and enter the final state.

When a TDS 4.2 client receives a table response (TDS 4.2 packet type %x04) from the server, it MUST behave as follows, according to the state of the TDS 4.2 client.

3.2.5.1 Sent Initial PRELOGIN Packet State

If the response contains a structurally valid PRELOGIN response indicating a success, the TDS 4.2 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 4.2 client MUST initiate a TLS/SSL handshake, send to the server a TLS/SSL message obtained from the TLS/SSL layer encapsulated in TDS 4.2 packets 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 4.2 client MUST send to the server a LOGIN message that includes either standard login and password or indicates that integrated authentication SHOULD be used, and enter the Sent LOGIN record state. The TDS 4.2 specification does not prescribe the authentication protocol if SSPI authentication is used. The current implementation supports NTLM (for more information, see [NTLM]) and Kerberos (for more information, see [RFC4120]).
- If encryption was not negotiated and the upper layer requested full encryption, then the TDS 4.2 client MUST close the underlying transport connection, indicate an error to the upper layer, and enter the final 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 4.2 client MUST close the underlying transport connection, indicate an error to the upper layer, and enter the final state.

3.2.5.2 Sent TLS/SSL Negotiation Packet State

If the response contains a structurally valid TLS/SSL response message (TDS 4.2 packet Type 0x12), the TDS 4.2 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 handshaking is needed, the TDS 4.2 client MUST send to the server the TLS/SSL message obtained from the TLS/SSL layer encapsulated in TDS 4.2 packets of Type PRELOGIN (0x12).
- If the TLS/SSL layer indicates successful completion of the TLS/SSL handshake, the TDS 4.2 client MUST send a login message to the server and enter the Sent LOGIN record state.
- If login-only encryption was negotiated in the Pre-Login message description as described in section 2.2, the first, and only the first, TDS 4.2 packet of the login message MUST be encrypted using TLS/SSL and encapsulated in a TLS/SSL message. All other TDS 4.2 packets sent or received MUST be in plaintext.

- If full encryption was negotiated as described in the Pre-Login message description in section 2.2.6.4, all subsequent TDS 4.2 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 4.2 client MUST close the underlying transport connection, indicate an error to the upper layer, and enter the final 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 4.2 client MUST close the underlying transport connection, indicate an error to the upper layer, and enter the final state.

3.2.5.3 Sent LOGIN Record State

If standard login is used and the response received from the server contains a structurally valid login response indicating a successful login, the TDS 4.2 client MUST indicate successful login completion to the upper layer and enter the Logged In state.

If SPNEGO authentication is used and the response received from the server contains a correct SSPI token (that is, the SSPI signature in the token matches the local value of the client), the TDS 4.2 client MUST send an SSPI message (TDS 4.2 packet type %x11) containing the initial data obtained from the applicable SSPI layer and enter the Sent SSPI Record with SPNEGO Packet state. The TDS 4.2 specification does not prescribe the authentication protocol if SSPI authentication is used. The current implementation supports NTLM (for more information, see [NTLM]) and Kerberos (for more information, see [RFC4120]).

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, or the SSPI signature received from server in the SSPI token does not match the TDS 4.2 client's local copy of the SSPI signature when SPNEGO authentication is used, the TDS 4.2 client MUST close the underlying transport connection, indicate an error to the upper layer, and enter the final state.

3.2.5.4 Sent SSPI Record with SPNEGO Packet State

If the response received from the server contains a structurally valid login response indicating a successful login, the TDS 4.2 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 4.2 client MUST send to the server an SSPI message (TDS 4.2 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 4.2 client MUST close the underlying transport connection, indicate an error to the upper layer, and enter the final state.

3.2.5.5 Logged In State

The TDS 4.2 client waits for notification from the upper layer. If the upper layer requests a message to be sent to the server, the TDS 4.2 client MUST send the appropriate request to the server and enter the Sent Client Request state. If the upper layer requests a termination of the connection, the TDS 4.2 client MUST disconnect from the server and enter the final state. If the TDS 4.2 client detects a connection error from the transport layer, the TDS 4.2 client MUST disconnect from the server and enter the final state.

3.2.5.6 Sent Client Request State

If the response received from the server contains a structurally valid response, the TDS 4.2 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 4.2 client supports Cancel Request, and the upper layer requests a Cancel Request to be sent to the server, the TDS 4.2 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 4.2 client MUST close the underlying transport connection, indicate an error to the upper layer, and enter the final 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, the TDS 4.2 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, the TDS 4.2 client MUST discard any data contained in the response, indicate the completion of the message to the upper layer together with the cause of the Attention (either an upper-layer cancellation as described in section 3.2.4 or a message 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 4.2 client MUST close the underlying transport connection, indicate an error to the upper layer, and enter the final state.

3.2.5.8 Final State

The connection is disconnected. All resources for this connection will be recycled by the TDS 4.2 server.

3.2.6 Timer Events

If a TDS 4.2 client implements the Connection Timer and the timer times out, the TDS 4.2 client MUST close the underlying connection, indicate the error to the upper layer, and enter the final state.

If a TDS 4.2 client implements the Client Request Timer and the timer times out, the TDS 4.2 client MUST send an Attention message to the server and enter the Sent Attention state.

If a TDS 4.2 client implements the Cancel Timer and the timer times out, the TDS 4.2 client MUST close the underlying connection, indicate the error to the upper layer, and enter the final state.

3.2.7 Other Local Events

Whenever an indication of a connection error is received from the underlying transport, the TDS 4.2 client MUST close the transport connection, indicate an error to the upper layer, stop any timers if they are running, and enter the final state. If TCP is used as the underlying transport, examples of events that may trigger such action—depending 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 4.2 on the server side.

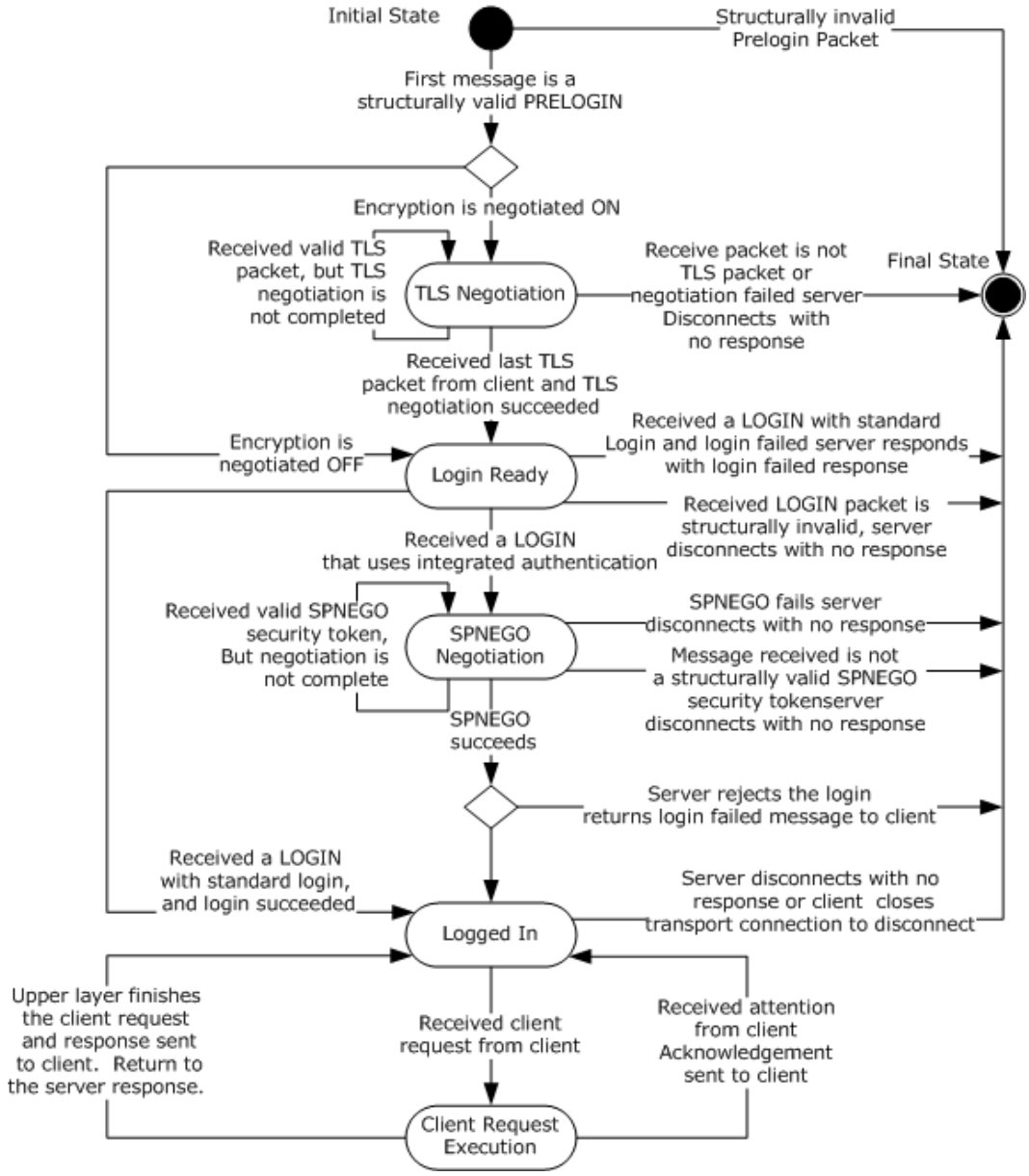

Figure 8: TDS 4.2 server state machine

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 4.2 protocol does not regulate any timer on a data stream. The TDS 4.2 server may implement a timer on any message.

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 initializes all resources required for this connection. The server is ready to receive a pre-login message.

3.3.4 Higher-Layer Triggered Events

A higher layer SHOULD terminate a TDS 4.2 connection when it needs to. When this happens, the server MUST terminate the connection and recycle all resources for this connection. No response is sent to the client.

3.3.5 Message Processing Events and Sequencing Rules

The processing of messages received from a TDS 4.2 client depends on the message type and the state the TDS 4.2 server is in. The message type is determined from the TDS 4.2 packet type and the token stream inside the TDS 4.2 packet payload, as described in section 2.2. The rest of this section describes message processing and the possible actions that can be taken on messages.

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

3.3.5.1 Initial State

The TDS 4.2 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. The TDS 4.2 server SHOULD close the underlying transport connection, indicate an error to the upper layer, and enter the final state if the first packet is not a structurally correct PRELOGIN packet. For instance, the PRELOGIN packet will not contain the client version as the first option token. Otherwise, the TDS 4.2 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 the 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 the unencrypted Login Ready state if encryption is not negotiated.

3.3.5.2 TLS/SSL Negotiation

If the next packet from the TDS 4.2 client is not a TLS/SSL negotiation packet or if the packet is not structurally correct, the TDS 4.2 server MUST close the underlying transport connection, indicate an error to the upper layer, and then enter the final state. A TLS/SSL negotiation packet is a PRELOGIN (0x12) packet header encapsulated with TLS/SSL payload. The TDS 4.2 server MUST exchange a TLS/SSL negotiation packet with the client and reenter this state until the TLS/SSL negotiation is successfully completed. Upon successful negotiation, the TDS 4.2 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 LOGIN packet with standard login is received, the TDS 4.2 server MUST respond to the TDS 4.2 client with a LOGINACK (0xAD), indicating that the login succeeded. The TDS 4.2 server MUST enter the Logged In state.
- If a valid LOGIN packet is received and integrated authentication is required by the TDS 4.2 client, the TDS 4.2 server MUST respond with an SSPI message containing the SSPI signature of the TDS 4.2 client and enter the SPNEGO Negotiation state.
- If a LOGIN packet with a standard login packet is received, but the login is invalid, the TDS 4.2 server MUST send an ERROR packet to the client. The TDS 4.2 server MUST close the underlying transport connection, indicate an error to the upper layer, and enter the final state.
- If the packet received is not a structurally valid LOGIN packet, the TDS 4.2 server will not send any response to the client. The TDS 4.2 server MUST close the underlying transport connection, indicate an error to the upper layer, and enter the final state.

3.3.5.4 SPNEGO Negotiation

This state is used to negotiate the security scheme between the client and server. The TDS 4.2 server processes the packet received according to the following rules:

- If the packet received is a structurally valid SPNEGO negotiation packet, the TDS 4.2 server delegates processing of the security token embedded in the packet to the SPNEGO layer. (For more information about SPNEGO, see [RFC4178].) The SPNEGO layer responds with one of three results, and the TDS 4.2 server continues processing according to the response as follows:
  - **Complete**: The TDS 4.2 server then sends the security token to the upper layer (typically a database server) for authorization. If the upper layer approves the security token, the TDS 4.2 server sends a LOGINACK message to the client 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 4.2 server closes the connection, and the TDS 4.2 server enters the final state.
  - **Continue**: The TDS 4.2 server sends a SPNEGO negotiation response to the client, embedding the new security token returned by SPNEGO as part of the Continue response. (For more information about SPNEGO, see [RFC4178].) The server then waits for a message from the client and reenters 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.
- If the packet received is not a structurally valid SPNEGO negotiation packet, the TDS 4.2 server will send no response to the client. The TDS 4.2 server MUST close the underlying transport connection, indicate an error to the upper layer, and enter the final state. (For more information about SPNEGO, see [RFC4178].)

3.3.5.5 Logged In

If a TDS 4.2 message of type 1, 3, 7, or 14 (see section 2.2.3.1.1) arrives, the TDS 4.2 server begins processing by raising an event to the upper layer containing the data of the client request and by entering the Client Request Execution state. If any other TDS 4.2 types arrive, the server MUST close the underlying transport connection and enter the final state.

The server MUST also enter the final state if the client closes the underlying transport connection or if the upper layer requests the TDS layer to close the connection. In this case, no response is sent to the client.

3.3.5.6 Client Request Execution

The TDS 4.2 server MUST continue to listen for messages from the client while awaiting notification of client request for completion from the upper layer. The TDS 4.2 server MUST also do one of the following:

- If the upper layer notifies the TDS 4.2 server that the client request has finished successfully, the TDS 4.2 server MUST send the results to the TDS 4.2 client and enter the Logged In state.
- If the upper layer notifies TDS 4.2 that an error has been encountered during the client request, the TDS 4.2 server MUST send an ERROR message to the TDS 4.2 client and enter the Logged In state.
- If an Attention packet is received during the execution of the current client request, it MUST deliver a cancel indication to the upper layer. If an Attention packet is received after the execution of the current client request, it MUST NOT deliver a cancel indication to the upper layer, because there is no existing execution to cancel. Instead, the TDS 4.2 server MUST send an attention acknowledgment to the TDS 4.2 client and enter the Logged In state.
- If another client request packet is received during the execution of the current client request, the TDS 4.2 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 4.2 server reenters the Client Request Execution state and processes the newly arrived message.

3.3.5.7 Final State

The connection is disconnected. All resources for this connection are recycled by the TDS 4.2 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 4.2 session without sending any response to the client. An 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 4.2 protocol. For each example, the binary TDS 4.2 message is provided, followed by the decomposition displayed in XML.

4.1 Pre-Login Request

The following is an example of the pre-login request that is sent from the client to the server.

```
12 01 00 34 00 00 01 00 00 00 15 00 06 01 00 1B
00 01 02 00 1C 00 0C 03 00 28 00 04 FF 08 00 01
55 00 00 00 4D 53 53 51 4C 53 65 72 76 65 72 00
80 19 00 00
```

```
<PacketHeader>
    <Type>
      <BYTE>12 </BYTE>
    </Type>
    <Status>
      <BYTE>01 </BYTE>
    </Status>
    <Length>
      <BYTE>00 </BYTE>
      <BYTE>34 </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 15</USHORT>
      </TokenPosition>
      <TokenLeng>
         <USHORT>00 06</USHORT>
      </TokenLeng>
      <TokenType>
         <BYTE>01 </BYTE>
      </TokenType>
      <TokenPosition>
         <USHORT>00 1B</USHORT>
      </TokenPosition>
      <TokenLeng>
         <USHORT>00 01</USHORT>
      </TokenLeng>
```

```
      <TokenType>
         <BYTE>02 </BYTE>
      </TokenType>
      <TokenPosition>
         <USHORT>00 1C</USHORT>
      </TokenPosition>
      <TokenLeng>
         <USHORT>00 0C</USHORT>
      </TokenLeng>
      <TokenType>
         <BYTE>03 </BYTE>
      </TokenType>
      <TokenPosition>
         <USHORT>00 28</USHORT>
      </TokenPosition>
      <TokenLeng>
         <USHORT>00 04</USHORT>
      </TokenLeng>
      <TokenType>
         <BYTE>FF </BYTE>
      </TokenType>
      <PreloginData>
         <BYTES>08 00 01 55 00 00 00 4D 53 53 51 4C 53 65 72
76 65 72 00 80 19 00 00</BYTE>
      </PreloginData>
    </Prelogin>
  </PacketData>
```

4.2 Login Request

The following is an example of the login request that is sent from the client to the server in two packets.

The following information is included in the first packet.

```
02 00 02 00 00 00 01 00 53 51 4C 50 4F 44 30 36
38 2D 30 35 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 0C 73 61 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 02 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 08 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 03 01 06 0A 09 01 01 00 00 00 00 00
00 00 00 00 4F 53 51 4C 2D 33 32 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 07 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

```
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 04 02 00 00 4D 53 44 42 4C 49 42 00 00 00
07 06 00 00 00 00 0D 11 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

The following information is included in the second packet.

```
02 01 00 47 00 00 01 00 00 00 00 00 00 00 00 01
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 35 31 32
00 00 00 03 00 00 00

  <PacketHeader>
    <Type>
      <BYTE>02 </BYTE>
    </Type>
    <Status>
      <BYTE>01 </BYTE>
    </Status>
    <Length>
      <BYTE>02 </BYTE>
      <BYTE>3F </BYTE>
    </Length>
    <SPID>
      <BYTE>00 </BYTE>
      <BYTE>00 </BYTE>
    </SPID>
    <Packet>
      <BYTE>01 </BYTE>
    </Packet>
    <Window>
      <BYTE>00 </BYTE>
    </Window>
  </PacketHeader>
  <PacketData>
    <Login>
      <HostName>
        <BYTES>53 51 4C 50 4F 44 30 36 38 2D 30 35 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 </BYTES>
      </HostName>
        <cbHostName>
          <BYTE>0C </BYTE>
        </cbHostName>
      <UserName>
        <BYTES>73 61 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 </BYTES>
      </UserName>
        <cbUserName>
          <BYTE>02 </BYTE>
```

```
        </cbUserName>
      <Password>
        <BYTES>59 75 6B 6F 6E 39 30 30 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 </BYTES>
      </Password>
        <cbPassword>
          <BYTE>08 </BYTE>
        </cbPassword>
      <HostProc>
        <BYTES>00 00 00 00 00 00 00 00 </BYTES>
      </HostProc>
        <FRESERVEDBYTE>
          <BYTES>00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 </BYTES>
        </FRESERVEDBYTE>
        <AppType>
          <BYTES>00 00 00 00 00 00 </BYTES>
        </AppType>
        <cbHostProc>
          <BYTE>00 </BYTE>
        </cbHostProc>
      <lInt2>
        <BYTE>03 </BYTE>
      </lInt2>
      <lInt4>
        <BYTE>01 </BYTE>
      </lInt4>
      <lChar>
        <BYTE>06 </BYTE>
      </lChar>
      <lFloat>
        <BYTE>0A </BYTE>
      </lfloat>
      <FRESERVEDBYTE>
        <BYTE>09 </BYTE>
      </FRESERVEDBYTE>
      <lUseDb>
        <BYTE>01 </BYTE>
      </lUseDb>
      <lDumpLoad>
        <BYTE>01 </BYTE>
      </lDumpLoad>
      <lInterface>
        <BYTE>00 </BYTE>
      </lInterface>
      <lType>
        <BYTE>00 </BYTE>
      </lType>
      <FRESERVEDBYTE>
        <BYTES>00 00 00 00 00 00 </BYTES>
      </FRESERVEDBYTE>
      <lDBLIBFlags>
        <BYTE>00 </BYTE>
      </lDBLIBFlags>
      <AppName>
        <BYTES>4F 53 51 4C 2D 33 32 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 </BYTES>
      </AppName>
        <cbAppName>
```

```
          <BYTE>07 </BYTE>
        </cbAppName>
      <ServerName>
        <BYTES>00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 </BYTES>
      </ServerName>
        <cbServerName>
          <BYTE>00 </BYTE>
        </cbServerName>
      <RemotePassword>
        <BYTES>00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 </BYTES>
      </RemotePassword>
        <cbRemotePassword>
          <BYTE>00 </BYTE>
        </cbRemotePassword>
      <TDSVersion>
        <BYTES>04 02 00 00 </BYTES>
      </TDSVersion>
      <ProgName>
        <BYTES>4D 53 44 42 4C 49 42 00 00 00 </BYTES>
      </ProgName>
        <cbProgName>
          <BYTE>07 </BYTE>
        </cbProgName>
      <ProgVersion>
        <BYTES>06 00 00 00 </BYTES>
      </ProgVersion>
      <FRESERVEDBYTE>
        <BYTE>00 </BYTE>
      </FRESERVEDBYTE>
      <lFloat4>
        <BYTE>0D </BYTE>
      </lFloat4>
      <lDate4>
        <BYTE>11 </BYTE>
      </lDate4>
      <Language>
        <BYTES>00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 </BYTES>
        <cbLanguage>
          <BYTE>00 </BYTE>
        </cbLanguage>
      </Language>
      <SetLang>
        <BYTE>01 </BYTE>
      </SetLang>
      <FRESERVEDBYTES>
```

```
        <BYTES>00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  00 00 00 00 00 00 00 00 00 00 </BYTES>
      </FRESERVEDBYTES>
      <PacketSize>
        <BYTES>35 31 32 00 00 00 </BYTES>
      </PacketSize>
        <cbPacketSize>
          <BYTE>03 </BYTE>
        </cbPacketSize>
      <Padding>
        <BYTES>00 00 00 </BYTES>
      </Padding>
    </Login>
  </PacketData>
```

4.3 Login Response

The following is an example of the login response that is sent from the server to the client.

```
04 01 00 E8 00 34 01 00 E3 0F 00 01 06 6D 61 73
74 65 72 06 6D 61 73 74 65 72 AB 39 00 45 16 00
00 02 00 25 00 43 68 61 6E 67 65 64 20 64 61 74
61 62 61 73 65 20 63 6F 6E 74 65 78 74 20 74 6F
20 27 6D 61 73 74 65 72 27 2E 08 41 42 43 44 45
46 47 31 00 01 00 E3 0D 00 02 0A 75 73 5F 65 6E
67 6C 69 73 68 00 AB 3B 00 47 16 00 00 01 00 27
00 43 68 61 6E 67 65 64 20 6C 61 6E 67 75 61 67
65 20 73 65 74 74 69 6E 67 20 74 6F 20 75 73 5F
65 6E 67 6C 69 73 68 2E 08 41 42 43 44 45 46 47
31 00 01 00 E3 09 00 03 05 69 73 6F 5F 31 01 00
AD 20 00 01 04 02 00 00 16 4D 69 63 72 6F 73 6F
66 74 20 53 51 4C 20 53 65 72 76 65 72 00 00 5F
0A 00 FF E3 09 00 04 03 35 31 32 03 35 31 32 FD
```

```
  <PacketHeader >
    <Type>
      <BYTE>04 </BYTE>
    </Type>
    <Status>
      <BYTE>01 </BYTE>
    </Status>
    <Length>
      <BYTE>00 </BYTE>
      <BYTE>E8 </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>0F 00 </USHORT>
        </Length>
        <EnvChangeData>
          <BYTES>01 06 6D 61 73 74 65 72 06 6D 61 73 74
65 72 </BYTES>
        </EnvChangeData>
      </ENVCHANGE>
      <INFO>
        <TokenType>
          <BYTE>AB </BYTE>
        </TokenType>
        <Length>
          <USHORT>39 00 </USHORT>
        </Length>
        <Number>
          <LONG>45 16 00 00 </LONG>
        </Number>
        <State>
          <BYTE>02 </BYTE>
        </State>
        <Class>
          <BYTE>00 </BYTE>
        </Class>
        <MsgText>
          <US_VARCHAR>
            <USHORT>25 00 </USHORT>
            <BYTES ascii="Changed database context to
'master'.">43 68 61 6E 67 65 64 20 64 61 74 61 62 61 73
65 20 63 6F 6E 74 65 78 74 20 74 6F 20 27 6D 61 73 74 65
72 27 2E </BYTES>
          </US_VARCHAR>
        </MsgText>
        <ServerName>
          <B_VARCHAR>
            <BYTE>08 </BYTE>
            <BYTES ascii="ABCDEFG1">41 42 43 44 45 46
47 31 </BYTES>
          </B_VARCHAR>
        </ServerName>
        <ProcName>
          <B_VARCHAR>
            <BYTE>00 </BYTE>
            <BYTES ascii="">
            </BYTES>
          </B_VARCHAR>
        </ProcName>
        <LineNumber>
          <USHORT>01 00 </USHORT>
        </LineNumber>
      </INFO>
      <ENVCHANGE>
```

```
          <TokenType>
            <BYTE>E3 </BYTE>
          </TokenType>
          <Length>
            <USHORT>0D 00 </USHORT>
          </Length>
          <EnvChangeData>
            <BYTES>02 0A 75 73 5F 65 6E 67 6C 69 73 68 00 </BYTES>
          </EnvChangeData>
        </ENVCHANGE>
        <INFO>
          <TokenType>
            <BYTE>AB </BYTE>
          </TokenType>
          <Length>
            <USHORT>3B 00 </USHORT>
          </Length>
          <Number>
            <LONG>47 16 00 00 </LONG>
          </Number>
          <State>
            <BYTE>01 </BYTE>
          </State>
          <Class>
            <BYTE>00 </BYTE>
          </Class>
          <MsgText>
            <US_VARCHAR>
              <USHORT>27 00 </USHORT>
              <BYTES ascii="Changed language setting to
  us_english.">43 68 61 6E 67 65 64 20 6C 61 6E 67 75 61
  67 65 20 73 65 74 74 69 6E 67 20 74 6F 20 75 73 5F 65 6E
  67 6C 69 73 68 2E </BYTES>
            </US_VARCHAR>
          </MsgText>
          <ServerName>
            <B_VARCHAR>
              <BYTE>08 </BYTE>
              <BYTES ascii="ABCDEFG1">41 42 43 44 45 46
  47 31 </BYTES>
            </B_VARCHAR>
          </ServerName>
          <ProcName>
            <B_VARCHAR>
              <BYTE>00 </BYTE>
              <BYTES ascii="">
              </BYTES>
            </B_VARCHAR>
          </ProcName>
          <LineNumber>
            <USHORT>01 00 </USHORT>
          </LineNumber>
        </INFO>
        <ENVCHANGE>
          <TokenType>
            <BYTE>E3 </BYTE>
          </TokenType>
          <Length>
            <USHORT>09 00 </USHORT>
```

```
          </Length>
          <EnvChangeData>
            <BYTES>03 05 69 73 6F 5F 31 01 00 </BYTES>
          </EnvChangeData>
        </ENVCHANGE>
        <LOGINACK>
          <TokenType>
            <BYTE>AD </BYTE>
          </TokenType>
          <Length>
            <USHORT>20 00 </USHORT>
          </Length>
          <Interface>
            <BYTE>01 </BYTE>
          </Interface>
          <TDSVersion>
            <DWORD>04 02 00 00 </DWORD>
          </TDSVersion>
          <ProgName>
            <B_VARCHAR>
              <BYTE>16 </BYTE>
              <BYTES ascii="Microsoft SQL Server..">4D 69 63 72
  6F 73 6F 66 74 20 53 51 4C 20 53 65 72 76 65 72 00 00 </BYTES>
            </B_VARCHAR>
          </ProgName>
          <ProgVersion>
            <DWORD>00 00 00 00 </DWORD>
          </ProgVersion>
        </LOGINACK>
        <ENVCHANGE>
          <TokenType>
            <BYTE>E3 </BYTE>
          </TokenType>
          <Length>
            <USHORT>09 00 </USHORT>
          </Length>
          <EnvChangeData>
            <BYTES>04 03 35 31 32 03 35 31 32 </BYTES>
          </EnvChangeData>
        </ENVCHANGE>
        <DONE>
          <TokenType>
            <BYTE>FD </BYTE>
          </TokenType>
          <Status>
            <USHORT>00 00 </USHORT>
          </Status>
          <CurCmd>
            <USHORT>00 00 </USHORT>
          </CurCmd>
          <DoneRowCount>
            <LONG>00 00 00 00 </LONG>
          </DoneRowCount>
        </DONE>
      </TableResponse>
    </PacketData>
```

4.4 SQL Batch Client Request

The following is an example of the client request that is sent from the client to the server.

```
01 01 00 1E 00 00 01 00 73 65 6C 65 63 74 20 63
6F 6C 31 20 66 72 6F 6D 20 66 6F 6F 0D 0A

  <PacketHeader>
    <Type>
      <BYTE>01 </BYTE>
    </Type>
    <Status>
      <BYTE>01 </BYTE>
    </Status>
    <Length>
      <BYTE>00 </BYTE>
      <BYTE>1E </BYTE>
    </Length>
    <SPID>
      <BYTE>00 </BYTE>
      <BYTE>00 </BYTE>
    </SPID>
    <Packet>
      <BYTE>01 </BYTE>
    </Packet>
    <Window>
      <BYTE>00 </BYTE>
    </Window>
  </PacketHeader>
  <PacketData>
    <SQLBatch>
      <SQLText>
        <BYTESTREAM>
          <BYTES>
73 65 6C 65 63 74 20 63 6F 6C 31 20 66 72 6F 6D 20 66 6F 6F
0D 0A  </BYTES>
        </BYTESTREAM>
      </SQLText>
    </SQLBatch>
  </PacketData>
```

4.5 SQL Batch Server Response

The following is an example of the server response that is sent from the server to the client.

```
04 01 00 26 00 33 01 00 A0 05 00 04 63 6F 6C 31
A1 05 00 07 00 08 00 38 D1 01 00 00 00 FD 10 00
C1 00 01 00 00 00
  <PacketHeader>
    <Type>
      <BYTE>04 </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>
    <TableResponse>
      <COLNAME>
        <TokenType>
          <BYTE>A0 </BYTE>
        </TokenType>
        <Length>
          <USHORT>05 00 </USHORT>
        </Length>
        <ColName>
          <B_VARCHAR>
            <BYTE>04 </BYTE>
            <BYTES ascii="col1">63 6F 6C 31 </BYTES>
          </B_VARCHAR>
        </ColName>
      </COLNAME>
      <COLFMT>
        <TokenType>
          <BYTE>A1 </BYTE>
        </TokenType>
        <Length>
          <USHORT>05 00 </USHORT>
        </Length>
        <ColumnData>
          <UserType>
            <USHORT>07 00 </USHORT>
          </UserType>
          <Flags>
            <USHORT>08 00 </USHORT>
          </Flags>
          <TYPE_INFO>
            <FIXEDLENTYPE>
              <BYTE>38 </BYTE>
            </FIXEDLENTYPE>
          </TYPE_INFO>
        </ColumnData>
      </COLFMT>
      <ROW>
        <TokenType>
          <BYTE>D1 </BYTE>
        </TokenType>
        <TYPE_VARBYTE>
          <BYTES>01 00 00 00 </BYTES>
        </TYPE_VARBYTE>
```

```
      </ROW>
      <DONE>
        <TokenType>
          <BYTE>FD </BYTE>
        </TokenType>
        <Status>
          <USHORT>10 00 </USHORT>
        </Status>
        <CurCmd>
          <USHORT>C1 00 </USHORT>
        </CurCmd>
        <DoneRowCount>
          <LONG>01 00 00 00 </LONG>
        </DoneRowCount>
      </DONE>
    </TableResponse>
  </PacketData>
```

4.6 RPC Client Request

The following is an example of the RPC request that is sent from the client to the server.

```
03 01 00 24 00 00 01 00 0A 70 5F 61 6C 6C 74 79
70 65 73 00 00 0A 40 62 69 67 69 6E 74 63 6F 6C
00 34 01 00
  <PacketHeader>
    <Type>
      <BYTE>03 </BYTE>
    </Type>
    <Status>
      <BYTE>01 </BYTE>
    </Status>
    <Length>
      <BYTE>00 </BYTE>
      <BYTE>24 </BYTE>
    </Length>
    <SPID>
      <BYTE>00 </BYTE>
      <BYTE>00 </BYTE>
    </SPID>
    <Packet>
      <BYTE>01 </BYTE>
    </Packet>
    <Window>
      <BYTE>00 </BYTE>
    </Window>
  </PacketHeader>
  <PacketData>
    <RPCRequest>
      <RPCReqBatch>
        <NameLenProcID>
          <ProcName>
            <B_VARCHAR>
              <BYTE>0A </BYTE>
              <BYTES ascii="p_alltypes">70 5F 61 6C 6C 74 79
70 65 73 </BYTES>
```

```
            </B_VARCHAR>
          </ProcName>
        </NameLenProcID>
        <OptionFlags>
          <fWithRecomp>
            <BIT>false</BIT>
          </fWithRecomp>
          <fNoMetaData>
            <BIT>false</BIT>
          </fNoMetaData>
        </OptionFlags>
        <ParameterData>
          <ParamMetaData>
            <B_VARCHAR>
              <BYTE>0A </BYTE>
              <BYTES ascii="@bigintcol">40 62 69 67 69 6E 74
63 6F 6C </BYTES>
            </B_VARCHAR>
            <StatusFlags>
              <fByRefValue>
                <BIT>false</BIT>
              </fByRefValue>
              <fDefaultValue>
                <BIT>false</BIT>
              </fDefaultValue>
              <fCookie>
                <BIT>false</BIT>
              </fCookie>
            </StatusFlags>
            <TYPE_INFO>
              <FIXEDLENTYPE>
                <BYTE>34 </BYTE>
              </FIXEDLENTYPE>
            </TYPE_INFO>
          </ParamMetaData>
          <ParamLenData>
            <TYPE_VARBYTE>
              <BYTES>01 00 </BYTES>
            </TYPE_VARBYTE>
          </ParamLenData>
        </ParameterData>
      </RPCReqBatch>
    </RPCRequest>
  </PacketData>
```

4.7 RPC Server Response

The following is an example of the RPC response that is sent from the server to the client.

```
04 01 00 1F 00 35 01 00 FF 11 00 C1 00 01 00 00
00 79 00 00 00 00 FE 00 00 E0 00 00 00 00 00 00

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

```
    <Status>
      <BYTE>01 </BYTE>
    </Status>
    <Length>
      <BYTE>00 </BYTE>
      <BYTE>1F </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>
          <LONG>01 00 00 00 </LONG>
        </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>
          <LONG>00 00 00 00 </LONG>
        </DoneRowCount>
      </DONEPROC>
    </TableResponse>
  </PacketData>
```

4.8 Attention Request

The following is an example of the Attention request that is sent from the client to the 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

The following is an example of the SSPI message carrying the SSPI payload that is sent from the client to the server.

```
11 01 00 3F 00 00 04 00 4E 54 4C 4D 53 53 50 00
01 00 00 00 97 B2 08 E2 07 00 07 00 30 00 00 00
08 00 08 00 28 00 00 00 06 00 71 17 00 00 00 0F
58 49 4E 57 45 49 48 32 52 45 44 4D 4F 4E 44

<PacketHeader>
  <Type>
    <BYTE>11 </BYTE>
  </Type>
  <Status>
    <BYTE>01 </BYTE>
  </Status>
  <Length>
    <BYTE>00 </BYTE>
    <BYTE>3F </BYTE>
  </Length>
  <SPID>
    <BYTE>00 </BYTE>
    <BYTE>00 </BYTE>
  </SPID>
  <Packet>
```

```
    <BYTE>04 </BYTE>
  </Packet>
  <Window>
    <BYTE>00 </BYTE>
  </Window>
</PacketHeader>
<PacketData>
  <SSPI>
    <BYTES>
4E 54 4C 4D 53 53 50 00 01 00 00 00 97 B2 08 E2 07 00 07 00
30 00 00 00 08 00 08 00 28 00 00 00 06 00 71 17 00 00 00 0F
58 49 4E 57 45 49 48 32 52 45 44 4D 4F 4E 44</BYTES>
  </SSPI>
</PacketData>
```

4.10 Bulk Load

The following is an example of the BULKLOADBCP request that is sent from the client to the server.

```
07 01 00 21 00 00 01 00 17 00 01 00 0F 00 00 00
00 00 00 00 00 00 00 17 00 65 62 63 64 65 02 14
0F

  <PacketHeader>
    <Type>
      <BYTE>07 </BYTE>
    </Type>
    <Status>
      <BYTE>01 </BYTE>
    </Status>
    <Length>
      <BYTE>00 </BYTE>
      <BYTE>21 </BYTE>
    </Length>
    <SPID>
      <BYTE>00 </BYTE>
      <BYTE>00 </BYTE>
    </SPID>
    <Packet>
      <BYTE>01 </BYTE>
    </Packet>
    <Window>
      <BYTE>00 </BYTE>
    </Window>
  </PacketHeader>
  <PacketData>
    <BulkLoadBCP>
      <RowData>
        <Length>
          <USHORT>17 00 </USHORT>
        </Length>
        <ColData>
          <NumVarCols>
            <BYTE>01 </BYTE>
          </NumVarCols>
          <RowNum>
```

```
            <BYTE>00 </BYTE>
          </RowNum>
          <FixedColData>
            <TYPE_VARBYTE>
              <BYTES>0F 00 00 00 </BYTES>
            </TYPE_VARBYTE>
          </FixedColData>
          <Paddings>
            <BYTES>00 00 00 00 00 00 00</BYTES>
          </Paddings>
          <RowLen>
            <USHORT>17 00 </USHORT>
          </RowLen>
          <VarColData>
              <BYTES>65 62 63 64 65 </BYTES>
          </VarColData>
          <Adjust>
            <BYTES>02 </BYTES>
          </Adjust>
          <Offset>
            <BYTES>14 0F </BYTES>
          </Offset>
        </ColData>
      </RowData>
    </BulkLoadBCP>
  </PacketData>
```

4.11 Transaction Manager Request

The following is an example of the transaction manager request that is sent from the client to the server.

```
0E 01 00 0C 00 00 01 00 00 00 00 00

<PacketHeader>
  <Type>
    <BYTE>0E </BYTE>
  </Type>
  <Status>
    <BYTE>01 </BYTE>
  </Status>
  <Length>
    <BYTE>00 </BYTE>
    <BYTE>0C </BYTE>
  </Length>
  <SPID>
    <BYTE>00 </BYTE>
    <BYTE>00 </BYTE>
  </SPID>
  <Packet>
    <BYTE>01 </BYTE>
  </Packet>
  <Window>
    <BYTE>00 </BYTE>
  </Window>
</PacketHeader>
```

```
<PacketData>
  <TransMgrReq>
    <RequestType>
      <USHORT>00 00 </USHORT>
    </RequestType>
    <RequestPayload>
        <TM_GET_DTC_ADDRESS>
        <US_VARBYTE>
          <USHORT>00 00 </USHORT>
          <BYTES></BYTES>
        </US_VARBYTE>
        </TM_GET_DTC_ADDRESS>
    </RequestPayload>
  </TransMgrReq>
</PacketData>
```

5 Security

5.1 Security Considerations for Implementers

As previously described in this document, the TDS 4.2 protocol provides facilities for authentication and channel encryption negotiation. If SSPI authentication is requested by the client application, the exact choice of security mechanisms is determined by the SSPI layer. Likewise, although the decision as to whether channel encryption should be used is negotiated in the TDS 4.2 layer, the exact choice of cipher suite is negotiated by the TLS/SSL layer.

5.2 Index of Security Parameters

Security parameter	Section
TLS Negotiation	2.2.6.4 PRELOGIN message 3.2.5.1 Sent Initial PRELOGIN Packet State (Client) 3.2.5.2 Sent TLS/SSL Negotiation Packet State (Client) 3.3.5.2 TLS/SSL Negotiation (Server)
SSPI Authentication	2.2.6.7 SSPI message 3.2.5.4 Sent SSPI Record with SPNEGO Packet Size 3.3.5.4 SPNEGO Negotiation
SQL Authentication	2.2.6.3 LOGIN message 3.3.5.3 Login Ready (Server)

6 Appendix A: Product Behavior

The information in this specification is applicable to the following Microsoft products or supplemental software. References to product versions include released service packs:

- Microsoft® SQL Server® 6.5
- Microsoft® SQL Server® 7
- 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 or Quick Fix Engineering (QFE) number appears with the product version, behavior changed in that service pack or QFE. The new behavior also applies to subsequent service packs of the product unless otherwise specified. If a product edition appears with the product version, behavior is different in that product edition.

Unless otherwise specified, any statement of optional behavior in this specification that is 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 the product does not follow the prescription.

<1> Section 2.1: For more information related to Microsoft-specific implementations, see [MSDN-NamedPipes].

<2> Section 2.2.2.6: If a stored procedure executes one or more other stored procedures, a DONEPROC token data stream signals the completion of each stored procedure. When executing SQL statements as a result of a trigger event, the completion of each of the SQL statements inside the trigger is indicated by a DONEINPROC token data stream.

<3> Section 2.2.6.7: The SSPI signature for DBLIB that is recognized by SQL Server is "d5bf8d50-451e-11d1-968d-e4b783000000". The SSPI token contains US_VARBYTE: that is, the length of the string followed by the string itself.

<4> Section 2.2.7.5: This element is not implemented in SQL Server.

<5> Section 2.2.7.7: This bit is not set by SQL Server, and should be considered reserved for future use.

<6> Section 2.2.7.8: This bit is not set by SQL Server, and should be considered reserved for future use.

<7> Section 2.2.7.9: This bit is not set by SQL Server, and should be considered reserved for future use.

<8> Section 2.2.7.11: Numbers less than 20001 are reserved by SQL Server.

<9> Section 2.2.7.11: SQL Server does not raise system errors with severities of 0 through 9.

<10> Section 2.2.7.11: For compatibility reasons, SQL Server converts severity 10 to severity 0 before returning the error information to the calling application.

<11> Section 2.2.7.12: Numbers less than 20001 are reserved by SQL Server.

<12> Section 2.2.7.17: This flag is not implemented in SQL Server.

7 Change Tracking

This section identifies changes that were made to the [MS-SSTDS] protocol document between the September 2010 and February 2011 releases. Changes are classified as New, Major, Minor, Editorial, or No change.

The revision class **New** means that a new document is being released.

The revision class **Major** means that the technical content in the document was significantly revised. 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.
- The removal of a document from the documentation set.
- Changes made for template compliance.

The revision class **Minor** means that the meaning of the technical content was clarified. Minor changes do not affect protocol interoperability or implementation. Examples of minor changes are updates to clarify ambiguity at the sentence, paragraph, or table level.

The revision class **Editorial** means that the language and formatting in the technical content was changed. Editorial changes apply to grammatical, formatting, and style issues.

The revision class **No change** means that no new technical or language changes were introduced. The technical content of the document is identical to the last released version, but minor editorial and formatting changes, as well as updates to the header and footer information, and to the revision summary, may have been made.

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

- New content added.
- Content updated.
- 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 are always classified with the change type **Editorially updated.**

Some important terms used in the change 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.

The changes made to this document are listed in the following table. For more information, please contact protocol@microsoft.com.

Section	Tracking number (if applicable) and description	Major change (Y or N)	Change type
2.2.2.2 Login Response	393106 Updated information about the DONE token data stream.	N	Content updated.
2.2.2.5 Return Parameters	550630 Clarified procedure.	N	Content updated.
2.2.3 Packets	395238 Updated packet description.	N	Content updated.
2.2.3.1.2 Status	393130 Clarified the description of the 0x01 value.	N	Content updated.
2.2.6.3 LOGIN	396416 Updated stream parameter details.	N	Content updated.
2.2.6.4 PRELOGIN	391386 Updated description of encryption.	N	Content updated.
2.2.6.5 RPC Request	394832 Updated stream-specific rules.	N	Content updated.
2.2.7.3 ALTROW	390380 Updated description of the token stream function.	N	Content updated.
3.2.2 Timers	390286 Clarified description of timers.	N	Content updated.
3.2.2	403257	N	Content

Section	Tracking number (if applicable) and description	Major change (Y or N)	Change type
Timers	Clarified description of timers.		updated.
3.3.5.4 SPNEGO Negotiation	399493 Updated description of the Complete result.	N	Content updated.
3.3.5.5 Logged In	394816 Updated description of the Logged In state.	N	Content updated.
3.3.5.6 Client Request Execution	394820 Removed content that specified that the server MUST close the connection and enter the final state if any other message type arrives.	N	Content removed.
4.1 Pre-Login Request	390315 Updated pre-login request example.	N	Content updated.

8 Index

U

V

W

Z