[MS-NNS]:
.NET NegotiateStream Protocol Specification

Revision Summary

Date	Revision History	Revision Class	Comments
07/20/2007	0.1	Major	MCPP Milestone 5 Initial Availability
09/28/2007	0.1.1	Editorial	Revised and edited the technical content.
10/23/2007	0.1.2	Editorial	Revised and edited the technical content.
11/30/2007	0.1.3	Editorial	Revised and edited the technical content.
01/25/2008	1.0	Major	Updated and revised the technical content.

Table of Contents

1 Introduction

The .NET NegotiateStream Protocol provides mutually authenticated and confidential communication over a TCP connection. It uses the **Simple and Protected Generic Security Service Application Program Interface (GSS-API) Negotiation (SPNEGO)** mechanism for security services (authentication, key derivation, and data encryption and decryption).

1.1 Glossary

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

Big-Endian
Kerberos
Little-Endian
Security Support Provider Interface (SSPI)
Windows Error Code

The following terms are specific to this document:

Simple and Protected Generic Security Service Application Program Interface (GSS-API) Negotiation (SPNEGO): A pseudo-security mechanism that enables GSS-API peers to determine in-band whether their credentials support a common set of one or more GSS-API security mechanisms; if so, the SPNEGO mechanism invokes the normal security context establishment for a selected common security mechanism.

MAY, SHOULD, MUST, SHOULD NOT, MUST NOT: These terms (in all caps) are used as specified 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.

[MS-ERREF] Microsoft Corporation, "Windows Error Codes", January 2007.

[MS-NLMP] Microsoft Corporation, "NT LAN Manager (NTLM) Authentication Protocol Specification", June 2007.

[MS-SPNG] Microsoft Corporation, "Simple and Protected Generic Security Service Application Program Interface Negotiation Mechanism (SPNEGO) Protocol Extensions", January 2007.

[RFC1964] Linn, J., "The Kerberos Version 5 GSS-API Mechanism", RFC 1964, June 1996, http://www.ietf.org/rfc/rfc1964.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

[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

1.2.2 Informative References

None.

1.3 Protocol Overview (Synopsis)

The .NET NegotiateStream Protocol was introduced to address the need for a simple and lightweight authentication and security mechanism between a client and a server when the client or server needs direct access to the TCP stream. A key benefit is that authentication is accomplished without the use of digital certificates. The .NET NegotiateStream Protocol provides a means for framing GSS-API (as specified in [RFC4178]) over a TCP stream. This is used to negotiate the security context for communication between a client and a server. The client and server can then exchange data protected by the negotiated security context.

The .NET NegotiateStream Protocol uses the SPNEGO mechanism (specified in [RFC4178]), which selects between **Kerberos** and NT LAN Manager (NTLM) authentication (specified in [MS-NLMP]), to determine which underlying security protocol to use. The .NET NegotiateStream Protocol can also forego negotiation with SPNEGO and use NTLM authentication directly when necessary.

This protocol enables:

- Client and/or server authentication
- Data confidentiality and integrity

The .NET NegotiateStream Protocol performs these functions in two phases: a security context negotiation phase and a data transfer phase.

The security context negotiation allows for the selection of mechanisms to protect the authenticity and confidentiality of data that will be subsequently exchanged. SPNEGO is used to select the underlying security protocol, and the security context is negotiated between the client and server in a set of messages. This context negotiation is initiated by the client, and several messages may be exchanged before the security context negotiation is complete. When the negotiation is completed, the client and server have agreed upon the necessary authentication, data integrity, and confidentiality mechanisms. These mechanisms will be used to secure subsequent data exchanges between the client and the server.

After the security context has been successfully negotiated, the client and server exchange data that is protected using the agreed-upon authentication, integrity, and confidentiality mechanisms. The server may initiate a data send to the client, or the client may initiate a data send to the server. A data send can happen at any time after the security context negotiation is complete.

An error in the negotiation process or the data sending process invalidates the stream, and a new security context must be renegotiated. The reason for the failure to negotiate a security context is communicated to the other machine taking part in the negotiation. An error in the data sending process can include sending data with an authentication, integrity, or confidentiality mechanism different from what was negotiated. The connection can also be closed with a TCP FIN message.

1.4 Relationship to Other Protocols

The .NET NegotiateStream Protocol relies on TCP for transport, and it relies on the SPNEGO security protocol and the NTLM security protocol ([MS-NLMP]) for authentication and message securing. The .NET NegotiateStream Protocol authenticates (and optionally adds data integrity and confidentiality support to) secure information transmitted between a client and a server over a raw TCP stream by using the SPNEGO GSS-API (as specified in [RFC4178]). The protocol also allows for NTLM to be used without negotiation.

In the case of negotiation, the .NET NegotiateStream Protocol calls for the use of SPNEGO GSS-API tokens for security context management and per-message protection purposes. These tokens follow the formats set for SPNEGO (specified in [RFC4178]). When negotiation is not used, the protocol uses NTLM tokens as specified in [MS-NLMP].

1.5 Prerequisites/Preconditions

Preconditions:

- Client and server support the SPNEGO GSS-API.
- A TCP connection has been established between client and server, and no data has been sent on the connection by the client or the server.

1.6 Applicability Statement

The .NET NegotiateStream Protocol is designed to secure information transmitted between a client and a server. The protocol provides security services without using digital certificates, and is thus useful to secure network traffic when the use of certificates is not an option. The .NET NegotiateStream Protocol uses SPNEGO (which selects between Kerberos and NTLM) to determine the underlying security protocol to use. Therefore, this protocol is best suited for environments in which a Kerberos infrastructure is deployed. Using SPNEGO, the .NET NegotiateStream Protocol enables:

- Client and/or server authentication
- Data integrity and confidentiality

1.7 Versioning and Capability Negotiation

- **Supported Transports:** This protocol must be implemented on top of TCP.
- **Protocol Versions:** A single version of the .NET NegotiateStream Protocol has been defined. The .NET NegotiateStream Protocol supports versioning and is currently at version 1.0. The versioning capability is outlined in section 2.2.
- **Security and Authentication Methods:** The .NET NegotiateStream Protocol supports the use of the SPNEGO and Microsoft NTLM protocols (as specified in [RFC4178] and [MS-NLMP], respectively).
- **Capability Negotiation:** This protocol has a single mode of operation, which is further described in sections 2 and 3 of this document.
  - When implemented on Windows 98 or Windows Me, the NTLM security package is used to generate the security tokens in the handshake messages (see section 2.2.1 for handshake message syntax).
  - When implemented on operating systems other than Windows 98 or Windows Me, the SPNEGO security package is used to generate the security tokens present in the handshake messages (see section 2.2.1 for handshake message syntax).

Version 1.0 is the only defined version of the .NET NegotiateStream Protocol. Therefore, no version negotiation semantics are described in this document. At present, a .NET NegotiateStream Protocol implementation should treat all messages received with a version number other than 1.0 as invalid. See the subsequent sections of this document for more information.

1.8 Vendor-Extensible Fields

The .NET NegotiateStream Protocol does not define any vendor-extensible fields.

This protocol uses Win32 error codes. These values are taken from the Windows error number space defined in [MS-ERREF]. Vendors SHOULD reuse those values with their indicated meanings. Choosing any other value runs the risk of a collision in the future.

1.9 Standards Assignments

No standards assignments have been made for this protocol.

2 Messages

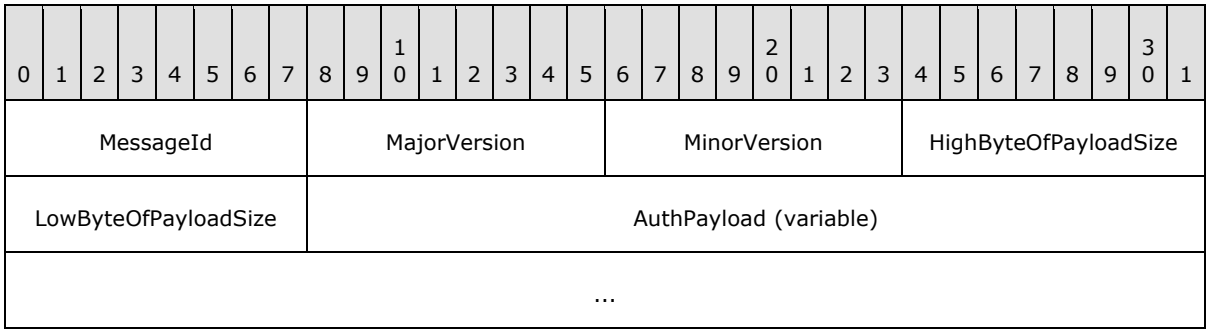
2.1 Transport

The .NET NegotiateStream Protocol transports messages using a TCP stream. The protocol specifies a framing for messages over a TCP stream (see section 2.2 for message syntax). The protocol does not define a mechanism to establish the TCP connection; rather, an established TCP connection is a precondition for this protocol. This established TCP connection can be over a prearranged port agreed upon and used by both the client and server.

2.2 Message Syntax

2.2.1 Handshake Message

The following message structure MUST be used during the handshake phase of the .NET NegotiateStream implementation.



MessageId (1 byte): An unsigned 8-bit integer that specifies the type of Handshake message. The values of this field MUST be one of the following (in decimal): 20, 21 or 22. The meaning of each value is as follows.

Type of Handshake	Description
HandshakeDone 0x14	Signals that the handshake has completed successfully.
HandshakeError 0x15	Signals that an error occurred during the handshake. The **Authpayload** contains a Win32 error code. See the description of the **AuthPayload** field for further information.
HandshakeInProgress 0x16	Signals that the message is part of the handshake phase, and is not the final message from the host. The final Handshake message from a host is always transferred in a HandshakeDone message.

MajorVersion (1 byte): An unsigned 8-bit integer that, along with the **MinorVersion** field, specifies the version of the .NET NegotiateStream Protocol being used. This field MUST be set to 1.

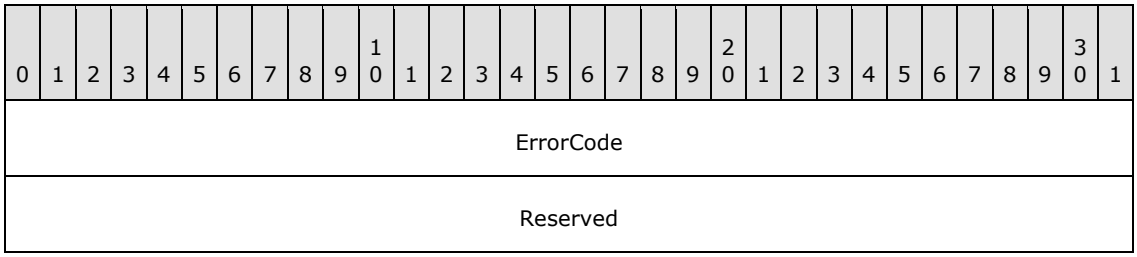
MinorVersion (1 byte): An unsigned 8-bit integer that, along with the **MajorVersion** field, defines the version of the .NET NegotiateStream Protocol being used. This field MUST be set to 0.

HighByteOfPayloadSize (1 byte): An unsigned 8-bit integer that, along with the **LowByteOfPayloadSize** field, defines the size, in bytes, of the **AuthPayload** field. This field represents the high-order byte of the payload size.

LowByteOfPayloadSize (1 byte): An unsigned 8-bit integer that, along with the **HighByteOfPayloadSize** field, defines the size, in bytes, of the **AuthPayload** field. This field represents the low-order byte of the payload size.

AuthPayload (variable): Contains the authentication tokens generated by the **Security Support Provider Interface (SSPI)** security packages (SPNEGO and NTLM) used by the .NET NegotiateStream Protocol. The format for these tokens is defined in [MS-SPNG]. When the Handshake message is a HandshakeDone message, this field is optional. When the Handshake message is a HandshakeInProgress message, this field is required and is of variable length. Finally, when the Handshake message is a HandshakeError message (and the payload size is >=8 bytes), the **AuthPayload** field should carry a **Windows error code** from the remote side's security package.

The following structure should be used to format this error code within the **AuthPayload** field:

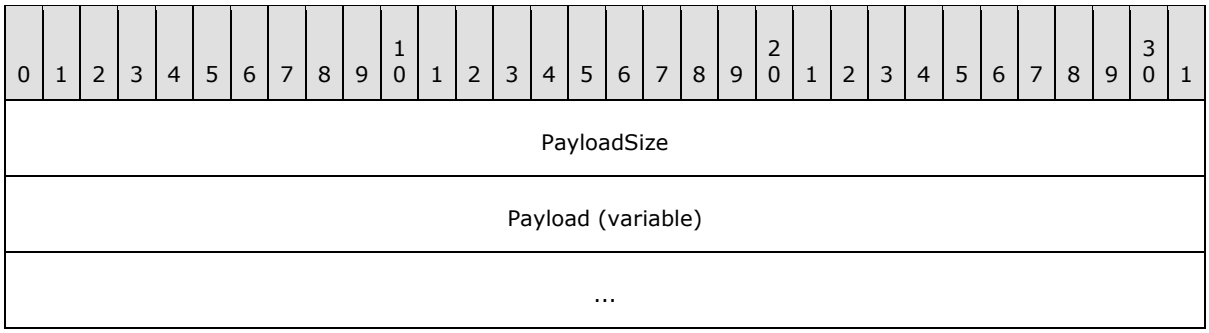


ErrorCode (4 bytes): This field carries the Win32 error code generated by the remote side's security package. This field is formatted using **big-endian** representation.

Reserved (4 bytes): This field is reserved and SHOULD be set to zero and the recipient MUST ignore the value.

2.2.2 DataMessage

The following defines the structure of the data exchange messages. These messages are used to transfer application-specific data after the handshake phase is complete. The .NET NegotiateStream Protocol only frames application data using the format noted below if the negotiation of security services during the handshake phase resulted in both the client and server agreeing to encrypt, sign, or encrypt and sign the data to be transferred. Thus, if the negotiated security context in the handshake phase results in a context that does not support message confidentiality or integrity, then the data transferred is not framed, and does not follow the format specified in this section (that is, application-supplied data is written directly to the underlying TCP stream).



PayloadSize (4 bytes): An unsigned 32-bit integer that defines the size, in bytes, of the **EncryptedPayload** field. The value in this field is formatted using **little-endian** representation. The maximum value for this field is 0xFC00 (that is, 63K, or 64,512).

Payload (variable): A buffer that contains the application-specific data to transfer between the client and server that has been secured by the selected security mechanism. When SPNEGO is selected as the security protocol, the structure and size of this field is compliant with the message structure defined in [RFC4178] (that is, the natively supported size using stream SSPI mode).

When NTLM is used as the security protocol, the structure and size of this field is as specified in [MS-NLMP] section 2.2.

The payload is in clear text if the negotiated security level is signature-only.

3 Protocol Details

3.1 Client Details

The following figure represents the client state machine for .NET NegotiateStream. The remainder of this section will discuss the state machine in depth.

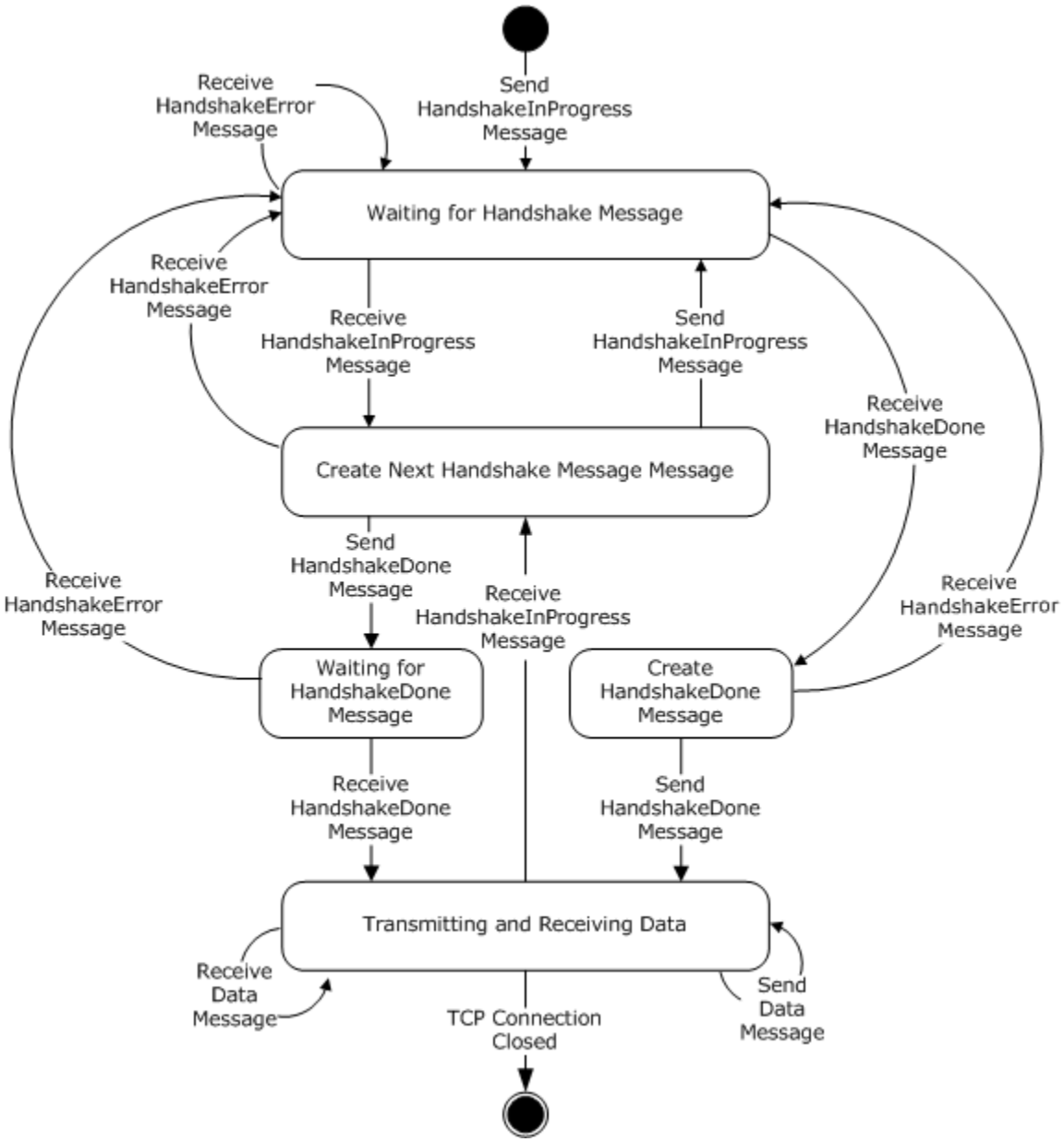


Figure 1: Client details

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. This document does not mandate that implementations adhere to this model, as long as their external behavior is consistent with that specified in this document.

3.1.1.1 Stream State

The .NET NegotiateStream Protocol uses the Stream State to keep track of the state of the stream. The possible values for the state of the stream are:

- Waiting to Authenticate
- Authenticate in Progress
- Authenticate Complete

3.1.1.2 Protection Levels

The .NET NegotiateStream Protocol uses Protection Levels to keep track of both the protection level desired and the protection level achieved during the security context negotiation. The possible values for the protection level desired and protection level achieved are:

- None
- Signed
- Encrypt
- EncryptAndSign

3.1.1.3 Mutual Authentication

The .NET NegotiateStream Protocol uses Mutual Authentication to keep track of both the mutual authentication desired and the mutual authentication achieved. Mutual authentication is either desired or not desired and is either achieved or not achieved.

3.1.1.4 Security Provider Context

The .NET NegotiateStream Protocol tracks the context of the current security provider chosen during the handshake phase.

3.1.1.5 Framing State Machine

The .NET NegotiateStream Protocol employs a framing state machine to handle the receiving and processing of full frames while in the handshake phase, and when the date payloads are signed, encrypted, or encrypted and signed.

3.1.2 Timers

The .NET NegotiateStream Protocol does not inherently require the use of timers. Protocols above and below this protocol layer are responsible for implementing timers for timeout events.

3.1.3 Initialization

The .NET NegotiateStream Protocol initialization for the client role is triggered by an application event. See section 3.1.4 for more details.

3.1.4 Higher-Layer Triggered Events

The client role in the .NET NegotiateStream Protocol initialization is triggered by the application. The initialization process is as follows:

1. It is the job of the application running in the client role to first establish a TCP connection to the server.

2. Then, before writing any data to the TCP stream, the client triggers the start of the protocol's handshake phase (at any time after the TCP connection is established) by generating and sending a Handshake message. The Handshake message sent MUST be a HandshakeInProgress message. The **AuthPayload** field of this initial HandshakeInProgress message contains the security token returned from the gss_init_sec_context function (specified in [RFC4178]).

3. If the gss_init_sec_context function returns an error code, then the client MUST create a HandshakeError message, placing the returned error code in the **AuthPayload** of the messages as described in section 2.2.1.

   If the gss_init_sec_context completes successfully, the client MAY choose not to proceed to the data transfer phase of the protocol.

4. If the client chooses not to proceed to the data transfer phase, then the client MUST create a HandshakeError message, placing the error code with decimal value 1790 (ERROR_TRUST_FAILURE in Windows error codes) in the **AuthPayload** field of the message, as described in section 2.2.1.

5. If the HandshakeError message is successfully sent to the server, then the client MAY retry authentication by repeating the initialization process from step 1.

3.1.5 Message Processing Events and Sequencing Rules

Handshake Phase

At any point in time, the client is either expecting a Handshake message or a DataMessage from the server. After the initiation of the protocol by the client (see section 3.1.4), the client waits to receive a response from the server before continuing the handshake phase of the protocol. Upon receiving a response the following actions SHOULD be taken:

- If the number of bytes received from the server is 0: This indicates that the server has initiated the closure of the underlying TCP connection. The client MUST treat the stream as being in an invalid state, and MUST no longer send or attempt to receive further .NET NegotiateStream messages on the TCP connection. The client SHOULD close the TCP connection at this point.

- If the response contains more than 0 bytes: The client MUST check the first 8 bytes of the message to see if it matches one of the three known message IDs for .NET NegotiateStream Handshake messages. If the message received is:

  - A HandshakeInProgress message: Upon receipt of a message of this type, the client MUST take the token from the **AuthPayload** field of the message and create a reply Handshake message by passing the token to the gss_init_sec_context function (specified in [RFC4178]). If this function generates an output token, it MUST be sent back to the server in the client's

reply. The type of Handshake message that the server replies with depends on the result returned from the gss_init_sec_context function call:

If the return code represents that the security context is complete, then the client MUST reply with a HandshakeDone message.

If the return code represents that the security context negotiation is not yet complete, the client MUST reply with a HandshakeInProgress message.

Finally, if the return code represents that an error occurred, the client MUST respond with a HandshakeError message, and place the Windows error code in the payload of the message.

- A HandshakeDone message: The protocol SHOULD proceed to the data transfer phase.
- A HandshakeError message: This message type can be received at any time during the handshake phase and therefore cannot, by definition, be received out of order. The client SHOULD then inspect the error code in the payload. If the error code is equal to 0x8009030C (LogonDenied) or 0x6FE (TrustFailure), then the client MAY restart the handshake phase (see section 3.1.4) or the client MAY close the TCP connection. The client MUST NOT repeatedly retry the handshake with the same failing credentials.
- None of the above handshake message types: In this case the client SHOULD restart the handshake phase (see section 3.1.4) or close the connection.

Data Transfer Phase

After the client receives a HandshakeDone message from the server and the handshake phase of the .NET NegotiateStream Protocol completes, the protocol MUST transition to the data transfer phase. In this phase, the client can send a DataMessage to the server at any time, and may receive a DataMessage from the server at any time. Upon receiving a response from the server, the client should inspect the security context negotiated during the authentication phase:

- If the handshake phase resulted in a security context that does not support message integrity or confidentiality, then the data received by the client is not framed, and all bytes read SHOULD be treated as application-level data and subsequently passed unmodified to the application.
- If the handshake phase resulted in a security context that supports data confidentiality, integrity, or both, then the client SHOULD expect to receive a DataMessage (section 2.2.2) that is framed with the security context negotiated. If the client does not receive a valid DataMessage, it MUST treat the stream as being in an invalid state, and MUST no longer send or attempt to receive further .NET NegotiateStream messages on the TCP connection. The client MUST close the TCP connection at this point. The server SHOULD perform the following steps to determine if the DataMessage is valid:

  1. Read the first 4 bytes of the message, which signify the size of the DataMessage.
  2. Retrieve an additional X bytes, where X is equal to the size read in step 1.
  3. Validate the bytes received in step 2 by either ensuring that they can be successfully decrypted (if the security negotiation resulted in encryption of the stream), or by ensuring that the signature is valid (if the security negotiation resulted in integrity of the stream). If the bytes received cannot be successfully decrypted or if the signature is invalid, the client MUST treat the DataMessage as invalid.

3.1.6 Timer Events

None.

3.1.7 Other Local Events

If the server closes the TCP connection (that is, it sends a TCP FIN or RST message) during the handshake phase of the protocol, then the client SHOULD treat the stream as being in an invalid state, and no longer send or attempt to receive further .NET NegotiateStream messages on the TCP connection. If the server sent a TCP FIN message, the client MAY close the TCP connection at this point (if it hasn't already done so).

3.2 Server Details

The following figure represents the server state machine for .Net NegotiateStream. The remainder of this section will discuss the state machine in depth.

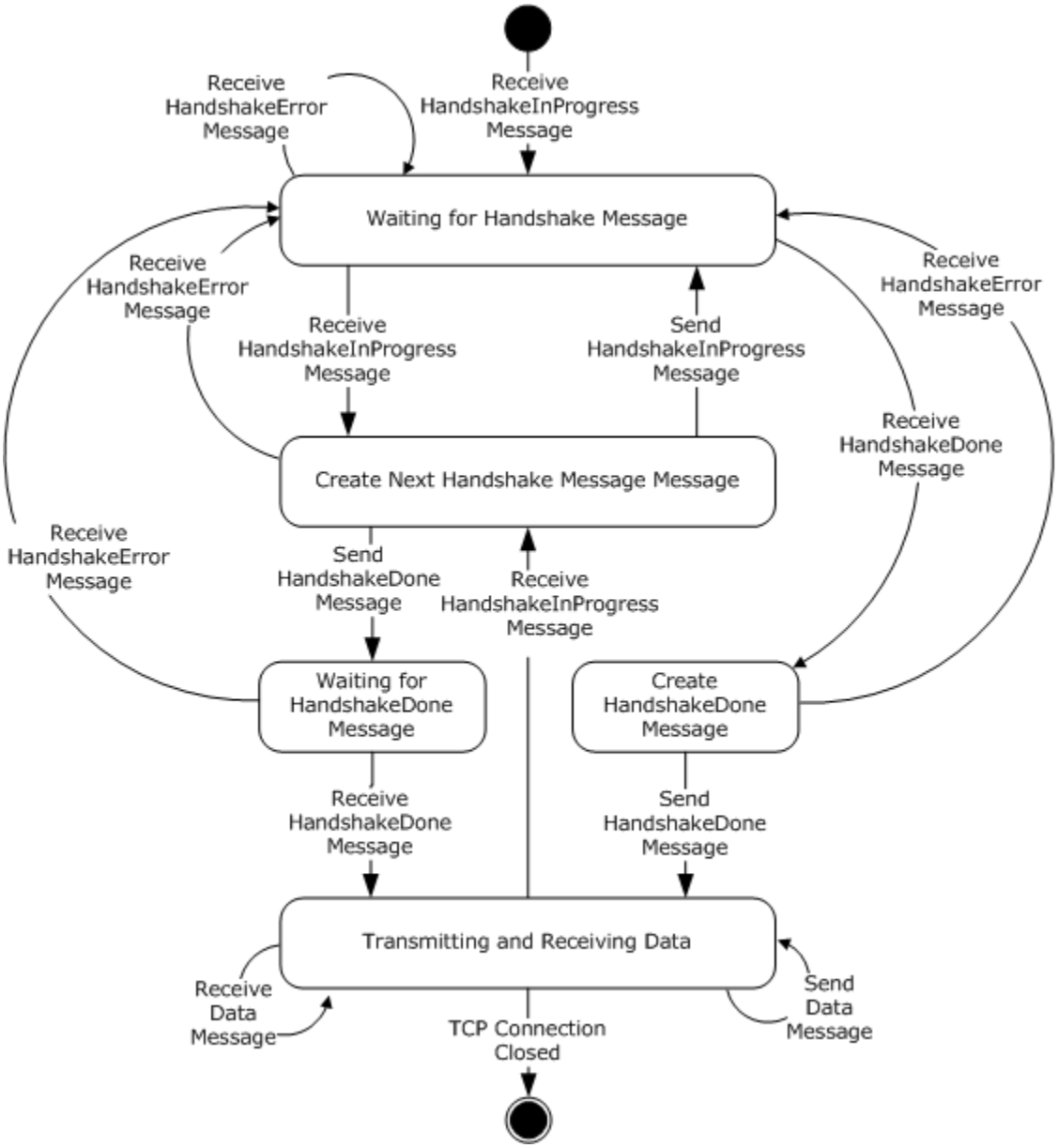


Figure 2: Server details

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. This document does not mandate that implementations adhere to this model as long as their external behavior is consistent with that specified in this document.

3.2.1.1 Stream State

The .NET NegotiateStream Protocol uses the Stream State to keep track of the state of the stream. The possible values for the state of the stream are:

- Waiting to Authenticate
- Authenticate in Progress
- Authenticate Complete

3.2.1.2 Protection Levels

The .NET NegotiateStream Protocol uses Protection Levels to keep track of both the protection level desired and the protection level achieved during the security context negotiation. The possible values for the protection level desired and protection level achieved are:

- None
- Signed
- Encrypt
- EncryptAndSign

3.2.1.3 Mutual Authentication

The .NET NegotiateStream Protocol uses Mutual Authentication to keep track of both the mutual authentication desired and the mutual authentication achieved. Mutual authentication is either desired or not desired, and is either achieved or not achieved.

3.2.1.4 Security Provider Context

The .NET NegotiateStream Protocol tracks the context of the current security provider chosen during the handshake phase.

3.2.1.5 Framing State Machine

The .NET NegotiateStream Protocol employs a framing state machine to handle the receiving and processing of full frames while in the handshake phase, and when the date payloads are signed, encrypted, or encrypted and signed.

3.2.2 Timers

The .NET NegotiateStream Protocol server role does not inherently require the use of timers. Protocols above and below this protocol layer are responsible for implementing timers for timeout events.

3.2.3 Initialization

The .NET NegotiateStream Protocol initialization for the server role is triggered by an application event. See section 3.2.4 for more details.

3.2.4 Higher-Layer Triggered Events

The server role in the .NET NegotiateStream Protocol initialization is triggered by the application. The server role of the .NET NegotiateStream Protocol MAY begin after a TCP stream (initiated by the client) has been established, and before writing or reading any data to or from the TCP stream. The server begins by waiting to receive a HandshakeInProgress message from the client.

If the gss_init_sec_context function returns an error code, then the server MUST create a HandshakeError message, placing the returned error code in the **AuthPayload** of the messages as described in section 2.2.1.

If the gss_init_sec_context function completes successfully, the server MAY choose not to proceed to the data transfer phase of the protocol. If the server chooses not to proceed to the data transfer phase, then the server MUST create a HandshakeError message, placing the error code with decimal value 1790 (ERROR_TRUST_FAILURE in Windows error codes) in the **AuthPayload** field of the message as described in section 2.2.1.

3.2.5 Message Processing Events and Sequencing Rules

Handshake Phase

As specified in section 3.2.4, the server role begins by waiting for a message from the client. The following actions SHOULD be taken by the server when a message arrives:

- If the number of bytes received from the client is 0: This indicates that the client has initiated the closure of the underlying TCP connection. The server MUST treat the stream as being in an invalid state, and MUST no longer send or attempt to receive further .NET NegotiateStream messages on the TCP connection. The server SHOULD close the TCP connection at this point.
- If the message contains more than 0 bytes, then the server MUST check the first 8 bytes of the message to see if it matches one of the three known message IDs for .NET NegotiateStream Handshake messages. If the message received is:
  - A HandshakeInProgress message: Upon receipt of a message of this type, the server MUST take the token from the **AuthPayload** field of the message and create a reply Handshake message, by passing the token to the gss_accept_sec_context function (specified in [RFC4178]). This function MAY generate an output token that MUST be sent back to the client in the server's reply. The type of Handshake message that the server replies with depends on the result returned from the gss_accept_sec_context function call:

    If the return code represents that the security context is complete, then the server MUST reply with a HandshakeDone message.

    If the return code represents that the security context negotiation is not yet complete, the server MUST reply with a HandshakeInProgress message.

    Finally, if the return code represents that an error occurred, the server SHOULD respond with a HandshakeError message and place the Windows error code in the payload of the message. After the server sends the HandshakeError message, the client MAY restart the handshake phase (see section 3.2.4).

- A HandshakeDone message: If the last message sent from the server to the client was a HandshakeDone message, then the protocol SHOULD proceed to the data transfer phase. If the last message sent from the server to the client was not a HandshakeDone message, then the server MUST create a HandshakeDone message and send it back to the client to signal completion of the handshake phase.
- A HandshakeError message: This message type can be received at any time during the handshake phase and therefore cannot, by definition, be received out of order. The server SHOULD then inspect the error code in the payload. If the error code is equal to 0x8009030C (LogonDenied) or 0x6FE (TrustFailure), then the server MAY restart the handshake phase (see section 3.2.4) or close the connection. The server MUST NOT repeatedly retry the handshake with the same failing credentials.
- None of the above Handshake message types: In this case the client SHOULD restart the handshake phase (see section 3.2.4) or close the connection.

Data Transfer Phase

After the server receives a HandshakeDone message from the client and the handshake phase of the .NET NegotiateStream Protocol completes, the protocol MUST transition to the data transfer phase. In this phase, the server MAY send a DataMessage to the client at any time, and MUST expect to receive a DataMessage from the client at any time. Upon receiving a message from the client, the server SHOULD inspect the security context negotiated during the authentication phase:

- If the handshake phase resulted in a security context that does not support message integrity or confidentiality, then the data received by the server is not framed, and all bytes read SHOULD be treat as application-level data and passed unmodified to the application.
- If the handshake phase resulted in a security context that supports data confidentiality, integrity or both, then the server SHOULD expect to receive a DataMessage (section 2.2.2) that is framed with the security context negotiated. If the server does not receive a valid DataMessage, it MUST treat the stream as being in an invalid state, and MUST no longer send or attempt to receive further .NET NegotiateStream messages on the TCP connection. The server MUST close the TCP connection at this point. The server SHOULD perform the following steps to determine if the DataMessage is valid:

  1. Read the first 4 bytes of the message, which signify the size of the DataMessage.
  2. Retrieve an additional X bytes, where X is equal to the size read in step 1.
  3. Validate the bytes received in step 2 by either ensuring that they can be successfully decrypted (if the security negotiation resulted in encryption of the stream), or by ensuring that the signature is valid (if the security negotiation resulted in integrity of the stream). If the bytes received cannot be successfully decrypted or if the signature is invalid, the client MUST treat the DataMessage as invalid.

3.2.6 Timer Events

None.

3.2.7 Other Local Events

If the client closes the TCP connection (that is, it sends a TCP FIN or RST message) during the handshake phase of the protocol, then the server SHOULD treat the stream as being in an invalid state, and no longer send or attempt to receive further .NET NegotiateStream messages on the TCP

connection. If the client sent a TCP FIN message, the server SHOULD close the TCP connection at this point (if it hasn't already done so).

4 Protocol Examples

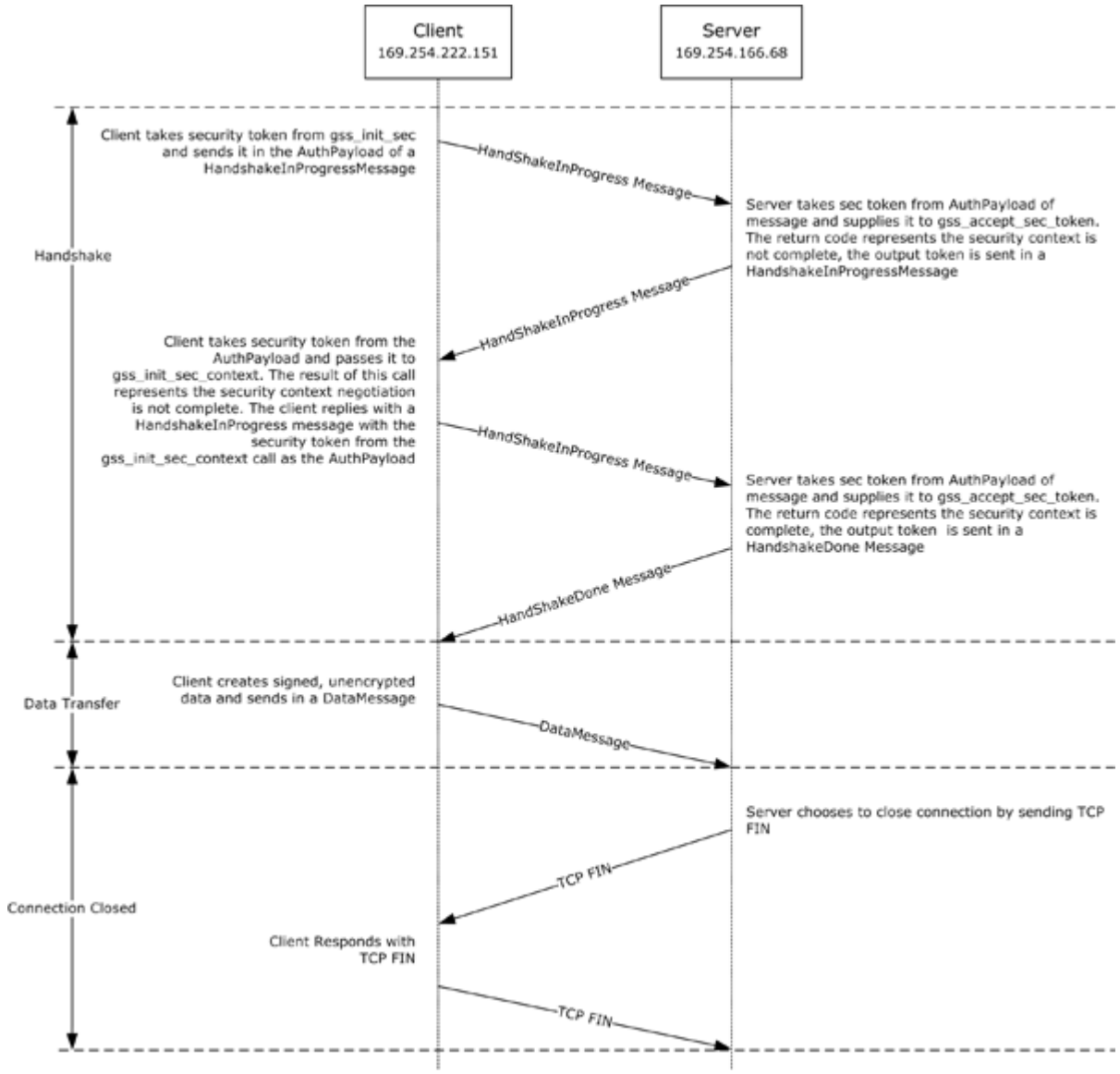


Figure 3: Protocol example

Figure 3 gives a simple example of handshake, data transfer, and closing of the connection.

1. To initiate the connection, the client first calls gss_init_sec_context to obtain the security token. This security token is placed in the **AuthPayload** field of a HandshakeInProgress message and sent to the server machine.

   Upon receipt of this HandshakeInProgress message, the server passes the **AuthPayload** to the gss_accept_sec_token function. In this example, this function returns an output token and indicates that the security context negotiation is not yet complete.

2. The server takes the token returned from gss_accept_sec_token, and places this in the **AuthPayload** field of a HandShakeInProgress message, and sends this message to the client.

3. The client receives the HandshakeInProgress message, and passes the token in the **AuthPayload** to gss_init_sec_context. The number of HandshakeInProgress messages exchanged between the client and server is dependent on the authentication level selected by the client and the authentication protocol which was negotiated between the client and server. In this example, the return code of gss_init_sec_context indicates that the handshake is not complete. The client takes the token returned from gss_init_sec_context, and places it in the **AuthPayload** field of a HandshakeInProgress message.

   The client sends the HandshakeInProgress message to the server.

4. After the server receives the HandshakeInProgress message from the client, the server passes the **AuthPayload** to the gss_accept_sec_token function. In this example, this function returns an output token and indicates that the security context negotiation is complete. The server takes the token returned from gss_accept_sec_token, and places this in the **AuthPayload** field of a HandShakeDone message, and sends this message to the client. The server now enters the data transfer phase.

5. When the client receives the HandshakeDone message from the server, it also transitions to the data transfer phase. In this example, the client chooses to send a single message to the server. The client creates this message as specified in section 2.2.2, using the negotiated security context from the handshake phase. The client then sends this message to the server.

6. The server receives the message, and in this example chooses to end the connection at this time. The server sends a TCP FIN to the client to indicate that the connection is closed.

7. The client chooses to respond with a TCP FIN message as well. At this point, the connection is closed.

5 Security

The following sections specify security considerations for implementers of the .NET NegotiateStream Protocol.

5.1 Security Considerations for Implementers

The .NET NegotiateStream Protocol is dependent on the security services of the SPNEGO security package of NTLM. Before using the .NET NegotiateStream Protocol, implementers should carefully review the characteristics of the security providers used by SPNEGO.

Implementers should ensure timely closure of connections in the case of an authentication failure, or when the data stream is determined to be invalid.

5.2 Index of Security Parameters

Security Parameter	Section
Client Role: Credentials, Message protection level	3.1.4
Server Role: Credentials, Message protection level	3.2.4

6 Appendix A: Windows Behavior

The .NET NegotiateStream Protocol is implemented by the NegotiateStream class in versions 2.0 and later of the .NET Framework. The operating systems that come with V2.0 (or later) of the .NET Framework installed are:

- Windows Vista

The following operating systems support the NegotiateStream class, but do not come with the required version of the .NET Framework installed by default:

- Windows 98
- Windows Me
- Windows 2000 SP4
- Windows XP
  - Media Center Edition
  - Windows XP 64-Bit Edition
  - Windows XP SP2
  - Windows XP Starter Edition
- Windows Server 2003
- Windows Vista

If the client specifies not to request message integrity or encryption, then the NTLM security package is used instead of the SPNEGO package.

All versions of Windows obey all the MUST and SHOULD statements outlined in this specification. The behavior of Windows regarding all MAY statements is controlled by the application that is using .NET NegotiateStream.

7 Index

T

V

W