[MS-GRVHENC]: HTTP Encapsulation of Simple Symmetric Transport Protocol (SSTP)

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 Open Specification Promise or the Community Promise. 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. For a list of Microsoft trademarks, visit www.microsoft.com/trademarks.
- **Fictitious Names.** The example companies, organizations, products, domain names, email 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.

Copyright © 2014 Microsoft Corporation.

Revision Summary

Date	Revision History	Revision Class	Comments
04/04/2008	0.1		Initial Availability
06/27/2008	1.0	Major	Revised and edited the technical content
12/12/2008	1.01	Editorial	Revised and edited the technical content
07/13/2009	1.02	Major	Changes made for template compliance
08/28/2009	1.03	Editorial	Revised and edited the technical content
11/06/2009	1.04	Editorial	Revised and edited the technical content
02/19/2010	2.0	Major	Updated and revised the technical content
03/31/2010	2.01	Editorial	Revised and edited the technical content
04/30/2010	2.02	Editorial	Revised and edited the technical content
06/07/2010	2.03	Editorial	Revised and edited the technical content
06/29/2010	2.04	Editorial	Changed language and formatting in the technical content.
07/23/2010	2.05	Minor	Clarified the meaning of the technical content.
09/27/2010	2.05	No change	No changes to the meaning, language, or formatting of the technical content.
11/15/2010	2.05	No change	No changes to the meaning, language, or formatting of the technical content.
12/17/2010	2.05	No change	No changes to the meaning, language, or formatting of the technical content.
03/18/2011	2.05	No change	No changes to the meaning, language, or formatting of the technical content.
06/10/2011	2.05	No change	No changes to the meaning, language, or formatting of the technical content.
01/20/2012	2.6	Minor	Clarified the meaning of the technical content.
04/11/2012	2.6	No change	No changes to the meaning, language, or formatting of the technical content.
07/16/2012	2.6	No change	No changes to the meaning, language, or formatting of the technical content.
10/08/2012	2.6	No change	No changes to the meaning, language, or formatting of the technical content.
02/11/2013	2.6	No change	No changes to the meaning, language, or formatting of the technical content.

Copyright © 2014 Microsoft Corporation.

Date	Revision History	Revision Class	Comments
07/30/2013	2.6	No change	No changes to the meaning, language, or formatting of the technical content.
11/18/2013	2.6	No change	No changes to the meaning, language, or formatting of the technical content.
02/10/2014	2.6	No change	No changes to the meaning, language, or formatting of the technical content.
04/30/2014	2.6	No change	No changes to the meaning, language, or formatting of the technical content.
07/31/2014	2.6	No change	No changes to the meaning, language, or formatting of the technical content.

Copyright © 2014 Microsoft Corporation.

Table of Contents

1 Introduction 12
1.1 Glossary 13
1.2 References 13
1.2.1 Normative References 14
1.2.2 Informative References 14
1.3 Protocol Overview (Synopsis) 15
1.3.1 HTTP Encapsulation Protocols 18
1.3.1.1 HTTP LongLived Encapsulation Connections 19
1.3.1.2 HTTP KeepAlive Encapsulation Connections 20
1.3.1.3 HTTP Polling Encapsulation Connections 22
1.3.2 Secure Tunnel Connections 24
1.3.3 SOCKS Connections 26
1.3.4 Performance Considerations 26
1.4 Relationship to Other Protocols 27
1.5 Prerequisites/Preconditions 28
1.6 Applicability Statement 28
1.7 Versioning and Capability Negotiation 28
1.8 Vendor-Extensible Fields 29
1.9 Standards Assignments 29

2 Messages 30
2.1 Transport 30
2.2 Message Syntax 30
2.2.1 Common HTTP Data Types 30
2.2.1.1 Encapsulation Data Types 30
2.2.1.1.1 Virtual-Connection-GUID 30
2.2.1.1.2 Relay-Server-Name 31
2.2.1.1.3 Encapsulation-Echo-String 31
2.2.1.1.4 Application-Data 31
2.2.1.1.4.1 SSTP_COMMAND 31
2.2.1.1.5 Server-User-Agent 32
2.2.1.2 Request-Header 32
2.2.1.2.1 Accept 33
2.2.1.2.2 Content-Type 33
2.2.1.2.3 User-Agent 33
2.2.1.2.4 Pragma 34
2.2.1.2.5 Expires 34
2.2.1.2.6 Connection 34
2.2.1.2.7 Host 34
2.2.1.2.8 Cache-Control 35
2.2.1.2.9 Proxy-Connection 35
2.2.1.3 Response Headers 35
2.2.1.3.1 Date 35
2.2.1.3.2 Server 36
2.2.1.4 Response Status Code and Reason Phrase 36
2.2.2 LongLived Encapsulation 36
2.2.2.1 LongLived-GET-Request 36
2.2.2.1.1 LongLived-GET-Request-URI 37
2.2.2.1.1.1 LongLived-Encapsulation-Version 37
2.2.2.1.1.2 LongLived-Encapsulation-Type-Token 38

Copyright © 2014 Microsoft Corporation.

2.2.2.1.1.3 LongLived-Encapsulation-Content-Length 38
2.2.2.1.1.4 LongLived-Encapsulation-Request-ID 38
2.2.2.1.2 LongLived-GET-Request Example 38
2.2.2.2 LongLived-POST-Request.. 38
2.2.2.2.1 LongLived-POST-Request-URI... 39
2.2.2.2.2 LongLived-Content-Length 39
2.2.2.2.3 LongLived-Entity-Body 40
2.2.2.2.4 LongLived-POST-Request Example 40
2.2.2.3 LongLived-GET-Response 40
2.2.2.3.1 Response-Status-Line 41
2.2.2.3.2 LongLived-GET-Response-Content-Length 41
2.2.2.3.3 LongLived-GET-Response Example 41
2.2.2.4 LongLived-POST-Response.. 42
2.2.2.4.1 LongLived-POST-Response-Content-Length 42
2.2.3 KeepAlive Encapsulation.. 42
2.2.3.1 KeepAlive-GET-Request 42
2.2.3.1.1 KeepAlive-Request-URI 43
2.2.3.1.1.1 KeepAlive-Encapsulation-Type-Token 43
2.2.3.1.1.2 KeepAlive-Encapsulation-Version 43
2.2.3.1.1.3 KeepAlive-Encapsulation-Request-ID 44
2.2.3.1.2 KeepAlive-GET-Request Example 44
2.2.3.2 KeepAlive-POST-Request 44
2.2.3.2.1 KeepAlive-Content-Length.. 45
2.2.3.2.2 KeepAlive-Entity-Body 45
2.2.3.2.3 KeepAlive-POST-Request.. 45
2.2.3.3 KeepAlive-GET-Response... 45
2.2.3.3.1 KeepAlive-GET-Response Example 46
2.2.3.4 KeepAlive-POST-Response 46
2.2.3.4.1 KeepAlive-POST-Response-Entity-Body 46
2.2.3.4.1.1 KeepAlive-POST-Response-No-Data 46
2.2.3.4.2 KeepAlive-POST-Response Example.. 47
2.2.4 Polling Encapsulation 47
2.2.4.1 Polling-POST-Request.. 47
2.2.4.1.1 Polling-Request-URI 47
2.2.4.1.2 Polling-Content-Length 48
2.2.4.1.3 Polling-Request-Entity-Body 48
2.2.4.1.3.1 Polling-Virtual-Connection-Message 48
2.2.4.1.3.1.1 Polling-Encapsulation-Version 48
2.2.4.1.3.1.2 Sequence-Number 49
2.2.4.1.3.1.3 Checksum 49
2.2.4.1.3.1.4 Relay-Server-URL 49
2.2.4.1.4 Polling-POST-Request Example 49
2.2.4.2 Polling-POST-Response.. 50
2.2.4.2.1 Polling-Response-Entity-Body 50
2.2.4.2.1.1 Polling-Virtual-Connection-Response-Message................................. 50
2.2.4.2.1.1.1 Max-Poll-Interval 50
2.2.4.2.1.1.2 Min-Poll-Interval 50
2.2.4.2.1.1.3 Poll-Repetition 51
2.2.4.2.2 Polling-POST-Response Example 51
2.2.5 Secure Tunnel Proxy 51
2.2.6 SOCKS Encapsulation 52

3 Protocol Details 55

Copyright © 2014 Microsoft Corporation.

3.1 LongLived Encapsulation Protocol Client Details 55
3.1.1 LongLived Client Abstract Data Model 55
3.1.1.1 Connection State Information 57
3.1.1.2 Proxy State Information 58
3.1.2 LongLived Client Timers 59
3.1.2.1 ConnectionEstablishment Timer 59
3.1.2.2 NetworkReceiveIO Timer 59
3.1.2.3 KeepAlive Timer 59
3.1.3 LongLived Client Initialization 59
3.1.3.1 Protocol Initialization 59
3.1.4 LongLived Client Higher-Layer Triggered Events 59
3.1.4.1 Establishing a LongLived Encapsulation Connection 59
3.1.4.1.1 Establishing GET Session without Proxy 60
3.1.4.1.2 Establishing GET Session with Proxy 60
3.1.4.1.3 Establishing POST Session without Proxy 61
3.1.4.1.4 Establishing POST Session with Proxy 61
3.1.4.2 Closing a LongLived Connection 61
3.1.4.3 Sending Application Data 62
3.1.5 LongLived Client Message Processing Events and Sequencing Rules 62
3.1.5.1 Receiving Data on the POST Session 62
3.1.5.1.1 LongLived-POST-Response Processing 62
3.1.5.1.1.1 Status code: 400 (Bad Request) 62
3.1.5.1.1.2 Status codes: 401 (Unauthorized) and 407 (ProxyAuthentication Required) 62
3.1.5.1.1.3 All Other Status Codes 63
3.1.5.1.2 POST Session Data Processing 63
3.1.5.2 Receiving Data on the GET Session 63
3.1.5.2.1 LongLived-GET-Response Processing 63
3.1.5.2.1.1 Status code: 200 (OK) 63
3.1.5.2.1.2 Status code: 400 (Bad Request) 64
3.1.5.2.1.3 Status codes: 401 (Unauthorized) and 407 (ProxyAuthentication Required) 64
3.1.5.2.1.4 All Other Status Codes 64
3.1.5.2.2 Receiving Application Data (GET Session Data Processing) 64
3.1.6 LongLived Client Timer Events 64
3.1.6.1 ConnectionEstablishment Timer Event 64
3.1.6.2 Network Receive IO Timer Event 65
3.1.6.3 KeepAlive Timer Event 65
3.1.7 LongLived Client Other Local Events 65
3.2 LongLived Encapsulation Protocol Server Details 65
3.2.1 LongLived Server Abstract Data Model 65
3.2.1.1 Connection State Information 65
3.2.2 LongLived Server Timers 66
3.2.2.1 ConnectionEstablishment Timer 66
3.2.2.2 Network Receive IO Timer 66
3.2.2.3 KeepAlive Timer 66
3.2.3 LongLived Server Initialization 66
3.2.3.1 Protocol Initialization 66
3.2.3.2 LongLived Listener 66
3.2.4 LongLived Server Higher-Layer Triggered Events 66
3.2.4.1 Closing a LongLived Connection 66
3.2.4.2 Sending Application Data 67
3.2.5 LongLived Server Message Processing Events and Sequencing Rules 67

Copyright © 2014 Microsoft Corporation.

3.2.5.1 GET Session Processing 67
3.2.5.1.1 Receiving a LongLived-GET-Request 67
3.2.5.1.1.1 Sending a LongLived-GET-Response with Status Code 200 68
3.2.5.1.1.2 Sending a LongLived-GET-Response with Status Code 400 69
3.2.5.1.2 Receiving Data on LongLived-GET-Request 69
3.2.5.2 POST Session Processing 69
3.2.5.2.1 Receiving a LongLived-POST-Request 69
3.2.5.2.1.1 Sending a LongLived-POST-Response because of a Protocol Error 70
3.2.5.2.2 Receiving Application Data 70
3.2.6 LongLived Server Timer Events 70
3.2.6.1 ConnectionEstablishment Timer Event 70
3.2.6.2 NetworkReceiveIO Timer Event 71
3.2.6.3 KeepAlive Timer Event 71
3.2.7 LongLived Server Other Local Events 71
3.3 KeepAlive Encapsulation Protocol Client Details 71
3.3.1 KeepAlive Client Abstract Data Model 71
3.3.1.1 Connection State Information 73
3.3.1.2 Proxy State Information 74
3.3.2 KeepAlive Client Timers 75
3.3.2.1 ConnectionEstablishment Timer 75
3.3.2.2 GetNetworkReceiveIO Timer 75
3.3.2.3 PostNetworkReceiveIO Timer 75
3.3.2.4 KeepAlive Timer 75
3.3.3 KeepAlive Client Initialization 75
3.3.3.1 Protocol Initialization 75
3.3.4 KeepAlive Client Higher-Layer Triggered Events 76
3.3.4.1 Establishing a KeepAlive Encapsulation Connection 76
3.3.4.1.1 Establishing GET Session without Proxy 76
3.3.4.1.2 Establishing GET Session with Proxy 76
3.3.4.1.3 Establishing POST Session without Proxy 77
3.3.4.1.4 Establishing POST Session with Proxy 77
3.3.4.2 Closing a KeepAlive Connection 77
3.3.4.3 Closing a KeepAlive POST Session 77
3.3.4.4 Closing a KeepAlive GET Session 78
3.3.4.5 Re-Opening a KeepAlive POST Session 78
3.3.4.6 Re-Opening a KeepAlive GET Session 78
3.3.4.7 Sending Application Data 78
3.3.4.7.1 Sending Application Data without Proxy 78
3.3.4.7.2 Sending Application Data with Proxy 79
3.3.5 KeepAlive Client Message Processing Events and Sequencing Rules 79
3.3.5.1 KeepAlive-POST-Response Processing 79
3.3.5.1.1 Status Code: 200 (OK) 80
3.3.5.1.1.1 Handshake POST Response Processing 80
3.3.5.1.1.2 Application Data Posted 80
3.3.5.1.2 Status code: 400 (Bad Request) 81
3.3.5.1.3 Status codes: 401 (Unauthorized) / 407 (ProxyAuthentication Required) 81
3.3.5.1.4 All Other Status Codes 81
3.3.5.2 KeepAlive-GET-Response Processing 81
3.3.5.2.1 Status code: 200 (OK) 81
3.3.5.2.1.1 Handshake GET Response Processing 81
3.3.5.2.1.2 Application Data GET Response Processing 82
3.3.5.2.2 Status code: 400 (Bad Request) 82

Copyright © 2014 Microsoft Corporation.

3.3.5.2.3 Status codes: 401 (Unauthorized) and 407 (ProxyAuthentication Required) ... 82
3.3.5.2.4 All Other Status Codes ... 83
3.3.5.3 Sending a KeepAlive-GET-Request ... 83
3.3.5.3.1 Sending Request for Application Data without Proxy ... 83
3.3.5.3.2 Sending Request for Application Data with Proxy ... 83
3.3.6 KeepAlive Client Timer Events ... 84
3.3.6.1 ConnectionEstablishment Timer Event ... 84
3.3.6.2 GetNetworkReceiveIO Timer Event ... 84
3.3.6.3 PostNetworkReceiveIO Timer Event ... 84
3.3.6.4 KeepAlive Timer Event ... 84
3.3.7 KeepAlive Client Other Local Events ... 84
3.3.7.1 Re-Opening the POST Session after a Transport Disconnect ... 84
3.4 KeepAlive Encapsulation Protocol Server Details ... 86
3.4.1 KeepAlive Server Abstract Data Model ... 86
3.4.1.1 Connection State Information ... 87
3.4.2 KeepAlive Server Timers ... 87
3.4.2.1 ConnectionEstablishment Timer ... 87
3.4.2.2 IdleConnection Timer ... 87
3.4.2.3 KeepAlive Timer ... 87
3.4.3 KeepAlive Server Initialization ... 87
3.4.3.1 Protocol Initialization ... 87
3.4.3.2 KeepAlive Listener ... 88
3.4.4 KeepAlive Server Higher-Layer Triggered Events ... 88
3.4.4.1 Closing a KeepAlive Connection ... 88
3.4.4.2 Closing a POST Session ... 88
3.4.4.3 Sending Application Data ... 88
3.4.5 KeepAlive Server Message Processing Events and Sequencing Rules ... 89
3.4.5.1 GET Session Processing ... 89
3.4.5.1.1 Receiving a KeepAlive-GET-Request (Handshake) ... 89
3.4.5.1.1.1 Handshake GET Response Processing ... 90
3.4.5.1.1.2 Sending a KeepAlive-GET-Response with Status code 400 ... 90
3.4.5.1.2 Receiving a KeepAlive-GET-Request for Application Data ... 90
3.4.5.2 POST Session Processing ... 91
3.4.5.2.1 Receiving a KeepAlive-POST-Request (KeepAlive Handshake) ... 92
3.4.5.2.2 Receiving a KeepAlive-POST-Request with Application Data ... 92
3.4.5.2.3 Sending a KeepAlive-POST-Response with Status code 200 ... 92
3.4.5.2.3.1 Handshake POST Response Processing ... 92
3.4.5.2.3.2 Application Data POST Response Processing ... 93
3.4.5.2.4 Sending a KeepAlive-POST-Response with Status Code 400 ... 93
3.4.6 KeepAlive Server Timer Events ... 93
3.4.6.1 ConnectionEstablishment Timer Event ... 93
3.4.6.2 IdleConnection Timer ... 93
3.4.6.3 KeepAlive Timer Event ... 94
3.4.7 KeepAlive Server Other Local Events ... 94
3.5 Polling Encapsulation Protocol Client Details ... 94
3.5.1 Polling Client Abstract Data Model ... 94
3.5.1.1 Connection State Information ... 96
3.5.1.2 Proxy State Information ... 98
3.5.1.3 Client State Information ... 98
3.5.2 Polling Client Timers ... 98
3.5.2.1 ConnectionEstablishment Timer ... 98
3.5.2.2 Network Receive IO Timer ... 98

Copyright © 2014 Microsoft Corporation.

3.5.2.3 Polling Encapsulation Timer........ 99
3.5.3 Polling Client Initialization 99
3.5.3.1 Protocol Initialization........ 99
3.5.4 Polling Client Higher-Layer Triggered Events........100
3.5.4.1 Establishing a Polling Encapsulation Connection........100
3.5.4.1.1 Establishing POST Session without Proxy........100
3.5.4.1.2 Establishing POST Session with Proxy 100
3.5.4.2 Closing a Polling Connection........101
3.5.4.3 Sending Application Data 101
3.5.4.3.1 Sending Application Data without Proxy 102
3.5.4.3.2 Sending Application Data through a Proxy 102
3.5.5 Polling Client Message Processing Events and Sequencing Rules........103
3.5.5.1 Polling-POST-Response Processing........103
3.5.5.1.1 Status code: 200 (OK) 103
3.5.5.1.1.1 When ConnectionState is Connecting (last handshake response) 103
3.5.5.1.1.2 When ConnectionState is Established (Receiving Application Data) 104
3.5.5.1.2 Status code: 400 (Bad Request) 105
3.5.5.1.2.1 When PostSessionState is Probing 105
3.5.5.1.2.2 All other PostSessionState States 106
3.5.5.1.3 Status codes: 401 (Unauthorized) and 407 (ProxyAuthentication Required) 106
3.5.5.1.4 All Other Status Codes 106
3.5.5.1.5 Closing the POST Session 106
3.5.5.1.6 Closing a Polling Connection because of Protocol Error 106
3.5.6 Polling Client Timer Events 106
3.5.6.1 ConnectionEstablishment Timer Event 106
3.5.6.2 NetworkReceiveIO Timer Event 107
3.5.6.3 Polling Encapsulation Timer........107
3.5.7 Polling Client Other Local Events 107
3.6 Polling Encapsulation Protocol Server Details........107
3.6.1 Polling Server Abstract Data Model 107
3.6.1.1 Connection State Information........107
3.6.2 Polling Server Timers........108
3.6.2.1 ConnectionEstablishment Timer........108
3.6.3 Polling Server Initialization 108
3.6.3.1 Protocol Initialization........108
3.6.3.2 Polling Encapsulation Listener 108
3.6.4 Polling Server Higher-Layer Triggered Events........108
3.6.4.1 Closing a Polling Connection........108
3.6.4.2 Closing a Polling Session 108
3.6.4.3 Sending Application Data 109
3.6.5 Polling Server Message Processing Events and Sequencing Rules 109
3.6.5.1 Receiving a Polling-POST-Request (Initial Handshake Request) 109
3.6.5.1.1 Sending a Polling-POST-Response with Status code 400 (Handshake).......110
3.6.5.2 Receiving a Polling-POST-Request (Last Handshake Request)........110
3.6.5.2.1 Sending a Polling-POST-Response with Status code 200 (OK) 110
3.6.5.3 Receiving a Polling-POST-Request (After Handshake)........111
3.6.6 Polling Server Timer Events 112
3.6.6.1 ConnectionEstablishment Timer Event 112
3.6.6.2 Polling Encapsulation Timer........112
3.6.7 Polling Server Other Local Events 112
3.7 Secure Tunnel Encapsulation of SSTP Protocol Client Details 112
3.7.1 Secure Tunnel Client Abstract Data Model 112

Copyright © 2014 Microsoft Corporation.

3.7.1.1 Connection State Information 113
3.7.1.2 Proxy State Information 114
3.7.2 Secure Tunnel Client Timers 114
3.7.2.1 ConnectionEstablishment Timer 114
3.7.2.2 NetworkReceiveIO Timer 114
3.7.2.3 KeepAlive Timer 115
3.7.3 Secure Tunnel Client Initialization 115
3.7.3.1 Protocol Initialization 115
3.7.3.2 Secure Tunnel Listener Endpoints 115
3.7.3.3 Timers Started 115
3.7.4 Secure Tunnel Client Higher-Layer Triggered Events 115
3.7.4.1 Establishing a Secure Tunnel Encapsulation Connection 115
3.7.4.1.1 Establishing a Secure Tunnel connection without proxy 115
3.7.4.1.2 Establishing a Secure Tunnel connection with a proxy 116
3.7.4.2 Closing a Secure Tunnel Connection 116
3.7.4.3 Sending Application Data 116
3.7.5 Secure Tunnel Client Message Processing Events and Sequencing Rules 116
3.7.5.1 HTTP Response Processing 116
3.7.5.1.1 Status code: 200 117
3.7.5.1.2 Status code: 400 (Bad Request) 117
3.7.5.1.3 Status codes: 401 (Unauthorized) and 407 (ProxyAuthentication Required) 117
3.7.5.1.4 All Other Status Codes 117
3.7.5.2 Application Data Processing 117
3.7.6 Secure Tunnel Client Timer Events 118
3.7.6.1 ConnectionEstablishment Timer Event 118
3.7.6.2 NetworkReceiveIO Timer Event 118
3.7.6.3 KeepAlive Timer Event 118
3.7.7 Secure Tunnel Client Other Local Events 118
3.8 Secure Tunnel Encapsulation of SSTP Protocol Server Details 118
3.8.1 Secure Tunnel Server Abstract Data Model 118
3.8.2 Secure Tunnel Server Timers 118
3.8.2.1 SSTP KeepAlive Timer 118
3.8.3 Secure Tunnel Server Initialization 119
3.8.3.1 Secure Tunnel Encapsulation Listener 119
3.8.4 Secure Tunnel Higher-Layer Triggered Events 119
3.8.5 Secure Tunnel Server Message Processing Events and Sequence Rules 119
3.8.6 Secure Tunnel Server Timer Events 119
3.8.7 Secure Tunnel Server Other Local Events 119
3.9 SOCKS Encapsulation of SSTP Protocol Client Details 119
3.9.1 SOCKS Client Abstract Data Model 119
3.9.1.1 Connection State Information 120
3.9.1.2 Proxy State Information 121
3.9.2 SOCKS Client Timers 121
3.9.2.1 ConnectionEstablishment Timer 121
3.9.2.2 NetworkReceiveIO Timer 121
3.9.2.3 KeepAlive Timer 121
3.9.3 SOCKS Client Initialization 122
3.9.3.1 SOCKS Protocol Initialization 122
3.9.4 SOCKS Client Higher-Layer Triggered Events 122
3.9.4.1 Establishing a SOCKS Encapsulation Connection 122
3.9.4.1.1 Establishing a SOCKS Encapsulation Connection 122
3.9.4.2 Closing a SOCKS Connection 122

Copyright © 2014 Microsoft Corporation.

3.9.4.3 Sending Application Data 123
3.9.5 SOCKS Client Message Processing Events and Sequencing Rules 123
3.9.5.1 SOCKS Connection Negotiation Processing 123
3.9.5.1.1 Version Identifier Response 123
3.9.5.1.2 Connect Request 124
3.9.5.1.3 Connect Response 124
3.9.5.2 Application Data Processing 124
3.9.6 SOCKS Client Timer Events 124
3.9.6.1 ConnectionEstablishment Timer Event 124
3.9.6.2 NetworkReceiveIO Timer Event 125
3.9.6.3 KeepAlive Timer Event 125
3.9.7 SOCKS Client Other Local Events 125
3.10 SOCKS Encapsulation of SSTP Protocol Server Details 125
3.10.1 SOCKS Server Abstract Data Model 125
3.10.2 SOCKS Server Timers 125
3.10.3 SOCKS Server Initialization 125
3.10.4 SOCKS Server Higher-Layer Triggered Events 125
3.10.5 SOCKS Server Message Processing Events and Sequencing Rules 125
3.10.6 SOCKS Server Timer Events 126
3.10.7 SOCKS Server Other Local Events 126

4 Protocol Examples 127
4.1 HTTP LongLived Encapsulation Examples 127
4.2 HTTP KeepAlive Encapsulation Examples 129
4.3 HTTP Polling Encapsulation Examples 134
4.4 Secure Tunnel Proxy Protocol Examples 142
4.5 SOCKS Proxy 142
4.6 Proxy Authentication using NTLM Example 143

5 Security 147
5.1 Security Considerations for Implementers 147
5.2 Index of Security Parameters 147

6 Appendix A: Product Behavior 148

7 Change Tracking 159

8 Index 160

Copyright © 2014 Microsoft Corporation.

1 Introduction

This document specifies protocols and methodologies to route Simple Symmetric Transport Protocol (SSTP) through firewalls and proxies. SSTP is defined separately in the Simple Symmetric Transport Protocol (SSTP). These protocols and methods are used to traverse firewalls and proxy servers. Multiple protocols are necessary because no one protocol is capable of traversing all firewalls and proxy configurations, because of lack of standards, different implementation characteristics and different transport restrictions common to various firewall and proxy implementations.

The protocols defined in this specification are:

- LongLived Encapsulation Protocol
- KeepAlive Encapsulation Protocol
- Polling Encapsulation Protocol

These three protocols use different forms of HTTP encapsulation and collectively referred to as the HTTP Encapsulation protocols.

Also, the use of two tunneling protocols is described:

- Secure Tunnel Proxy Protocol
- SOCKS Protocol

Collectively, these protocols are known as the HTTP Encapsulation of SSTP protocols.

The focus of this document is the encapsulation of the SSTP protocol, but these protocols could encapsulate any protocol.

The LongLived, KeepAlive and Polling encapsulation protocols provide an alternative transport mechanism to TCP for encapsulating SSTP protocols within HTTP. Using HTTP as a transport allows SSTP application data to seamlessly traverse firewalls and proxies. This is accomplished by wrapping SSTP commands inside of HTTP messages. The main benefit of HTTP encapsulation is that it makes it possible to route data across network topologies that allow HTTP communications that require little or no network configuration changes.

The Secure Tunnel Proxy Protocol and SOCKS Protocol are proxy negotiation protocols based on Internet standard protocols that use TCP as a transport. The benefit of using these industry standard protocols is to allow the SSTP data stream to tunnel through firewalls and proxies. The Secure Tunnel Proxy uses an HTTP protocol, intended for use by SSL, to negotiate a secure tunnel through an HTTP proxy. The SOCKS protocol uses a binary protocol commonly implemented by HTTP servers.

Firewall traversal is accomplished using LongLived, KeepAlive and Polling encapsulation protocols without proxy negotiation. These protocols enable end-to-end communication through firewalls that inspect HTTP traffic or block non-port 80 traffic.

Proxy traversal is accomplished using any of the protocols defined in this specification. These protocols provide a proxy negotiation mechanism. When a proxy is traversed for SSTP communication, clients first establish a connection to a proxy. Proxy negotiation includes a message exchange between client and proxy that includes the target servers name and port number. The proxy then establishes a TCP connection with the target server on the specified port. After successfully negotiating the proxy connection, the proxy transfers the application data between the client and target server. Proxies do not do OSI model Level 3 routing as do firewalls. Instead, data is transferred across two TCP connections at the application layer. For additional security, proxies

Copyright © 2014 Microsoft Corporation.

commonly support proxy authentication which introduces additional headers and message exchanges as part of proxy negotiation.

Clients commonly attempt proxy access serially and use the first encapsulation method that succeeds. This specification documents each of these protocols in detail in subsequent sections.

Sections 1.8, 2, and 3 of this specification are normative and can contain the terms MAY, SHOULD, MUST, MUST NOT, and SHOULD NOT as defined in [RFC2119]. Sections 1.5 and 1.9 are also normative but does not contain those terms. All other sections and examples in this specification are informative.

1.1 Glossary

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

firewall rule
fully qualified domain name (FQDN)
HTTP proxy
Hypertext Transfer Protocol (HTTP)
Secure Sockets Layer (SSL)
Transmission Control Protocol (TCP)
tunnel

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

basic authentication scheme
connection
endpoint
HTTP encapsulation
Internet Assigned Numbers Authority (IANA)
keepalive message
session
Simple Symmetric Transport Protocol (SSTP)
SOCKS proxy

The following terms are specific to this document:

access protocols: A set of protocols that are supported by proxies to enable protocol clients and protocol servers to communicate with and share proxy services. A single proxy can support multiple proxy protocols, such as an HTTP proxy that is configured to support HTTP with proxy headers, secure tunnel proxy, and SOCKS.

timeout: An integer value, measured in seconds, that indicates the duration of an instance of session data.

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

References to Microsoft Open Specification documents do not include a publishing year because links are to the latest version of the documents, which are updated frequently. References to other documents include a publishing year when one is available.

Copyright © 2014 Microsoft Corporation.

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.

[ISO/IEC 7498-1:1994] International Organization for Standardization, "Information technology -- Open Systems Interconnection -- Basic Reference Model: The Basic Model", ISO/IEC 7498-1:1994, June 1996, http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=20269

[MS-GRVSSTP] Microsoft Corporation, "Simple Symmetric Transport Protocol (SSTP)".

[MS-GRVSSTPS] Microsoft Corporation, "Simple Symmetric Transport Protocol (SSTP) Security Protocol".

[RFC1123] Braden, R., "Requirements for Internet Hosts - Application and Support", STD 3, RFC 1123, October 1989, http://www.ietf.org/rfc/rfc1123.txt

[RFC1928] Leech, M., Ganis, M., Lee, Y., et al., "SOCKS Protocol Version 5", RFC 1928, March 1996, http://www.rfc-editor.org/rfc/rfc1928.txt

[RFC1929] Leech, M., "Username/Password Authentication for SOCKS V5", RFC 1929, March 1996, http://www.rfc-editor.org/rfc/rfc1929.txt

[RFC1945] Berners-Lee, T., Fielding, R., and Frystyk, H., "Hypertext Transfer Protocol -- HTTP/1.0", RFC 1945, May 1996, http://www.ietf.org/rfc/rfc1945.txt

[RFC2068] Fielding, R., Gettys, J., Mogul, J., et al., "Hypertext Transfer Protocol -- HTTP/1.1", RFC 2068, January 1997, http://www.ietf.org/rfc/rfc2068.txt

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

[RFC2617] Franks, J., Hallam-Baker, P., Hostetler, J., et al., "HTTP Authentication: Basic and Digest Access Authentication", RFC 2617, June 1999, http://www.ietf.org/rfc/rfc2617.txt

[RFC3986] Berners-Lee, T., Fielding, R., and Masinter, L., "Uniform Resource Identifier (URI): Generic Syntax", STD 66, RFC 3986, January 2005, http://www.ietf.org/rfc/rfc3986.txt

[RFC4559] Jaganathan, K., Zhu, L., and Brezak, J., "SPNEGO-based Kerberos and NTLM HTTP Authentication in Microsoft Windows", RFC 4559, June 2006, http://www.ietf.org/rfc/rfc4559.txt

[RFC5234] Crocker, D., Ed., and Overell, P., "Augmented BNF for Syntax Specifications: ABNF", STD 68, RFC 5234, January 2008, http://www.rfc-editor.org/rfc/rfc5234.txt

[TCPPROXY] Luotonen, A., "Tunneling TCP based protocols through Web proxy servers", February 1998, http://tools.ietf.org/html/draft-luotonen-web-proxy-tunneling-00

1.2.2 Informative References

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

[MS-OFCGLOS] Microsoft Corporation, "Microsoft Office Master Glossary".

[RFC1961] McMahon, P., "GSS-API Authentication Method for SOCKS Version 5", RFC 1961, June 1996, http://www.rfc-editor.org/rfc/rfc1961.txt

Copyright © 2014 Microsoft Corporation.

[RFC2459] Housley, R., Ford, W., Polk, W., and Solo, D., "Internet X.509 Public Key Infrastructure Certificate and CRL Profile", RFC 2459, January 1999, http://www.ietf.org/rfc/rfc2459.txt

[RFC2616] Fielding, R., Gettys, J., Mogul, J., et al., "Hypertext Transfer Protocol -- HTTP/1.1", RFC 2616, June 1999, http://www.ietf.org/rfc/rfc2616.txt

[RFC3143] Cooper, I., and Dilley, J., "Known HTTP Proxy/Caching Problems", RFC 3143, June 2001, http://www.rfc-editor.org/rfc/rfc3143.txt

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

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

[SSLPROXY] Luotonen, A., "Tunneling SSL Through a WWW Proxy", March 1997, http://tools.ietf.org/html/draft-luotonen-ssl-tunneling-03

1.3 Protocol Overview (Synopsis)

Hypertext Transfer Protocol (HTTP), as described in [RFC1945], is both an application layer protocol and a transport. HTTP facilitates communication between clients and servers over multi-tier network architectures that use firewalls and proxies.

Firewalls allow a client and server to communicate directly with one another as long as they use a protocol that has been explicitly allowed by the **firewall rules**. Firewalls are used to enforce corporate policies and can inspect HTTP payload content. Firewalls typically limit protocol use using two main schemes. The first is based on limiting the destination machines to well-known port addresses. The second scheme inspects the packets flowing over a TCP [RFC793] **connection (1)** to validate that the connection is sending packets that are legal for the specified protocol and for the defined firewall policies. Firewalls are generally not detectable.

Proxies typically provide value-added services, such HTML caching, authentication, and auditing services. Using a layered approach, a proxy works in concert with the firewall to provide and enforce protocol specific rules. In the OSI model described in [ISO/IEC 7498-1:1994], proxies route traffic Layer 7, the application layer. The impact of Layer 7 routing is that proxies introduce a tiered architecture, and the proxy requires an extra hop for all client-server traffic. There are many different proxy types and **access protocols**, especially for HTTP. Firewall architectures typically use a small subset of the available proxy access protocols.

HTTP uses the **Transmission Control Protocol (TCP)** as its underlying transport. Clients establish TCP connections with servers listening on the well-known TCP port 80. Port 80/TCP is the default port assigned to HTTP by the **Internet Assigned Numbers Authority (IANA)**. See the following figure.

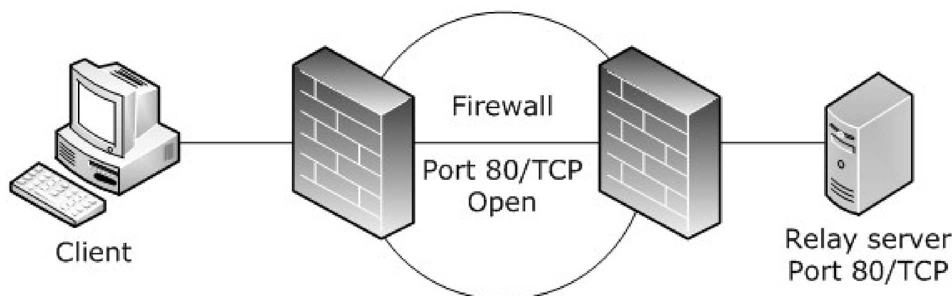

Figure 1: Firewall infrastructure

Copyright © 2014 Microsoft Corporation.

Depending on the network infrastructure, clients can be blocked from establishing communication directly with servers. Within such infrastructures, firewalls and proxies can provide the only means of communication with a remote server. If a client is unable to communicate directly with a server and instead tries to establish communication with the server via a proxy, it first opens a TCP connection with the proxy using the proxy's well known port. The client then exchanges information about the target server with the proxy, such as its **fully qualified domain name (FQDN)**, IP address, and port number. Upon successful completion of the proxy negotiation handshake, the proxy opens a TCP connection with the target server on behalf of the client. Target servers typically have no knowledge that they are communicating with a client via a proxy. Ports 80/TCP and 8080/TCP are IANA assigned ports for HTTP. The IANA ports for **Secure Sockets Layer (SSL)** and **SOCKS proxy** are 443/TCP and 1080/TCP respectively. The preceding figure and the following figure show typical firewall and proxy configurations.

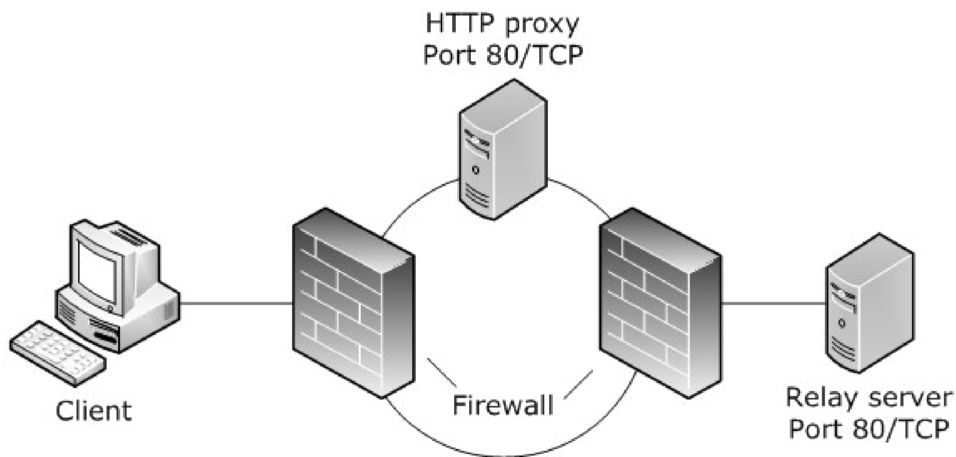

Figure 2: Firewall and proxy infrastructure

There are two HTTP versions in wide use today, HTTP 1.0 [RFC1945] and HTTP 1.1 [RFC2068]. **HTTP encapsulation** of **Simple Symmetric Transport Protocol (SSTP)** is based on HTTP 1.0 because of firewall and proxy traversal dependencies. HTTP 1.0 was chosen because it provides the widest degree of compatibility, which maximizes the chances of establishing an SSTP connection.

HTTP encapsulation of SSTP is designed to specifically encapsulate SSTP. SSTP is documented separately in [MS-GRVSSTP]. SSTP uses TCP as its default transport. A single SSTP connection is layered on a single TCP connection. SSTP allows multiple SSTP **sessions** to flow between clients and servers. Each session represents an independent communication path between two resources. See the following figure.

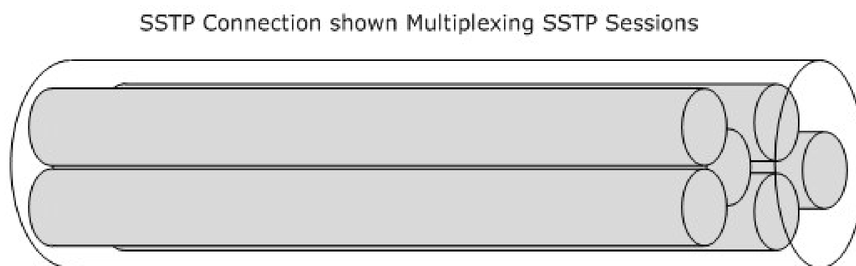

Figure 3: Relationship of SSTP connections and sessions

Copyright © 2014 Microsoft Corporation.

SSTP message data is optionally encrypted by application level protocols using SSTP as a transport. SSTP authentication is provided by SSTP security, a security sub-protocol of SSTP, which is documented separately in [MS-GRVSSTPS].

HTTP encapsulation of SSTP specifies how an SSTP client and server communicate with one another across a network boundary that prevents direct SSTP connectivity on the default 2492/TCP port used by SSTP. These protocols are used when the SSTP protocol fails to establish end-to-end connectivity with a server. This document defines multiple protocols that can be used to navigate through firewalls and proxies. Some protocols, such as Secure Tunnel and SOCKS, negotiate connections with proxies to allow SSTP data to pass through firewalls and proxies. These protocols essentially **tunnel** though intervening firewalls and proxies. Other protocols, such as the HTTP encapsulation protocols, replace the TCP transport with HTTP, to provide a reliable full-duplex connection-oriented stream, using only the HTTP protocol as a transport. Because proxy implementations vary widely, a suite of HTTP encapsulation protocols are defined to overcome common firewall and proxy restrictions.

These protocols deploy a variety of encapsulation and tunneling techniques to route SSTP across a network boundary that only allows HTTP traffic. These transports are less efficient than SSTP over TCP for a number of reasons, such as the extra proxy hop and overhead required for HTTP encapsulation and connection management.

There are three HTTP encapsulation protocols: LongLived, KeepAlive, and Polling and two tunneling protocols, Secure Tunnel and SOCKS. Each of these protocols is optimized for different proxy architectures. The following table summarizes the various transports supporting SSTP connections.

Client and Server Protocols	Functions	Listening Ports Used
SSTP	Used by clients and servers to transport SSTP messages. **Firewall Traversal:** Requires firewall rule to allow SSTP Port 2492/TCP. **Proxy Traversal:** None.	Servers default well known port: 2492/TCP
SSTP over SSL Port	Used by clients to transport SSTP messages to servers when port 2492/TCP is blocked by a firewall/proxy. Uses alternate SSTP port. **Firewall Traversal:** Requires firewall rule to allow SSL Port 443/TCP. **Proxy Traversal:** None. To Proxy SSTP over the SSL Port see Secure Tunnel Proxy Protocol. **Comments:** Supports direct connections between client and server on the SSL port. Data stream is SSTP protocol messages; no SSL protocol is used.	Servers default well known port: SSL 443/TCP
Secure Tunnel Proxy	Used by clients to transport SSTP messages to servers when port 2492/TCP is blocked by a firewall/proxy. Uses **HTTP proxy**. **Firewall Traversal:** See SSTP over SSL Port. **Proxy Traversal:** Requires HTTP Connect Method negotiation with proxy. Also requires a firewall rule to allow traffic originating from the proxy with destination port of 443/TCP. **Comments:** Proxy negotiation message exchange is followed by SSTP command data stream with no additional HTTP or SSL	HTTP proxy default well known port: SSL 443/TCP Servers default well known port: 443/TCP

Copyright © 2014 Microsoft Corporation.

Client and Server Protocols	Functions	Listening Ports Used
	framing. Servers do not detect that connection is with proxy.	
SOCKS	Used by clients to transport SSTP messages to servers when port 2492/TCP is blocked by a firewall or proxy. Uses SOCKS protocol [RFC1928] to pass through firewalls and proxies. **Firewall Traversal:** Requires firewall rule to allow SOCKS Port 1080/TCP. **Proxy Traversal:** Requires SOCKS proxy message exchange. Also requires a firewall rule to allow traffic originating from the proxy with destination port of 2492/TCP. Comments: Proxy negotiation message exchange is followed by SSTP command data stream with no additional SOCKS specific messages. Servers do not detect that connection is with proxy.	SOCKS proxy well known port: 1080/TCP Servers default well known port SSTP: 2492/TCP
HTTP Encapsulation of SSTP	Used by clients to transport SSTP messages to servers when port 2492/TCP is blocked by a firewall or proxy. Used as an HTTP transport to encapsulate SSTP messages. **Firewall Traversal:** Requires a firewall rule to allow HTTP Port 80/TCP. **Proxy Traversal:** Supports proxy traversal through encapsulation of SSTP within HTTP requests and responses. Also requires firewall rule to allow traffic originating from the proxy with destination port of 80/TCP. **Comments:** SSTP data stream is encapsulated using one of the following HTTP encapsulation protocols: LongLived, KeepAlive, Polling. Servers do not detect that connection is with proxy.	HTTP proxy default well known port: 80/TCP HTTP proxy alternate well known: port 8080/TCP Servers default well known port: 80/TCP

1.3.1 HTTP Encapsulation Protocols

This document defines three HTTP encapsulation protocols, LongLived, KeepAlive and Polling. Each of these HTTP encapsulation protocols is designed to replace TCP as the transport for SSTP. Multiple encapsulation protocols exist because of different proxy implementations and lack of proxy standards. All of the HTTP encapsulated connections specified in this document are designed to traverse firewalls and HTTP proxies.

These encapsulation protocols are used to navigate firewalls and proxies when both are working together. When allowed by firewall rules, these encapsulation protocols can be used to connect directly to a target server on port 80/TCP. If a direct connection to the target server is blocked by a firewall, these encapsulation protocols can be used to traverse an HTTP proxy. The server listens on the well-known HTTP port 80/TCP. The HTTP proxy typically listens on the well-known HTTP port 80/TCP or the alternate well known port 8080/TCP.

SSTP is the preferred protocol for client and server communication because it avoids the overhead associated with HTTP encapsulation of SSTP. HTTP Encapsulation of SSTP protocols are adaptable to various types of network topologies that block SSTP traffic on 2492/TCP. Because the encapsulation protocols are optimized for different network topologies, each protocol has its own advantages and constraints. This adaptability comes with the cost of additional protocol overhead, in the form of additional headers, message exchanges and connection management. For performance reasons, described in section 1.3.4, HTTP encapsulation connections are used when direct SSTP connections to a server are blocked and when both Secure Tunnel and SOCKS proxy connections fail.

Copyright © 2014 Microsoft Corporation.

1.3.1.1 HTTP LongLived Encapsulation Connections

LongLived Encapsulation Protocol is an HTTP based protocol used for firewall and proxy traversal. It provides an HTTP transport which can also negotiate and authenticate with HTTP proxies. LongLived Encapsulation is designed to specifically be used with HTTP proxies that do not buffer inbound or outbound proxy traffic. Proxies that buffer data can have a negative impact on the performance of the LongLived protocol. Proxy data buffering is designed to be efficient for typical HTTP request/response exchanges. Buffering is an efficient mechanism for proxies that support a large number of connections. Proxy buffering can cause problems for clients using the LongLived protocol because it streams a large amount of data within each HTTP request and response message. Proxy buffering introduces network latency and can cause network IO stalls. Stalls occur because an application processing SSTP commands on one session block waiting for related SSTP commands to arrive on the other session. If the delay caused by the application reaches a response timeout threshold, a LongLived client will terminate the connection.

LongLived Encapsulation of SSTP uses two half-duplex sessions, a GET session and a POST session, to support the full-duplex requirements of SSTP. The POST session is used by the client to send data to the server while the GET session is used by the server to send data to the client. Together these sessions provide a single full-duplex LongLived connection, which supports a single SSTP Connection. See the following figure.

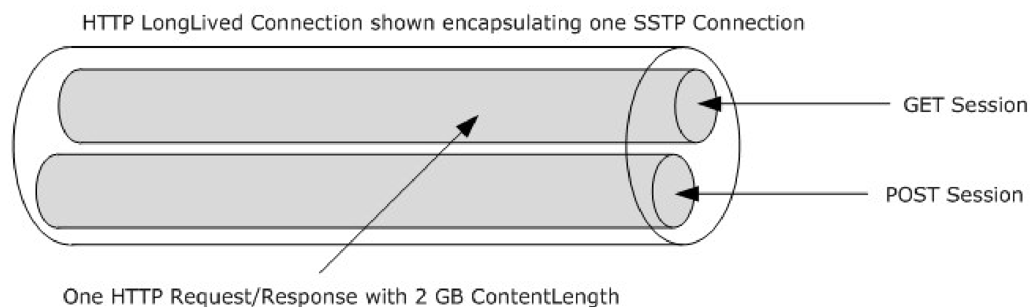

Figure 4: LongLived Encapsulation connection

LongLived protocol information is described as part of the URI [RFC3986]. The client sends an Encapsulation-Echo-String on the POST session, which the server echoes back to the client on the GET session. The LongLived handshake uses the Encapsulation-Echo-String to test for short timeouts common with proxies that are caching. Proxies that use short timeouts when caching close the TCP connection before the Encapsulation-Echo-String has a chance to complete a round trip. No application data is sent or received until after the Encapsulation-Echo-String round trip to avoid flushing the proxy caches, thereby defeating the short timeout test.

Each session can send approximately 2 gigabytes (0x7ffff000 bytes) of data, as specified in the content length header on the initial GET and POST requests. This allows each **endpoint (3)** to independently send up to 0x7ffff000 (2147479552 decimal) octets of data per session. Each session carries one large GET response or POST request with the encapsulated data streaming within the entity body of the corresponding GET response or POST request.

The following figure shows how the GET response and POST request are issued as part of the HTTP LongLived Encapsulation connection establishment.

Copyright © 2014 Microsoft Corporation.

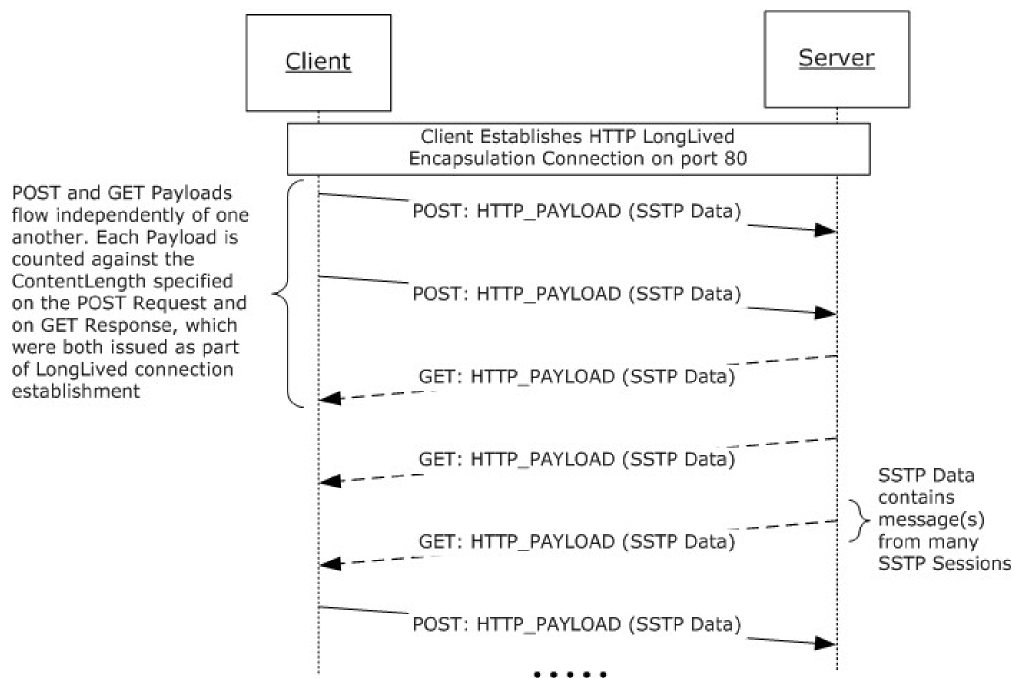

Figure 5: HTTP LongLived Encapsulation message exchange

When a GET response or POST request reaches its content length limit, the client and server closes the session and physical TCP connection. The client opens a new LongLived connection, and new GET response or POST request are issued on the session. This process repeats each time a request/response reaches the specified content length limit.

1.3.1.2 HTTP KeepAlive Encapsulation Connections

KeepAlive Encapsulation Protocol is an HTTP-based protocol used for firewall and proxy traversal. It provides an HTTP transport that also allows authentication with HTTP proxies. KeepAlive encapsulation is designed to be used with HTTP servers and proxies that support persistent connections. KeepAlive connections provide acceptable performance when used with some proxies that buffer inbound and outbound traffic and support persistent connections.

KeepAlive encapsulation uses more frequent HTTP request and response message exchanges to traverse proxies that do not allow LongLived encapsulation connections because of proxy buffering. Multiple request and response messages are sent over the wire, with only a single request or response outstanding at a time, on each session. KeepAlive encapsulation of SSTP uses two half-duplex sessions, GET and POST, to support the full-duplex requirements of SSTP. Each session is a separate TCP connection which, when combined, provide a virtual KeepAlive connection. The POST session is used by the client to send data to the server while the GET session is used by the server to send data to the client. Together these sessions provide a single full-duplex KeepAlive connection, which supports a single SSTP Connection. See the following figure.

Copyright © 2014 Microsoft Corporation.

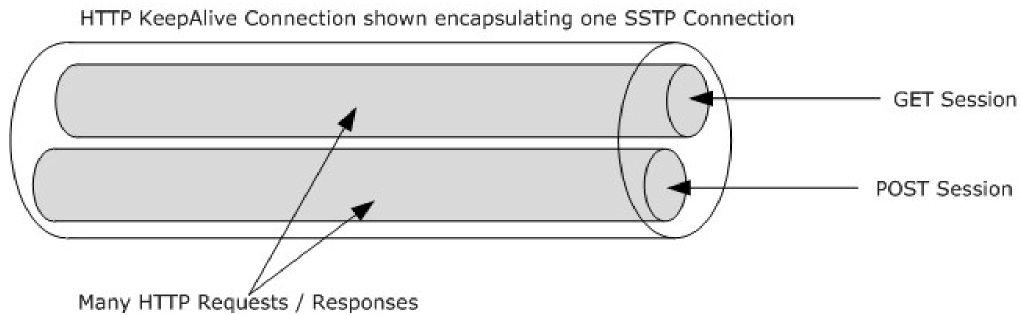

Figure 6: HTTP KeepAlive connection

GET and POST request and response messages are issued independently of one another, each on its own session. The KeepAlive encapsulation protocol sends many requests and responses over the same session.

Each request or response that contains data has a chunk of the SSTP data stream encapsulated in an HTTP entity body, and is sent on the GET/POST session. When the request and response message exchange is complete, the next chunk of the SSTP data stream is sent. In this context, a chunk is a buffer's worth of data, where the buffer size is implementation-defined. The POST and GET sessions are dependent on one another, which means that an SSTP command request can be sent across one session and the SSTP command response can flow across the other session. The following figure illustrates the KeepAlive encapsulation message flow.

HTTP KeepAlive encapsulation of SSTP uses the HTTP 1.0 Connection request header with the KeepAlive connection token, as described in [RFC2068], section 19.7.1. The Connection header, which is the persistent connection extension described in [RFC2068], is not part of the original HTTP 1.0 specification, and is not necessarily honored by all proxies [RFC3143].

If the POST session is closed because of a TCP disconnect, the client is capable of re-opening the POST session and re-binding it to the existing KeepAlive connection. This ability is asymmetrical, and does not apply to the GET session. If the GET session is closed, the client tears down the KeepAlive connection. The client can open a new KeepAlive connection if desired.

Copyright © 2014 Microsoft Corporation.

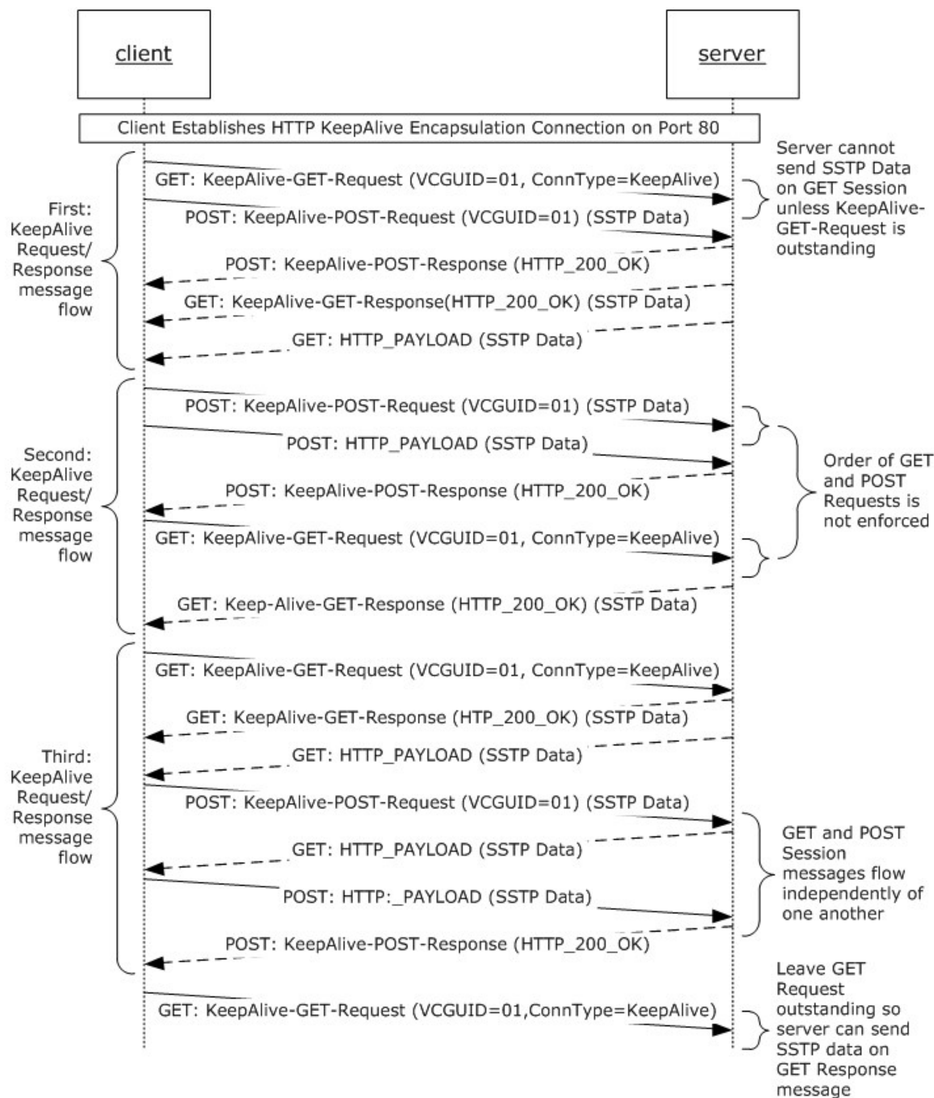

Figure 7: HTTP KeepAlive Encapsulation message flows

1.3.1.3 HTTP Polling Encapsulation Connections

The Polling Encapsulation Protocol is an HTTP based protocol for firewall and proxy traversal. It provides an HTTP transport which can also negotiate and authenticate with HTTP proxies. Polling Encapsulation is designed to interoperate with the widest possible range of proxy implementations. However with this ubiquity comes the cost of performance.

Polling Encapsulation of SSTP differs from the previous HTTP encapsulation protocols in that it uses a single session. A Polling Encapsulation connection is virtualized across many short lived POST request/responses, where each request/response pair of messages uses a separate TCP connection. See the following figure.

Copyright © 2014 Microsoft Corporation.

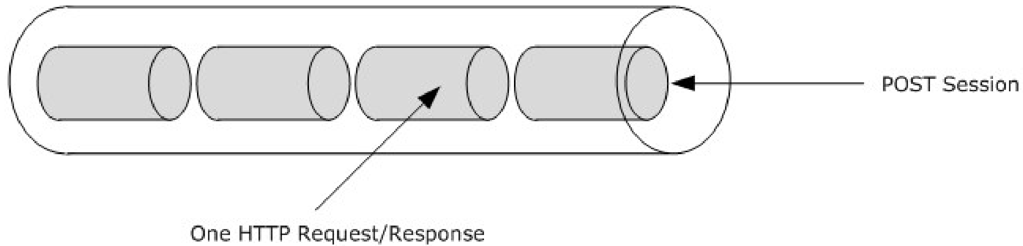

Figure 8: HTTP polling connection

POST requests for an encapsulated connection are associated using a virtual connection identifier. These requests are used by the client to send data to the server while POST responses are used by the server to send data to the client. A new POST request is sent only after the previous POST response is received.

Polling requests use a simple URI [RFC3986] and a minimum number of request headers. Virtual connection information is sent on every request and response. This virtual connection information is embedded in the entity body, preceding the encapsulated messages. The content length header includes the length of virtual connection information as well as the length of the application data (SSTP data stream chunk). The Polling session uses traditional HTTP request/response semantics which means that the session operates in a half-duplex mode. The server can only send data to the client on a POST response, which requires that the client has issued a POST request. The other encapsulation protocols specified in this document are full-duplex. This half-duplex constraint on Polling Encapsulation means the client and server communication streams cannot operate independently of one another. To allow a full-duplex protocol such as SSTP to communicate over this half-duplex session, a polling model is required. Polling solves the challenge of how the server sends a message to the client when the client has no messages to send to the server. In the case where the client has no data to send to the server, the client periodically polls the server via a POST request. This polling request allows the server to send data to the client on the POST response. The following figure shows the Polling Encapsulation message flow.

Copyright © 2014 Microsoft Corporation.

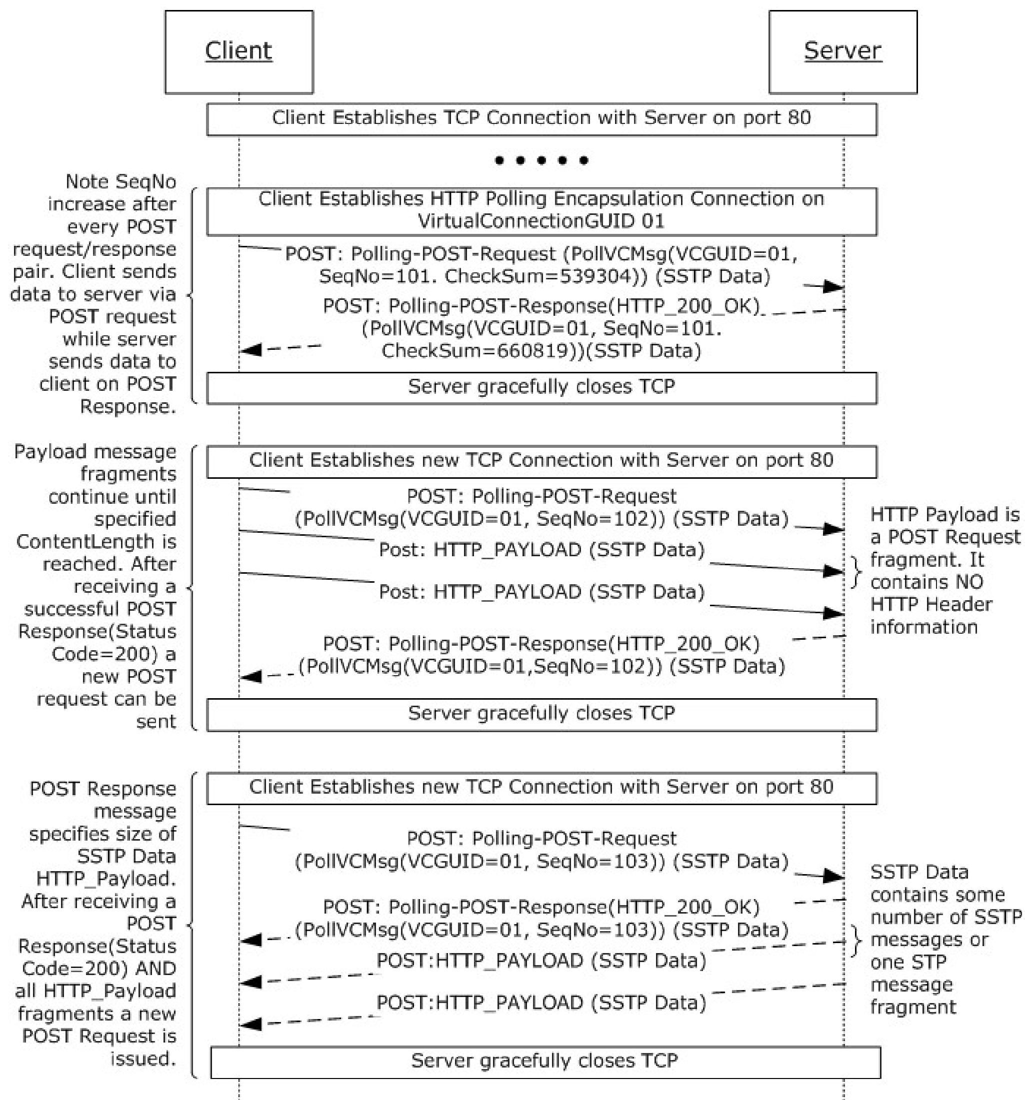

Figure 9: HTTP Polling Encapsulation message flow

1.3.2 Secure Tunnel Connections

The Secure Tunnel Proxy Protocol, also known as the SSL Tunnel Protocol [SSLPROXY], is an internet draft standard described in Secure Tunnel Proxy Protocol [TCPPROXY]. It was originally designed to allow SSL traffic through an HTTP proxy and uses the well-known port 443/TCP. SSL requires a tunnel because traffic is encrypted. Without a tunnel the HTTP proxy would need the client's and server's X.509 keys [RFC2459] to decrypt and parse the SSL stream, which would weaken the security of the system. The Secure Tunnel Proxy Protocol solves this problem by using the HTTP Connect Method [RFC1945] for proxy negotiation. The initial handshake negotiates a tunnel connection with the proxy, before the stream in encrypted. A Secure Tunnel handshake [TCPPROXY], section 3.1, stipulates that once the handshake is finished, all subsequent data is to be ignored by the proxy, with the intent that all data after the plaintext handshake is SSL encrypted. From this point onward the Secure Tunnel Proxy Protocol simply proxies the application data stream

Copyright © 2014 Microsoft Corporation.

between the client and server. This specification substitutes an SSTP data steam for the SSL data stream after completion of the Secure Tunnel handshake.

The Secure Tunnel Proxy Protocol has evolved over time to become a general purpose tunnel mechanism for permitting non-HTTP protocols, such as SSTP, to traverse an HTTP proxy. The following figure shows this tunnel mechanism. The Secure Tunnel Proxy can listen on any port while the Connect Method allows clients to select any target port on the remote server. The application data that follows the Connection Method handshake can be an SSL data stream or any data stream. SSL encryption is optional, but some Secure Tunnel Proxy implementations still attempt to validate the presence of an SSL handshake within the data stream to provide increased security.

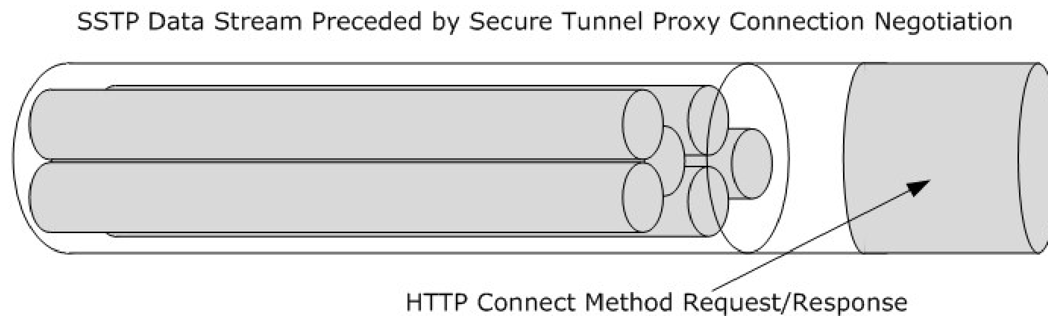

Figure 10: Relationship of SSTP connections/sessions and Secure Tunnel Proxy handshake

In this specification a Secure Tunnel connection is used to explicitly traverse firewall and proxies where direct connections are not possible. Although the Secure Tunnel handshake uses HTTP, the application data following the Secure Tunnel proxy protocol handshake contains the SSTP data stream. The SSTP data stream is possible because the proxy treats all data following the handshake as SSL data which implicitly cannot be decrypted. The SSTP data stream does not need the added security provided by SSL because it has already been authenticated by SSTP Security [MS-GRVSSTPS] and encrypted and integrity protected by the SSTP application layer.

During the Secure Tunnel connection handshake, the client specifies the server's well known port 443/TCP. Although the SSTP 2492/TCP port could have been specified, port 443/TCP is used in favor of 2492/TCP to avoid firewall and proxy configurations that block the Secure Tunnel Proxy connections on ports other than 443/TCP. Although the secure tunnel connection uses port 443/TCP, which is the well-known SSL port, the connection does not use the SSL protocol and the server does not support the SSL handshake on the port.

A server has no knowledge that it is communicating with a Secure Tunnel Proxy. Meanwhile the Secure Tunnel Proxy has no knowledge that it is communicating using the SSTP Protocol [MS-GRVSSTP].

A variant of the Secure Tunnel Proxy Protocol is used by the client for direct end-to-end communication when no intermediate proxy is involved. The client connections to the server use the well-known SSL 443/TCP port to send and receive SSTP protocol messages. In this case, the client requires that the server's SSL Tunnel listener does not support SSL handshake or SSL messages. The client requires that the SSL listener is essentially the same as the SSTP listener except it uses the SSL port. This mode is referred to as SSTP over SSL, which defines a technique, not a protocol.

Copyright © 2014 Microsoft Corporation.

1.3.3 SOCKS Connections

The SOCKS proxy is an internet standard as described in SOCKS Protocol Version 5 [RFC1928]. SOCKS is designed to be a general purpose application proxy which is also known as a circuit level proxy. SOCKS uses its own binary protocol, which is not based on HTTP. The SOCKS protocol allows any application that supports the SOCKS protocol to negotiate proxy connections. A SOCKS connection works similar to a Secure Tunnel Connection, where there is a proxy negotiation handshake followed by the application data stream. From this point forward the SOCKS proxy transfers the application data stream between the client and server. The following figure shows this connection.

The SOCKS Protocol is designed to provide an application neutral proxy protocol for firewall and proxy traversal. SOCKS provides a proxy traversal handshake so tunneled application protocols can negotiate and authenticate with SOCKS proxies. SOCKS connections are NOT used for direct end-to-end communications. Rather, they are used for proxy traversal only. The application data stream following the SOCKS handshake is application defined; in this specification the application data is the SSTP data stream. During the SOCKS handshake, the client specifies the server's FQDN and the SSTP well known port 2492/TCP on the SOCKS Connect request [RFC1928], section 4. Using the connection information that is provided, the SOCKS proxy establishes an SSTP connection with the target server on the well-known port 2492/TCP. This is shown in the following figure.

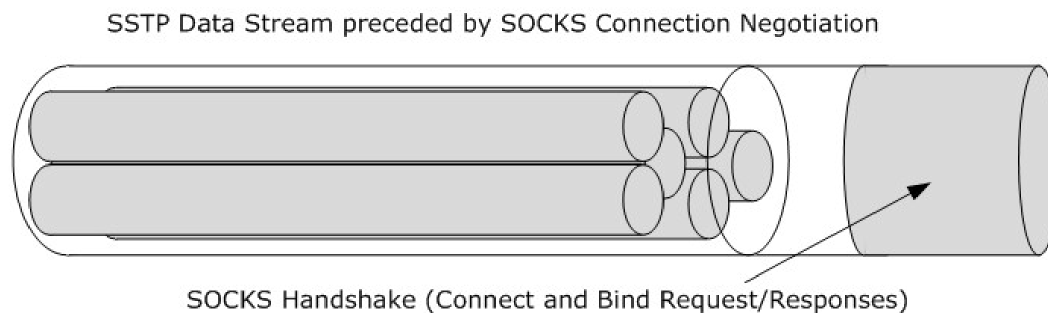

Figure 11: Relationship of SSTP connections/sessions and SOCKS handshake

The target server does not take part in the SOCKS handshake as the handshake is carried out only between the client and SOCK proxy. SOCKS connections persist for the life of the encapsulated protocol connection.

1.3.4 Performance Considerations

Secure Tunnel and SOCKS are the best performing protocols within the HTTP encapsulation of SSTP suite of protocols. They incur minimum handshake overhead during connection establishment. They use TCP as the transport and incur no additional per-message overhead transferring the application data. A SOCKS or Secure Tunnel Proxy connection is created for the life of the SSTP connection. Discounting the extra hop through a proxy, the SOCKS and Secure Tunnel Proxy Protocols provide an efficient conduit for SSTP data. Except for the proxy handshake that occurs just before SSTP connection establishment, there is no protocol overhead when using the SOCKS or Secure Tunnel Proxy Protocol.

LongLived encapsulation of SSTP is the most efficient of the HTTP encapsulation protocols. LongLived uses two TCP connections bound into one virtual LongLived encapsulation connection. Large content lengths allow this protocol to stream application data across both sessions with low protocol overhead. Because it is common for SSTP connections to transfer less than the specified content length of data, there is often no need to create subsequent LongLived connections. Except

Copyright © 2014 Microsoft Corporation.

for the requirement of two TCP connections per LongLived connection, LongLived encapsulation introduces minimal overhead.

KeepAlive is less efficient than LongLived encapsulation, because of an increased framing overhead. There are two main reasons why KeepAlive connections introduce more overhead than LongLived connections. First, KeepAlive encapsulation breaks the application data into small pieces where each chunk is encapsulated within an HTTP request. As a result, HTTP headers take up a larger percentage of the connection's throughput. The second reason is latency; KeepAlive encapsulation allows only one outstanding request per session. Each request waits for the corresponding HTTP response before issuing a new request. HTTP-persistent connections provide a performance boost by reducing the TCP connection overhead, which would otherwise be required for every new request.

HTTP polling encapsulation is the least efficient of all the encapsulation protocols. When other encapsulation protocols fail because of individual firewall and proxy implementations, polling encapsulation often succeeds because of its traditional HTTP request and response semantics and connection behavior. There are three main sources of polling inefficiencies. First, polling encapsulation requires each new HTTP request or response to be a new TCP connection, which implies that TCP connections are established before each request and closed after each response. Second, each SSTP data chunk is encapsulated within a POST request or response message, which increases polling protocol overhead by decreasing, as a percentage, the amount of application data on the wire. Third, polling encapsulation uses one half-duplex session, where other encapsulation protocols use two half-duplex sessions to provide a virtual full-duplex session. Half-duplex mode introduces latency as each endpoint waits for its response. Even when the client has no data to send to the server, it needs to send a POST request to poll for data. To minimize the overhead of polling, in the absence of application data arriving from the server, the client polls less and less frequently using a back-off algorithm.

1.4 Relationship to Other Protocols

The HTTP based protocols depend on the HTTP 1.0 protocol [RFC1945]. Where noted the protocol makes use of additional HTTP 1.1 request headers as specified in HTTP 1.1 [RFC2616]. These headers are ignored by HTTP 1.0 servers, but can be interpreted by proxies. Proxies which accept inbound HTTP 1.0 connections from clients can establish HTTP 1.1 outbound connections to servers. These proxies can use the HTTP 1.1 protocol headers found on the HTTP 1.0 connections or silently drop them. A proxy's exact behavior in this situation is implementation specific because these headers are only treated as hints [RFC2068].

The HTTP Encapsulation of SSTP protocols is layered on top of the TCP protocol and by default uses the IANA-registered ports of 80/HTTP, 443/SSL and 8080/HTTP. When used with the SOCKS protocol for firewall and proxy transversal, the IANA-registered port of 1080/TCP is used. These protocols are used either as a transport or for proxy negotiation to encapsulate or tunnel the SSTP protocol [MS-GRVSSTP]. They are used when direct SSTP connections to a server cannot be established over the default SSTP port. Using HTTP as a transport or the defined tunnel protocols is not as efficient as using TCP because of the significant overhead caused by encapsulation and connection management of the HTTP Encapsulation of SSTP connections. For performance reasons when an SSTP connection is available, the client will always choose it over the equivalent HTTP Encapsulation of SSTP connection.

The following figure shows the relationship between these protocols in the HTTP Encapsulation of SSTP protocol stack:

Copyright © 2014 Microsoft Corporation.

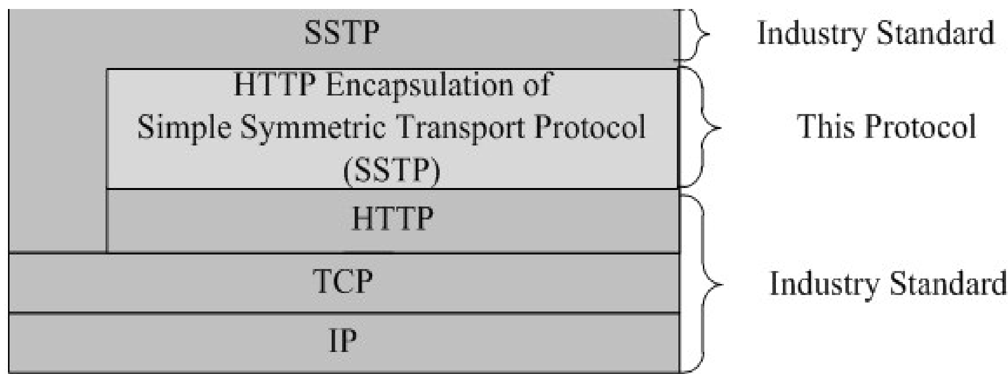

Figure 12: This protocol in relation to other protocols

1.5 Prerequisites/Preconditions

To use the protocols defined by this specification, the client needs to be able to establish a connection to a server over TCP/IP using the well-known HTTP or SSL ports.

In order for one of these protocols to succeed, the client needs to be able to establish a TCP/IP connection through any firewall or proxy that is present. Firewall traversal is usually transparent to the client application. For proxy traversal the client requires proxy connection information to successfully establish proxy connections. This proxy connection information includes the proxy's FQDN or IP address, the proxy's well known port number, and the proxy type (HTTP, SSL or SOCKS). The proxy type provides the client with the correct proxy protocol to use.

If a firewall or proxy is present, then the firewall or proxy device needs to be configured to proxy data from clients to servers. These configuration requirements are generic client-server requirements which include enabling at least one of the proxy protocols (HTTP, SSL, SOCKS) shared in common with the client. The firewall or proxy needs to be able to resolve server names [RFC1123] to IP addresses and establish connections to servers using TCP/IP. The firewall/proxy also needs to be configured to allow connections to at least one of the server's well known ports (HTTP or SSL).

1.6 Applicability Statement

HTTP Encapsulation of SSTP protocols are designed to provide an alternate transport for SSTP. Protocols such as SSTP that use TCP as a transport can fail to establish connections when firewalls, proxies, or routing restrictions, or both are in place. However, it is common for firewalls to allow HTTP traffic via one or more of the methods specified here, while at the same time blocking other application ports. In concert with firewalls, proxies can be inserted between clients and servers to act as gateways. Proxies provide a means to enforce security policies and can provide inspection of application protocol payloads. HTTP Encapsulation of SSTP protocols provide a conduit for SSTP messages to traverse firewalls and proxies that permit HTTP traffic, and to increase the likelihood of successful end-to-end communication.

1.7 Versioning and Capability Negotiation

This document covers versioning issues in the following areas:

- **Supported Transports:** HTTP Encapsulation of SSTP protocols rely on a number of transports to provide reliable end-to-end connectivity as described in section 2.1.

Copyright © 2014 Microsoft Corporation.

- **Protocol Versions:** Secure Tunnel Proxy Protocol relies on HTTP version 1.0 [RFC1945]. SOCKS relies on SOCKS Version 5 [RFC1928]. HTTP Encapsulation protocols rely on HTTP version 1.0 [RFC1945]. Some HTTP version 1.1 Request-Headers [RFC2068], section 5.3, are used for proxy navigation. HTTP proxies that strictly follow HTTP 1.0 will ignore the HTTP 1.1 Request-Headers. HTTP proxies that do not strictly follow HTTP 1.0 [RFC1945] can use HTTP 1.1 Request-Headers as part of proxy traversal requests. The HTTP LongLived and KeepAlive Encapsulation protocols use a version value of "2.0" as specified in the request URI, in sections 2.2.2.1.1.1 and section 2.2.3.1.1.2.The HTTP Polling Encapsulation protocol uses version value of "1.2", as specified in the virtual connection entity body (section 2.2.4.1.3.1.1).

- **Security and Authentication Methods:** SOCKS authentication as described in [RFC1929]. The HTTP Encapsulation protocols and the Secure Tunnel Proxy Protocol support HTTP access authentication, as specified in section 11 of the HTTP 1.0 specification [RFC1945].

- **Localization:** None.

- **Capability Negotiation:** None.

1.8 Vendor-Extensible Fields

HTTP Encapsulation of SSTP protocol uses and specifies the following vendor extensible fields as specified in sections 2.2.1.2.3 and 2.2.1.3.2 respectively.

User-Agent product token value: "Mozilla/4.0 (compatible; MSIE 5.5; Win32)"

HTTP Response Server header server-product-name token value: "Groove-Relay/12.0" or "GrooveRelay/14.0"

HTTP Encapsulation of SSTP protocol supports HTTP extensible headers and header-entity fields as specified in HTTP 1.0 [RFC1945] sections 5.2, 6.2 and 7.1.

1.9 Standards Assignments

HTTP Encapsulation of SSTP uses standard IANA port assignments for HTTP, SSL and SOCKS. These standard port assignments used the IANA assigned ports as specified in the following table.

Parameter	Value
IANA assigned port for SSTP	2492
IANA assigned port for SSL	443
IANA assigned port for HTTP	80
IANA assigned port for HTTP–Alternate (Alternate for port 80)	8080
IANA assigned port for SOCKS	1080

Copyright © 2014 Microsoft Corporation.

2 Messages

2.1 Transport

The HTTP encapsulation of SSTP Protocol uses the HTTP 1.0 Protocol as a transport. While no port has been reserved specifically for The HTTP encapsulation of SSTP Protocol, it uses well known HTTP port 80. The Secure Tunnel Proxy Protocol uses HTTP 1.0 as a transport between a client and an HTTP proxy, and uses TCP as transport between an HTTP proxy and a server listening on port 443. The SOCKS Connection<1> uses TCP as transport between the client and the SOCKS proxy and between the SOCKS proxy and the server listening on port 2492.

2.2 Message Syntax

Section 2.2.1 defines the data types that are common to the LongLived, KeepAlive and Polling encapsulation protocols. Section 2.2.1.1 defines the Encapsulation Data Types. These data types are specific to the HTTP Encapsulation of SSTP protocol. Section 2.2.1.2 defines the HTTP Request Header, section 2.2.1.3 defines the HTTP Response Headers, and section 2.2.1.4 defines the Response Status Code and Reason Phrase. Sections 2.2.2 to 2.2.6 define the message syntax for LongLived Encapsulation, KeepAlive Encapsulation, Polling Encapsulation, Secure Tunnel, and SOCKS Connection, respectively.

This section defines the syntax of the HTTP headers and the messages in the Augmented Backus-Naur Form (ABNF) as specified in [RFC5234]. SOCKS as specified in [RFC1928], a binary protocol, is specified using binary block diagrams.

The following are the three HTTP Encapsulation Protocols:

- LongLived Encapsulation
- KeepAlive Encapsulation
- Polling Encapsulation

In addition to the HTTP Encapsulation of SSTP Protocol, this section also defines the message syntax for the following proxy negotiation protocols:

- Secure Tunnel
- SOCKS

This section also provides examples of the protocols and encapsulation to highlight the message structure as sent to or received from the wire. To make it more readable, the CRLF tokens in the examples of HTTP messages are replaced by a new-line token. An empty line indicates additional CRLF token.

2.2.1 Common HTTP Data Types

2.2.1.1 Encapsulation Data Types

2.2.1.1.1 Virtual-Connection-GUID

The Virtual-Connection-GUID is a GUID that identifies a connection to the server.

Virtual-Connection-GUID = 39(ALPHA / DIGIT)

Copyright © 2014 Microsoft Corporation.

Example: hczn5kctbrpxfgkgxzqs6zmkp9uwvswszvs6f72

2.2.1.1.2 Relay-Server-Name

The Relay-Server-Name defines the host domain name of the server in FQDN format.

Relay-Server-Name = <A legal Internet host domain name>; [RFC1123], section 2.1

An Example of Relay-Server-Name is as following:

Relay-Server-Name = "Server.Domain.com"

2.2.1.1.3 Encapsulation-Echo-String

The Encapsulation-Echo-String is exchanged between a client and a server. This message is exchanged as a payload in the HTTP Request/Response during initial encapsulation connection establishment.

Encapsulation-Echo-String = "GroovePing: 1.0,"ping-data

ping-data = 1*CHAR

Example: GroovePing: 1.0,Ping

2.2.1.1.4 Application-Data

Application-Data refers to a payload consisting of one or more Simple Symmetric Transport Protocol (SSTP) Commands. The SSTP Commands are treated as an opaque block and MUST NOT have any effect on the encapsulation behavior.

Application-Data = 1*(SSTP_COMMAND); section 2.2.1.1.4.1

2.2.1.1.4.1 SSTP_COMMAND

SSTP is an application-layer protocol designed to allow two programs to engage in bi-directional, asynchronous communication. For more information about SSTP protocol command definitions and command exchange sequences, refer to the "Simple Symmetric Transport Protocol (SSTP)" [MS-GRVSSTP], sections 3.1.5, 3.2.5 and 3.3.5.

```
SSTP_COMMAND = (Connect
    / ConnectResponse
    / ConnectAuthenticate
    / ConnectClose
    / Open
    / FanoutOpen
    / OpenResponse
    / Attach
    / AttachResponse
    / AttachAuthenticate
    / Register
    / RegisterResponse
    / Message
    / Data
    / EndMessage
    / Noop
    / Close
    / SessionStatus)
```


Copyright © 2014 Microsoft Corporation.

```
Connect = 1*OCTET
ConnectResponse = 1*OCTET
ConnectAuthenticate = 1*OCTET
ConnectClose        = 1*OCTET
Open               = 1*OCTET
FanoutOpen          = 1*OCTET
OpenResponse          = 1*OCTET
Attach              = 1*OCTET
AttachResponse= 1*OCTET
AttachAuthenticate= 1*OCTET
Register           = 1*OCTET
RegisterResponse = 1*OCTET
Message           = 1*OCTET
Data               = 1*OCTET
EndMessage          = 1*OCTET
Noop               = 1*OCTET
Close               = 1*OCTET
SessionStatus          = 1*OCTET
```

Application-Data can include a fragment of an SSTP_COMMAND. In such cases, the receiving client or the server MUST wait until it receives a complete command before taking further action.

2.2.1.1.5 Server-User-Agent

The Server-User-Agent request header field contains the target server's FQDN.

Server-User-Agent = "UserAgent:" server-name

server-name = Relay-Server-Name

Example1: UserAgent: server.domain.com

2.2.1.2 Request-Header

The HTTP Encapsulation of SSTP Protocol uses several HTTP request headers defined by the HTTP 1.0 protocol (see [RFC1945], section 5.2) and the HTTP 1.1 protocol (see [RFC2068], section 5.3). Most of the HTTP headers used by the HTTP Encapsulation of SSTP Protocol are further constrained in how they can be used. This section defines all of the HTTP headers used by the HTTP Encapsulation of SSTP, and all of these HTTP headers apply only to the HTTP Requests unless otherwise specified.

A client SHOULD NOT send any HTTP header not specified in this section. It is possible that a proxy between a client and a server will add additional headers. For example, a proxy can add an additional header indicating the method it used to serve the request made by the client. In this case, the server or the client can receive HTTP headers not specified in this section. In such cases, the server MUST interpret the header in accordance with the HTTP 1.0 or HTTP 1.1 protocol. If the server receives an HTTP header that is specified in this section but that contains a value that is not specified in this section, the header SHOULD be ignored.

Following is the Common ABNF definition used for the HTTP request headers and HTTP response headers:

SP = " "; Space

CR = %x0D ; carriage return

Copyright © 2014 Microsoft Corporation.

LF = %x0A; linefeed

CRLF = CR LF

NUL = ""; NULL Character '\0'

DIGIT = "0" / "1" / "2" / "3" / "4" / "5" / "6" / "7" / "8" / "9"

OCTET = %x00-FF; 8 bits of data

ALPHA = %x41-5A / %x61-7A; A-Z / a-z

CHAR = %x01-7F; any 7-bit US-ASCII character, excluding NUL

wkday = "Mon" / "Tue" / "Wed" / "Thu" / "Fri" / "Sat" / "Sun"

date-month-year = 2DIGIT SP month SP 4DIGIT

month = "Jan" / "Feb" / "Mar" / "Apr" /"May" / "Jun" / "Jul" / "Aug"/ "Sep" / "Oct" / "Nov" / "Dec"

time = 2DIGIT ":" 2DIGIT ":" 2DIGIT

HTTP-date = wkday "," SP date-month-year SP time SP "GMT"

HTTP-Version = DIGIT "." DIGIT

HTTP-URL = "http://" Relay-Server-Name" ; section 2.2.1.1.2

2.2.1.2.1 Accept

The Accept request-header field specifies the media types (see [RFC1945], section 3.6) the requesting application is willing to accept for a response to the request.

Accept = "Accept:" media-range CRLF; as specified in [RFC1945], section D.2.1

media-range = "*/*"

Example: Accept: */*

2.2.1.2.2 Content-Type

The Content-Type request-header field specifies the media type (see [RFC1945], section 3.6) of the Entity-Body sent to the recipient.

Content-Type = "Content-Type:" media-type CRLF; as specified in [RFC1945], section 10.5

media-type = "application/octet-stream"

This media-type specifies that the response consists of a sequence of OCTETs.

Example: Content-Type: application/octet-stream

2.2.1.2.3 User-Agent

The User-Agent request-header field contains information about the user agent originating the request.

User-Agent = "User-Agent:" 1*product CRLF; as specified in [RFC1945], section 10.15

Copyright © 2014 Microsoft Corporation.

product = 1*CHAR

The client sets the token value to "Mozilla/4.0 (compatible; MSIE 5.5; Win32)", but the implementer of the HTTP Encapsulation of SSTP Protocol can choose to use their product name as token value<2>.

Example: User-Agent: Mozilla/4.0 (compatible; MSIE 5.5; Win32)

2.2.1.2.4 Pragma

The Pragma request-header field contains implementation specific directives (see [RFC1945], section 10.12). The client uses this request-header to indicate that the proxy between the client and the server SHOULD NOT cache the data.

Pragma = «Pragma:» pragma-directive CRLF; as specified in [RFC1945], section 10.12

pragma-directive = "no-cache"

Example: Pragma: no-cache

2.2.1.2.5 Expires

The Expires header field provides the date/time after which the entity SHOULD be considered stale. The server is a data-producing application; any data produced by the server MUST NOT be cached.

Expires = "Expires" ":" HTTP-date CRLF; as specified in [RFC1945], section 10.7

The applications SHOULD be tolerant of non-valid date formats. The "Expires" value of zero or other non-valid date is equivalent to making the entity expire immediately. The client sets the value of this header to zero indicating that the proxy SHOULD not buffer any information in the request.

Example: Expires: 0

2.2.1.2.6 Connection

The Connection header field allows the sender to select options that are desired for the particular connection. This field can be used both as a Request-header field as well as a Response header field. The header field indicates to the proxy that the client is requesting a persistent connection to the server.

Connection = "Connection:" groove-connection-token CRLF ; as specified in [RFC2068], section 14.10

groove-connection-token = "Keep-Alive"

Example: Connection: Keep-Alive

2.2.1.2.7 Host

The Host Request-header field specifies the server of the resource being requested.

```
Host-address = (OCTET "." OCTET "." OCTET "." OCTET)
    / Relay-Server-Name              ; section 2.2.1.1.2
port = 1*DIGIT
Host = "Host:" host-address [":" port] CRLF; (see [RFC2068], section 14.23)
```


Copyright © 2014 Microsoft Corporation.

Example1: Host: 10.150.1.226

Example2: Host: server.domain.net

2.2.1.2.8 Cache-Control

This header is used to request that any intermediate caching server such as a proxy, not cache the content of the HTTP Response, as specified in [RFC2068], section 14.9.

```
Cache-Control = 2(«Cache-Control:» cache-directive CRLF)
cache-directive = "no-cache"; (see [RFC2068], section 14.9.1)
    / "max-age=" delta-seconds; (see [RFC2068], section 14.9.3)
delta-seconds = 1*DIGIT
```

If the "no-cache" directive is specified for Cache-Control, the server MUST NOT use its cache to respond to any request from the client. The "max-age" directive can be set to 0 to force the intermediate caching server to re-validate the cache entry before responding to any request.

Some proxies honor one cache directive while other proxies honor the other. The client MUST send both of the cache directives as part of the request to be compatible with the widest variety of proxies. This header is repeated for the two values of Cache-directive.

Example: Cache-Control: max-age=0 CRLF Cache-Control: no-cache CRLF<3>

2.2.1.2.9 Proxy-Connection

The client adds the proxy-Connection header to the Request-headers requesting the proxy to keep this connection alive. This header is only added if the client detects that a proxy will be making a connection on its behalf.

Proxy-Connection = "Proxy-Connection:" proxy-connection-directive CRLF

proxy-connection-directive = "Keep-Alive"

Example: Proxy-Connection: Keep-Alive CRLF

2.2.1.3 Response Headers

The HTTP Encapsulation of SSTP protocol uses a subset of the HTTP Response Headers defined by the HTTP 1.0 Protocol (see [RFC1945], section 10).

A server SHOULD NOT send any HTTP Response Header not specified in this section. An HTTP Response Header received by the client not specified in this list SHOULD be interpreted in accordance to the HTTP 1.0 Protocol. If the client or the server receives an HTTP header that is specified in this section but that contains a value that is not specified in this section, the header SHOULD be ignored.

2.2.1.3.1 Date

The Date Response header field specifies the date at which the message was created by the sending server.

Date = "Date:" HTTP-date; (see [RFC1945], section 10.6)

Example: Date: Mon, 17 Dec 2007 22:18:31 GMT

Copyright © 2014 Microsoft Corporation.

2.2.1.3.2 Server

The Server Response header field specifies the software that handled the request and originated the response.

Server = "Server:" server-product-name "/" version CRLF; (see [RFC1945], section 10.14)

server-product-name = 1*CHAR

The version uses the <Major Version>.<Minor Version> numbering scheme.

Version = 1*DIGIT "." 1*DIGIT

The server sets the token value to a string identifying the server product. The implementers SHOULD set the token value to a string identifying their product and product version.<4>

2.2.1.4 Response Status Code and Reason Phrase

The HTTP Encapsulation of SSTP does not use all HTTP response codes defined by the HTTP 1.0 Protocol and HTTP 1.1 Protocol. The server MUST always send the Response status codes and the reason phrases from the ones listed in this section.

If a client receives a status code and reason phrase not specified in the following list, it SHOULD be interpreted as a failure status code.

Response-Status-Code-And-Reason-Phrase = Response-Status-Code SP Reason-Phrase

Response-Status-Code = 3DIGIT

Reason-Phrase = *<TEXT, excluding CR, LF>

The following is the list of the Response-Code-And-Reason-Phrase defined by the HTTP Encapsulation of SSTP Protocol:

```
Response-Status-Code-And-Reason-Phrase =
    "200 OK"
    / "302 Moved Temporarily"
    / "304 Not Modified"
    / "400 Bad Request"
    / "401 Unauthorized"
    / "403 Not Valid"
    / "404 Not Found"
    / "407 ProxyAuthentication Required"
    / "408 Request Time-out"
    / "500 Internal Server Error"
    / "501 Not Implemented"
    / "505 HTTP Version Not Supported")
```

2.2.2 LongLived Encapsulation

2.2.2.1 LongLived-GET-Request

The LongLived-GET-Request is sent from a client to a server to retrieve the information identified by the LongLived-GET-Request-URI. The server MUST respond with a LongLived-GET-Response Message (section 2.2.2.3).

Copyright © 2014 Microsoft Corporation.

```
LongLived-GET-Request = LongLived-GET-Request-Line
    LongLived-GET-Request-Required-Headers
    CRLF
LongLived-GET-Request-Line = "GET" SP
    LongLived-GET-Request-URI SP; section 2.2.2.1.1.1
    HTTP-Version CRLF
LongLived-GET-Request-Required-Headers = (
    Accept; section 2.2.1.2.1
    Content-Type; section 2.2.1.2.2
    User-Agent; section 2.2.1.2.3
    Pragma; section 2.2.1.2.4
    Expires; section 2.2.1.2.5
    Host; section 2.2.1.2.7
    Cache-Control); section 2.2.1.2.8
```

2.2.2.1.1 LongLived-GET-Request-URI

The LongLived-GET-Request-URI is a Uniform Resource Identifier (URI) (see [RFC3986]) that identifies the resource upon which to apply the request.

```
LongLived-GET-Request-URI = LongLived-GET-Request-absoluteURI
    /LongLived-GET-Request-relative-path
```

The format of the URI depends on the nature of the request. The LongLived-GET-Request-absoluteURI MUST be used if a client is connecting to a proxy to communicate with a server. The LongLived-GET-Request-relative-path MUST be used if the client is directly connecting to the server.

```
LongLived-GET-Request-relative-path =
     "/" LongLived-Encapsulation-Version; section 2.2.2.1.1.1.1
     "/" Relay-Server-Name; section 2.2.1.1.2
     "/" Virtual-Connection-GUID; section 2.2.1.1.1
     "," LongLived-Encapsulation-Type-Token; section 2.2.2.1.1.1.2
     "," LongLived-Encapsulation-Content-Length; section 2.2.2.1.1.1.3
     ["," LongLived-Encapsulation-Request-ID]; section 2.2.2.1.1.1.4

LongLived-GET-Request-absoluteURI = HTTP-URL; section 2.2.1.2
     LongLived-GET-Request-relative-path
```

Example (LongLived-GET-Request-relative-path):
/2.0/server.domain.net/hczn5kctbrpxfgkgxzqs6zmkp9uwvswszvs6f72,ConnType=LongLived,Content Length=2147479552

Example (LongLived-GET-Request-absoluteURI):
http://server.domain.net/2.0/server.domain.net/hczn5kctbrpxfgkgxzqs6zmkp9uwvswszvs6f72,Conn Type=LongLived,ContentLength=2147479552, ID=ugqrvphxsc2yqfjqh8ijah6crkziz8qrspvh9ja

2.2.2.1.1.1 LongLived-Encapsulation-Version

The LongLived-Encapsulation-Version indicates the version of the Encapsulation and uses a *"<Major Version >.<Minor Version>"* numbering scheme.

LongLived-Encapsulation-Version = 1*DIGIT "." 1*DIGIT

LongLived-Encapsulation-Version MUST be set to 2.0

Copyright © 2014 Microsoft Corporation.

Example: 2.0

2.2.2.1.1.2 LongLived-Encapsulation-Type-Token

The LongLived-Encapsulation-Type-Token defines the type of encapsulation used by the connection. It is defined as following:

LongLived-Encapsulation-Type-Token = "ConnType=LongLived"

Example: ConnType=LongLived

2.2.2.1.1.3 LongLived-Encapsulation-Content-Length

The LongLived-Encapsulation-Content-Length field specifies the maximum number of OCTETs that a server can send to a client as a response to one request. LongLived-Encapsulation-Content-Length MUST be set to 2147479552<5>.

LongLived-Encapsulation-Content-Length = "ContentLength=" 1*DIGIT

Example: ContentLength=2147479552

2.2.2.1.1.4 LongLived-Encapsulation-Request-ID

The client appends LongLived-Encapsulation-Request-ID to the LongLived-Encapsulation-GET-URI to uniquely identify a request on the proxy. This is done to prevent the caching proxies from returning stale data to the client. This token MUST NOT be included if the client is directly connected to the server.

LongLived-Encapsulation-Request-ID = "ID=" 39(ALPHA / DIGIT)

Example: ID=ugqrvphxsc2yqfjqh8ijah6crkziz8qrspvh9ja

2.2.2.1.2 LongLived-GET-Request Example

The following is an example of a LongLived-GET-Request:

```
----------------------------------Message START-----------------------------------
GET
/2.0/server.domain.net/hczn5kctbrpxfgkgxzqs6zmkp9uwvswszvs6f72,ConnType=LongLived,ContentLeng
th=2147479552 HTTP/1.0
Accept: */*
Content-Type: application/octet-stream
User-Agent: Mozilla/4.0 (compatible; MSIE 5.5; Win32)
Host: 10.150.1.226
Pragma: no-cache
Cache-Control: no-cache
Expires: 0
Cache-Control: max-age=0
----------------------------------Message END-------------------------------------
```

2.2.2.2 LongLived-POST-Request

The LongLived-POST-Request is sent from a client to a server and it MUST include the LongLived-Request-Line and the LongLived-Entity-Body.

Copyright © 2014 Microsoft Corporation.

```
LongLived-POST-Request = LongLived-POST-Request-Line
    LongLived-POST-Request-Required-Headers
    CRLF
    LongLived-Entity-Body; section 2.2.2.1.2.3
LongLived-POST-Request-Line = "POST" SP
    LongLived-POST-Request-URI SP; section 2.2.2.1.2.1
    HTTP-Version; (see [RFC1945], section 3.1)
    CRLF
```

The HTTP-Version MUST be set to "HTTP/1.0."

```
LongLived-POST-Request-Required-Headers = (
    Accept; section 2.2.1.2.1
    Content-Type; section 2.2.1.2.2
    User-Agent; section 2.2.1.2.3
    Server-User-Agent; section 2.2.1.1.5
    LongLived-Content-Length; section 2.2.2.1.2.2
    Pragma; section 2.2.1.2.4
    Expires; section 2.2.1.2.5
    Cache-Control); section 2.2.1.2.8
```

2.2.2.2.1 LongLived-POST-Request-URI

The LongLived-POST-Request-URI is a Uniform Resource Identifier (URI) [RFC3986] that identifies the resource upon which to apply the request.

```
LongLived-POST-Request-URI = LongLived-POST-Request-URI-absoluteURI
     / LongLived-POST-Request-URI-relative-path
```

The format of the URI depends on the nature of the request. The LongLived-POST-Request-URI-absoluteURI MUST be used if a client is connecting to a proxy. The LongLived-POST-Request-URI-relative-path URI MUST be used if the client is directly connecting to the server.

```
LongLived-POST-Request-URI-relative-path =
    "/" LongLived-Encapsulation-Version; section 2.2.2.1.1.1.1
    "/" Relay-Server-Name; section 2.2.1.1.2
    "/" Virtual-Connection-GUID; section 2.2.1.1.1
    "," LongLived-Encapsulation-Type-Token; section 2.2.2.1.1.1.2
LongLived-POST-Request-URI-absoluteURI = HTTP-URL; section 2.2.1.2
    LongLived-POST-Request-URI-relative-path
```

Example (LongLived-POST-Request-URI-relative-path):
/2.0/server.domain.net/hczn5kctbrpxfgkgxzqs6zmkp9uwvswszvs6f72,ConnType=LongLived

Example (LongLived-POST-Request-URI-absoluteURI):
http://server.domain.net/2.0/server.domain.net/hczn5kctbrpxfgkgxzqs6zmkp9uwvswszvs6f72,ConnType=LongLived

2.2.2.2.2 LongLived-Content-Length

The LongLived-Content-Length header field specifies the number of OCTETs present in the LongLived-Entity-Body and can be used in the Request-header as well as Response header.

Copyright © 2014 Microsoft Corporation.

LongLived-Content-Length = "Content-Length:" 1*DIGIT

The LongLived-Content-Length value MUST be set to 2147479552 indicating that the Client or the server could send a LongLived-Entity-Body up to 2147479552 OCTETs.

Example: Content-Length: 2147479552

2.2.2.2.3 LongLived-Entity-Body

The LongLived-Entity-Body is sent with a LongLived-POST-Request and LongLived-GET-Response.

```
LongLived-Entity-Body = 1*(Encapsulation-Echo-String; section 2.2.1.1.3
    / Application-Data); section 2.2.1.1.4
    CRLF
```

The LongLived-Entity-Body MUST be fragmented into multiple messages, each of which is either Encapsulation-Echo-String or Application-Data. As part of the protocol handshake, the Encapsulation-Echo-String MUST be sent as the first fragment of LongLived-Entity-Body. Subsequent fragments MUST be set to Application-Data.

2.2.2.2.4 LongLived-POST-Request Example

The following is an example of a LongLived-POST-Request:

```
----------------------------------Message START-----------------------------------
POST /2.0/server.domain.net/hczn5kctbrpxfgkgxzqs6zmkp9uwvswszvs6f72,ConnType=LongLived
HTTP/1.0
Accept: */*
Content-Type: application/octet-stream
User-Agent: Mozilla/4.0 (compatible; MSIE 5.5; Win32)
UserAgent: server.domain.net
Content-Length: 2147479552
Pragma: no-cache
Cache-Control: no-cache
Expires: 0
Cache-Control: max-age=0

GroovePing: 1.0,Ping
----------------------------------Message END------------------------------------
```

2.2.2.3 LongLived-GET-Response

The LongLived-GET-Response is the HTTP response message from the server to the client on the connection on which a LongLived-GET-Request (section 2.2.2.1) was sent.

```
LongLived-GET-Response = Response-Status-Line; section 2.2.2.1.3.1
    LongLived-GET-Response-Required-Headers
    CRLF
    LongLived-Entity-Body; section 2.2.2.1.2.3
LongLived-GET-Response-Required-Headers = (
    Date; section 2.2.1.3.1
    Server; section 2.2.1.3.2
    Connection; section 2.2.1.2.6
    LongLived-GET-Response-Content-Length); section 2.2.2.3.2
```


Copyright © 2014 Microsoft Corporation.

2.2.2.3.1 Response-Status-Line

The Response-Status-Line is the first line sent from the server to the client and it consists of the protocol version followed by numeric status code and its corresponding reason phrase (see [RFC1945], section 6.1).

```
Response-Status-Line = HTTP-Version SP; (see [RFC1945], section 3.1)
    Response-Status-Code-And-Reason-Phrase; section 2.2.1.4
    CRLF
```

The HTTP-Version MUST be set to "HTTP/1.0".

Example: HTTP/1.0 200 OK

2.2.2.3.2 LongLived-GET-Response-Content-Length

The LongLived-GET-Response-Content-Length header field specifies the number of OCTETs present in the LongLived-Entity-Body and can be used in the Request-header as well as Response header.

```
LongLived-GET-Response-Content-Length = "Content-Length:" 1*DIGIT
```

The LongLived-GET-Response-Content-Length value MUST be set to one of two values:

- Value equals 2147479552.
- Value equals 0.

A LongLived-GET-Response-Content-Length value equal to 2147479552 indicates that the Client or the server could send a LongLived-Entity-Body up to 2147479552 OCTETs.

Example: Content-Length: 2147479552

A LongLived-GET-Response-Content-Length value equal to 0 indicates that the Client or the server MUST not send a LongLived-Entity-Body.

Example: Content-Length: 0

2.2.2.3.3 LongLived-GET-Response Example

The following is an example of a LongLived-GET-Response:

```
----------------------------------Message START-----------------------------------
HTTP/1.0 200 OK
Date: Wed, 26 Dec 2007 20:31:28 GMT
Server: Groove-Relay/12.0
Connection: Keep-Alive
Content-Length: 2147479552

GroovePing: 1.0,Ping

----------------------------------Message END-------------------------------------
```


Copyright © 2014 Microsoft Corporation.

2.2.2.4 LongLived-POST-Response

The LongLived-POST-Response refers to the response message from the server to the client on the connection on which a Request with POST Method was sent. The HTTP Encapsulation of SSTP Protocol MUST NOT send any response on this connection during normal operation (when no errors have occurred). The proxy or server can send this response on the connection if there is an error during connection establishment.

```
LongLived-POST-Response = Response-Status-Line; section 2.2.2.1.3.1
    LongLived-POST-Response-Required-Headers
    CRLF
LongLived-POST-Response-Required-Headers = (
    Date; section 2.2.1.3.1
    Server; section 2.2.1.3.2
    Connection; section 2.2.1.2.6
    LongLived-POST-Response-Content-Length); section 2.2.2.1.4.1
```

2.2.2.4.1 LongLived-POST-Response-Content-Length

The LongLived-POST-Response-Content-Length specifies the number of OCTETs present in the LongLived-POST-Response as Entity-Body.

LongLived-POST-Response-Content-Length = "Content-Length: 0" CRLF

Example: Content-Length: 0

2.2.3 KeepAlive Encapsulation

2.2.3.1 KeepAlive-GET-Request

The KeepAlive-GET-Request message is sent from a client to a server and it MUST include the KeepAlive-GET-Request-Line and the KeepAlive-GET-Request-Required-Headers.

```
KeepAlive-GET-Request = KeepAlive-GET-Request-Line
    KeepAlive-GET-Request-Required-Headers
    [KeepAlive-GET-Request-Other-Headers]
    CRLF
KeepAlive-GET-Request-Line = "GET" SP
    KeepAlive-Request-URI SP; section 2.2.2.2.1.1
    HTTP-Version; (see [RFC1945], section 3.1)
    CRLF
```

The HTTP-Version MUST be set to "HTTP/1.0."

```
KeepAlive-GET-Request-Required-Headers = (
    Accept; section 2.2.1.2.1
    Content-Type; section 2.2.1.2.2
    User-Agent; section 2.2.1.2.3
    Pragma; section 2.2.1.2.4
    Expires; section 2.2.1.2.5
    Host; section 2.2.1.2.7
    Connection; section 2.2.1.2.6
    Cache-Control); section 2.2.1.2.8
```


Copyright © 2014 Microsoft Corporation.

Note that the following header is only present in case a client is connecting to a proxy.

```
KeepAlive-GET-Request-Other-Headers = Proxy-Connection; section 2.2.1.2.9
```

2.2.3.1.1 KeepAlive-Request-URI

The KeepAlive-Request-URI is a Uniform Resource Identifier (URI) as specified in [RFC3986] that identifies the resource upon which to apply the request.

```
KeepAlive-Request-URI = KeepAlive-Request-absoluteURI
    / KeepAlive-Request-relative-path
```

The format of the URI depends on the nature of the request. The KeepAlive-Request-absoluteURI MUST be used if a proxy is making the connection to the server on behalf of the client. The KeepAlive-Request-relative-path URI MUST be used if the client is directly connecting to the server.

```
KeepAlive-Request-relative-path =
    "/" KeepAlive-Encapsulation-Version; section 2.2.2.2.1.1.2
    "/" Relay-Server-Name; section 2.2.1.1.2
    "/" Virtual-Connection-GUID; section 2.2.1.1.1
    "," KeepAlive-Encapsulation-Type-Token; section 2.2.2.2.1.1.1
    ["," KeepAlive-Encapsulation-Request-ID]; section 2.2.2.2.1.1.3
KeepAlive-Request-absoluteURI = HTTP-URL; section 2.2.1.2

    KeepAlive-Request-relative-path
```

Example (KeepAlive-Request-relative-path):

/2.0/server.domain.net/kicxp8rrgwqdwhf7c6xsgbagmcdnxm9phtvbj5a,ConnType=KeepAlive

Example (KeepAlive-Request-absoluteURI):
http://server.domain.net/2.0/server.domain.net/kicxp8rrgwqdwhf7c6xsgbagmcdnxm9phtvbj5a,ConnType=KeepAlive,ID=ugqrvphxsc2yqfjqh8ijah6crkziz8qrspvh9ja

2.2.3.1.1.1 KeepAlive-Encapsulation-Type-Token

The KeepAlive-Encapsulation-Type-Token defines the type of encapsulation used by the connection. It is defined as follows:

KeepAlive-Encapsulation-Type-Token = "ConnType=KeepAlive"

Example: ConnType=KeepAlive

2.2.3.1.1.2 KeepAlive-Encapsulation-Version

The Encapsulation-Version indicates the version of the encapsulation and uses a "<Major Version >.<Minor Version>" numbering scheme.

KeepAlive-Encapsulation-Version = 1*DIGIT "." 1*DIGIT

Encapsulation-Version MUST be set to 2.0

Example: 2.0

Copyright © 2014 Microsoft Corporation.

2.2.3.1.1.3 KeepAlive-Encapsulation-Request-ID

The client appends KeepAlive-Encapsulation-Request-ID to a KeepAlive-Request-URI to uniquely identify a request on the proxy. This is done to prevent the caching proxies from returning stale data to the client. This token MUST NOT be included if the client is directly connected to the server. This token is only included for the KeepAlive-GET-Request-Line.

KeepAlive-Encapsulation-Request-ID = "ID=" 39(ALPHA / DIGIT)

Example: ID=ugqrvphxsc2yqfjqh8ijah6crkziz8qrspvh9ja

2.2.3.1.2 KeepAlive-GET-Request Example

The following is an example of complete KeepAlive-GET-Request:

```
---------------------------------Message START---------------------------------
GET /2.0/server.domain.net/kicxp8rrgwqdwhf7c6xsgbagmcdnxm9phtvbj5a,ConnType=KeepAlive
HTTP/1.0
Accept: */*
Content-Type: application/octet-stream
User-Agent: Mozilla/4.0 (compatible; MSIE 5.5; Win32)
Host: 10.150.1.226
Pragma: no-cache
Cache-Control: no-cache
Expires: 0
Connection: Keep-Alive
Cache-Control: max-age=0
---------------------------------Message END-------------------------------------
```

2.2.3.2 KeepAlive-POST-Request

The KeepAlive-POST-Request is sent from the client to the server and it MUST include the KeepAlive-POST-Request line, the KeepAlive-POST-Required-Headers, and the KeepAlive-POST-Request-Entity-Body.

```
KeepAlive-POST-Request = KeepAlive-POST-Request-Line
    KeepAlive-POST-Request-Required-Headers
    [KeepAlive-POST-Request-Other-Headers]
    CRLF
    KeepAlive-Entity-Body; section 2.2.2.2.2.2
KeepAlive-POST-Request-Line = "POST" SP
    KeepAlive-Request-URI SP; section 2.2.2.2.1.1
    HTTP-Version; (see [RFC1945],  section 3.1)
    CRLF
The HTTP-Version MUST be set to "HTTP/1.0."
KeepAlive-POST-Request-Required-Headers = (
    Accept; section 2.2.1.2.1
    Content-Type; section 2.2.1.2.2
    User-Agent; section 2.2.1.2.3
    Server-User-Agent; section 2.2.1.1.5
    KeepAlive-Content-Length; section 2.2.2.2.2.1
    Pragma; section 2.2.1.2.4
    Expires; section 2.2.1.2.5
    Connection; section 2.2.1.2.6
    Cache-Control); section 2.2.1.2.8
```


Copyright © 2014 Microsoft Corporation.

Note that the following header is only present when a client is connecting to a proxy.

KeepAlive-POST-Request-Other-Headers = Proxy-Connection; section 2.2.1.2.9

2.2.3.2.1 KeepAlive-Content-Length

The Content-Length header field specifies the number of OCTETs present in the Entity-Body.

KeepAlive-Content-Length = "Content-Length" ":" 1*DIGIT

Content-Length of greater than or equal to 0 is a valid value.

2.2.3.2.2 KeepAlive-Entity-Body

The Entity-Body is sent as part of the KeepAlive-POST-Request and KeepAlive-GET-Response.

```
KeepAlive-Entity-Body = Encapsulation-Echo-String; section 2.2.1.1.3
    / Application-Data; section 2.2.1.1.4
```

As part of the protocol handshake, the Encapsulation-Echo-String MUST be sent as the first fragment of LongLived-Entity-Body. Subsequent fragments MUST be set to Application-Data.

2.2.3.2.3 KeepAlive-POST-Request

The following is an example of a complete KeepAlive-POST-Request:

```
----------------------------------Message START-----------------------------------POST
/2.0/server.domain.net/kicxp8rrgwqdwhf7c6xsgbagmcdnxm9phtvbj5a,ConnType=KeepAlive HTTP/1.0
Accept: */*
Content-Type: application/octet-stream
User-Agent: Mozilla/4.0 (compatible; MSIE 5.5; Win32)
UserAgent: server.domain.net
Connection: Keep-Alive
Content-Length: 22
Pragma: no-cache
Cache-Control: no-cache
Expires: 0
Cache-Control: max-age=0

GroovePing: 1.0,Ping

----------------------------------Message END-------------------------------------
```

2.2.3.3 KeepAlive-GET-Response

The KeepAlive-GET-Response is sent from the server to the client in response to the KeepAlive-GET-Request (section 2.2.3.1).

```
KeepAlive-GET-Response = Response-Status-Line; section 2.2.2.1.3.1
    KeepAlive-GET-Response-Required-Headers
    CRLF
             KeepAlive-Entity-Body; section 2.2.2.2.2.2
KeepAlive-GET-Response-Required-Headers = (
    Date; section 2.2.1.3.1
    Server; section 2.2.1.3.2
```


Copyright © 2014 Microsoft Corporation.

```
    Connection; section 2.2.1.2.6
    KeepAlive-Content-Length); section 2.2.2.2.2.1
```

2.2.3.3.1 KeepAlive-GET-Response Example

The following is an example of KeepAlive-GET-Response message:

```
----------------------------------Message START-----------------------------------
HTTP/1.0 200 OK
Date: Wed, 26 Dec 2007 19:50:26 GMT
Server: Groove-Relay/12.0
Connection: Keep-Alive
Content-Length: 15

GroovePing: 1.0,Ping

----------------------------------Message END-------------------------------------
```

2.2.3.4 KeepAlive-POST-Response

The KeepAlive-POST-Response is sent from the server to the client on the connection on which a KeepAlive-POST-Request (section: 2.2.3.2) was sent.

```
KeepAlive-POST-Response = Response-Status-Line; section 2.2.2.1.3.1
    KeepAlive-POST-Response-Required-Headers
    CRLF
              [KeepAlive-POST-Response-Entity-Body]
KeepAlive-POST-Response-Required-Headers = (
    Date; section 2.2.1.3.1
    Server; section 2.2.1.3.2
    Connection; section 2.2.1.2.6
    KeepAlive-Content-Length); section 2.2.2.2.2.1
```

2.2.3.4.1 KeepAlive-POST-Response-Entity-Body

The KeepAlive-POST-Response-Entity-Body is sent from the server to the client with a KeepAlive-POST-Response.

KeepAlive-POST-Response-Entity-Body = [KeepAlive-POST-Response-No-Data] ; section 2.2.3.4.1.1

The server MUST set the Entity-Body of the first POST Response to KeepAlive-POST-Response-No-Data. The subsequent POST Responses MUST NOT include KeepAlive-POST-Response-Entity-Body.

2.2.3.4.1.1 KeepAlive-POST-Response-No-Data

The KeepAlive-POST-Response-Entity-Body is only sent as the entity body of the first message sent by the server.

KeepAlive-POST-Response-No-Data = "<HTML></HTML>"

Copyright © 2014 Microsoft Corporation.

2.2.3.4.2 KeepAlive-POST-Response Example

Following is an example of a KeepAlive-POST-Response message:

```
---------------------------------Message START----------------------------------
HTTP/1.0 200 OK
Date: Wed, 26 Dec 2007 19:50:26 GMT
Server: Groove-Relay/12.0
Connection: Keep-Alive
Content-Length: 15

<HTML></HTML>

---------------------------------Message END------------------------------------
```

2.2.4 Polling Encapsulation

2.2.4.1 Polling-POST-Request

The Polling-POST-Request is sent from the client to the server and it includes the request line, the request-header, and the Entity-Body.

```
Polling-POST-Request = Polling-POST-Request-URI
    Polling-Request-Headers
    CRLF
    Polling-Request-Entity-Body; section 2.2.2.3.1.3
Polling-POST-Request-URI = "POST" SP
    Polling-Request-URI SP; section 2.2.2.3.1.1
    HTTP-Version; (see [RFC1945],  section 3.1)
    CRLF
```

The HTTP-Version MUST be set to "HTTP/1.0."

```
Polling-Request-Headers = (
    Accept; section 2.2.1.2.1
    Content-Type; section 2.2.1.2.2
    User-Agent; section 2.2.1.2.3
    Polling-Content-Length; section 2.2.2.3.1.2
    Pragma; section 2.2.1.2.4
    Expires; section 2.2.1.2.5
    Host; section 2.2.1.2.7
    Cache-Control); section 2.2.1.2.8
```

2.2.4.1.1 Polling-Request-URI

The Polling-Request-URI identifies the resource on the server upon which to apply the request.

```
Polling-Request-URI = Polling-Request-absoluteURI
    / Polling-Request-relative-path
```

The format of the URI depends on the nature of the request. The Polling-Request-absoluteURI MUST be used if a proxy is making the connection to the server on behalf of the client. The Polling-Request-relative-path MUST be used if the client is directly connecting to the server.

Copyright © 2014 Microsoft Corporation.

Polling-Request-relative-path = "/"

Polling-Request-absoluteURI = HTTP-URL

The followings are the examples Polling-Request-URI:

Example (Polling-Request-absoluteURI) = http://server.domain.com

Example (Polling-Request-relative-path) = /

2.2.4.1.2 Polling-Content-Length

The Polling-Content-Length header field specifies the number of OCTETs present in the Entity-Body including the Polling-Virtual-Connection-Message and Application-Data.

Polling-Content-Length = "Content-Length:" 1*DIGIT

Polling-Content-Length is equal to the length of Polling-Virtual-Connection-Message plus the length of the Application_Data. The maximum length of Polling-Content-Length header is 32768 OCTETs.

2.2.4.1.3 Polling-Request-Entity-Body

The Polling-Request-Entity-Body is sent from the client to the server with an HTTP POST Request.

```
Polling-Request-Entity-Body = Polling-Virtual-Connection-Message; section 2.2.2.3.1.3.1
    / Application-Data; section 2.2.1.1.4
```

It is possible for the Polling-Request-Entity-Body to contain only the Polling-Virtual-Connection-Message when the client has no data to transfer to the server.

The value in the Polling-Content-Length header field MUST include the size of both the Polling-Virtual-Connection-Message and the Application-Data.

2.2.4.1.3.1 Polling-Virtual-Connection-Message

The Polling-Virtual-Connection-Message is the additional data pre-pended to the Application-Data to uniquely identify the Entity-Body on a client and a server.

```
Polling-Virtual-Connection-Message =
    Polling-Encapsulation-Version NUL; section 2.2.2.3.1.3.1.1
    Relay-Server-URL NUL; section 2.2.2.3.1.3.1.4
    Virtual-Connection-GUID NUL; section 2.2.1.1.1
    Sequence-Number NUL; section 2.2.2.3.1.3.1.2
    Checksum NUL; section 2.2.2.3.1.3.1.3
```

2.2.4.1.3.1.1 Polling-Encapsulation-Version

The Polling-Encapsulation-Version indicates the version of the Encapsulation and uses a *"<Major Version >.<Minor Version>"* numbering scheme.

Polling-Encapsulation-Version = 1*DIGIT "." 1*DIGIT

Polling-Encapsulation-Version MUST be set to 1.2

Example: 1.2

Copyright © 2014 Microsoft Corporation.

2.2.4.1.3.1.2 Sequence-Number

Each message sent using the HTTP Polling Encapsulation includes a sequence number. The Sequence-Number starts from 0 and is incremented for every message sent thereafter on the same virtual connection.

Sequence-Number = 1*DIGIT

2.2.4.1.3.1.3 Checksum

The checksum field in the Polling-Virtual-Connection-Message provides the receiving device with a way to validate the consistency of the received data. The checksum is computed strictly using the Application-Data. Any Polling-Virtual-Connection-Message or any of the other HTTP request/response headers MUST NOT be included when computing the checksum.

Checksum = 1*DIGIT

The following pseudo-code specifies the checksum generation:

On First SignedByte of Application-Data initialize value of CheckSum:

CheckSum = 0

For each subsequent SignedByte in Application Data:

Checksum = Checksum + ((SignedByte + 1) * (index of SignedByte + 1))

2.2.4.1.3.1.4 Relay-Server-URL

The Relay-Server-URL is a unique identifier for the server device. The Relay-Server-URL is specified with the "grooveDNS" schema.

```
Relay-Server-URL = "grooveDNS://"
    Relay-Server-Name; section 2.2.1.1.2
```

Example: grooveDNS://server.domain.net

2.2.4.1.4 Polling-POST-Request Example

An example of Polling-Request without any Application-Data is as follows:

```
---------------------------------Message START----------------------------------
POST / HTTP/1.0
Accept: */*
Content-Type: application/octet-stream
User-Agent: Mozilla/4.0 (compatible; MSIE 5.5; Win32)
Host: 10.150.1.226
Content-Length: 79
Pragma: no-cache
Cache-Control: no-cache
Expires: 0
Cache-Control: max-age=0

1.2.grooveDNS://server.domain.net.a5s2fj8q55cxne2v4wr48ad9ciffsznzq9apczi.0.0.
---------------------------------Message END------------------------------------
```


Copyright © 2014 Microsoft Corporation.

2.2.4.2 Polling-POST-Response

The Polling-POST-Response is the response message from the server to the client in response to the Polling-POST-Request (see section 2.2.4.1).

```
Polling-POST-Response = Response-Status-Line; section 2.2.2.1.3.1
    Polling-POST-Response-Required-Headers
    CRLF
    Polling-Response-Entity-Body; section 2.2.2.3.2.1
Polling-POST-Response-Required-Headers = (
    Date; section 2.2.1.3.1
    Server; section 2.2.1.3.2
    Connection; section 2.2.1.2.6
    Polling-Content-Length); section 2.2.2.3.1.2
```

2.2.4.2.1 Polling-Response-Entity-Body

The Entity-Body is sent from the server to the client with an HTTP POST Response.

```
Polling-Response-Entity-Body =
    Polling-Virtual-Connection-Message; section 2.2.2.3.1.3.1
    Polling-Virtual-Connection-Response-Message; section 2.2.2.3.2.1.1
    [Application-Data]
```

It is possible for the Entity-Body to contain only the Polling-Virtual-Connection-Message and Polling-Virtual-Connection-Response-Message when the server has no data to transfer to the client.

2.2.4.2.1.1 Polling-Virtual-Connection-Response-Message

The Polling-Virtual-Connection-Response-Message is sent from the server the client.

```
Polling-Virtual-Connection-Response-Message =
    Max-Poll-Interval ","; section 2.2.2.3.2.1.1.1
    Min-Poll-Interval ","; section 2.2.2.3.2.1.1.2
    Poll-Repetition NUL; section 2.2.2.3.2.1.1.3
```

2.2.4.2.1.1.1 Max-Poll-Interval

The Max-Poll-Interval is sent by the server to the client specifying the maximum time (in seconds) the client SHOULD wait before polling for the available data on the server.

Max-Poll-Interval = 1*DIGIT

The default value for Max-Poll-Interval is 120 seconds.

2.2.4.2.1.1.2 Min-Poll-Interval

The Min-Poll-Interval is sent by the server to the client, specifying the minimum amount of time (in seconds) the client SHOULD wait before sending another poll request.

Min-Poll-Interval = 1*DIGIT

The default value for Min-Poll-Interval is 5 seconds.

Copyright © 2014 Microsoft Corporation.

2.2.4.2.1.1.3 Poll-Repetition

Poll-Repetition specifies the number of times the client SHOULD poll for each interval value. The interval value varies between the Min-Poll-Interval (see section 2.2.4.2.1.1.2) and the Max-Poll-Interval (see section 2.2.2.3.2.1.1.1) based on the frequency of Application-Data received from the server. The interval value is initially set to Min-Poll-Interval and is incremented (doubled, up to Max-Poll-Interval) every time the Poll-Repetition count is reached. This progression continues for as long as the server does not return any Application-Data. When a poll to the server returns Application-Data, the interval value is reset to Min-Poll-Interval and the Poll-Repetition starts over again from 0.

Poll-Repetition = 1*DIGIT

The default value for Poll-Repetition is 3.

2.2.4.2.2 Polling-POST-Response Example

The following is an example of a Polling-POST-Response without any Application-Data:

```
----------------------------------Message START-----------------------------------HTTP/1.0
200 OK
Date: Wed, 26 Dec 2007 19:01:56 GMT
Server: Groove-Relay/12.0
Connection: Keep-Alive
Content-Length: 88

1.2.grooveDNS://server.domain.net.a5s2fj8q55cxne2v4wr48ad9ciffsznzq9apczi.16.0.120,5,3
----------------------------------Message END-------------------------------------
```

2.2.5 Secure Tunnel Proxy

Use of the Secure Tunnel Proxy Protocol for encapsulation is implemented in accordance with the [TCPPROXY] which defines a mechanism for TCP based protocols to communicate through an HTTP proxy.

The client sends an HTTP request to the proxy, requesting the proxy to establish a connection to the server. The proxy evaluates the request from the client, attempts to create a connection to the server, and responds to the client with a status code and reason phrase indicating the status of the connection to the server. The Secure Tunnel Proxy negotiation messages are specified in [TCPPROXY], section 3.1.

The following is an example of an initialization request, sent from the client to the proxy:

```
----------------------------------Message START-----------------------------------CONNECT
server.domain.net:443 HTTP/1.0
User-Agent:Mozilla/4.0 (compatible; MSIE 5.5; Win32)
proxy-Connection: Keep-Alive
Pragma: no-cache

----------------------------------Message END-------------------------------------
```

The following is an example of an initialization response, from the proxy indicating a successful connection establishment to the server:

```
----------------------------------Message START-----------------------------------
HTTP/1.0 200 Connection established
```


Copyright © 2014 Microsoft Corporation.

```
-------------------------------Message END----------------------------------
```

After the initialization process, the application data can be exchanged between the server and the client.

2.2.6 SOCKS Encapsulation

Use of the SOCKS Protocol for encapsulation is implemented in accordance to [RFC1928], which enables a client-server application to use the service of a proxy.

The client MUST negotiate the use of an appropriate authentication method with the SOCKS proxy as specified in [RFC1928] section 3.

The following are examples of SOCKS proxy negotiation messages exchanged between the client and the SOCKS proxy.

The following table shows the message sent from the client to the SOCKS proxy.

0	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6	7	8	9	20	1	2	3	4	5	6	7	8	9	30	1
Socks Proxy Version								A								B (variable)															
...																															

Socks Proxy Version (1 byte): Set to 05.

A - Authentication Method Count (1 byte): Set to 02.

B - Authentication Supported by the Client (variable): Set to 00 02. In this message, the client indicates that it can support two authentication methods: "No Authentication" (00) and "Username/Password Authentication" (02).

The following table shows the message sent from the SOCKS proxy to the client.

0	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6	7	8	9	20	1	2	3	4	5	6	7	8	9	30	1
A								B																							

A - Socks Proxy Version (1 byte): Set to 05.

B - Authentication Method Selected by Proxy (1 byte): Set to 00. The server replies with the authentication method supported by the server: "No Authentication" (00).

After initial authentication, the client then requests the SOCKS proxy to create a connection to the server.

The following are examples of the messages exchanged between the client and the SOCKS proxy.

The following table shows the format of a message sent from the client to the SOCKS proxy.

Copyright © 2014 Microsoft Corporation.

In this example, the client is requesting the SOCKS proxy to create a TCP connection to the server at destination address (server.domain.net) and destination port (2492).

<table>
<tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>1
0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>2
0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>3
0</td><td>1</td></tr>
<tr><td colspan="8">Socks Proxy Version</td><td colspan="8">CommandID</td><td colspan="8">Reserved</td><td colspan="8">Address Type</td></tr>
<tr><td colspan="32">Destination Address (variable)</td></tr>
<tr><td colspan="32">...</td></tr>
<tr><td colspan="16">Destination Port</td></tr>
</table>

Socks Proxy Version (1 byte): Set to 05.

CommandID (1 byte): Set to 01.

Reserved (1 byte): Set to 00.

Address Type (1 byte): Set to 03.

Destination Address (variable): Set to 6D 6F 6E 73 74 65 72 32 2E 72 65 6C 61 79 2E 65 74. The maximum size of this field is 256 bytes.

Destination Port (2 bytes): Set to 09 BC.

The following table shows the message sent from the SOCKS proxy to the client in response to the connection request.

In this example, the **ResponseID** (00) indicates that the SOCKS proxy is successful in creating a connection to the server. In addition the response message includes the server address (10.150.2.110) and port (1046).

After the successful response from the SOCKS proxy, the application data (SSTP) can be exchanged between the server and the client transparently to the SOCKS proxy.

<table>
<tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>1
0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>2
0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>3
0</td><td>1</td></tr>
<tr><td colspan="8">Socks Proxy Version</td><td colspan="8">ResponseID</td><td colspan="8">Reserved</td><td colspan="8">Address Type</td></tr>
<tr><td colspan="32">Server Bound Address</td></tr>
<tr><td colspan="16">Server Bound port</td></tr>
</table>

Socks Proxy Version (1 byte): Set to 05.

ResponseID (1 byte): Set to 00.

Reserved (1 byte): Set to 00.

Address Type (1 byte): Set to 01.

Copyright © 2014 Microsoft Corporation.

Server Bound Address (4 bytes): Set to 0A 96 02 6E.

Server Bound port (2 bytes): Set to 04 16.

Copyright © 2014 Microsoft Corporation.

3 Protocol Details

3.1 LongLived Encapsulation Protocol Client Details

LongLived Encapsulation Protocol is an HTTP based protocol used for firewall<6> and proxy traversal<7>. It provides an HTTP transport which can also negotiate and authenticate with HTTP proxies. LongLived Encapsulation is designed to specifically be used with HTTP proxies that do not buffer inbound or outbound proxy traffic.

LongLived Encapsulation provides a virtual connection composed of two HTTP sessions: the first is a GET session and the second is a POST session. Each of these HTTP sessions is layered on a dedicated TCP connection. The POST session is used to send SSTP commands and data from the client to the server, while the GET session is used by the client to receive SSTP commands and data from the server. Each session can send or receive up to the maximum of 0x7ffff000/(2147479552 decimal) bytes of data, as specified by the content length.

If the send operation on a POST session would exceed the content length, the client MUST close the GET and POST sessions and then reestablish a new virtual LongLived connection. If the next GET session response would exceed the maximum number of bytes, then the server MUST close the GET and POST sessions. The client SHOULD respond to the server's closing of the virtual LongLived connection by initiating new GET and POST requests.

3.1.1 LongLived Client Abstract Data Model

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

The following diagram provides a detailed look at the LongLived client session state machine.

Copyright © 2014 Microsoft Corporation.

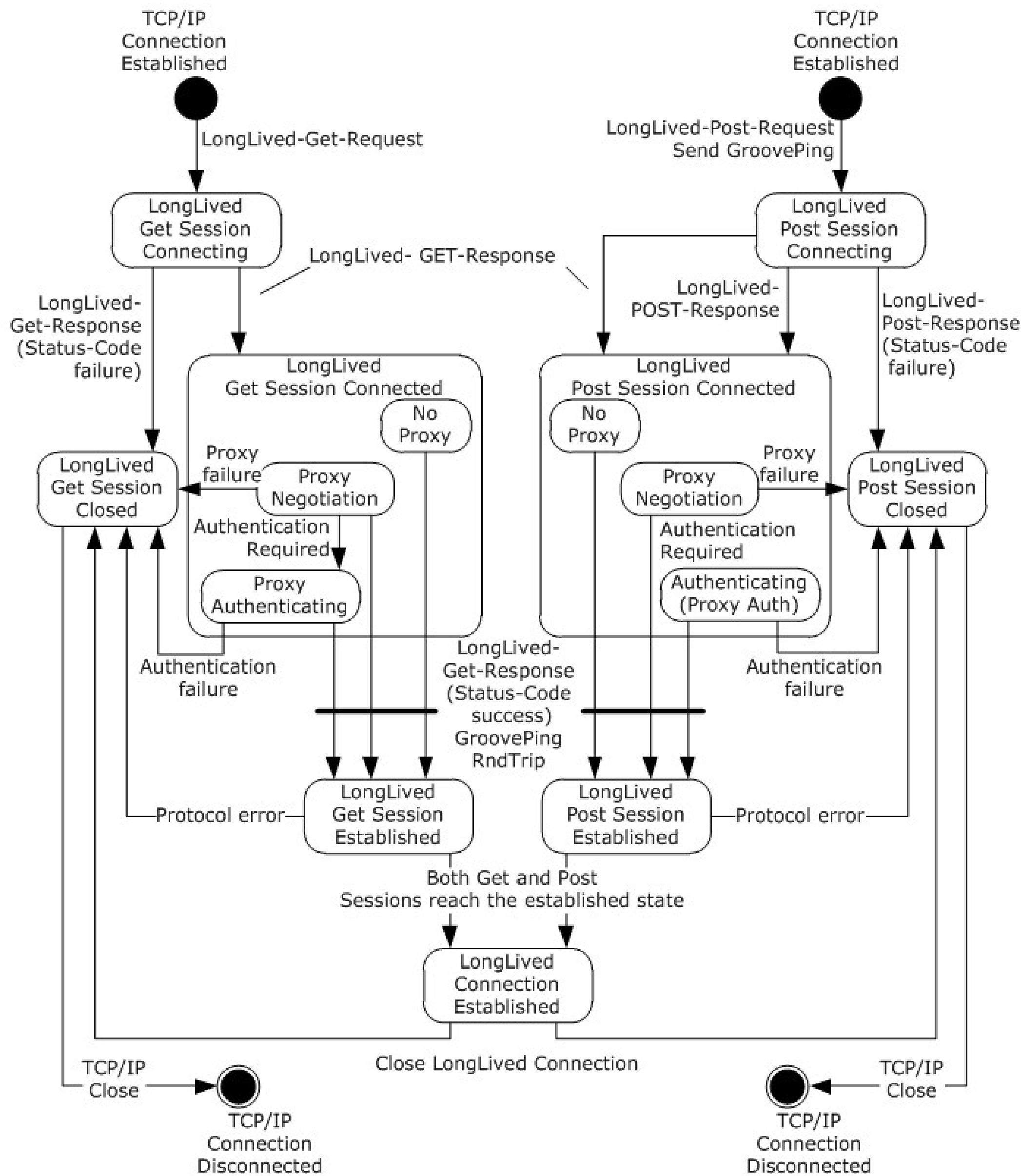

Figure 13: Client LongLived session state diagram

For a detailed view of the LongLived client connection state machine, see the following diagram.

Copyright © 2014 Microsoft Corporation.

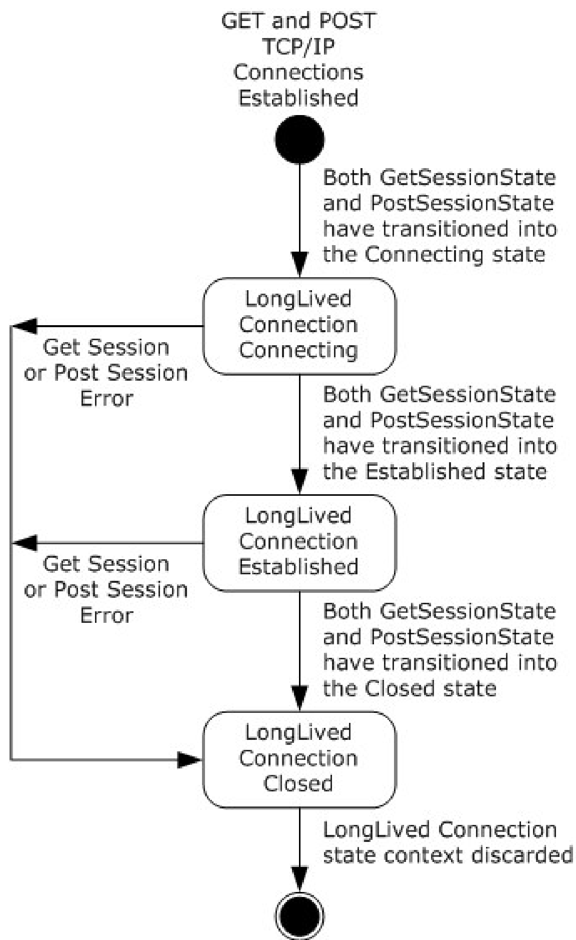

Figure 14: Client LongLived connection state diagram

3.1.1.1 Connection State Information

The state information detailed in this section defines the context needed to manage a single LongLived virtual connection. When a LongLived connection is terminated, this state information is no longer relevant and SHOULD be discarded.

A client SHOULD support LongLived connections to multiple servers concurrently. A client SHOULD support one LongLived virtual connection (2 TCP connections) to the same target server (see ServerHost state information). In all cases, each LongLived connection MUST maintain separate connection state variable information.

ServerPort: The well-known port number of the target server. By default this is the HTTP well known port 80/TCP.

ServerHost: The host name of the target server. Name is in the form of a FQDN or IP Address. There is no default value.

GetSessionState: The variable used to maintain the current disposition of the GET session.

Copyright © 2014 Microsoft Corporation.

There are four possible states: 'Connecting', 'Connected', 'Established' and ‘Closed’. The 'Connecting' state indicates that the GET session request has been sent and the LongLived handshake has started. The 'Connected' state indicates that proxy negotiations are in progress. Non-proxy connections immediately transition through the 'Connected' state, bypassing the proxy Negotiation state. The 'Established' state indicates the GET session response has been received. The ‘Closed’ state indicates a session cleanup in progress.

PostSessionState: The variable used to maintain the current disposition of the POST session. There are four possible states: 'Connecting', 'Connected', 'Established' and ‘Closed’. The 'Connecting' state indicates that the POST session request has been sent and the LongLived handshake has started. The 'Connected' state indicates that proxy negotiations are in progress. Non-proxy connections immediately transition through to the 'Connected' state, bypassing the proxy Negotiation state. The 'Established' state indicates the POST session response has been received. The ‘Closed’ state indicates a session cleanup in progress.

ConnectionState: The variable used to maintain the current disposition of the virtual LongLived connection. There are three possible states: 'Connecting' and 'Established', and 'Closed'. The 'Connecting' state indicates that the GET/POST session creation is in progress. The 'Established' state indicates that both GET/POST sessions have been successfully created, and application data can begin to flow over the virtual connection. The 'Closed' state indicates that the connection can no longer send or receive application data; the virtual connection sessions are closed.

VirtualConnectionGUID: A GUID used to uniquely identify the virtual connection. This GUID is generated by the client when initiating the encapsulation connection. The GUID is exchanged between the client and the server and MUST be unique within each server. There is no default value.

ConnectionContentLength: The maximum content length value specified by the application to be used by the GET and POST session.

PostContentLength: The current number of entity body octets sent over the POST session. It is used by both the client and server to keep track how many octets that have been sent or received on the POST session.

GetContentLength: The current number of entity body bytes received over the GET session. It is used by both the client and server to keep track of how many octets have been received or sent on the GET session.

ProxyConnection: The indicator of whether the current connection is a connection to a proxy or a direct connection to a server. The value is set to TRUE after the client determines that a proxy is to be used. The default value is FALSE.

3.1.1.2 Proxy State Information

The state information detailed in this section defines the context clients need to use to establish connections with proxies. This proxy configuration information MUST be provided by the application to the LongLived client prior to connection establishment. The source of this configuration information is external to the LongLived Protocol<8>.

ProxyServerPort: The well-known port number of the target proxy. It is used for establishing a TCP connection to a proxy. By default this is the HTTP well known port 80/TCP or the HTTP alternate well known port 8080/TCP.

ProxyServerHostName: The host name of the target proxy. The name is in the form of an FQDN or an IP Address. If the name is an FQDN, then the client MUST resolve this name to its IP Address. There is no default value.

Copyright © 2014 Microsoft Corporation.

ProxyAuthRequired: A variable used to indicate whether a proxy requires authentication. The client sets this variable to TRUE when it discovers that the proxy needs authentication during its first negotiation with the proxy. When the client initiates a new virtual connection through the same proxy, it SHOULD provide the cached credentials without waiting to be challenged to avoid the overhead of additional message exchanges.

3.1.2 LongLived Client Timers

3.1.2.1 ConnectionEstablishment Timer

The ConnectionEstablishment timer SHOULD be used by clients to limit the amount of time LongLived connection negotiations take to complete. This timer measures the time it takes for a connection to move from the connecting' to the established state. The recommended timeout value is 30 seconds. ConnectionEstablishment timer event processing is handled as specified in section 3.1.6.1.

3.1.2.2 NetworkReceiveIO Timer

The **NetworkReceiveIO** timer SHOULD be used by clients to limit the amount of time a client waits for the network to receive IO to complete. This timer applies to the GET session only. A **NetworkReceiveIO** timer SHOULD NOT be started on the POST session because a receive on the POST session is expected to never complete unless there are session errors or TCP disconnects. The timer duration SHOULD be greater than the **KeepAlive** timer specified in section 3.1.2.3. The recommended **timeout** value is 5 minutes. The **NetworkReceiveIO** timer event processing is handled as specified in section 3.1.6.2.

3.1.2.3 KeepAlive Timer

The HTTP Encapsulation protocols do not define a KeepAlive timer. The underlying encapsulated protocol MUST implement a KeepAlive timer. The SSTP protocol uses the KeepAlive mechanism provided by the SSTP_NOOP command (see [MS-GRVSSTP] section 2.2.13)<9>.

The **keepalive message** serves to keep the LongLived connection from being closed by firewalls and proxies. All LongLived Connections SHOULD use KeepAlive timers, regardless of whether the client detects if a connection is a proxy connection or not, as some firewalls and proxies are undetectable. The recommended client KeepAlive timeout value is 45 seconds<10>. KeepAlive timer event processing is handled as specified in section 3.1.6.3.

3.1.3 LongLived Client Initialization

3.1.3.1 Protocol Initialization

The LongLived Protocol is not initialized until a request to open an encapsulated connection is received by the client. The variables defined by the abstract data model are initialized to their default values when a LongLived connection request is made.

3.1.4 LongLived Client Higher-Layer Triggered Events

3.1.4.1 Establishing a LongLived Encapsulation Connection

When an application requests a LongLived Connection, the LongLived protocol layer MUST initialize the LongLived connection state variables as specified in the abstract data model (see section 3.1.1). After the connection state variables are initialized, the LongLived protocol enters into the connection establishment phase<11>.

Copyright © 2014 Microsoft Corporation.

The client opens two connections to the server, one for GET session and one for POST session, as specified in sections 3.1.4.1.1 and 3.1.4.1.2. If proxy configuration information is supplied, the client MUST go through the proxy for each connection by performing proxy negotiation as specified in sections 3.1.4.1.2 and 3.1.4.1.4.

The ConnectionEstablishment timer SHOULD be started.

The client SHOULD set the ConnectionContentLength to 0x7ffff000/(2147479552 decimal) octets as specified in section 2.2.2.1.1.3.

The client MUST generate a new virtual connection GUID and store it in the VirtualConnectionGUID state variable.

The ConnectionState MUST be set to 'Connecting'.

3.1.4.1.1 Establishing GET Session without Proxy

The GetSessionState MUST be set to 'Connecting'.

The client MUST construct the LongLived-GET-Request-URI as the LongLived-GET-Request-Relative-URI, with the ServerHost as Relay-Server-Name, with Virtual-Connection-GUID as the VirtualConnectionGUID variable, and the LongLived-Encapsulation-Content-Length as the ConnectionContentLength variable.

The client MUST construct a LongLived-GET-Request as specified in section 2.2.2.1, with required headers.

The client specifies the HOST header and sets the value to equal the ServerHost variable as specified in section 2.2.1.2.7.

The client MUST establish a TCP connection to the server identified with ServerHost and ServerPort and send the LongLived-Get-Request.

3.1.4.1.2 Establishing GET Session with Proxy

The GetSessionState MUST be set to 'Connecting'.

The client sets the ProxyConnection to TRUE.

The client generates a Request ID GUID.

The client MUST construct the LongLived GET-Request-URI as the LongLived-GET-Request-Absolute-URI, with the ServerHost as Relay-Server-Name, virtual connection GUID using the VirtualConnectionGUID variable and the preceding generated Request ID GUID for the LongLived-Encapsulation-Request-ID.

The client MUST construct a LongLived-GET-Request as specified in section 2.2.2.1, with the required headers.

The client specifies the HOST header and sets the value to equal the ServerHost variable as specified in section 2.2.1.2.7.

If the ProxyAuthRequired is set, the client MUST add additional proxy authentication<12> headers to the request.

The client MUST establish a TCP connection to the server identified with ProxyServerHostName and ProxyServerPort and send the LongLived-GET-Request.

Copyright © 2014 Microsoft Corporation.

3.1.4.1.3 Establishing POST Session without Proxy

The PostSessionState MUST be set to 'Connecting'.

The client MUST construct the LongLived-POST-Request-URI as the LongLived-POST-Request-Relative-URI, with the ServerHost as Relay-Server-Name, virtual connection GUID as VirtualConnectionGUID variable, and the LongLived-Encapsulation-Content-Length as ConnectionContentLength variable.

The client MUST construct a LongLived-POST-Request with required headers, as specified in section 2.2.2.2.

The LongLived-Entity-Body MUST contain the Encapsulation-Echo-String message.

The client MUST construct the LongLived-Content-Length header with the value of the ConnectionContentLength variable.

The client MUST establish a TCP connection to the server identified with ServerHost and ServerPort and send the LongLived-POST-Request.

3.1.4.1.4 Establishing POST Session with Proxy

The PostSessionState MUST be set to 'Connecting'.

The client sets the ProxyConnection to TRUE.

The client generates a Request ID GUID.

The client MUST construct the LongLived-POST-Request-URI as the LongLived-POST-Request-Absolute-URI, with the ServerHost as Relay-Server-Name, virtual connection GUID using the VirtualConnectionGUID variable and the preceding generated Request ID GUID for the LongLived-Encapsulation-Request-ID.

The client MUST construct a LongLived-POST-Request as specified in section 2.2.2.2, with required headers.

If the ProxyAuthRequired is set, the client MUST add additional proxy authentication headers to the request.

The LongLived-Entity-Body MUST contain the Encapsulation-Echo-String message.

The client MUST construct the LongLived-Content-Length header with the value of the ConnectionContentLength variable.

The client MUST establish a TCP connection to the server identified with ProxyServerHostName and ProxyServerPort and send the LongLived-POST-Request.

3.1.4.2 Closing a LongLived Connection

The client SHOULD close the POST and GET sessions by sending a graceful TCP disconnect on each session to the server. The connection then transitions into the connection 'Closed' state.

TCP client or server connections are closed using either a Graceful Close (TCP FIN Flag set to 1) or Abortive Close (TCP Reset Flag set to 1) mechanism. Closing the connection gracefully provides clients and servers with the benefit of never having to resend unacknowledged payload data. Abortive Closed connections are efficient in connection tear down but can require the client or server to resend payload data which has not been acknowledged at the TCP level. HTTP Encapsulation of

Copyright © 2014 Microsoft Corporation.

SSTP Protocols relies on SSTP to ensure that any unacknowledged payload data is later resent. The choice of connection close mechanism is one of efficiency (efficient connection tear down verses efficient byte transmission). This document does not mandate that implementations adhere to one approach or another. Implementations can choose to close HTTP Encapsulated Connections either gracefully or abortively based on the implementation's requirements.

3.1.4.3 Sending Application Data

The LongLived connection MUST be in the 'Established' state to send application data. If the virtual connection is still being established, the client MUST buffer the data and wait until the connection is in the 'Established' state.

The client sends the SSTP stream data over the POST session. The SSTP stream data is in the LongLived-Entity-Body fragment (see section 2.2.2.2.3).

The client MUST keep track of the number of octets sent, as defined by the PostContentLength state variable, to ensure that the content length is not exceeded. It is a protocol error if PostContentLength exceeds the value specified in ConnectionContentLength.

If sending the entity body fragment would cause the PostContentLength to exceed the ConnectionContentLength, the client SHOULD close the LongLived connection and immediately re-establish a new LongLived virtual connection with the target server as specified in section 3.1.4.1 <13>.

3.1.5 LongLived Client Message Processing Events and Sequencing Rules

Unlike traditional HTTP clients, LongLived clients MUST NOT use the content length header to determine the length of the message and MUST NOT wait to receive the entire content before returning received data to the application. After the LongLived connection handshake (see section 3.1.4.1), clients SHOULD return the application data to the application layer as the data is received.

3.1.5.1 Receiving Data on the POST Session

Upon receiving data on the POST session, the client MUST first check to see if the data starts with the HTTP response status line, as specified in section 2.2.2.4. If so, the client processing proceeds as in section 3.1.5.1.1. If not, the client processing proceeds as in section 3.1.5.1.2.

3.1.5.1.1 LongLived-POST-Response Processing

The HTTP response header MUST be parsed, and the status code and reason phrase extracted (see section 2.2.1.4).

A client SHOULD NOT receive a LongLived-POST-Response message (see section 2.2.2.4), on the POST session unless there is a proxy authentication challenge or a protocol error.

3.1.5.1.1.1 Status code: 400 (Bad Request)

The server has rejected the connection request because of a protocol error or because an encapsulation version is not equal to the required value (see section 2.2.2.1.1.1.1). The client MUST close the virtual LongLived connection (see section 3.2.4.1).

3.1.5.1.1.2 Status codes: 401 (Unauthorized) and 407 (ProxyAuthentication Required)

HTTP status code values of 401 (Unauthorized) or 407 (ProxyAuthentication Required)

Copyright © 2014 Microsoft Corporation.

indicate that the proxy requires the client to authenticate to gain access to the proxy. Common authentication schemes include Basic and Digest, as specified in [RFC2617], and NTLM HTTP Authentication, as specified in [RFC4559].

The client sets the ProxyAuthRequired state variable to TRUE. Subsequent connection attempts to the same proxy SHOULD avoid the proxy challenge message by sending the proxy authentication credentials as part of the LongLived-POST-Request.

Depending on the authentication method, multiple round trips can happen to complete the authentication process. That is, the client MUST expect to get multiple 401 and 407 messages. It MUST follow [RFC2617] and [RFC4559] to set proper authentication headers and retry the proxy connection.

For processing required to retry the proxy connection, see section 3.1.4.1.4.

The ConnectionEstablishment timer SHOULD be restarted before re-attempting the LongLived handshake (see section 3.1.4.1).

3.1.5.1.1.3 All Other Status Codes

All other status codes are fatal; the virtual connection MUST be closed as specified in section 3.1.4.2.

3.1.5.1.2 POST Session Data Processing

The client MUST never receive application data on the POST session. This event is a protocol error. The LongLived connection MUST be closed as specified in section 3.2.4.1.

3.1.5.2 Receiving Data on the GET Session

Upon receiving data on the GET session, the client MUST first check to see if the data starts with the HTTP response status line, as specified in section 2.2.2.3.1. If so, the client processing proceeds as specified in section 3.1.5.2.1. If not, the client processing proceeds as specified in section 3.1.5.2.2.

3.1.5.2.1 LongLived-GET-Response Processing

The HTTP response header MUST be parsed and the status code and response body extracted.

The receipt of a LongLived-GET-Response message on the GET session causes both the GET session and POST session to transition into the 'Connected' state.

3.1.5.2.1.1 Status code: 200 (OK)

The client MUST compare the response body string against the Encapsulation-Echo-String sent earlier on the POST session in sections 3.1.4.1.2 and 3.1.4.1.4. If they are not equal, it is a violation of the protocol and the connection MUST be closed (see section 3.1.4.2).

The receipt of Encapsulation-Echo-String on the GET session completes the virtual connection establishment. The GET and POST sessions move to the ‘Established’ state, and the ConnectionState transitions into the ‘Established’ state. The connection is ready to send and receive application data as entity body fragments.

The ConnectionEstablishment timer SHOULD be stopped and no further timer expiration processing is performed.

The NetworkReceiveIO and KeepAlive timer SHOULD be started.

Copyright © 2014 Microsoft Corporation.

If there is any buffered data to send, the client MUST now send it, as specified in section 3.1.4.3.

3.1.5.2.1.2 Status code: 400 (Bad Request)

The server has rejected the connection request because of a protocol error or because an encapsulation version does not equal the required value (see section 2.2.2.1.1.1.1). The client MUST close all connections associated with this virtual connection (see section 3.2.4.1).

3.1.5.2.1.3 Status codes: 401 (Unauthorized) and 407 (ProxyAuthentication Required)

HTTP status code values of 401 (Unauthorized) or 407 (ProxyAuthentication Required)

indicate that the proxy requires the client to authenticate to gain access to the proxy. Common authentication schemes include Basic and Digest, as specified in [RFC2617], and NTLM HTTP Authentication, as specified in [RFC4559].

The client sets the ProxyAuthRequired state variable to TRUE. Subsequent connection attempts to the same proxy SHOULD avoid the proxy challenge message by sending the proxy authentication credentials as part of the LongLived-GET-Request.

Depending on the authentication method, multiple round trips can happen to complete the authentication process. That is, the client MUST expect to get multiple 401 and 407 messages. The client MUST follow [RFC2617] and [RFC4559] to set proper authentication headers and retry the proxy connection.

For processing required to retry the proxy connection, see section 3.3.4.1.2.

The ConnectionEstablishment timer SHOULD be restarted before re-attempting the LongLived handshake (see section 3.1.4.1).

3.1.5.2.1.4 All Other Status Codes

All other status codes are fatal; the virtual connection MUST be closed as specified in section 3.1.4.2.

3.1.5.2.2 Receiving Application Data (GET Session Data Processing)

The LongLived connection MUST be in the 'Established' state to receive application data. If the connection state is not ’Established�ṣ, this is a violation of the protocol. The data MUST be discarded and the connection closed.

The client receives the SSTP stream data over the GET session. The SSTP stream data is contained within the LongLived-GET-Response-Entity-Body. The client MUST pass the application data to a higher layer for processing.

The NetworkReceiveIO Timer MUST be restarted after each LongLived-GET-Response-Entity-Body fragment is received.

3.1.6 LongLived Client Timer Events

3.1.6.1 ConnectionEstablishment Timer Event

The ConnectionEstablishment Timer Event fires when enforcing a limit on the time it takes to establish a LongLived connection with the server. If this timer expires before the LongLived

Copyright © 2014 Microsoft Corporation.

connection enters the 'Established' state, the virtual LongLived connection SHOULD be closed by the client.

3.1.6.2 Network Receive IO Timer Event

The NetworkReceiveIO Timer Event fires when an entity body fragment is not received on the GET session within the specified amount of time. If the NetworkReceiveIO event triggers, the client SHOULD close the LongLived connection, but can retry immediately as specified in section 3.1.4.1. This timer event will not occur on well behaved connections, as the encapsulated protocol SHOULD have implemented the KeepAlive timer as specified in section 3.1.2.3, which SHOULD have used a timer value smaller than the NetworkReceiveIO timer value.

3.1.6.3 KeepAlive Timer Event

A KeepAlive Timer Event SHOULD trigger an encapsulated protocol message such as an SSTP_NOOP command to be sent across the wire by LongLived Client implementations<14>. Well behaved connections will see this event every KeepAlive timer interval when the session is idle. After firing, the KeepAlive timer SHOULD be restarted.

3.1.7 LongLived Client Other Local Events

A transport disconnect event on either session causes the client to close the virtual LongLived connection, thereby closing both sessions. The ConnectionState transitions into the 'Closed' state. All connection state information MUST be discarded.

The client SHOULD close the LongLived connection, but can retry immediately (see section 3.1.4.1). If the retry attempt fails, the client SHOULD let the higher layer decide whether to wait before establishing a new LongLived virtual connection to the target server again, or switch to using a different encapsulation protocol to establish a connection to the server.

3.2 LongLived Encapsulation Protocol Server Details

3.2.1 LongLived Server Abstract Data Model

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

3.2.1.1 Connection State Information

See section 3.1.1.1 for a list of connection state variables that are shared with the client.

VirtualConnectionGUIDList: The global list of virtual connection GUIDs of all active connections. This list allows the application to quickly look up a virtual connection GUID to determine if it is a known virtual connection GUID. This list also contains a reference to the per-connection state variables for the associated GUID.

GetSessionReady: The state variable enforces the requirement that the server MUST NOT send application data on the GET session until the client has received the LongLived-GET-Response containing the Encapsulation-Echo-String. The first packet of application data received by the server on the POST session is an implied acknowledgement that indicates that the client has received the Encapsulation-Echo-String. The default state is FALSE.

Copyright © 2014 Microsoft Corporation.

3.2.2 LongLived Server Timers

3.2.2.1 ConnectionEstablishment Timer

The ConnectionEstablishment timer SHOULD be used by the server to limit the amount of time a LongLived connection handshake takes to complete. This timer measures the time it takes for a connection to move from the connecting to the established state. This is a per-connection timer whose recommended timeout value is 90 seconds. The ConnectionEstablishment timer event processing is handled as specified in section 3.2.3.

3.2.2.2 Network Receive IO Timer

The NetworkReceiveIO timer SHOULD be used by the server to determine if a connection has become idle. This timer is set after the LongLived handshake is finished. The timer MUST be greater than the KeepAlive timer used by the server, specified in section 3.2.2.3. This is a per-connection timer. The recommended timeout value is 90 seconds. The

NetworkReceiveIO timer event processing is handled as specified in section 3.2.6.2.

3.2.2.3 KeepAlive Timer

HTTP Encapsulation protocols do not support a native KeepAlive timer, but rely on the encapsulated protocol to provide a KeepAlive mechanism. Encapsulated protocols SHOULD implement their own KeepAlive mechanisms. The SSTP protocol provides its own KeepAlive mechanism using the SSTP_NOOP command<15>. This data serves to keep the LongLived connection from being closed by firewalls and proxies. All LongLived connections SHOULD use KeepAlive timers, regardless of whether or not the client detects if a connection is a proxy connection, as some firewalls and proxies are undetectable. The default server KeepAlive timeout value is 45 seconds. The maximum KeepAlive value is limited by proxy implementations. The KeepAlive timer event processing is handled as specified in section 3.2.6.3.

3.2.3 LongLived Server Initialization

3.2.3.1 Protocol Initialization

When the server starts, it MUST initialize the HTTP stack<16>.

A LongLived connection protocol is not initialized until a request to open an encapsulated connection is received by the server. The variables defined by the abstract data model are initialized when the LongLived connection request is received.

3.2.3.2 LongLived Listener

The Server MUST create a listener socket on the LongLived port. The LongLived connection port is typically the well-known HTTP port 80/TCP. Alternate ports MAY be used, but non-default port information MUST be conveyed to the client independently of the LongLived protocol.

3.2.4 LongLived Server Higher-Layer Triggered Events

3.2.4.1 Closing a LongLived Connection

The server MUST close the POST and GET sessions by sending a graceful TCP disconnect on each session. The ConnectionState then transitions into the ‘Closed’ state. All connection state information SHOULD be discarded.

Copyright © 2014 Microsoft Corporation.

TCP client or server connections are closed using either a Graceful Close (TCP FIN Flag set to 1) or Abortive Close (TCP Reset Flag set to 1) mechanism. Closing the connection gracefully provides clients and servers with the benefit of never having to resend unacknowledged payload data. Abortive Closed connections are efficient in connection tear down but can require the client or server to resend payload data which has not been acknowledged at the TCP level. HTTP Encapsulation of SSTP Protocols relies on SSTP to ensure that any unacknowledged payload data is later resent. The choice of connection close mechanism is one of efficiency (efficient connection tear down verses efficient byte transmission). This document does not mandate that implementations adhere to one approach or another. Implementations can choose to close HTTP Encapsulated Connections either gracefully or abortively based on the implementation's requirements.

3.2.4.2 Sending Application Data

The LongLived connection MUST be in the 'Established' state to send application data.

If the GetSessionReady state variable is set to FALSE, the application data MUST be buffered in the order that it was sent. If GetSessionReady is TRUE, the server MUST send the data.

The server sends the SSTP stream data over the GET session. The SSTP stream data is contained within the LongLived-Entity-Body fragment (see section 2.2.2.2.3).

The server MUST keep track of the number of octets sent, as defined by the GetContentLength state variable, and ensure that the content length is not exceeded. It is a protocol error if GetContentLength exceeds the value specified in ConnectionContentLength. If sending the entity body fragment would cause the GetContentLength to exceed the ConnectionContentLength, the server MUST close the LongLived connection immediately and let the client establish a new LongLived virtual connection with the target server<17>.

3.2.5 LongLived Server Message Processing Events and Sequencing Rules

Unlike traditional HTTP servers, LongLived servers MUST NOT use the content length header to specify the length of the message.

3.2.5.1 GET Session Processing

Upon receiving data on the GET session, the server MUST first check to see if the data starts with an HTTP GET request line as specified in section 2.2.2.1. If so, the server processing continues as specified in section 3.2.5.1.1. If not, application data is received, and the server processing continues as specified in section 3.2.5.1.2.

3.2.5.1.1 Receiving a LongLived-GET-Request

Upon receipt of a LongLived-GET-Request (see section 2.2.2.1), the server transitions the GetSessionState to 'Connected'. If the PostSessionState is ‘Connected’, the ConnectionState transitions into the 'Connected' state. If the PostSessionState is uninitialized, the ConnectionEstablishment timer is started.

The server validates the LongLived-GET-Request message (see section 2.2.2.1) using the following procedure:

1. The server MUST validate the LongLived-GET-Request-URI (see section 2.2.2.1.1) and extract the version, server name, virtual connection GUID, encapsulation type, content length, and Request ID. If the parsing fails, it is a protocol error and the server MUST close the connections (see section 3.2.4.1). The content length is saved in the variable ConnectionContentLength.

Copyright © 2014 Microsoft Corporation.

2. The server SHOULD<18> check the LongLived-Encapsulation-Version and, if the value does not equal the required value (see section 2.2.2.1.1.1.1), send a LongLived-GET-Response with a status code of 400. See section 3.2.5.1.1.2.

3. If the encapsulation type is not LongLived, it is a protocol error and the virtual connection MUST be closed.

4. The server SHOULD<19> verify that the server name in the message equals its own name and, if they are not equal, close the virtual connection.

5. The server SHOULD ignore the Request ID.

6. The server MUST examine the Virtual-Connection-GUID to validate that the LongLived-GET-Request is a new connection request. Virtual-Connection-GUID SHOULD be maintained in the VirtualConnectionGUIDList. If the PostSessionState is 'Connected', the virtual connection GUID SHOULD be found in the VirtualConnectionGUIDList. If the PostSessionState is uninitialized, the Virtual-Connection-GUID SHOULD be added to the VirtualConnectionGUIDList. If the Virtual-Connection-GUID is found and the PostSessionState is not ‘Connected’, this is a protocol error. See section 3.2.5.1.1.2.

If the PostSessionState is not ‘Connected’, the processing stops here.

If the PostSessionState is 'Connected', the virtual connection is established. The server transitions GetSessionState, PostSessionState and ConnectionState to the 'Established' state.

The ConnectionEstablishment timer is stopped and no timer expiration processing is performed.

The server as described in section 3.2.5.1.1.1 to complete the handshake by sending a LongLived-GET-Response.

3.2.5.1.1.1 Sending a LongLived-GET-Response with Status Code 200

The server MUST send a LongLived-GET-Response message on the GET session to complete the LongLived Encapsulation connection handshake.

A successful LongLived-GET-Response message MUST contain a Response-Status-Line (see section 2.2.2.3.1) with a status code equal to 200 (OK).

The LongLived-GET-Response message sent to the client MUST contain the Encapsulation-Echo-String within the LongLived-Entity-Body.

The LongLived-Content-Length response header value MUST be set to the ConnectionContentLength value.

The LongLived connection response MUST construct the extended HTTP 1.0 response as specified in section 2.2.2.3.

The server sends a LongLived-GET-Response on the GET session.

The connection is now ready to receive the SSTP data stream as LongLived-Entity-Body fragments. The server MUST NOT send any data to client until after receiving the first non-Encapsulation-Echo-String entity body.

A NetworkReceiveIO timer SHOULD be started on the POST session. The KeepAlive timer is started after the LongLived Connection handshake completes and moves into the established state. The KeepAlive timer is set and maintained by the protocol encapsulated by the LongLived connection (for example SSTP).

Copyright © 2014 Microsoft Corporation.

3.2.5.1.1.2 Sending a LongLived-GET-Response with Status Code 400

If the protocol version does not equal the required LongLived version value (see section 2.2.2.1.1.1.1), the server sends on the GET session a LongLived-GET-Response message which MUST contain a Response-Status-Line (see section 2.2.2.3.1) with a status code equal to 400 (Bad Request).

The LongLived-GET-Response message with Status code of 400 can also be sent on protocol errors.

3.2.5.1.2 Receiving Data on LongLived-GET-Request

This event is a protocol error. The server MUST close the virtual LongLived connection (see section 3.2.4.1).

3.2.5.2 POST Session Processing

Upon receiving data on the POST session, the server MUST first check to see if the data starts with HTTP POST request line, as specified in section 2.2.2.2. If so, the server processing follows section 3.2.5.2.1. If not, application data is received, the server processing proceeds as specified in section 3.2.5.2.2.

3.2.5.2.1 Receiving a LongLived-POST-Request

Upon receipt of a LongLived-POST-Request, the server transitions the PostSessionState to 'Connected'. If the GetSessionState is 'Connected', the ConnectionState transitions into the 'Connected' state. If the GetSessionState is uninitialized, the ConnectionEstablishment timer is started.

The server validates the LongLived-POST-Request message (see section 2.2.2.2) using the following procedure:

The server MUST validate the URI (see section 2.2.2.2.1) and extract the version, server name, virtual connection GUID, encapsulation type, and content length. If the parsing fails, it is a protocol error and the server MUST close the connections (see section 3.2.4.1). The content length is saved in the variable ConnectionContentLength.

1. The server SHOULD<20> check the LongLived-Encapsulation-Version . If the version does not equal the required value (see section 2.2.2.1.1.1.1), the server MUST send a LongLived-POST-Response with a status code of 400. See section 3.2.5.2.1.1.

2. If the encapsulation type is not LongLived, it is a protocol error and the virtual connection MUST be closed.

3. The server SHOULD<21> verify that the server name in the message equals its own name and, if they are not equal, close the virtual connection.

4. The server MUST examine the Virtual-Connection-GUID to validate that the LongLived-POST-Request is a new connection request. The virtual connection GUID SHOULD be maintained in the VirtualConnectionGUIDList. If the GetSessionState is 'Connected', the Virtual-Connection-GUID SHOULD be found in the VirtualConnectionGUIDList. If the GetSessionState is uninitialized, the Virtual-Connection-GUID SHOULD be added to VirtualConnectionGUIDList. If the Virtual-Connection-GUID is found and GetSessionState is not 'Connected', this is a protocol error. See section 3.2.5.2.1.1.

Copyright © 2014 Microsoft Corporation.

The LongLived-POST-Request MUST contain the Encapsulation-Echo-String (see section 2.2.2.2.3) in the message entity body. The Encapsulation-Echo-String is saved so it can later be echoed back to the client on the LongLived-GET-Response message (see section 3.2.5.1.1.1).

If the GetSessionState is not 'Connected', the processing stops here.

If the GetSessionState is 'Connected', the virtual connection is established. The server transitions the GetSessionState, PostSessionState, ConnectionState to the 'Established' state.

The server clears the ConnectionEstablishment timer.

The server continues with the procedure in section 3.2.5.1.1.1 to complete the handshake by sending a LongLived-GET-Response.

3.2.5.2.1.1 Sending a LongLived-POST-Response because of a Protocol Error

If the protocol version does not equal the required LongLived version value (see section 2.2.2.1.1.1.1) or if protocol errors occur during the handshake, the server optionally can send a LongLived-POST-Response message which contains a Response-Status-Line (see section 2.2.2.4) with a status code equal to 400 (Bad Request).

Regardless of protocol version not equaling the required LongLived version value or a protocol error occurring during the handshake, the server MUST close the virtual LongLived connection as specified in section 3.2.4.1.

3.2.5.2.2 Receiving Application Data

If any data is received before the ConnectionState is 'Established', it is a violation of protocol and the server MUST close the virtual LongLived connection, as specified in section 3.2.4.1.

The server sets the GetSessionReady state to TRUE so that the server can send application data at any time.

The server receives encapsulated application data over the POST session. The server passes the application data to a higher layer for processing. The server can keep track of the amount of octets received with the PostContentLength variable, to ensure that it does not exceed the ConnectionContentLength value. If the PostContentLength exceeds this value, it is a protocol violation and is handled as specified in section 3.2.5.2.1.1.

If there is buffered data to be sent to the client, the sever can now send the data in one chunk<22> as specified in section 3.2.4.2.

The NetworkReceiveIO timer MUST be restarted after each entity body fragment is received.

3.2.6 LongLived Server Timer Events

3.2.6.1 ConnectionEstablishment Timer Event

The ConnectionEstablishment Timer Event fires when the ConnectionEstablishment timer for a given LongLived connection expires before the connection can be established. If this timer expires before the LongLived connection enters the established state, all established TCP connections on the LongLived virtual connection SHOULD be closed by the server.

Copyright © 2014 Microsoft Corporation.

3.2.6.2 NetworkReceiveIO Timer Event

The NetworkReceiveIO Timer Event fires when no data is received on the POST session within the NetworkReceiveIO interval. If the NetworkReceiveIO event triggers, the server SHOULD close the LongLived connection (see section 3.2.4.1).

3.2.6.3 KeepAlive Timer Event

The KeepAlive timer fires every KeepAlive interval to trigger a send of a KeepAlive Message on the GET Session from the server to the client. A KeepAlive Event SHOULD trigger the server to send an encapsulated protocol message, such as an SSTP_NOOP command, to the client<23>. Well behaved KeepAlive connections will see this event every KeepAlive interval when the session is idle. After the KeepAlive event timer fires, the timer SHOULD be restarted.

3.2.7 LongLived Server Other Local Events

A transport disconnect event on one session causes the server to close the virtual LongLived connection (see section 3.2.4.1).

3.3 KeepAlive Encapsulation Protocol Client Details

KeepAlive Encapsulation Protocol is an HTTP-based protocol used for firewall and proxy traversal. It provides an HTTP transport which can also negotiate and authenticate with HTTP proxies. KeepAlive Encapsulation provides a virtual connection composed of two HTTP sessions, a GET session and the POST session. Each of these HTTP sessions is layered on a dedicated TCP connection. The POST session is used to send SSTP commands and data from the client to the server, while the GET session is used by the client to receive SSTP commands and data from the server. Each session is capable of sending/receiving multiple request/response messages over a single TCP connection.

3.3.1 KeepAlive Client Abstract Data Model

This section specifies a conceptual model of possible data organization that an implementation maintains to participate in the KeepAlive encapsulation protocol. The specified organization is provided to facilitate the explanation of how the protocol behaves. This document does not mandate that implementations adhere to this model as long as their external behavior is consistent with that specified in this document. See the following figure for a detailed view of the KeepAlive client session state machine.

Copyright © 2014 Microsoft Corporation.

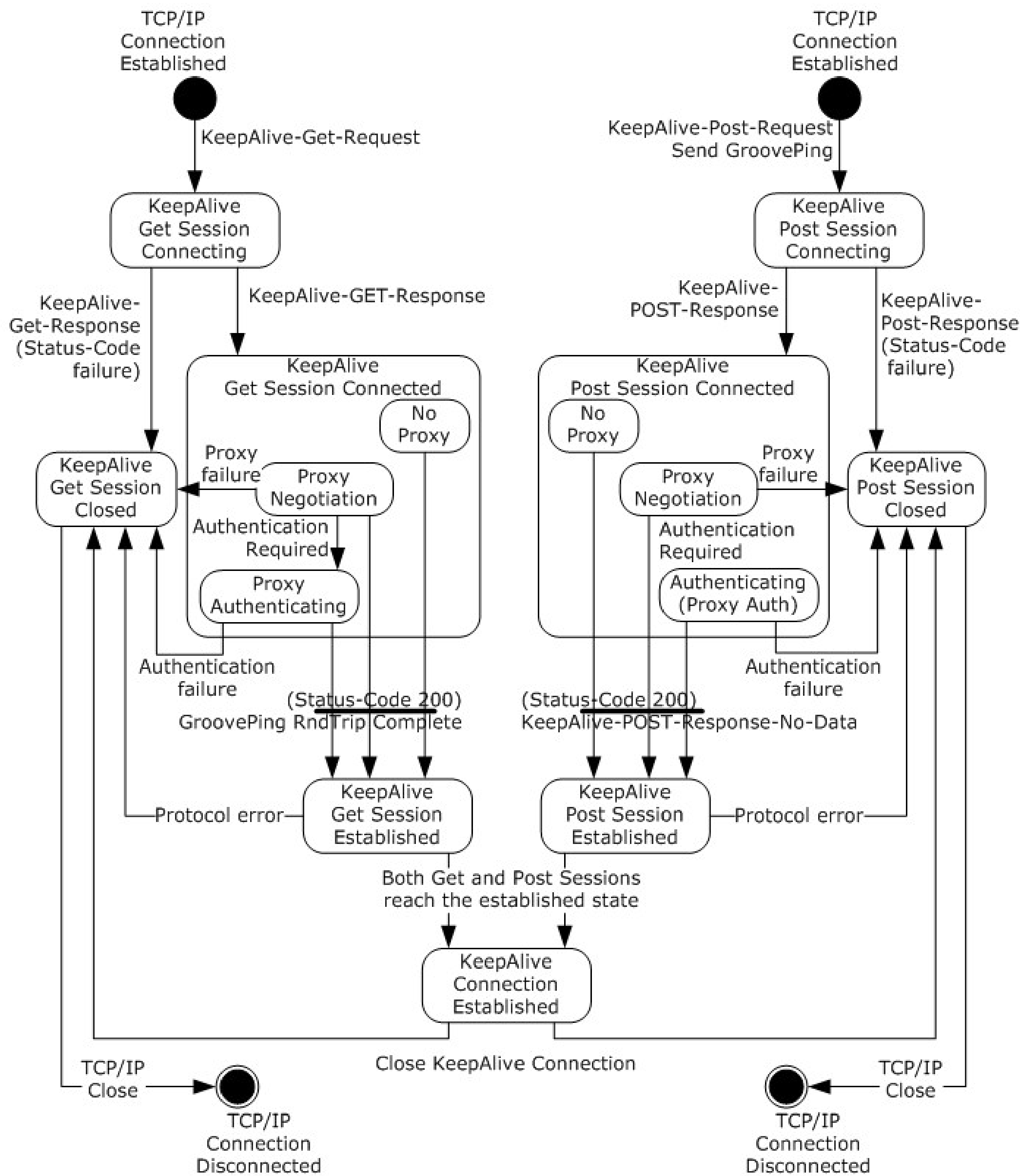

Figure 15: Client KeepAlive session state diagram

For a detailed view of the KeepAlive client connection state machine, see the following diagram.

Copyright © 2014 Microsoft Corporation.

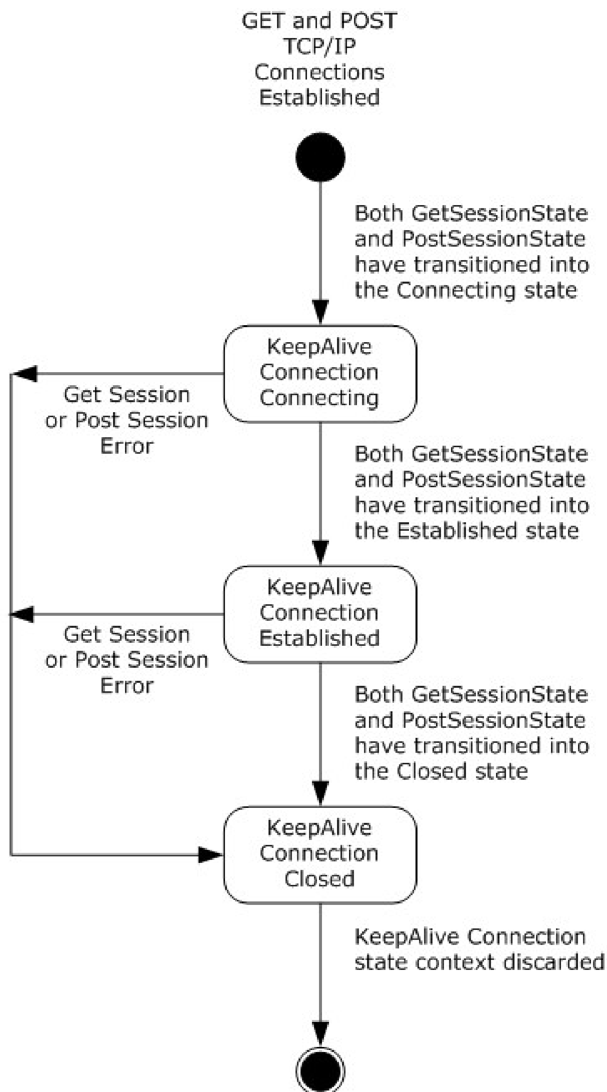

Figure 16: Client KeepAlive connection state diagram

3.3.1.1 Connection State Information

The following details about the state information define the context needed to manage a KeepAlive virtual connection. Unless otherwise noted, the following connection state variables are scoped to a single KeepAlive connection. When a KeepAlive connection is terminated, this state information is no longer relevant and SHOULD be discarded.

A client SHOULD support multiple KeepAlive connections to multiple servers concurrently. A client SHOULD support one KeepAlive virtual connection (2 TCP connections) to the same target server (see ServerHost state information). In all cases, each KeepAlive connection MUST maintain separate connection state variable information.

ServerPort: The well-known port number of the target server. By default this is the HTTP well known port 80/TCP.

Copyright © 2014 Microsoft Corporation.

ServerHost: The host name of the target server, in the form of an FQDN or IP Address. There is no default value.

VirtualConnectionGUID: A GUID used to uniquely identify the virtual connection. This GUID is generated by the client when initiating the encapsulation connection. The GUID is exchanged between the client and server and MUST be unique within each server. There is no default value.

ClientOKtoSend: The state variable enforces the requirement that the client MUST NOT send application data when there is an outstanding KeepAlive-POST-Request. Only when the KeepAlive-POST-Response is received can the client send a new KeepAlive-POST-Request.

GetSessionState: The variable used to maintain the current disposition of the GET session. There are four possible states: 'Connecting', 'Connected', 'Established', and 'Closed'. The 'Connecting' state indicates that the GET session request has been sent and the KeepAlive handshake has started. The 'Connected' state indicates that proxy negotiations are in progress. Non-proxy connections immediately transition through the 'Connected' state. The 'Established' state indicates that the GET session response has been received. The 'Closed' state indicates a session cleanup in progress.

PostSessionState: The variable used to maintain the current disposition of the POST session. There are four possible states: 'Connecting', 'Connected', 'Established', and 'Closed'. The 'Connecting' state indicates that the POST session request has been sent and the KeepAlive handshake has started. The 'Connected' state indicates that proxy negotiations are in progress. Non-proxy connections immediately transition through the 'Connected' state. The 'Established' state indicates the POST session response has been received. The 'Closed' state indicates a session cleanup in progress.

ConnectionState: The variable used to maintain the current disposition of the virtual KeepAlive connection. There are three possible states: 'Connecting', 'Established', and 'Closed'. The 'Connecting' indicates that the GET/POST session creation is in progress. The 'Established' state indicates that both GET/POST sessions have been successfully created and application data can begin to flow over the virtual connection. The 'Closed' state indicates that the connection can no longer send or receive application data; the virtual connection sessions are closed.

ProxyConnection: The indicator of whether the current connection is a connection to a proxy or a direct connection to a server. The value is set to TRUE after the client determines that a proxy is to be used. The default value is FALSE.

3.3.1.2 Proxy State Information

The following details about the state information define the context clients need to establish connections with proxies. This proxy configuration information MUST be provided to the client prior to connection establishment. The source of this configuration information is external to the KeepAlive Protocol<24>.

ProxyServerPort: The well-known port number of the target proxy. It is used for establishing a TCP connection to a proxy. By default this is the HTTP well known port 80/TCP or the HTTP alternate well known port 8080/TCP.

ProxyServerHostName: The host name of the target proxy. The name is in the form of an FQDN or an IP Address. If the name is an FQDN, then the client MUST resolve this name to its IP Address. There is no default value.

ProxyAuthRequired: A variable used to indicate if a proxy requires authentication. The client sets this variable to TRUE when it discovers that the proxy needs authentication during its first negotiation with the proxy. When the client initiates a new virtual connection through the same

Copyright © 2014 Microsoft Corporation.

proxy, it SHOULD provide the cached credentials without waiting to be challenged to avoid the overhead of additional message exchanges.

3.3.2 KeepAlive Client Timers

3.3.2.1 ConnectionEstablishment Timer

The ConnectionEstablishment timer SHOULD be used by clients to limit the amount of time a virtual connection negotiation takes to complete. This timer measures the time it takes for a connection to move from the non-established state to the established state. The recommended timeout value is 30 seconds. The ConnectionEstablishment timer event processing is handled as specified in section 3.3.6.1.

3.3.2.2 GetNetworkReceiveIO Timer

The GetNetworkReceiveIO timer SHOULD be used by clients to limit the amount of time a client waits for a KeepAlive connection GET session response. This timer is set after sending the KeepAlive-GET-Request when the KeepAlive connection handshake is finished. The timer SHOULD be greater than the KeepAlive timer (see section 3.3.2.4). The recommended timeout value is 5 minutes. The GetNetworkReceiveIO timer event processing is handled as specified in section 3.3.6.2.

3.3.2.3 PostNetworkReceiveIO Timer

The PostNetworkReceiveIO timer SHOULD be used by clients to limit the amount of time a client waits for a KeepAlive connection POST session response. This timer is set after sending the KeepAlive-POST-Request when the KeepAlive connection handshake is finished. The timer SHOULD be greater than the KeepAlive timer (see section 3.3.2.4). The recommended timeout value is 5 minutes. The PostNetworkReceiveIO timer event processing is handled as specified in section 3.3.6.3.

3.3.2.4 KeepAlive Timer

HTTP Encapsulation protocols do not support a native KeepAlive timer, but rather rely on the encapsulated protocol to provide a KeepAlive mechanism. Encapsulated protocols SHOULD implement their own KeepAlive mechanisms. The SSTP protocol provides its own KeepAlive mechanism using the SSTP_NOOP command<25>. This data serves to keep the KeepAlive connection from being closed by firewalls and proxies. All KeepAlive Connections SHOULD use KeepAlive timers, regardless of whether or not the client detects if a connection is a proxy connection, as some firewalls and proxies are undetectable. The default client KeepAlive timeout value is 45 seconds. The KeepAlive timer event processing is handled as specified in section 3.3.6.4.

3.3.3 KeepAlive Client Initialization

3.3.3.1 Protocol Initialization

The KeepAlive protocol is not initialized until a request to open an encapsulated connection is made by the application. The variables defined by the abstract data model are initialized to their default values when a KeepAlive connection request is made.

Copyright © 2014 Microsoft Corporation.

3.3.4 KeepAlive Client Higher-Layer Triggered Events

3.3.4.1 Establishing a KeepAlive Encapsulation Connection

When the application requests a KeepAlive connection, the KeepAlive protocol layer MUST initialize the KeepAlive connection state variables as specified in the abstract data model (see section 3.3.1). After the connection state variables are initialized, the KeepAlive protocol enters into the connection establishment phase. Initialization SHOULD include fetching any proxy configuration information<26>.

The ConnectionState MUST be set to 'Connecting'. The ConnectionEstablishment timer SHOULD be started.

The client opens two connections to the server, one for a GET session and one for a POST session, as specified in sections 3.3.4.1.1 and 3.3.4.1.3. If proxy configuration information is supplied, the client MUST go through the proxy for each connection as specified in sections 3.3.4.1.2 and 3.3.4.1.4.

The client generates a new virtual connection GUID, stores it in the VirtualConnectionGUID variable, and sets the GetSessionState and PostSessionState to 'Connecting'.

3.3.4.1.1 Establishing GET Session without Proxy

The client MUST construct the KeepAlive-GET-Request-URI as the KeepAlive-GET-Request-Relative-URI, with the ServerHost as Relay-Server-Name, and the VirtualConnectionGUID variable as Virtual-Connection-GUID.

The client specifies the HOST header and sets the value to equal the ServerHost variable as specified in section 2.2.1.2.7.

The client MUST construct a KeepAlive-GET-Request as specified in section 2.2.3.1 with required headers.

The client MUST establish a TCP connection to the server identified with ServerHost and ServerPort and send the KeepAlive-Get-Request.

3.3.4.1.2 Establishing GET Session with Proxy

The client sets the ProxyConnection to TRUE.

The client generates a Request ID GUID.

The client MUST construct the KeepAlive-GET-Request-URI as the KeepAlive-GET-Request-Absolute-URI, with the ServerHost as Relay-Server-Name, the VirtualConnectionGUID variable as Virtual-Connection-GUID, and the preceding generated Request ID GUID as the KeepAlive-Encapsulation-Request-ID.

The client specifies the HOST header and sets the value to equal the ServerHost variable as specified in section 2.2.1.2.7.

The client specifies the proxy-Connection header with the Keep-Alive value as specified in section 2.2.1.2.9.

The client MUST construct a KeepAlive-GET-Request as specified in section 2.2.3.1 with required headers.

Copyright © 2014 Microsoft Corporation.

If the ProxyAuthRequired is set, the client MUST add additional proxy authentication headers to the request.

The client MUST establish a TCP connection to the server identified with ProxyServerHostName and ProxyServerPort and send the KeepAlive-GET-Request.

3.3.4.1.3 Establishing POST Session without Proxy

The client MUST construct the KeepAlive-POST-Request-URI as the KeepAlive-POST-Request-Relative-URI, with the ServerHost as Relay-Server-Name and the VirtualConnectionGUID variable as Virtual-Connection-GUID.

The client MUST construct a KeepAlive-POST-Request as specified in section 2.2.3.2.with required headers.

The KeepAlive-Entity-Body MUST contain the Encapsulation-Echo-String message.

The client MUST establish a TCP connection to the server identified with ServerHost and ServerPort and send the KeepAlive-POST-Request.

3.3.4.1.4 Establishing POST Session with Proxy

The client sets the ProxyConnection to TRUE.

The client generates a Request ID GUID.

The client MUST construct the KeepAlive-GET-Request-URI as the KeepAlive-POST-Request-Absolute-URI, with the ServerHost as Relay-Server-Name, the VirtualConnectionGUID variable as Virtual-Connection-GUID, and the newly generated Request ID GUID for the KeepAlive-Encapsulation-Request-ID.

The client specifies the proxy-Connection header with the Keep-Alive value as specified in section 2.2.1.2.9.

The client MUST construct a KeepAlive-POST-Request as specified in section 2.2.3.2.with required headers.

If the ProxyAuthRequired is set, the client MUST add additional proxy authentication headers to the request.

The KeepAlive-Entity-Body MUST contain the Encapsulation-Echo-String message.

The client MUST establish a TCP connection to the server identified with ProxyServerHostName and ProxyServerPort and send the KeepAlive-POST-Request.

3.3.4.2 Closing a KeepAlive Connection

The client SHOULD close the KeepAlive virtual connection by closing both the POST and GET sessions as specified in sections 3.3.4.3 and 3.3.4.4.

The ConnectionState then transitions into the 'Closed' state. The connection state variables SHOULD be discarded.

3.3.4.3 Closing a KeepAlive POST Session

The client SHOULD close the POST session by sending a graceful TCP disconnect. The PostSessionState is set to the 'Closed' state.

Copyright © 2014 Microsoft Corporation.

TCP client or server connections are closed using either a Graceful Close (TCP FIN Flag set to 1) or Abortive Close (TCP Reset Flag set to 1) mechanism. Closing the connection gracefully provides clients and servers with the benefit of never having to resend unacknowledged payload data. Abortive Closed connections are efficient in connection tear down but can require the client or server to resend payload data which has not been acknowledged at the TCP level. HTTP Encapsulation of SSTP Protocols relies on SSTP to ensure that any unacknowledged payload data is later resent. The choice of connection close mechanism is one of efficiency (efficient connection tear down verses efficient byte transmission). This document does not mandate that implementations adhere to one approach or another. Implementations can choose to close HTTP Encapsulated Connections either gracefully or abortively based on the implementation's requirements.

3.3.4.4 Closing a KeepAlive GET Session

The client SHOULD close the GET session by sending a graceful TCP disconnect. The GetSessionState is set to the 'Closed' state.

TCP client or server connections are closed using either a Graceful Close (TCP FIN Flag set to 1) or Abortive Close (TCP Reset Flag set to 1) mechanism. Closing the connection gracefully provides clients and servers with the benefit of never having to resend unacknowledged payload data. Abortive Closed connections are efficient in connection tear down but can require the client or server to resend payload data which has not been acknowledged at the TCP level. HTTP Encapsulation of SSTP Protocols relies on SSTP to ensure that any unacknowledged payload data is later resent. The choice of connection close mechanism is one of efficiency (efficient connection tear down verses efficient byte transmission). This document does not mandate that implementations adhere to one approach or another. Implementations can choose to close HTTP Encapsulated Connections either gracefully or abortively based on the implementation's requirements.

3.3.4.5 Re-Opening a KeepAlive POST Session

The client SHOULD re-open the KeepAlive POST session, as directed by the application layer, after the closing of the KeepAlive POST session (see section 3.3.4.3). The POST session MUST NOT be re-opened after the KeepAlive virtual connection has been closed (see section 3.3.4.2).

3.3.4.6 Re-Opening a KeepAlive GET Session

The GET session MUST NOT be re-opened, after closing of the KeepAlive GET session.

3.3.4.7 Sending Application Data

To send, the KeepAlive connection MUST be in the 'Established' state, and the ClientOKtoSend state MUST be TRUE. If either condition is not met, the client MUST buffer the data and the processing stops.

If the client is able to send, the client MUST set the ClientOKtoSend state to FALSE. The client sends the application data as specified in section 3.3.4.7.1. If the ProxyConnection state variable is set to TRUE, the client MUST instead send the application data with additional proxy headers (see section 3.3.4.7.2).

3.3.4.7.1 Sending Application Data without Proxy

The client sends the SSTP stream data over the POST session within a KeepAlive-POST-Request. The SSTP stream data is contained within the KeepAlive-Entity-Body. This content length MUST NOT exceed the 32768 octet limit imposed by the KeepAlive protocol.

The client constructs a KeepAlive-POST-Request as defined in section 2.2.3.2.

Copyright © 2014 Microsoft Corporation.

The client MUST construct the KeepAlive-POST-Request-URI as the KeepAlive-POST-Request-Relative-URI, with the ServerHost as Relay-Server-Name and the VirtualConnectionGUID variable as Virtual-Connection-GUID.

The KeepAlive-Content-Length header is set equal to the application data length.

The client specifies the HOST header and sets the value to equal the ServerHost variable as specified in section 2.2.1.2.7.

The client MUST send the KeepAlive-POST-Request message on the POST session.

The client SHOULD start the PostNetworkReceiveIO timer.

3.3.4.7.2 Sending Application Data with Proxy

The client sends the SSTP stream data over the POST session within a KeepAlive-POST-Request. The SSTP stream data is contained within the KeepAlive-Entity-Body. This content length MUST NOT exceed the 32768 octet limit imposed by the KeepAlive protocol.

The client constructs a KeepAlive-POST-Request as defined in section 2.2.3.2.

The client generates a Request ID GUID.

The client MUST construct the KeepAlive-POST-Request-URI as the KeepAlive-POST-Request-Absolute-URI, with the ServerHost as Relay-Server-Name, the VirtualConnectionGUID variable as Virtual-Connection-GUID, and the preceding generated Request ID GUID as the KeepAlive-Encapsulation-Request-ID.

The KeepAlive-Content-Length header is set equal to the application data length.

The client specifies the proxy-Connection header with the Keep-Alive value as specified in section 2.2.1.2.9.

The client specifies the HOST header and sets the value to equal the ServerHost variable as specified in section 2.2.1.2.7.

If the ProxyAuthRequired is set, the client MUST add additional proxy authentication headers to the request.

The client MUST send the KeepAlive-POST-Request message on the POST session.

The client SHOULD start the PostNetworkReceiveIO timer.

3.3.5 KeepAlive Client Message Processing Events and Sequencing Rules

3.3.5.1 KeepAlive-POST-Response Processing

Upon receiving data on the POST session, the client MUST scan the data to verify that it has received an HTTP response status line, as specified in section 2.2.2.3.1. If not, this is a protocol error on the POST session, and the client MUST close the virtual KeepAlive connection (see section 3.3.4.2).

The HTTP response header MUST be parsed and the status code and response body extracted.

Copyright © 2014 Microsoft Corporation.

3.3.5.1.1 Status Code: 200 (OK)

Status Code 200 is processed differently depending on state of the KeepAlive connection. If the KeepAlive virtual connection handshake is in progress, that is, the PostSessionState is 'Connecting', then Status Code 200 is processed as specified in section 3.3.5.1.1.1, otherwise the response is processed as specified in section 3.3.5.1.1.2.

3.3.5.1.1.1 Handshake POST Response Processing

If the ConnectionState is 'Established' and the PostSessionState is 'Connecting', the POST session has been successfully re-opened., the PostSessionState MUST transition to the 'Established' state.

If the PostSessionState is 'Connecting', the receipt of a KeepAlive-POST-Response causes the PostSessionState to transition to 'Connected'.

If the ConnectionState is 'Connecting', the client SHOULD validate that the response entity body contains the KeepAlive-POST-Response-No-Data message. If they are not the same it is a protocol error and the connection MUST be closed (see section 3.3.4.2).

The receipt of KeepAlive-POST-Response-No-Data on the POST session completes the POST session establishment. The client transitions the PostSessionState to 'Established'.

If the GetSessionState is not 'Established', the response processing stops here.

If the GetSessionState is 'Established', then the virtual connection is established and the server transitions ConnectionState to the 'Established' state.

The ConnectionState timers SHOULD be stopped and no further timer expiration processing is performed. The KeepAlive timer SHOULD be started. The ClientOKtoSend state MUST be set to TRUE. The client is now ready to send application data.

If there is buffered data to send, the data MUST be sent now as specified in section 3.3.4.7.

A KeepAlive-GET-Request MUST be sent (see section 3.3.5.3) to allow the server to send data back to the client.

3.3.5.1.1.2 Application Data Posted

The ConnectionState MUST be in the 'Established' state. If not, it is a protocol error and the virtual connection MUST be closed (see section 3.3.4.2).

The receipt of a status code of 200 indicates that the previously sent application data has been received by the server.

The KeepAlive-Content-Length MUST be 0. If not, it is a protocol error and the virtual connection MUST be closed.(see section 3.3.4.2).

The PostNetworkReceiveIO timer SHOULD be stopped and no further timer expiration processing is performed.

The ClientOKtoSend state MUST be set to TRUE. If there is buffered data to send, the data MUST be sent now as specified in section 3.3.4.7.

Copyright © 2014 Microsoft Corporation.

3.3.5.1.2 Status code: 400 (Bad Request)

The server has rejected the connection request because an encapsulation version does not equal the required value (see section 2.2.2.2.1.1.2) or because of a protocol error. The client MUST close all connections associated with this virtual connection (see section 3.3.4.2).

3.3.5.1.3 Status codes: 401 (Unauthorized) / 407 (ProxyAuthentication Required)

HTTP status code values of 401 (Unauthorized) or 407 (ProxyAuthentication Required)

indicate that the proxy requires the client to authenticate. Common authentication schemes include Basic and Digest, as specified in [RFC2617], and NTLM HTTP Authentication, as specified in [RFC4559].

The client sets the ProxyAuthRequired state variable to TRUE. Subsequent connection attempts to the same proxy SHOULD avoid the proxy challenge message by sending the proxy authentication credentials as part of the KeepAlive-POST-Request.

Depending on the authentication method, multiple round trips can happen to complete the authentication process. That is, the client MUST expect to get multiple 401 and 407 messages. It MUST follow [RFC2617] and [RFC4559] to set proper authentication headers and retry the proxy connection.

For processing required to retry the proxy connection, see section 3.3.4.1.4.

The ConnectionEstablishment timer SHOULD be restarted before re-attempting the KeepAlive connection handshake.

3.3.5.1.4 All Other Status Codes

All other status codes are fatal, the virtual connection MUST be closed as specified in section 3.3.4.2.

3.3.5.2 KeepAlive-GET-Response Processing

Upon receiving data on the GET session, the client MUST scan the data to verify that it has received an HTTP response status line, as specified in section 2.2.2.3.1. If not, a protocol error exists on the GET session, and the client MUST close the virtual KeepAlive connection (see section 3.3.4.2).

The HTTP response header MUST be parsed, and the status code and response body extracted.

3.3.5.2.1 Status code: 200 (OK)

Status Code 200 is processed differently depending on state of the KeepAlive connection. If the ConnectionState is 'Connecting', then Status Code 200 is processed as specified in section 3.3.5.2.1.1; otherwise see section 3.3.5.2.1.2 to handle the response with application data.

3.3.5.2.1.1 Handshake GET Response Processing

The receipt of a KeepAlive-GET-Response message on the GET session causes the GetSessionState variable to transition into the 'Connected' state.

The client MUST compare the response body string against the Encapsulation-Echo-String sent earlier on the POST session. If they are not equal, it is a violation of the protocol and the connection MUST be closed (see section 3.3.4.2).

Copyright © 2014 Microsoft Corporation.

The receipt of Encapsulation-Echo-String on the GET session completes the GET session establishment. The client transitions GetSessionState to 'Established'.

If the PostSessionState is not 'Established', the processing stops here.

If the PostSessionState is 'Established', then the virtual connection is established. The client transitions ConnectionState to the 'Established' state.

The ConnectionEstablishment timer SHOULD be stopped and no further timer expiration processing is performed. The KeepAlive timer SHOULD be started. The ClientOKtoSend state MUST be set to TRUE. The client is ready to send and receive application data.

If there is buffered data, the data MUST be sent now as defined in section 3.3.4.7.

The client MUST send a KeepAlive-GET-Request as specified in section 3.3.5.3 to allow the server to send data back to the client.

3.3.5.2.1.2 Application Data GET Response Processing

The receipt of a KeepAlive-GET-Response message with status code of 200 indicates application data has arrived.

The KeepAlive-Content-Length MUST be greater than zero.

The KeepAlive-GET-Response-Entity-Body contains the application data which is passed to the application layer for processing.

The GetNetworkReceiveIO timer SHOULD be cleared.

The client MUST immediately send a KeepAlive-GET-Request (see section 3.3.5.3) on the GET session so the client can receive more application data from the server.

3.3.5.2.2 Status code: 400 (Bad Request)

The server has rejected the connection request because an encapsulation version does not equal the required value (see section 2.2.2.2.1.1.2). The client MUST close all connections associated with this virtual connection (see section 3.3.4.2) and SHOULD NOT retry this connection attempt.

3.3.5.2.3 Status codes: 401 (Unauthorized) and 407 (ProxyAuthentication Required)

HTTP status code values of 401 (Unauthorized) or 407 (Proxy Authentication Required)

indicate that the proxy requires the client to authenticate. Common authentication schemes include Basic and Digest, as specified in [RFC2617], and Negotiated or NTLM HTTP Authentication, as specified in [RFC4559].

The client sets the ProxyAuthRequired state variable to TRUE. Subsequent connection attempts to the same proxy SHOULD avoid the proxy challenge message by sending the proxy authentication credentials as part of the KeepAlive-GET-Request.

Depending on the authentication method, multiple round trips can happen to complete the authentication process. That is, the client MUST expect to get multiple 401 and 407 messages. It MUST follow [RFC2617] and [RFC4559] to set proper authentication headers and retry the proxy connection.

For processing required to retry the proxy connection, see section 3.3.4.1.2.

Copyright © 2014 Microsoft Corporation.

The ConnectionEstablishment timer SHOULD be reset before re-attempting the KeepAlive handshake (see section 3.3.4.1).

3.3.5.2.4 All Other Status Codes

All other status codes are fatal, the virtual connection MUST be closed as specified in section 3.3.4.2.

3.3.5.3 Sending a KeepAlive-GET-Request

The client sends the KeepAlive-GET-Request message as specified in section 3.3.5.3.2. If the ProxyConnection state variable is set to TRUE, the client MUST instead send the KeepAlive-GET-Request message with additional proxy headers (see section 3.3.5.3.2).

3.3.5.3.1 Sending Request for Application Data without Proxy

The client constructs a KeepAlive-GET-Request as defined in section 2.2.3.1.

The client MUST construct the KeepAlive-GET-Request-URI as the KeepAlive-GET-Request-Relative-URI, with the ServerHost as Relay-Server-Name and the VirtualConnectionGUID variable as Virtual-Connection-GUID.

The client specifies the HOST header and sets the value to equal the ServerHost variable as specified in section 2.2.1.2.7.

The client MUST send the KeepAlive-GET-Request message on the GET session.

The client MUST start the GetNetworkReceiveIO timer.

3.3.5.3.2 Sending Request for Application Data with Proxy

The client constructs a KeepAlive-GET-Request as defined section 2.2.3.1.

The client generates a Request ID GUID.

The client MUST construct the KeepAlive-GET-Request-URI as the KeepAlive-GET-Request-Absolute-URI, with the ServerHost as Relay-Server-Name, the VirtualConnectionGUID variable as the Virtual-Connection-GUID, and the preceding generated Request ID GUID for the KeepAlive-Encapsulation-Request-ID.

The client specifies the HOST header and sets the value to equal the ServerHost variable as specified in section 2.2.1.2.7.

The client specifies the proxy-Connection header with the Keep-Alive value as specified in section 2.2.1.2.9.

If the ProxyAuthRequired is set, the client MUST add additional proxy authentication headers to the request.

The client MUST send the KeepAlive-GET-Request message on the GET session.

The client MUST start the GetNetworkReceiveIO timer.

Copyright © 2014 Microsoft Corporation.

3.3.6 KeepAlive Client Timer Events

3.3.6.1 ConnectionEstablishment Timer Event

The ConnectionEstablishment timer event fires when the ConnectionEstablishment timer for a given KeepAlive connection expires before the connection can be established. If this timer expires before the KeepAlive connection enters the 'Established' state, the virtual KeepAlive connection SHOULD be closed by the client.

3.3.6.2 GetNetworkReceiveIO Timer Event

The GetNetworkReceiveIO timer event fires when a GET session response is not received within the GetNetworkReceiveIO interval. If the GetNetworkReceiveIO event triggers, the client SHOULD close the KeepAlive connection and MAY retry immediately. Well behaved connections will not see this event, so clients can expect this event to be transient, and SHOULD always attempt to establish a new KeepAlive connection with the target server.

3.3.6.3 PostNetworkReceiveIO Timer Event

The PostNetworkReceiveIO timer event fires when a POST session response is not received within the PostNetworkReceiveIO interval. If the PostNetworkReceiveIO event triggers, the client SHOULD close the KeepAlive connection and MAY retry immediately. Well behaved connections will not see this event, so clients can expect this event to be transient, and SHOULD always attempt to establish a new KeepAlive connection with the target server.

3.3.6.4 KeepAlive Timer Event

A KeepAlive Event SHOULD trigger an encapsulated protocol message such as an SSTP_NOOP command to be sent across the wire<27>. Well behaved connections will see this event every KeepAlive timer interval when the session is idle. The timer SHOULD be restarted each time it fires.

3.3.7 KeepAlive Client Other Local Events

If the POST session receives a transport disconnect, the client SHOULD set the PostSessionState to 'Closed' and attempt to re-open the POST session (see section 3.3.7.1). If re-opening the POST session fails, the KeepAlive connection MUST be closed (see section 3.3.4.2). The application layer can immediately attempted a new KeepAlive connection as specified in section 3.3.4.1.

If the GET session receives a transport disconnect, the KeepAlive connection MUST be closed (see section 3.3.4.2). The application layer can immediately attempted a new KeepAlive connection as specified in section 3.3.4.1.

The client SHOULD let the higher layer decide whether to wait before establishing a new KeepAlive virtual connection to the target server, or use a different encapsulation protocol to establish a connection to the server.

3.3.7.1 Re-Opening the POST Session after a Transport Disconnect

On a POST session TCP disconnect event, the client can re-open the POST session without reestablishing a new virtual KeepAlive connection. Re-opening the POST session can only be reattempted if the GET session remains connected. If the GET session also receives a TCP disconnect, the KeepAlive virtual connection SHOULD be closed (see section 3.3.4.2).

Copyright © 2014 Microsoft Corporation.

The PostSessionState MUST be in the 'Closed' state and the KeepAlive ConnectionState MUST be in the 'Established' state. Otherwise it is a protocol error and the virtual connection MUST be closed (see section 3.3.4.2).

To re-open the POST session, the client MUST set the PostSessionState to 'Connecting' and follow section 3.3.4.5 to reopen POST TCP connection. The following figure shows the PostSessionState transitions.

The client MUST use the existing virtual connection GUID in the VirtualConnectionGUID state variable to open the POST session.

The Encapsulation-Echo-String message MUST NOT be sent. Instead the previously failed, buffered, KeepAlive-POST-Request-Entity-Body SHOULD be resent. If no KeepAlive-POST-Request was outstanding when the TCP disconnect occurred, then the client SHOULD send the next available chunk of application data.

The client MUST set the ClientOKtoSend to TRUE and sends the application data as specified in section 3.2.4.2.

Copyright © 2014 Microsoft Corporation.

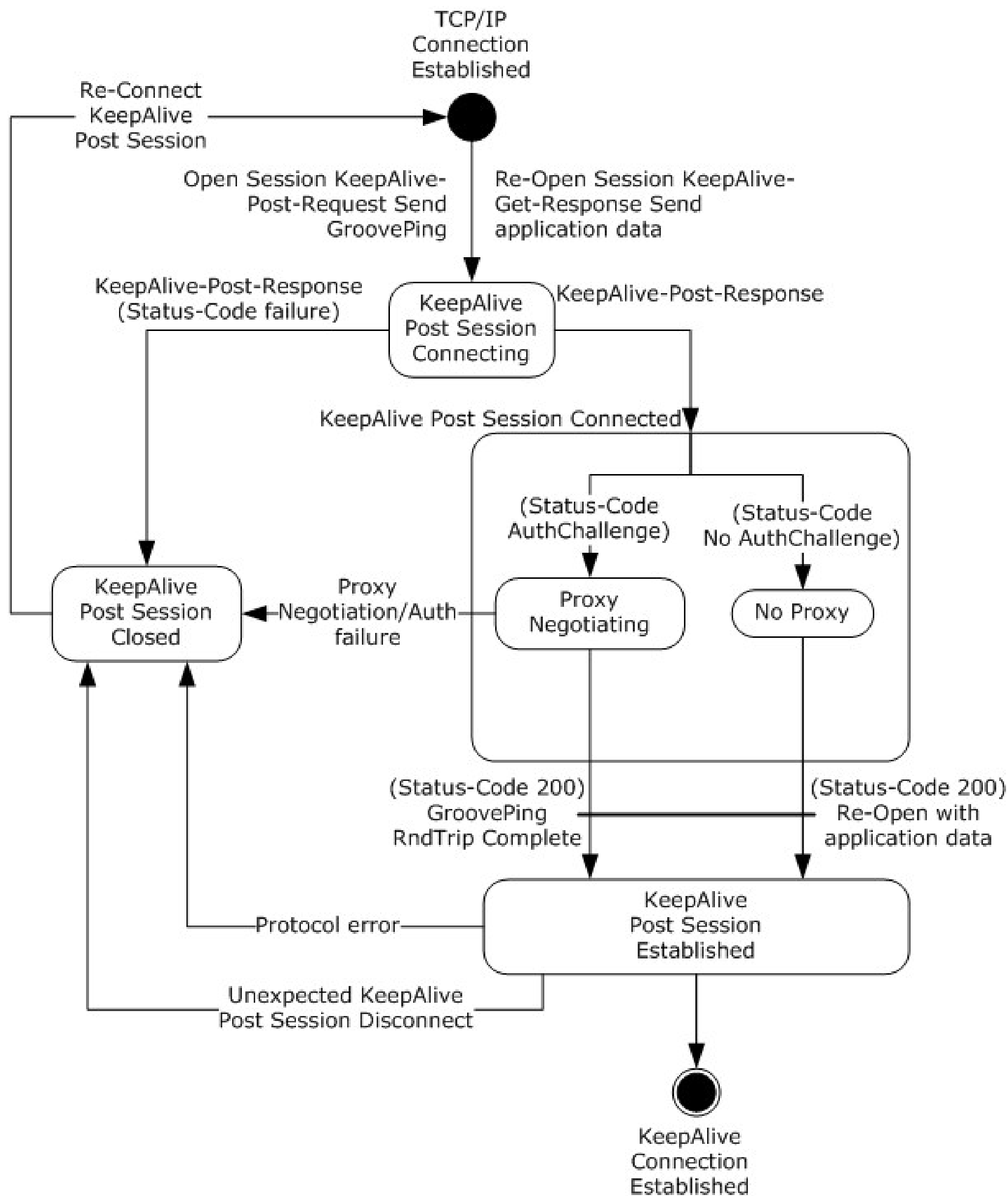

Figure 17: Client re-opening the post-session state diagram

3.4 KeepAlive Encapsulation Protocol Server Details

3.4.1 KeepAlive Server Abstract Data Model

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

Copyright © 2014 Microsoft Corporation.

3.4.1.1 Connection State Information

See section 3.3.1.1 for a list of connection state variables that are shared with the client.

VirtualConnectionGUIDList: The global list of virtual connection GUIDs of all active connections. This list allows the application to quickly lookup a virtual connection GUID to determine if it is a known virtual connection GUID. This list also contains a reference to the per-connection state variables for the associated GUID.

ServerOKtoSend: The state variable enforces the requirement that the server MUST NOT send application data on a KeepAlive-GET-Response until after receiving a KeepAlive-GET-Request. All subsequent KeepAlive-GET-Response messages MUST only be sent in response to an outstanding KeepAlive-GET-Request messages.

GetSessionReady: The state variable enforces the requirement that the server MUST NOT send application data on the GET session until the client has received the KeepAlive-POST-Response containing the Encapsulation-Echo-String. The first packet of application data received by the server on the POST session is an implied acknowledgement, which indicates that the client has received the Encapsulation-Echo-String. A value of TRUE indicates that the server is ready to send application data. The default state is FALSE.

3.4.2 KeepAlive Server Timers

3.4.2.1 ConnectionEstablishment Timer

The ConnectionEstablishment timer SHOULD be used by the server to limit the amount of time a KeepAlive connection handshake takes to complete. This timer measures the time it takes for a connection to move from the connected to the established state. This is a per-connection timer. The recommended timeout value is 90 seconds. The ConnectionEstablishment timer event processing is handled as specified in section 3.4.6.1.

3.4.2.2 IdleConnection Timer

The IdleConnection timer SHOULD be used by the server to determine if a connection has become idle. This timer is set after the KeepAlive handshake is finished. The timer SHOULD be greater than the KeepAlive timer used by the client, as specified in section 3.3.2.4. This is a per-connection timer. The recommended timeout value is 90 seconds. The IdleConnection timer event processing is handled as specified in section 3.4.6.2.

3.4.2.3 KeepAlive Timer

HTTP Encapsulation protocols do not support a native KeepAlive timer, but rely on the encapsulated protocol to provide a KeepAlive mechanism. Encapsulated protocols SHOULD implement their own KeepAlive mechanisms. The SSTP protocol provides its own KeepAlive mechanism using the SSTP_NOOP command<28>. This data serves to keep the KeepAlive connection from being closed by firewalls and proxies. All KeepAlive Connections SHOULD use KeepAlive timers, regardless of whether or not the client detects if a connection is a proxy connection, as some firewalls and proxies are undetectable. The default client KeepAlive timeout value is 45 seconds. The KeepAlive timer event processing is handled as specified in section 3.4.6.3.

3.4.3 KeepAlive Server Initialization

3.4.3.1 Protocol Initialization

When the server starts it MUST initialize the HTTP stack<29>.

Copyright © 2014 Microsoft Corporation.

A KeepAlive connection protocol is not initialized until a request to open an encapsulated connection is received by the server. The variables defined by the abstract data model are initialized when the KeepAlive connection request is received.

3.4.3.2 KeepAlive Listener

The Server MUST open a listener socket on the KeepAlive port. The KeepAlive connection port uses the well-known HTTP port 80/TCP. Alternate ports MAY be used, but non-default port information MUST be conveyed to the client.

3.4.4 KeepAlive Server Higher-Layer Triggered Events

3.4.4.1 Closing a KeepAlive Connection

The server MUST close the POST and GET sessions by sending a graceful TCP disconnect on each session. The ConnectionState then transitions into the 'Closed' state. All connection state information SHOULD be discarded.

TCP client or server connections are closed using either a Graceful Close (TCP FIN Flag set to 1) or Abortive Close (TCP Reset Flag set to 1) mechanism. Closing the connection gracefully provides clients and servers with the benefit of never having to resend unacknowledged payload data. Abortive Closed connections are efficient in connection tear down but can require the client or server to resend payload data which has not been acknowledged at the TCP level. HTTP Encapsulation of SSTP Protocols relies on SSTP to ensure that any unacknowledged payload data is later resent. The choice of connection close mechanism is one of efficiency (efficient connection tear down verses efficient byte transmission). This document does not mandate that implementations adhere to one approach or another. Implementations can choose to close HTTP Encapsulated Connections either gracefully or abortively based on the implementation's requirements.

3.4.4.2 Closing a POST Session

The server MUST close the POST session by sending a graceful TCP disconnect. The connection then transitions into the 'Closed' state.

The PostSessionState MUST be set to 'Closed'.

All connection state information MUST be retained.

3.4.4.3 Sending Application Data

To send, the KeepAlive ConnectionState MUST be in the 'Established' state, and the GetSessionReady state MUST be TRUE, and the ServerOKtoSend state MUST be TRUE. If any of the conditions are not met, the application data MUST be buffered in the order it was received.

If the application data needs to be buffered, the processing stops here.

To send, the server sends the SSTP stream data over the GET session framed within a KeepAlive-GET-Responses. The SSTP stream data is contained within the KeepAlive-GET-Response-Entity-Body. The server MUST construct a KeepAlive-Content-Length header that is equal to the length of the entity body. This content length MUST NOT exceed the 32768 octet limit imposed by the KeepAlive protocol.

The Response-Status-Line MUST be set to a status code of 200 and Reason-Phrase of "OK"

as specified in section 2.2.2.3.1.

Copyright © 2014 Microsoft Corporation.

The server constructs a KeepAlive-GET-Response as defined in section 2.2.3.3.

The Connection header with Keep-Alive token MUST be specified as specified in section 2.2.1.3 for persistent connection interoperability with HTTP 1.1 proxies as recommended in [RFC2068].

The server MUST set ServerOKtoSend to FALSE.

The server MUST send the KeepAlive-GET-Response message on the GET session.

3.4.5 KeepAlive Server Message Processing Events and Sequencing Rules

3.4.5.1 GET Session Processing

Upon receiving data on the GET session, the server MUST first parse the data to verify that it starts with HTTP GET request line, as specified in section 2.2.3.1.

The server then checks if the ConnectionState is 'Connecting'; if so, the KeepAlive-GET-Request message is handled as specified in section 3.4.5.1.1, otherwise see section 3.4.5.1.2.

3.4.5.1.1 Receiving a KeepAlive-GET-Request (Handshake)

Upon receipt of a KeepAlive-GET-Request, the server transitions the ConnectionState and GetSessionState to the 'Connected' state. If the GetSessionState is uninitialized, the ConnectionEstablishment timer MUST be started.

The server validates the KeepAlive-GET-Request message (see section 2.2.3.1) using the following procedure:

1. The server MUST validate the KeepAlive-GET-Request-URI (see section 2.2.3.1.1) and extract the version, server name, virtual connection GUID, encapsulation type, content length, and Request ID. If the parsing fails, it is a protocol error and the server MUST close the connections (see section 3.3.4.2).

2. The server SHOULD<30> check the KeepAlive-Encapsulation-Version and send a KeepAlive-Response with a status code of 400. See section 3.4.5.1.1.2 if the version does not equal the required version (see section 2.2.2.2.1.1.2).

3. If the encapsulation type is not KeepAlive, it is a protocol error and the virtual connection MUST be closed.

4. The server SHOULD<31> verify that the server name in the message equals its own name and close the virtual connection if the names are not equal.

5. The server SHOULD ignore the Request ID.

6. The server MUST examine the Virtual-Connection-GUID to validate that the KeepAlive-GET-Request is a new connection request. The Virtual-Connection-GUID SHOULD be maintained in the VirtualConnectionGUIDList. If the PostSessionState is 'Connected', then the Virtual-Connection-GUID SHOULD be found in the VirtualConnectionGUIDList. If the PostSessionState is uninitialized, the Virtual-Connection-GUID MUST be new and the Virtual-Connection-GUID SHOULD be added to VirtualConnectionGUIDList. If the PostSessionState is anything else, it is a protocol error and the server closes the virtual KeepAlive connection (see section 3.3.4.2).

If the PostSessionState is not ‘Connected’, the processing stops here.

Copyright © 2014 Microsoft Corporation.

If the PostSessionState is 'Connected', the virtual connection is established. The server transitions the GetSessionState and the PostSessionState and the ConnectionState to the 'Established' state. The server clears the ConnectionEstablishment timer.

The server continues as described in section 3.4.5.1.1.1 to complete the handshake by sending a KeepAlive-GET-Response.

3.4.5.1.1.1 Handshake GET Response Processing

The server MUST send a KeepAlive-GET-Response message on the GET session to complete the KeepAlive Encapsulation connection handshake.

A successful KeepAlive-GET-Response message MUST contain a Response-Status-Line (see section 2.2.2.3.1) with a status code equal to 200 (OK).

The KeepAlive-GET-Response message sent to the client MUST contain the Encapsulation-Echo-String received on the first KeepAlive-POST-Request.

The KeepAlive-Content-Length response header value MUST be set to the length of the Encapsulation-Echo-String.

The KeepAlive connection response MUST construct the extended HTTP 1.0 response as specified in section 2.2.3.3.

The server sends a KeepAlive-GET-Response on the GET session.

The KeepAlive timer MUST be started.

3.4.5.1.1.2 Sending a KeepAlive-GET-Response with Status code 400

If the protocol version does not equal the required KeepAlive version value (see section 2.2.2.2.1.1.2), the server SHOULD<32> sends a KeepAlive-GET-Response message with a Response-Status-Line (see section 2.2.2.3.1) that contains a status code of 400 and phrase of "Bad Request".

The KeepAlive-GET-Response message with Status code of 400 and phrase of "Bad Request" can also be sent on protocol errors.

The KeepAlive-GET-Response MUST be sent on the GET session. The KeepAlive virtual connection MUST be closed (see section 3.3.4.2).

The server sets ServerOKtoSend to FALSE.

3.4.5.1.2 Receiving a KeepAlive-GET-Request for Application Data

The server validates the KeepAlive-GET-Request message (see section 2.2.3.1) using the following procedure:

1. The server MUST validate the KeepAlive-GET-Request-URI (see section 2.2.3.1.1) and extract the version, server name, virtual connection GUID, encapsulation type, content length, and Request ID. If the parsing fails, it is a protocol error and the server MUST close the connections (see section 3.4.4.1).

2. The server SHOULD<33> check the KeepAlive-Encapsulation-Version and if the version does not equal the required value (see section 2.2.2.2.1.1.2), send a KeepAlive-Response with a status code of 400. See section 3.4.5.1.1.2.

Copyright © 2014 Microsoft Corporation.

3. If the encapsulation type is not KeepAlive, it is a protocol error and the virtual connection MUST be closed.

4. The server SHOULD<34> verify that the server name in the message equals its own name and close the virtual connection if they are not equal.

5. The server SHOULD ignore Request ID

6. The server MUST examine the Virtual-Connection-GUID. The Virtual-Connection-GUID SHOULD be bound in the VirtualConnectionGUIDList.

If any of the preceding validations fails, it is a protocol error, and the virtual KeepAlive connection MUST be closed (see section 3.4.4.1).

The GetSessionReady and the ServerOKtoSend state variables MUST be set to TRUE.

The server MUST follow section 3.4.4.3 to send a KeepAlive-GET-Response on the GET session if there is buffered application data to send to the client.

If there is no application data buffered, the server SHOULD wait for application data. The KeepAlive timer makes sure that there is always some data to be sent.

3.4.5.2 POST Session Processing

Upon receiving data on the POST session, the server MUST first check to see if the data starts with an HTTP POST request line, as specified in section 2.2.3.2. Otherwise it is handled as a protocol error as specified in section 3.4.5.2.4.

The server validates the KeepAlive-POST-Request message (see section 2.2.3.2) using the following procedure:

1. The server MUST validate the KeepAlive-Request-URI (see section 2.2.3.1.1) and extract the version, server name, virtual connection GUID, encapsulation type, and content length. If the parsing fails, it is a protocol error and the server MUST close the connections (see section 3.4.4.2).

2. The server SHOULD<35> check the KeepAlive-Encapsulation-Version and, if the version does not equal the required value (see section 2.2.2.2.1.1.2), send a KeepAlive-POST-Response with a status code of 400. See section 3.4.5.2.4.

3. If the encapsulation type is not KeepAlive, it is a protocol error and the virtual connection MUST be closed.

4. The server SHOULD<36> verify that the server name in the message equals its own name, and, if they are not equal, close the virtual connection.

If the ConnectionState is 'Connecting', the KeepAlive-POST-Request message is further handled as specified in section 3.4.5.2.1.

If the ConnectionState is 'Established', the server handles the KeepAlive-POST-Request message and application data as specified in section 3.4.5.2.2.

All other states are protocol errors.

Copyright © 2014 Microsoft Corporation.

3.4.5.2.1 Receiving a KeepAlive-POST-Request (KeepAlive Handshake)

Upon receipt of a KeepAlive-POST-Request, the server moves the ConnectionState to 'Connected', transitions the PostSessionState to 'Connecting', and starts the ConnectionEstablishment timer if the GetSessionState is uninitialized.

The server validates the KeepAlive-POST-Request message as specified in section 2.2.3.2.

The server MUST examine the Virtual-Connection-GUID to validate that the KeepAlive-POST-Request is a new connection request. The Virtual-Connection-GUID SHOULD be maintained in a VirtualConnectionGUIDList. If the GetSessionState is 'Connected', then the Virtual-Connection-GUID SHOULD be found in the VirtualConnectionGUIDList. If the GetSessionState is uninitialized, the Virtual-Connection-GUID MUST be new and the Virtual-Connection-GUID SHOULD be added to the VirtualConnectionGUIDList. Otherwise it is a protocol error.

The KeepAlive-POST-Request MUST contain the Encapsulation-Echo-String (see section 2.2.1.1.3) in the message entity body. The Encapsulation-Echo-String MUST be saved so it can later be echoed back to the client on the KeepAlive-GET-Response message. If the Encapsulation-Echo-String is missing it is a protocol error.

The server transitions the PostSessionState to the "Connected" state. The server MUST send the KeepAlive-POST-Response as specified in section 3.4.5.2.3.1.

If the GetSessionState is not 'Connected', the processing stops here.

If the GetSessionState is 'Connected', the virtual connection is established. The server transitions the GetSessionState, the PostSessionState, and the ConnectionState to the 'Established' state. The ConnectionEstablishment timer MUST be cleared. The server continues as described in section 3.4.5.1.1.1 to complete the handshake by sending a KeepAlive-GET-Response.

3.4.5.2.2 Receiving a KeepAlive-POST-Request with Application Data

Application data is received when the ConnectionState is in the 'Established' state. Otherwise it is a violation of protocol and the server MUST close the virtual KeepAlive connection, as specified in section 3.4.4.1.

The IdleConnection timer MUST be cleared.

The server validates the KeepAlive-POST-Request message as defined in section 2.2.3.2.

The server MUST examine the Virtual-Connection-GUID to validate that it is an existing virtual connection GUID in the VirtualConnectionGUIDList.

The server can validate that the specified KeepAlive-Content-Length is equal to the length of the KeepAlive-POST-Response-Entity-Body. If the lengths are not equal, it is a protocol error and it MUST be handled as specified in section 3.4.4.1.

The server passes the application data to a higher layer for processing.

The server MUST send the KeepAlive-POST-Response as specified in section 3.4.5.2.3.2.

3.4.5.2.3 Sending a KeepAlive-POST-Response with Status code 200

3.4.5.2.3.1 Handshake POST Response Processing

A successful KeepAlive-POST-Response message MUST contain a Response-Status-Line (see section 2.2.2.3.1) with a status code equal to 200 (OK).

Copyright © 2014 Microsoft Corporation.

The KeepAlive-POST-Response message sent to the client MUST contain the KeepAlive-POST-Response-No-Data within the KeepAlive-POST-Response-Entity-Body.

The KeepAlive-Content-Length response header value MUST be set to the length of the KeepAlive-POST-Response-No-Data message.

The KeepAlive connection response MUST construct the extended HTTP 1.0 response as specified in section 2.2.3.4.

The server sends a KeepAlive-POST-Response on the POST session.

The KeepAlive timer is started.

3.4.5.2.3.2 Application Data POST Response Processing

A successful KeepAlive-POST-Response message MUST contain a Response-Status-Line (see section 2.2.2.3.1) with a status code equal to 200 (OK).

The KeepAlive-POST-Response message sent to the client MUST contain an empty body.

The KeepAlive-Content-Length response header value MUST be to 0.

The KeepAlive connection response MUST construct the extended HTTP 1.0 response as specified in section 2.2.3.4.

The server sends a KeepAlive-POST-Response on the POST session.

A IdleConnection timer SHOULD be restarted on the POST session.

3.4.5.2.4 Sending a KeepAlive-POST-Response with Status Code 400

On protocol version that are not equal to the required KeepAlive version value (see section 2.2.2.2.1.1.2) or protocol error, the server sends a KeepAlive-POST-Response message. The Response-Status-Line (see section 2.2.2.3.1) MUST contain a status code of 400 and phrase of "Bad Request".

The KeepAlive-POST-Response message with Status code of 400 and phrase of "Bad Request" can also be sent on protocol errors.

The KeepAlive-POST-Response MUST be sent on the POST session. The KeepAlive virtual connection MUST be closed (see section 3.4.4.2).

3.4.6 KeepAlive Server Timer Events

3.4.6.1 ConnectionEstablishment Timer Event

The ConnectionEstablishment Timer Event fires when the ConnectionEstablishment timer for a given KeepAlive connection expires before the connection can be established. If this timer expires before the KeepAlive connection enters the established state, all established TCP connections on the KeepAlive virtual connection SHOULD be closed by the server.

3.4.6.2 IdleConnection Timer

The IdleConnection timer fires when the IdleConnection interval expires with any interleaving KeepAlive-GET-Request. If this timer expires, all the virtual KeepAlive connections SHOULD be closed by the server. The timer SHOULD be started after receiving a KeepAlive-GET-Request.

Copyright © 2014 Microsoft Corporation.

3.4.6.3 KeepAlive Timer Event

The KeepAlive timer fires every KeepAlive interval to trigger a send of a KeepAlive-Message across the GET Session from the server to the client. A KeepAlive event SHOULD trigger the server to send an encapsulated protocol message such as an SSTP_NOOP command<37>. Well behaved KeepAlive connections will see this event every KeepAlive interval when the session is idle.

3.4.7 KeepAlive Server Other Local Events

Transport disconnect events are handed differently depending on the session.

If the POST session receives a transport disconnect, the server SHOULD close the POST session (see section 3.4.4.2). The client can then re-open the POST session or close the KeepAlive connection.

If the GET session receives a transport disconnect, the server MUST close the KeepAlive connection (see section 3.3.4.2).

If both sessions simultaneously receive transport disconnects, the KeepAlive connection MUST be closed (see section 3.3.4.2).

3.5 Polling Encapsulation Protocol Client Details

The Polling Encapsulation Protocol is an HTTP based protocol used for firewall and proxy traversal. It provides an HTTP transport which can also negotiate and authenticate with HTTP proxies. Polling Encapsulation provides a virtual connection composed of a single HTTP session. This POST session is layered across multiple TCP connections, where each connection is serialized. The life of a TCP connection is a single HTTP request/response.

POST requests are used to send the application data from the client to the server, while POST responses are used by the client to receive application data from the server. To encapsulate a full-duplex protocol such as SSTP, the client MUST periodically poll the server. The client polls the server with a POST request, which allows the server to respond with a POST response.

3.5.1 Polling Client Abstract Data Model

This section specifies a conceptual model of possible data organization that an implementation maintains to participate in the Polling encapsulation protocol. The specified organization is provided to facilitate the explanation of how the protocol behaves. This document does not mandate that implementations adhere to this model as long as their external behavior is consistent with that specified in this document. The following figure shows a detailed view of the state machine for the polling client session.

Copyright © 2014 Microsoft Corporation.

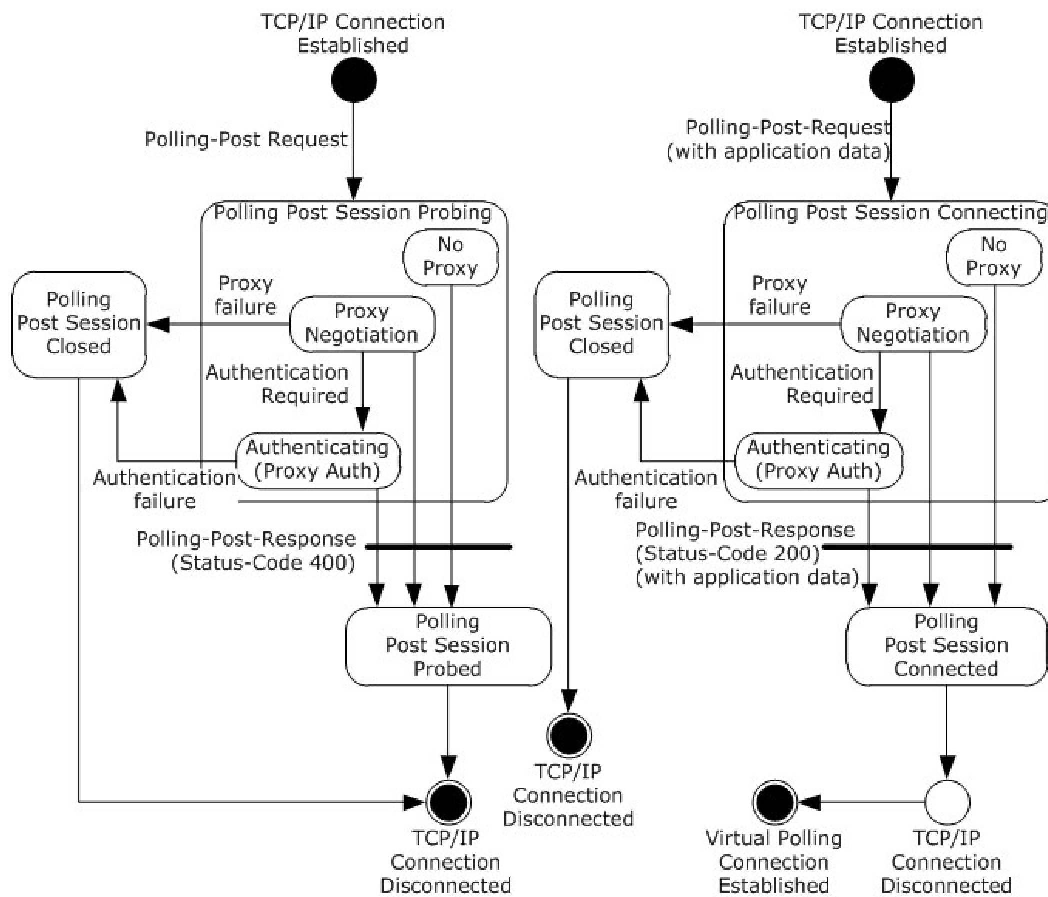

Figure 18: Client polling session state diagram

For a detailed view of the state machine of the polling client connection, see the following diagram.

Copyright © 2014 Microsoft Corporation.

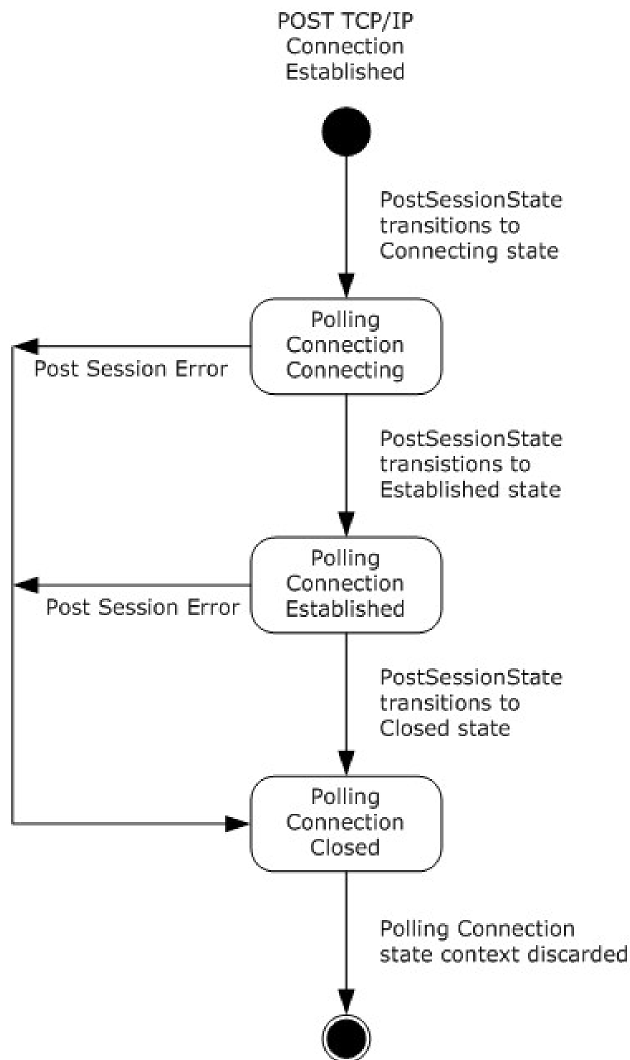

Figure 19: Client polling connection state diagram

3.5.1.1 Connection State Information

The following details about the state information define the context needed to manage a Polling connection. Unless otherwise noted, the following connection state variables are scoped to a single Polling Encapsulation connection. When a Polling virtual connection is terminated, this state information is no longer relevant and SHOULD be discarded.

A client SHOULD support multiple Polling connections to multiple servers concurrently. A client SHOULD support one Polling connection to the same target server (see ServerHost state information). In all cases, each Polling connection MUST maintain separate connection state variable information.

ServerPort: The well-known port number of the target server. By default this is the HTTP well known port 80/TCP.

Copyright © 2014 Microsoft Corporation.

ServerHost: The host name of the target server, in the form of an FQDN or IP Address. There is no default value.

PostSessionState: The variable used to maintain the current disposition of the POST session. There are six possible states: 'Probing', 'Probed', 'Connecting', 'Connected', 'Established', and 'Closed'. The 'Probing' state indicates that the first Polling-POST-Request message of the Polling handshake has been sent. The 'Probed' state indicates that the first Polling-POST-Response message of the Polling handshake has been received. The 'Connecting' state indicates that the Polling-POST-Request with application data was sent on the Polling handshake. The 'Connected' state indicates that proxy negotiations are in progress. Non-proxy connections immediately transition through the 'Connected' state. The 'Established' state indicates that the POST session response was received. The 'Closed' state indicates that session cleanup is in progress.

ConnectionState: The variable used to maintain the current disposition of the

virtual Polling connection. There are three possible states: 'Connecting', 'Established', and 'Closed'. The 'Connecting' state indicates that POST session creation is in progress. The 'Established' state indicates that the POST sessions have been successfully created and application data MAY begin to flow over the virtual connection. The 'Closed' state indicates that the connection can no longer send or receive application data, and the virtual connection session MUST be closed.

ClientOKtoSend: The state variable that enforces the requirement that the client MUST NOT send application data if a Polling-POST-Request is outstanding. This state is set to FALSE each time a Polling-POST-Request is sent, and set to TRUE each time a Polling-POST-Response is received.

VirtualConnectionGUID: A GUID used to uniquely identify the virtual connection. This GUID is generated by the client when initiating the encapsulation connection. The GUID is exchanged between the client and server and MUST be unique within each server. There is no default value.

ProxyConnection: The indicator of whether the connection is a connection using a proxy or a direct connection to a server. The value is set to TRUE after the client determines that a proxy is to be used. The default value is FALSE.

RequestSequenceNum: The current sequence number of the request, as it appears on the wire. It is used to ensure sequencing of request messages. Its initial value starts at 0 and is incremented by 1 on each new request message. The sequence number SHOULD NOT repeat for the life of the virtual connection.

ResponseSequenceNum: The current sequence number of the response. It is used to ensure sequencing of response messages. Its initial value starts at 0 and is incremented by 1 for each new response message. The sequence number SHOULD NOT repeat for the life of the virtual connection.

The PollingMinRepetitionInterval, PollingMaxRepetitionInterval and PollingRepetitionCount values are used by Polling Encapsulation connections to determine the frequency of poll requests for Polling Encapsulation connections. They are sent by the server to the client on each response message.

PollingMinRepetitionInterval: The minimal amount of time in seconds to poll for data on an idle session. It is used by Polling Encapsulation connection logic to send the next polling POST command. The minimum value is used as the low water mark interval in the back off timer calculation (see section 3.5.2.3). The back off calculation is equal to two times the current interval. This value is sent by the server to the client on every POST response<38>.

PollingMaxRepetitionInterval: The maximum amount of time in seconds to poll for data on an idle session. It is used by the application to send the next polling POST command. The maximum value is used as the high water mark interval in the back off timer calculation (see section 3.5.2.3). This value is sent from the server to the client on every POST response<39>.

Copyright © 2014 Microsoft Corporation.

PollingRepetitionCount: The number of times a client is to poll the server at the current poll interval. It is used by Polling Encapsulation connection logic to keep track of the current poll repetition count. This current poll repetition count is initialized to 0 and incremented each time by 1, until it reaches the PollingRepetitionCount value at which point it recalculates (see section 3.5.2.3) the repetition interval based on the PollingMinRepetitionInterval and PollingMaxRepetitionInterval values. If no application data is received before this limit is reached, the back off algorithm recalculates the poll interval. This value is sent by the server to the client on every POST response<40>.

3.5.1.2 Proxy State Information

The state information detailed in this section defines the context clients need to use to establish connections with proxies. This proxy configuration information MUST be provided to the client prior to connection establishment. The source of this configuration information is external to the Polling Protocol<41>.

ProxyServerPort: The well-known port number of the target proxy. It is used for establishing a TCP connection to a proxy. By default this is the HTTP well known port 80/TCP or the HTTP alternate well known port 8080/TCP.

ProxyServerHostName: The host name of the target proxy. The name is in the form of an FQDN or an IP Address. If the name is an FQDN, then the client MUST resolve this name to its IP Address. There is no default value.

ProxyAuthRequired: A variable used to indicate if a proxy requires authentication. The client sets this variable to TRUE when it discovers that the proxy needs authentication during its first negotiation with the proxy. When the client initiates a new virtual connection through the same proxy, it SHOULD provide the cached credentials without waiting to be challenged to avoid the overhead of additional message exchanges.

3.5.1.3 Client State Information

VirtualConnectionGUIDList: The global list of virtual connection GUIDs of all active connections. This list allows the application to quickly lookup a virtual connection GUID to determine if it is a known virtual connection GUID. This list also contains a reference to the per-connection state variables for the associated GUID.

3.5.2 Polling Client Timers

3.5.2.1 ConnectionEstablishment Timer

The ConnectionEstablishment timer can be used to limit the amount of time a Polling connection negotiation takes to complete. This timer measures the time it takes for a connection to move from the connecting to the established state. The recommended timeout value is 3 minutes. The ConnectionEstablishment timer event processing is handled as specified in section 3.5.6.1).

3.5.2.2 Network Receive IO Timer

The NetworkReceiveIO timer SHOULD be used to limit the amount of time a client waits for a Polling connection's POST session network receive operation to complete. The recommended timeout value is 120 seconds. The NetworkReceiveIO timer event processing is handled as specified in section 3.5.6.2.

Copyright © 2014 Microsoft Corporation.

3.5.2.3 Polling Encapsulation Timer

The polling encapsulation timer is used by clients to determine the next interval for polling the server. Because the POST session is half-duplex, a polling mechanism is needed to allow the server to send data to the client, even when the client has no data to send the server. The timer value algorithm has a built in back off mechanism to reduce the overhead of the client polling the server for application data.

The Poll Timer is comprised of three values that are updated by the client on every response received from the server. These values are MaxPollInterval, MinPollInterval, and PollRepetitions. MinPollInterval is measured in seconds and defines the starting poll interval value. PollRepetitions defines the amount of times the client polls using the initial starting value. MaxPollInterval is measured in seconds and defines the poll interval ceiling. Together these values are used to implement the back off algorithm used to manage the frequency at which a client polls the server for data.

The Poll timer determines the current poll interval to be used for the next poll request. The current poll interval value is initialized with the MinPollInterval value. Using the current poll interval, the client polls the number of times specified by PollRepetitions. When the current repetition count reaches the PollRepetitions value, the current poll value is doubled and becomes the new poll interval. This interval is also used by the client to poll the server the number of times specified by PollRepetitions. The doubling of the current poll interval continues until the interval exceeds the MaxPollInterval value. When the MaxPollInterval value is exceeded, the client continues to poll the server indefinitely, using the current value. Polling SHOULD continue at the current poll interval, until the client has data to send to the server, at which point the client resets the current poll interval back to MinPollInterval value. The back off algorithm resets and starts over again.

Once the client establishes a Polling connection, the per-connection polling interval is updated with the server's timer values. The server Poll timer values are returned on the last Polling-POST-Response message of the Polling connection handshake and on every subsequent Polling-POST-Response message. The poll value applies to the next poll operation. Clients SHOULD refresh their local Poll timer values after every Polling-POST-Response, if the timer values changed in the latest Polling-POST-Response message from the server. The Poll Timer values are scoped to a single connection. When the client receives data from the server, the Poll timer values are reset back to the Poll timer values found in the current Polling-POST-Response.

The recommended Poll timer values used for Polling for application data are specified in section 2.2.4.2.1.1. The maximum poll interval value used by polling connections is constrained by limits imposed by firewall and proxies<42>. The Poll timer event processing is handled as specified in section 3.5.6.3. The recommended MaxPollInterval, MinPollInterval, and PollRepetitions values are 120, 5, and 3, respectively.

3.5.3 Polling Client Initialization

3.5.3.1 Protocol Initialization

The Polling Encapsulation protocol is not initialized until a request to open an encapsulated connection is made by the client. The variables defined by the abstract data model are initialized to their default values when a Polling connection request is made.

Copyright © 2014 Microsoft Corporation.

3.5.4 Polling Client Higher-Layer Triggered Events

3.5.4.1 Establishing a Polling Encapsulation Connection

When applications requests a Polling Connection, the Polling protocol layer MUST initialize the Polling connection state variables as specified in the abstract data model (see section 3.5.1). After the connection state variables are initialized, the Polling protocol enters into the connection establishment phase. Initialization SHOULD include fetching any proxy configuration information<43>.

The ConnectionEstablishment timer SHOULD be started.

The client generates a new virtual connection GUID and stores it in the VirtualConnectionGUID variable.

The client set PostSessionState to 'Probing'.

The client opens one connection and a POST session to a target determined by the availability of proxy configuration information. If the client has proxy configuration information available; it uses the port and hostname found in the ProxyServerPort and ProxyServerHostName state information and uses section 3.5.4.1.2 to open a POST session. If no proxy configuration information is available; the client uses the port and hostname found in the ServerPort and ServerHostName state information and uses section 3.5.4.1.1 to open a POST session.

3.5.4.1.1 Establishing POST Session without Proxy

The client MUST construct the Polling-POST-Request-URI as the Polling-POST-Request-Relative-URI (see section 2.2.4.1.1).

The client MUST construct a Polling-POST-Request as specified in section 2.2.4.1 with required headers.

The client MUST construct the Polling-Virtual-Connection-Message as specified in section 2.2.4.1.3.1. This message is embedded within the Polling-Request-Entity-Body

The client MUST construct the Polling-Virtual-Connection-Message with the ServerHost as Relay-Server-Name and VirtualConnectionGUID as Virtual-Connection-GUID.

If the PostSessionState is 'Probing', the Sequence-Number and Checksum MUST both be set to 0. The Polling-Content-Length header MUST contain the length of the Polling-Virtual-Connection-Message.

If the PostSessionState is 'Probed', the Sequence-Number is set to 0, and the Checksum MUST be calculated over the length of the application data that is to be sent, as specified in section 2.2.4.1.3.1.3.

The Polling-Content-Length header MUST contain the length of the Polling-Request-Entity-Body.

The client MUST establish a TCP connection to the server using ServerHost and ServerPort and send the Polling-POST-Request.

3.5.4.1.2 Establishing POST Session with Proxy

The client sets the ProxyConnection to TRUE.

The client MUST construct the Polling-POST-Request-URI as the Polling-POST-Request-Absolute-URI (see section 2.2.4.1.1), with the ServerHost as the HTTP-URL target host name.

Copyright © 2014 Microsoft Corporation.

The client MUST construct a Polling-POST-Request with required headers, as specified in section 2.2.4.1.

The client MUST construct the Polling-Virtual-Connection-Message as specified in section 2.2.4.1.3.1. This message is embedded within the Polling-Request-Entity-Body.

The client MUST construct the Polling-Virtual-Connection-Message with the ServerHost as Relay-Server-Name and VirtualConnectionGUID as Virtual-Connection-GUID.

If the PostSessionState is 'Probing', the Sequence-Number and Checksum MUST both be set to 0. The Polling-Content-Length header MUST contain the length of the Polling-Virtual-Connection-Message.

If the PostSessionState is 'Probed', Sequence-Number is set to 0. The Checksum MUST be calculated (see section 2.2.4.1.3.1.3) over the length of the application data that is to be sent.

The Polling-Content-Length header MUST contain the length of the Polling-Request-Entity-Body.

If the ProxyAuthRequired is set, the client MUST add additional proxy authentication headers to the request.

The client MUST establish a TCP connection to the server identified with ProxyServerHostName and ProxyServerPort and send the Polling-POST-Request.

3.5.4.2 Closing a Polling Connection

The client SHOULD close the POST session by sending a graceful TCP disconnect.

The ConnectionState then transitions into the 'Closed' state. All connection state information SHOULD be discarded.

TCP client or server connections are closed using either a Graceful Close (TCP FIN Flag set to 1) or Abortive Close (TCP Reset Flag set to 1) mechanism. Closing the connection gracefully provides clients and servers with the benefit of never having to resend unacknowledged payload data. Abortive Closed connections are efficient in connection tear down but can require the client or server to resend payload data which has not been acknowledged at the TCP level. HTTP Encapsulation of SSTP Protocols relies on SSTP to ensure that any unacknowledged payload data is later resent. The choice of connection close mechanism is one of efficiency (efficient connection tear down verses efficient byte transmission). This document does not mandate that implementations adhere to one approach or another. Implementations can choose to close HTTP Encapsulated Connections either gracefully or abortively based on the implementation's requirements.

3.5.4.3 Sending Application Data

The Polling Connection MUST be in the 'Established' state to send application data. If the virtual connection is still being established, the client MUST buffer the data in the order it was received.

If ClientOKtoSend is FALSE, the client MUST buffer the application data. The application data MUST be sent on the next Polling-POST-Request, after the outstanding Polling-POST-Response is received.

If ClientOKtoSend is TRUE, the client sets ClientOKtoSend to FALSE. The client sends the application data as specified in section 3.5.4.3.1. If ProxyConnection state variable is set to TRUE, the client MUST instead send the application data with additional proxy headers (see section 3.5.4.3.2).

The Polling Timer MUST be restarted.

Copyright © 2014 Microsoft Corporation.

3.5.4.3.1 Sending Application Data without Proxy

The application data is included within a Polling-POST-Request.

The client MUST construct the Polling-POST-Request-URI as the Polling-POST-Request-Relative-URI.

The client MUST construct a Polling-POST-Request with required headers, as specified in section 2.2.4.1.

The client MUST construct the Polling-Virtual-Connection-Message as specified in section 2.2.4.1.3.1. This message is embedded within the Polling-Request-Entity-Body. The client MUST construct the Polling-Virtual-Connection-Message with the ServerHost as Relay-Server-Name, and VirtualConnectionGUID state variable as Virtual-Connection-GUID.

The first Polling-POST-Request after the Polling handshake MUST have a Sequence-Number of 1. Sequence-Number values MUST increase by 1 after each Polling-POST-Request sent. The last SequenceNumber sent SHOULD be stored in the RequestSequenceNum state variable and be incremented by 1 on every Polling-POST-Response sent.

The Polling-Virtual-Connection-Message Checksum field value MUST be calculated over the application data of each Polling-POST-Request message sent, as specified in section 2.2.4.1.3.1.2.

The application data, if present, is appended after the Polling-Virtual-Connection-Message; together they comprise the Polling-POST-Request-Entity-Body.

The client MUST construct a Polling-Content-Length header that is equal to the length of the

Polling-Virtual-Connection-Message plus the length of the application data. This content length MUST NOT exceed the 32768 octet limit imposed by the Polling protocol. The client opens one connection, a POST session, to the server as in sections 3.5.4.1.1 and 3.5.4.1.2, based on whether it has been given the proxy configuration information.

The NetworkReceiveIO timer SHOULD be restarted.

The client MUST establish a TCP connection to the server identified with ServerHost and ServerPort and send the Polling-POST-Request.

If there is no application data, the Polling timer MUST be restarted with the current polling interval.

3.5.4.3.2 Sending Application Data through a Proxy

The client sets the ProxyConnection to TRUE.

The application data, if present, MUST be included within a Polling-POST-Request.

The client MUST construct the Polling-POST-Request-URI as the Polling-POST-Request-Absolute-URI, with the ServerHost as the HTTP-URL target host name.

The client MUST construct a Polling-POST-Request with required headers, as specified in section 2.2.4.1.

The client MUST construct the Polling-Virtual-Connection-Message as specified in section 2.2.4.1.3.1. This message is embedded within the Polling-Request-Entity-Body

The client MUST construct the Polling-Virtual-Connection-Message with the ServerHost as Relay-Server-Name, and VirtualConnectionGUID state variable as Virtual-Connection-GUID.

Copyright © 2014 Microsoft Corporation.

The first Polling-POST-Request after the Polling handshake MUST have a Sequence-Number of 1. Sequence-Number values MUST increase by 1 for each Polling-POST-Request sent. The last SequenceNumber sent SHOULD be stored in the RequestSequenceNum state variable and be incremented by 1 on every Polling-POST-Response sent.

The Polling-Virtual-Connection-Message Checksum field value MUST be calculated over the application data of each Polling-POST-Request message sent, as specified in section 2.2.4.1.3.1.2.

The application data, if present, is appended after the Polling-Virtual-Connection-Message; together they comprise the Polling-POST-Request-Entity-Body.

The client MUST construct a Polling-Content-Length header that is equal to the length of the Polling-Virtual-Connection-Message plus the length of the application data chunk. This content length MUST NOT exceed the 32768 octet limit imposed by the Polling protocol. If the content length would exceed the 32768 size limit, the Polling-Response-Entity-Body MUST be broken into 32768 chunks.

The NetworkReceiveIO timer SHOULD be restarted.

If ProxyAuthRequired is set, the client MUST add additional proxy authentication headers to the request.

The client MUST establish a TCP connection to the server identified with ProxyServerHostName and ProxyServerPort, and send the Polling-POST-Request.

If there is no application data, the Polling timer MUST be started with the current polling interval.

3.5.5 Polling Client Message Processing Events and Sequencing Rules

3.5.5.1 Polling-POST-Response Processing

Upon receiving data on the POST session, the client MUST scan the data to verify that it has received an HTTP response status line, as specified in section 2.2.2.2.3. If not, a protocol error exists on the POST session, and the client MUST close the virtual Polling connection (see section 3.5.4.2).

The HTTP response header MUST be parsed and the status code and response body extracted.

The NetworkReceiveIO timer SHOULD be stopped and no further timer expiration processing is performed.

If the PostSessionState is in the 'Connecting' state, the receipt of a Polling-POST-Response causes the PostSessionState to transition to 'Connected'.

3.5.5.1.1 Status code: 200 (OK)

Status code 200 is handled differently based on the ConnectionState value. If ConnectionState is ‘Connecting’, the client processing proceeds as specified in section 3.5.5.1.1.1. If the ConnectionState is ‘Established’, processing continues as specified in section 3.5.5.1.1.2. All other ConnectionState values are not valid and are protocol errors; the client MUST close the virtual Polling connection (see section 3.5.4.2).

3.5.5.1.1.1 When ConnectionState is Connecting (last handshake response)

The **Polling-POST-Response** message of the polling handshake MUST have a **Polling-POST-Response-Entity-Body** that contains application data.

Copyright © 2014 Microsoft Corporation.

The client validates the **Polling-POST-Response** message, as specified in section 2.2.4.2, as follows:

1. The client MUST validate the polling **Polling-Virtual-Connection-Message**, as specified in section 2.2.4.1.3.1, and extract the version, server name, virtual connection GUID, sequence number and checksum. If parsing fails, it is a protocol error and the server MUST close the connection, as specified in section 3.5.4.2. The sequence number value is saved in the variable **ResponseSequenceNum**.
2. The client SHOULD<44> check the **Polling-Encapsulation-Version** and, if the version does not equal the required value (see section 2.2.2.3.1.3.1.1), close the polling virtual connection, as specified in section 3.5.4.2.
3. The client SHOULD<45> verify that the server name in the message equals its own name and, if the names are not equal, close the virtual connection.
4. The client MUST examine the **Virtual-Connection-GUID** to validate that it is found in the **VirtualConnectionGUIDList**. Otherwise it is a protocol error, and the client MUST close the polling connection, as specified in section 3.5.4.2.
5. The **Sequence-Number** MUST be verified and stored in the **ResponseSequenceNum.** The sequence number MUST be zero ("0").
6. The **Checksum** field MUST be calculated with the application data and verified as specified in section 2.2.4.1.3.1.3.

The receipt of application data on the POST session completes POST session establishment. The client sets **PostSessionState** and **ConnectionState** both to "Established".

The ConnectionEstablishment timer is stopped and no further timer expiration processing is performed.

The Polling Encapsulation Poll timer MUST be started if there is no application data to send now.

The clients sets the **ClientOKtoSend** state variable to "true".

The client MUST close the POST session as specified in section 3.5.5.1.5.

The application data is passed to the application layer to be processed and consumed.

The client is ready to send application data. If there is any buffered data to send, the client MUST send it now, as specified in section 3.5.4.3.

3.5.5.1.1.2 When ConnectionState is Established (Receiving Application Data)

The client validates the **Polling-POST-Response** message, as specified in section 2.2.4.2, as follows:

1. The client MUST validate the **Polling-Virtual-Connection-Message**, as specified in section 2.2.4.1.3, and extract the version, server name, virtual connection GUID, sequence number, and checksum. If parsing fails, it is a protocol error and the client MUST close the connections, as specified in section 3.5.4.2.
2. The client SHOULD<46> check the **Polling-Encapsulation-Version** and, if the version does not equal the required value (see section 2.2.2.3.1.3.1.1), close the polling virtual connection, as specified in section 3.5.4.2.

Copyright © 2014 Microsoft Corporation.

3. The client SHOULD<47> verify that the **Relay-Server-URL** in the message equals its own name and, if the names are not equal. close the virtual connection, as specified in section 3.5.4.2.
4. The client MUST examine the **Virtual-Connection-GUID** to validate that it is found in the **VirtualConnectionGUIDList**. Otherwise it is a protocol error and the client MUST close the polling connection, as specified in section 3.5.4.2.
5. The value of **Sequence-Number** MUST be verified and stored in **ResponseSequenceNum.**
6. The received sequence number MUST be equal to the value stored in the **ResponseSequenceNum** plus 1. Verification failure results in polling virtual connection closure, as specified in section 3.5.4.2.
7. The **Checksum** field MUST be calculated with the application data and verified as specified in section 2.2.4.1.3.1.3. The **Checksum** value is calculated only in the presence of application data and does not include the **Polling-Virtual-Connection-Request-Messages**. If there is no application data, the **Checksum** field MUST be zero ("0").

If **Polling-Content-Length** is greater than the length of **Polling-Virtual-Connection-Message** plus the length of **Polling-Virtual-Connection-Response-Message**, there is application data. Otherwise, the **Polling-POST-Response** message contains no application data.

Any verification failure results in polling virtual connection closure, as specified in section 3.5.4.2.

The clients sets the **ClientOKtoSend** state variable to "true".

The Polling timer MUST be started if there is no data to be sent now.

The client MUST refresh **PollingMinRepetitionInterval**, **PollingMinRepetitionInterval**, and **PollingRepetitionCount** values, if they have changed, with the contents of the **Polling-Virtual-Connection-Response-Message**.

The application data is passed to the application layer to be processed and consumed.

The client MUST close the POST session, as specified in section 3.5.5.1.5.

If there is buffered data, the client MUST send the data as specified in section 3.5.4.3.

3.5.5.1.2 Status code: 400 (Bad Request)

For processing a Polling-POST-Response with 400 status code when the PostSessionState is in the 'Probing' state, see section 3.5.5.1.2.1. For all other PostSessionState states, see section 3.5.5.1.2.2.

3.5.5.1.2.1 When PostSessionState is Probing

The receipt of a Polling-Response-Response transitions the PostSessionState to 'Probed'.

The client MUST close the Poll session.

The client MUST send the second and last Polling-POST-Request of the Polling connection handshake as specified in section 3.5.4.1.1.

If ProxyConnection is TRUE, then processing continues as specified in section 3.5.4.1.2.

Copyright © 2014 Microsoft Corporation.

3.5.5.1.2.2 All other PostSessionState States

The server has rejected the Polling-POST-Request because an encapsulation version does not equal the required value (see section 2.2.2.3.1.3.1.1) or because of a protocol error. The client MUST close the Polling connection (see section 3.6.4.1).

3.5.5.1.3 Status codes: 401 (Unauthorized) and 407 (ProxyAuthentication Required)

HTTP status code values of 401 (Unauthorized) or 407 (ProxyAuthentication Required)

indicate that the proxy requires the client to authenticate. Common authentication schemes include Basic and Digest, as specified in [RFC2617], and NTLM HTTP Authentication, as specified in [RFC4559].

The client sets the ProxyAuthRequired state variable to TRUE. Subsequent connection attempts to the same proxy SHOULD avoid the proxy challenge message by sending the proxy authentication credentials as part of the Polling-POST-Request.

Depending on the authentication method, multiple round trips can happen to complete the authentication process. That is, the client MUST expect to get multiple 401 and 407 messages. It MUST follow [RFC2617] and [RFC4559] to set authentication headers and retry the proxy connection.

For processing required to retry the proxy connection, see section 3.5.4.1.2.

The ConnectionEstablishment timer SHOULD be reset before re-attempting the Polling handshake (see section 3.5.4.1).

3.5.5.1.4 All Other Status Codes

All other status codes are fatal; the virtual connection MUST be closed as specified in section 3.6.4.1.

3.5.5.1.5 Closing the POST Session

The client SHOULD close the POST session by sending a graceful TCP disconnect.

All connection state information MUST NOT be discarded.

3.5.5.1.6 Closing a Polling Connection because of Protocol Error

On protocol errors the client can receive a Polling-POST-Response with a status code indicating the reason for closure.

The client MUST close the virtual Polling Connection as specified in section 3.5.4.2.

3.5.6 Polling Client Timer Events

3.5.6.1 ConnectionEstablishment Timer Event

The ConnectionEstablishment Timer Event fires when the ConnectionEstablishment timer for a given Polling connection expires before the connection can be established. If this timer expires before the Polling connection enters the 'Established' state, the virtual Polling connection SHOULD be closed by the client.

Copyright © 2014 Microsoft Corporation.

3.5.6.2 NetworkReceiveIO Timer Event

The NetworkReceiveIO Timer Event fires when a POST session response is not received within the NetworkReceiveIO interval. If the NetworkReceiveIO event triggers, the client SHOULD close the Polling connection. The client can retry the connection again at a later time. The NetworkReceiveIO SHOULD be reset on subsequent POST session responses. Well behaved connections will not see this event, so clients that expect this event to be transient SHOULD always attempt to establish a new Polling connection with the target server.

3.5.6.3 Polling Encapsulation Timer

The Polling timer uses a back off algorithm as specified in section 3.5.2.3.When the Polling timer event fires, the client MUST send a Polling-POST-Request poll message to the server. The current polling interval MUST be recalculated based on the algorithm specified in section 3.5.2.3.

The client sends (see section 3.5.4.3) the Poll requests with no application data.

3.5.7 Polling Client Other Local Events

On transport disconnect events, the virtual Polling Connection MUST be closed as specified in section 3.5.4.2.

The application layer SHOULD attempt to re-establish the Polling virtual connection. Upon repeated connection failures, the application layer SHOULD implement a back off algorithm for re-establishing the Polling connection to the target server.

3.6 Polling Encapsulation Protocol Server Details

3.6.1 Polling Server Abstract Data Model

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

3.6.1.1 Connection State Information

See section 3.5.1.1 for a list of connection state variables that are shared with the client.

VirtualConnectionGUIDList: The global list of virtual connection GUIDs of all active connections. This list allows the application to quickly lookup a virtual connection GUID to determine if it is a known virtual connection GUID. This list also contains a reference to the per-connection state variables for the associated GUID.

ServerOKtoSend: The state variable enforces the requirement that the server MUST NOT send application data on a Polling-POST-Response until receiving a Polling-POST-Request. All subsequent Polling-POST-Response messages MUST only be sent in response to an outstanding Polling-POST-Request.

Copyright © 2014 Microsoft Corporation.

3.6.2 Polling Server Timers

3.6.2.1 ConnectionEstablishment Timer

The ConnectionEstablishment timer can be used to limit the amount of time a Polling connection negotiation takes to complete. This timer measures the time it takes for a connection to move from the non-established state to the established state. The recommended timeout value is 3 minutes. The ConnectionEstablishment timer event processing is handled as specified in section 3.6.6.1.

3.6.3 Polling Server Initialization

3.6.3.1 Protocol Initialization

When the server starts it MUST initialize the HTTP stack[48].

A Polling connection protocol is not initialized until a request to open an encapsulated connection is received by the server. The variables defined by the abstract data model are initialized when the Polling connection request is received.

3.6.3.2 Polling Encapsulation Listener

The Server MUST open a listener socket on the Polling port. The Polling connection port uses the well-known HTTP port 80/TCP. Alternate ports MAY be used, but non-default port information MUST be conveyed to the client.

3.6.4 Polling Server Higher-Layer Triggered Events

3.6.4.1 Closing a Polling Connection

The server MUST close the POST session by sending a graceful TCP disconnect. The ConnectionState then transitions into the 'Closed' state.

Connection State variables are discarded.

TCP client or server connections are closed using either a Graceful Close (TCP FIN Flag set to 1) or Abortive Close (TCP Reset Flag set to 1) mechanism. Closing the connection gracefully provides clients and servers with the benefit of never having to resend unacknowledged payload data. Abortive Closed connections are efficient in connection tear down but can require the client or server to resend payload data which has not been acknowledged at the TCP level. HTTP Encapsulation of SSTP Protocols relies on SSTP to ensure that any unacknowledged payload data is later resent. The choice of connection close mechanism is one of efficiency (efficient connection tear down verses efficient byte transmission). This document does not mandate that implementations adhere to one approach or another. Implementations can choose to close HTTP Encapsulated Connections either gracefully or abortively based on the implementation's requirements.

3.6.4.2 Closing a Polling Session

The server MUST close the POST session by sending a graceful TCP disconnect. The connection then transitions into the connection closed state.

The PostSessionState is set to the 'Closed' state.

All connection state information MUST NOT be discarded.

Copyright © 2014 Microsoft Corporation.

3.6.4.3 Sending Application Data

The data to be sent SHOULD be buffered, waiting for the next POST request to come in. After processing of the request and before sending the response, the server always checks to see if there is buffered data to be sent.

3.6.5 Polling Server Message Processing Events and Sequencing Rules

Upon receiving data on the POST session, the server MUST first check to see if the data starts with an HTTP POST request line, as specified in section 2.2.4.1.1. If not then it is a protocol error, and the virtual Polling connection MUST be closed (see section 3.6.4.1).

The server MUST validate the Polling Polling-Virtual-Connection-Message by performing the following:

1. The server extracts the version, server name, virtual connection GUID, sequence number, and checksum. If the parsing fails, it is a protocol error and the server MUST close the virtual Polling connection (see section 3.6.4.1).
2. The server SHOULD<49> check the Polling-Encapsulation-Version and, if the version does not equal the required value (see section 2.2.2.3.1.3.1.1), close (see section 3.6.4.1) the virtual Polling connection.
3. The server SHOULD<50> verify that the server name in the message equals its own name.

The server extracts the Virtual-Connection-GUID from the Polling-Virtual-Connection-Message. The server performs a lookup on the global VirtualConnectionGUIDList values to:

1. Determine if the Virtual-Connection-GUID is a new connection or existing connection.
2. Retrieve the ConnectionState and PostSessionState connection state variables for the existing connection.

A new Virtual-Connection-GUID event is handled as specified in section 3.6.5.1. Existing Virtual-Connection-GUIDs events whose ConnectionState is 'Connecting' are processed as specified in section 3.6.5.2. Existing Virtual-Connection-GUID event whose ConnectionState is 'Established' are processed as specified in section 3.6.5.3.

3.6.5.1 Receiving a Polling-POST-Request (Initial Handshake Request)

Upon receipt of a Polling-POST-Request the server sets the PostSessionState to 'Probing', the ConnectionState to "Connect" and starts the ConnectionEstablishment timer.

The server validates the Polling-POST-Request message as defined in section 2.2.2.1.

The server MUST validate the remain Polling-Virtual-Connection-Message (see section 2.2.4.1.3.1)fields, not already validated in section 3.6.5.

1. The Sequence-Number MUST be verified and stored in the ResponseSequenceNum**.** The sequence number MUST be 0<51>.
2. The Checksum MUST be 0.
3. The Polling-POST-Request MUST NOT contain application data. The server validates the lack of application data by examining the Polling-Content-Length value whose length MUST be equal to the length of the Polling-Virtual-Connection-Message.

Copyright © 2014 Microsoft Corporation.

If any of the preceding validations procedures fails, the virtual Polling Connection MUST be closed as specified in section 3.6.4.1.

The server transitions PostSessionState to the "Probed" state and ConnectionState to the 'Connecting" state. The server continues to complete the handshake by sending a Polling-POST-Response, as specified in section 3.6.5.2.1.

3.6.5.1.1 Sending a Polling-POST-Response with Status code 400 (Handshake)

The initial Polling-POST-Response message (see section 3.6.5.1) of the polling Handshake overloads the 400 status to indicate a successful response.

A successful response MUST contain the a Response-Status-Line (see section 2.2.2.3.1) with a status code of 400 and phrase of "Bad Request".

The Polling-POST-Response-Required-Headers MUST be specified.

The Polling-Virtual-Connection-Message MUST NOT be sent on this response. Application data MUST NOT be supplied. The Polling-Content-Length is set to 0.

The Polling-POST-Response MUST be sent on the POST session. The server MUST close the Polling session (see section 3.6.4.2).

3.6.5.2 Receiving a Polling-POST-Request (Last Handshake Request)

The server MUST validate the **Polling-Virtual-Connection-Message**, as specified in section 2.2.4.1.3.1.

1. The **Polling-POST-Request Sequence-Number** field value on the received message MUST be equal to 0. If the **Sequence-Number** field value is not equal to 0 it is a protocol error and the Polling connection MUST be closed as specified in section 3.6.4.2.
2. The **Polling-POST-Request** MUST contain application data. The server validates the presence of application data by examining the **Polling-Content-Length** value. The length MUST be greater than the length of the **Polling-Virtual-Connection-Message**. If not, it is a protocol error, and the virtual connection is closed, as specified in section 3.6.4.1.
3. The **Polling-POST-Request Checksum** field value MUST be calculated as specified in section 2.2.4.1.3.1.3 over the length of the application data received. The calculated **Checksum** value MUST be equal to the **Checksum** value found in the **Polling-POST-Request** field.

If any of the preceding validation procedures fail, the virtual polling connection MUST be closed, as specified in section 3.6.4.1.

The server sets **ConnectionState** to "Established" and sets **PostSessionState** to "Established".

The application data is handed off to the application layer for further processing.

The server continues as described in section 3.6.5.2.1 to complete the handshake, by sending a **Polling-POST-Response** with a status code of "200".

3.6.5.2.1 Sending a Polling-POST-Response with Status code 200 (OK)

A **Polling-POST-Response** message with a status code of "200" is sent as a successful response to the second and last **Polling-Post-Request** of the polling connection handshake or as a successful response to a **Polling-POST-Request** after the handshake.

Copyright © 2014 Microsoft Corporation.

If **ConnectionState** is "Connected", the **Sequence-Number** field specified on the **Polling-Virtual-Connection-Message** uses the value of **ResponseSequenceNum**, which is set to zero. If **ConnectionState** is "Established", the **ResponseSequenceNum** field is incremented by 1 and used as the **Sequence-Number** field value.

The **Polling-POST-Response** MUST include a **Response-Status-Line**, as specified in section 2.2.2.3.1, with a status code of 200 and a phrase of "OK".

The **Polling-POST-Response-Required-Headers** MUST be included.

The Polling-Virtual-Connection-Message MUST be prepared using the ServerHost as Relay-Server-Name, and VirtualConnectionGUID as Virtual-Connection-GUID. The Polling-POST-Request Checksum field value MUST be calculated (see section 2.2.4.1.3.1.3) over the length of the application data that is to be sent. If there is no application data to send, the Checksum field MUST be set to 0.

All buffered application data and any new application data SHOULD be included in the Polling-POST-Response-Entity-Body. If there is no application data to send, the Polling-POST-Response message MUST be sent without application data. If the content length would exceed the 32768 size limit, the Polling-Response-Entity-Body MUST be broken into 32768 chunks.

The Polling-Virtual-Connection-Response-Message MUST be prepared using the PollingMinRepetitionInterval, PollingMinRepetitionInterval, and PollingRepetitionCount connection state variables. Server implementations MAY use various load balancing techniques to control the clients poll timer interval (see section 3.5.2.3)<52>.

The Polling-Content-Length is set to the length of the Polling-Virtual-Connection-Response-Message plus the length of the buffered application data.

If there is no application data to send, the Polling time MUST be started with the current polling interval.

The Polling-POST-Response MUST be sent on the POST session. The Polling session MUST be closed (see section 3.6.4.2).

3.6.5.3 Receiving a Polling-POST-Request (After Handshake)

Polling-POST-Request events that are received while the virtual Polling Connection is in the 'Established' state define how the server processes application data and Poll requests.

Once the ConnectionState is in the 'Established' state all Polling-POST-Requests MUST contain Application data, otherwise it is a protocol error, and the server MUST close the virtual polling connection as specified in section 3.6.4.1.

The server MUST validate the Polling-Virtual-Connection-Message (see section 2.2.4.1.3.1) using the following steps:

1. The Polling-POST-Request Sequence-Number field value on the received message SHOULD be compared against the expected Sequence-Number found in the RequestSequenceNum connection variable. Non-valid values are protocol errors. The new Sequence-Number SHOULD be stored in the RequestSequenceNum.

2. The Polling-POST-Request Checksum field value MUST be calculated over the length of the application data received on each Polling-POST-Request message received, as specified in section 2.2.4.1.3.1.3. The calculated Checksum value MUST be equal to the Checksum value found in the Polling-POST-Request field. If there is no application data, the Checksum MUST be 0.

Copyright © 2014 Microsoft Corporation.

The server validates the presence of application data by examining the Polling-Content-Length value. Application data MUST be present when the content length is greater than the length of the Polling-Virtual-Connection-Message.

If any of the preceding validation procedures fail, the virtual Polling Connection MUST be closed as specified in section 3.6.4.1.

Any application data is handed off to the application layer for further processing.

The server MUST respond to the Polling-POST-Request message with a Polling-POST-Response message containing a status code of 200, as specified in section 3.6.5.2.1. If the server has no buffered application data to send on the Polling-POST-Response, then it MUST send the Polling-POST-Response (see section 3.6.5.2.1) with no application data.

3.6.6 Polling Server Timer Events

3.6.6.1 ConnectionEstablishment Timer Event

The ConnectionEstablishment Timer Event fires when enforcing a limit on the time it takes to establish a Polling connection with the server. If this timer expires before the Polling connection enters the established state, the virtual connection SHOULD be closed by the client. The timer SHOULD be set before starting the Polling Connection handshake.

3.6.6.2 Polling Encapsulation Timer

None.

3.6.7 Polling Server Other Local Events

On transport disconnect events, the virtual Polling Session MUST be closed as specified in section 3.6.4.2.

3.7 Secure Tunnel Encapsulation of SSTP Protocol Client Details

Secure Tunnel Encapsulation is layered on a single TCP connection. After the TCP connection is established, the HTTP Connect method flows over the connection and is used to negotiate a tunnel connection through the proxy. When the Secure Tunnel connection handshake is complete, the SSTP data stream flows across the Secure Tunnel connection, through the proxy, to the server. The connection is a full duplex connection. SSTP commands and data from the client to the server flow in their SSTP format using TCP as a transport.

A Secure Tunnel connection is negotiated between the client and the proxy. The server is not involved in the proxy negotiation. Direct connections between a client and server do not perform the Secure Tunnel handshake. Instead, direct connections send and receive the SSTP data stream over a TCP connection, just as they do for SSTP over 2492/TCP. The only difference is that the target port is 443/TCP with no SSL handshake (see [SSL3], section 5.5) or protocol (see [SSL3]).

3.7.1 Secure Tunnel Client Abstract Data Model

This section specifies a conceptual model of possible data organization that an implementation maintains to participate in the Secure Tunnel Proxy protocol. The specified organization is provided to facilitate the explanation of how the protocol behaves. This document does not mandate that implementations adhere to this model as long as their external behavior is consistent with that specified in this document. The following figure shows a detailed view of the client state machine for Secure Tunnel.

Copyright © 2014 Microsoft Corporation.

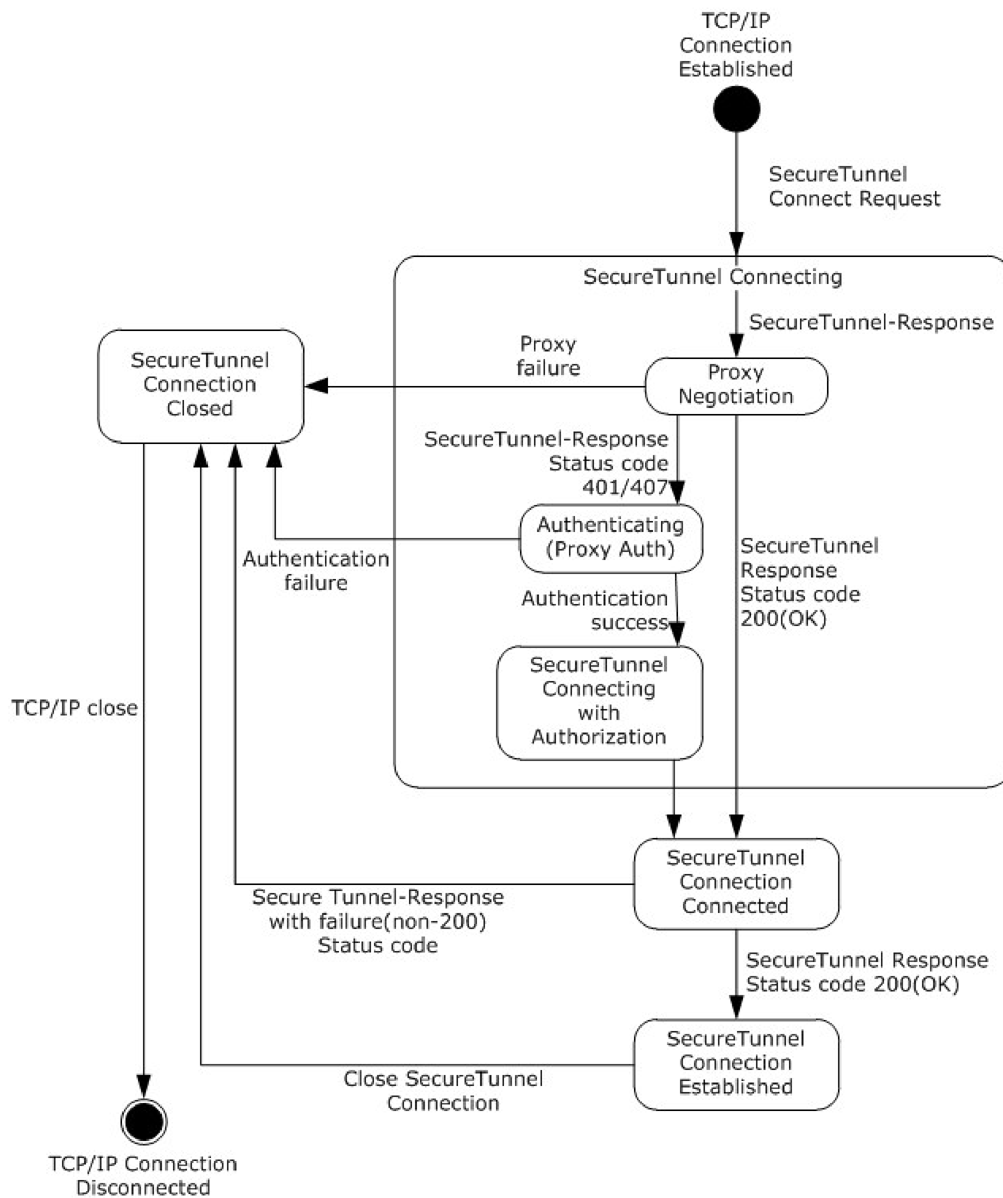

Figure 20: Client Secure Tunnel state diagram

3.7.1.1 Connection State Information

The following details about the state information define the context needed to manage a Secure Tunnel connection. Unless otherwise noted, the following connection state variables are scoped to a single Secure Tunnel connection. When a Secure Tunnel connection is terminated, this state information is no longer relevant and SHOULD be discarded.

A client SHOULD support multiple Secure Tunnel connections to multiple servers concurrently. A client SHOULD support one Secure Tunnel connection to the same target server (see ServerHost state information). In all cases, each Secure Tunnel connection MUST maintain separate connection state variable information.

Copyright © 2014 Microsoft Corporation.

ServerPort: The well-known port number of the target server. By default this is the SSL well known port 443/TCP.

ServerHost: The host name of the target server, in the form of an FQDN or IP Address. There is no default value.

ConnectionState: The variable used to maintain the current disposition of the Secure Tunnel connection. There are four possible states: 'Connecting', 'Connected', 'Established', and 'Closed'. The 'Connecting' indicates that the connection creation is in progress. The 'Connected' state indicates that the connection has finished any proxy negotiation. The 'Established' state indicates that connection has been successfully created, and application data MAY begin to flow over the connection. The 'Closed' state indicates the connection can no longer send or receive application data, and the connection is closed.

ProxyConnection: The indicator of whether the current connection is a connection to a proxy or a direct connection to a server. The value is set to TRUE after the client determines that a proxy is to be used. The default value is FALSE.

3.7.1.2 Proxy State Information

The following details about the state information define the context clients need to establish connections with proxies. This proxy configuration information MUST be provided to the client prior to connection establishment. The source of this configuration information is external to the Secure Tunnel Proxy Protocol<53>.

ProxyServerPort: The well-known port number of the target proxy. It is used for establishing a TCP connection to a proxy. By default this is the SSL well known port 443/TCP.

ProxyServerHostName: The host name of the target proxy. The name is in the form of an FQDN or an IP Address. If the name is an FQDN, then the client MUST resolve this name to its IP Address. There is no default value.

ProxyAuthRequired: A variable used to indicate if a proxy requires authentication. The client sets this variable to TRUE when it discovers that the proxy needs authentication during its first negotiation with the proxy. When the client initiates a new virtual connection through the same proxy, it SHOULD provide the cached credentials without waiting to be challenged, to avoid the overhead of additional message exchanges.

3.7.2 Secure Tunnel Client Timers

3.7.2.1 ConnectionEstablishment Timer

The ConnectionEstablishment timer SHOULD be used by clients to limit the amount of time a Secure Tunnel connection negotiation takes to complete. This timer measures the time it takes for a connection to move from the connecting to the established state. The recommended timeout value is 90 seconds. The ConnectionEstablishment timer event processing is handled as specified in section 3.7.6.1.

3.7.2.2 NetworkReceiveIO Timer

The NetworkReceiveIO timer SHOULD be used by clients to limit the amount of time a client waits for a Secure Tunnel connection network receive to complete. This timer is set on the first network IO after the Secure Tunnel connection handshake is finished. The NetworkReceiveIO timer value SHOULD be larger than the KeepAlive timer to avoid premature timeouts. The recommended

Copyright © 2014 Microsoft Corporation.

timeout value is 90 seconds. The NetworkReceiveIO timer event processing is handled as specified in section 3.7.6.2.

3.7.2.3 KeepAlive Timer

HTTP Encapsulation protocols do not support a native KeepAlive timer, but rather rely on the encapsulated protocol to provide a KeepAlive mechanism. Encapsulated protocols SHOULD implement their own KeepAlive mechanisms. The SSTP protocol provides its own KeepAlive mechanism using the SSTP_NOOP command<54>. The default client KeepAlive timeout value is 45 seconds. KeepAlive timers with short intervals SHOULD be used to interoperate with firewalls and proxies. The maximum KeepAlive value is limited by proxy implementations. The KeepAlive timer event processing is handled as specified in section 3.7.6.3.

3.7.3 Secure Tunnel Client Initialization

3.7.3.1 Protocol Initialization

The protocol is not initialized until a request to open an encapsulated connection is made by the application layer. The variables defined by the abstract data model are initialized when a Secure Tunnel connection request is made.

3.7.3.2 Secure Tunnel Listener Endpoints

None.

3.7.3.3 Timers Started

None.

3.7.4 Secure Tunnel Client Higher-Layer Triggered Events

3.7.4.1 Establishing a Secure Tunnel Encapsulation Connection

When the application layer requests a Secure Tunnel connection, the Secure Tunnel Proxy Protocol layer MUST initialize the Secure Tunnel connection state variables as specified in the abstract data model (see section 3.7.1). After the connection state variables are initialized, the Secure Tunnel Proxy Protocol enters into the connection establishment phase. Initialization SHOULD include fetching any proxy configuration information<55>.

The client opens a connection to the server as define in section 3.7.4.1.2. If proxy configuration is provided, processing continues as specified in section 3.7.4.1.2.

The ConnectionEstablishment timer SHOULD be started.

3.7.4.1.1 Establishing a Secure Tunnel connection without proxy

The ConnectionState MUST be set to 'Connecting'.

The client MUST establish a TCP connection to the server identified with ServerHost and ServerPort. The client uses the SSTP protocol over the Secure Tunnel port.

When the TCP connection is successfully established, the ConnectionState MUST transition to the 'Connected' state.

Copyright © 2014 Microsoft Corporation.

3.7.4.1.2 Establishing a Secure Tunnel connection with a proxy

The client sets the ProxyConnection to TRUE.

The client MUST construct a Connect Request as defined in [TCPPROXY], section 3.1, providing ServerHost and ServerPort.

The client includes the Proxy-Connection header with the Keep-Alive value as specified in section 2.2.1.2.9.

The client includes the required headers as specified in [TCPPROXY], section 3.1.

If ProxyAuthRequired is set, the client MUST add additional proxy authentication headers to the request.

The client MUST establish a TCP connection to the server identified with ProxyServerHostName and ProxyServerPort, and send the HTTP Connect Request.

3.7.4.2 Closing a Secure Tunnel Connection

The client MUST close the Secure Tunnel connection by sending a graceful TCP disconnect.

The ConnectionState then transitions into the 'Closed' state. The connection state variables SHOULD be discarded.

TCP client or server connections are closed using either a Graceful Close (TCP FIN Flag set to 1) or Abortive Close (TCP Reset Flag set to 1) mechanism. Closing the connection gracefully provides clients and servers with the benefit of never having to resend unacknowledged payload data. Abortive Closed connections are efficient in connection tear down but can require the client or server to resend payload data which has not been acknowledged at the TCP level. HTTP Encapsulation of SSTP Protocols relies on SSTP to ensure that any unacknowledged payload data is later resent. The choice of connection close mechanism is one of efficiency (efficient connection tear down verses efficient byte transmission). This document does not mandate that implementations adhere to one approach or another. Implementations can choose to close HTTP Encapsulated Connections either gracefully or abortively based on the implementation's requirements.

3.7.4.3 Sending Application Data

The Secure Tunnel connection MUST be in the 'Established' state to send application data. If the connection is still being established, the client MUST buffer the data .

If the client is in the 'Established' state, the client sends application data over the connection.

The KeepAlive and NetworkReceiveIO timers MUST be restarted.

3.7.5 Secure Tunnel Client Message Processing Events and Sequencing Rules

Data received while the ConnectionState is not 'Established' is part of the Secure Tunnel connection negotiation handshake as specified in section 3.7.5.1. Data received while the ConnectionState is 'Established' is handled as specified in section 3.7.5.2.

3.7.5.1 HTTP Response Processing

The HTTP response header MUST be parsed and the status code and response body extracted.

The receipt of an HTTP Response transitions the ConnectionState to 'Connected'.

Copyright © 2014 Microsoft Corporation.

3.7.5.1.1 Status code: 200

The ConnectionState transitions into the 'Established' state.

The ConnectionEstablishment timer is stopped and no further timer expiration processing is performed.

The KeepAlive timer and NetworkReceiveIO timer SHOULD both be started.

If there is any buffered data to send, the client MUST now send it, as specified in section 3.7.4.3.

3.7.5.1.2 Status code: 400 (Bad Request)

The proxy has rejected the connection request. The client MUST close the Secure Tunnel connection (see section 3.7.4.2).

3.7.5.1.3 Status codes: 401 (Unauthorized) and 407 (ProxyAuthentication Required)

HTTP status code values of 401 (Unauthorized) or 407 (ProxyAuthentication Required) indicate that the proxy requires the client to authenticate. Common authentication schemes include Basic and Digest, as specified in [RFC2617], and NTLM HTTP Authentication, as specified in [RFC4559].

The client sets the ProxyAuthRequired state variable to TRUE. Subsequent connection attempts to the same proxy SHOULD avoid the proxy challenge message by sending the proxy authentication credentials as part of the Connect Request (see [TCPPROXY], section 3.1).

Depending on the authentication method, multiple round trips can happen to complete the authentication process. That is, the client MUST expect to get multiple 401 and 407 messages. It MUST follow [RFC2617] and [RFC4559] to set authentication headers and retry the proxy connection.

For processing required to retry the proxy connection, see section 3.7.4.1.2.

The ConnectionEstablishment timer SHOULD be reset before re-attempting the Secure Tunnel handshake (see section 3.7.4.1).

3.7.5.1.4 All Other Status Codes

All other status codes are fatal; the virtual connection MUST be closed as specified in section 3.7.4.2.

3.7.5.2 Application Data Processing

If the connection state is not "Established", this is a protocol error; the data MUST be discarded and the connection closed, as specified in section 3.7.4.2.

The client receives application data over the connection.

The application data is passed to the application layer for processing.

The **NetworkReceiveIO** timer MUST be restarted after each transport receive operation completes.

Copyright © 2014 Microsoft Corporation.

3.7.6 Secure Tunnel Client Timer Events

3.7.6.1 ConnectionEstablishment Timer Event

The ConnectionEstablishment Timer Event fires when the ConnectionEstablishment timer for a given Secure Tunnel connection expires before the connection can be established. If this timer expires before the Secure Tunnel connection enters the 'Established' state, the TCP connection SHOULD be closed by the client.

3.7.6.2 NetworkReceiveIO Timer Event

The NetworkReceiveIO Timer Event fires when a network receive does not complete within the specified amount of time. If the NetworkReceiveIO Event triggers, the client SHOULD close the Secure Tunnel connection and MAY retry immediately.

3.7.6.3 KeepAlive Timer Event

A KeepAlive Event SHOULD trigger an encapsulated protocol message such as an SSTP_NOOP command to be sent across the wire<56>. Well behaved connections will see this event every KeepAlive timer interval.

3.7.7 Secure Tunnel Client Other Local Events

On transport disconnect events all connection state information SHOULD be discarded.

The client SHOULD close the Secure Tunnel connection and can retry immediately. If the retry attempt fails, the client SHOULD let the higher layer decide whether to wait before establishing a new Secure Tunnel virtual connection to the target server, or use a different encapsulation protocol to establish a connection to the server.

3.8 Secure Tunnel Encapsulation of SSTP Protocol Server Details

From the server's standpoint, the Secure Tunnel Proxy only exchanges SSTP messages with the server. The traffic between the proxy and the server is SSTP protocol traffic on a non-standard port for SSTP: 443/TCP. The server does not take part in the Secure Tunnel handshake. Secure Tunnel encapsulation occurs between the client and proxy server as part of the Secure Tunnel handshake. All messages that flow between the client and server are standard SSTP messages. See SSTP [MS-GRVSSTP] for server processing rules. All SSTP server processing rules apply except for those listed in sections 3.8.2 and 3.2.3.

3.8.1 Secure Tunnel Server Abstract Data Model

None.

3.8.2 Secure Tunnel Server Timers

The following SSTP timers are used in conjunction with the client's Secure Tunnel timers.

3.8.2.1 SSTP KeepAlive Timer

The HTTP Encapsulation protocols do not define a KeepAlive timer. The underlying encapsulated protocol MUST implement a KeepAlive timer. The SSTP protocol uses the KeepAlive mechanism provided by the SSTP_NOOP command<57>.

Copyright © 2014 Microsoft Corporation.

The KeepAlive message keeps the Secure Tunnel connection from being closed by firewalls and proxies because of connection inactivity. All Secure Tunnel connections SHOULD use KeepAlive timers, regardless of whether the client detects if a connection is a proxy connection or not, as some firewall and proxies are undetectable. The recommended client KeepAlive timeout value is 45 seconds<58>.

3.8.3 Secure Tunnel Server Initialization

3.8.3.1 Secure Tunnel Encapsulation Listener

The server MUST open a listener socket on the Secure Tunnel port. The Secure Tunnel connection port SHOULD be the well-known SSL port 443/TCP.

The SSL protocol handshake is not used on this port. This listener communicates using the SSTP protocol.

The Secure Tunnel listener supports both Secure Tunnel Proxy connections and direct client to server connections. From the perspective of the Secure Tunnel listener, there is no difference between a Secure Tunnel direct connection and a Secure Tunnel Proxy connection.

3.8.4 Secure Tunnel Higher-Layer Triggered Events

None.

3.8.5 Secure Tunnel Server Message Processing Events and Sequence Rules

None.

3.8.6 Secure Tunnel Server Timer Events

None.

3.8.7 Secure Tunnel Server Other Local Events

None.

3.9 SOCKS Encapsulation of SSTP Protocol Client Details

SOCKS Encapsulation is layered on a single TCP connection. After the TCP connection is established, the SOCKS handshake negotiates a tunnel through the proxy to the server. When the SOCKS connection handshake is complete, the SSTP data stream flows across the SOCKS connection, with no additional SOCK protocol messages.

The SOCKS connection is a full duplex connection. SSTP commands and data from the client to the server flow in their SSTP format using the SSTP Listener and well known port number 2492/TCP.

3.9.1 SOCKS Client Abstract Data Model

This section specifies a conceptual model of possible data organization that an implementation maintains to participate in the SOCKS protocol. The specified organization is provided to facilitate the explanation of how the protocol behaves. This document does not mandate that implementations adhere to this model as long as their external behavior is consistent with that specified in this document. The following figure provides a detailed look at the SOCKS client state machine.

Copyright © 2014 Microsoft Corporation.

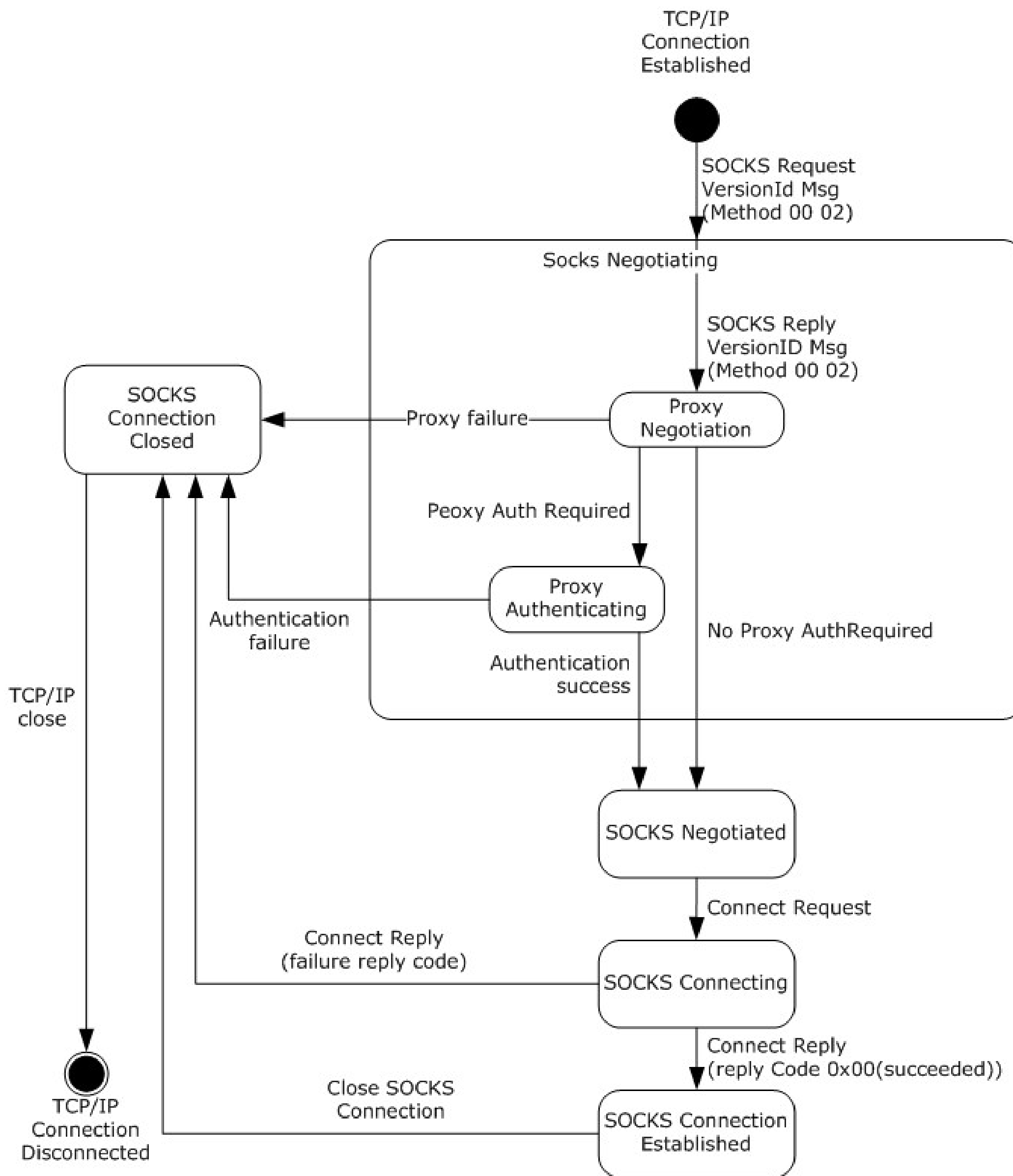

Figure 21: SOCKS client state diagram

3.9.1.1 Connection State Information

The state information detailed in this section defines the context needed to manage a SOCKS connection. Unless otherwise noted, the following connection state variables are scoped to a single SOCKS connection. When a SOCKS connection is terminated, this state information is no longer relevant and SHOULD be discarded.

A client SHOULD support multiple SOCKS connections to multiple servers concurrently. A client SHOULD support one SOCKS connection to the same target server (see ServerHost state

Copyright © 2014 Microsoft Corporation.

information). Each SOCKS connection MUST maintain separate connection state variable information.

ServerPort: The well-known port number of the target server. By default this is the SSTP well known port 2492/TCP.

ServerHost: The host name of the target server, in the form of an FQDN or IP Address. There is no default value.

ConnectionState: The variable used to maintain the disposition of the SOCKS connection. There are five possible states: 'Negotiating', 'Negotiated', 'Connecting', 'Established', and 'Closed'. The 'Negotiating' state indicates that the SOCKS connection is negotiating its authentication methods. The 'Negotiated' state indicates that the client has successfully finished authentication. The 'Connecting' state indicates that the SOCKS proxy connection to the target server is in progress. The 'Established' state indicates that the SOCKS connection with the proxy has successfully established a connection with the target server and application data MAY begin to flow. The 'Closed' state indicates that the connection can no longer send or receive application data; the SOCKS connection is closed.

3.9.1.2 Proxy State Information

The following details about the state information define the context clients need to establish connections with proxies<59>. This proxy configuration information MUST be provided to the client prior to connection establishment. This source of this configuration information is external to the SOCKS Protocol<60>.

ProxyServerPort: The well-known port number of the SOCKS server. It is used for establishing a TCP connection. By default the well-known port is 1080/TCP.

ProxyServerHostName: The host name of the SOCKS server. The name is in the form of an FQDN or an IP Address. There is no default value.

ProxyAuthRequired: A variable used to indicate that proxy authentication is required. The client sets this variable to TRUE when it discovers that the proxy needs authentication.

3.9.2 SOCKS Client Timers

3.9.2.1 ConnectionEstablishment Timer

The ConnectionEstablishment timer SHOULD be used by clients to limit the amount of time a SOCKS connection negotiation takes to complete. This timer measures the time it takes for a connection to move from the negotiating state to the established state. The recommended timeout value is 90 seconds. The ConnectionEstablishment timer event processing is handled as specified in section 3.9.6.1.

3.9.2.2 NetworkReceiveIO Timer

The NetworkReceiveIO timer SHOULD be used by clients to limit the amount of time a client waits for a SOCKS connection network receive to complete. The recommended timeout value is 60 seconds. The NetworkReceiveIO timer event processing is handled as specified in section 3.9.6.2.

3.9.2.3 KeepAlive Timer

HTTP Encapsulation protocols do not support a built-in KeepAlive timer, but instead rely on the encapsulated protocol to provide a KeepAlive mechanism. An encapsulated protocol SHOULD

Copyright © 2014 Microsoft Corporation.

implement its own KeepAlive mechanism. The SSTP protocol provides its own KeepAlive mechanism (see [MS-GRVSSTP] section 2.2.13), using the SSTP_NOOP command<61>. The default client KeepAlive timeout value is 45 seconds. KeepAlive timers with short intervals SHOULD be used to maximize compatibility with a variety of firewalls and proxies. The KeepAlive timer event processing is handled as specified in section 3.9.6.3.

3.9.3 SOCKS Client Initialization

3.9.3.1 SOCKS Protocol Initialization

The SOCKS Protocol is not initialized until a request to open an encapsulated connection is made by the application layer.

3.9.4 SOCKS Client Higher-Layer Triggered Events

3.9.4.1 Establishing a SOCKS Encapsulation Connection

When the application layer requests a SOCKS connection, the SOCKS protocol layer MUST initialize the SOCKS state variables as defined in the abstract data model (see section 3.9.1). After the connection state variables are initialized, the SOCKS protocol enters the connection establishment phase. Initialization SHOULD include fetching the proxy configuration information<62>.

The SOCKS handshake occurs between the client and proxy; there are no direct connections to servers.

3.9.4.1.1 Establishing a SOCKS Encapsulation Connection

The ConnectionState MUST be set to 'Negotiating'.

The client checks the SOCKS proxy configuration information before establishing any TCP connections. If a proxy is NOT configured, then SOCKS connections MUST NOT be attempted.

The client prepares a SOCKS Identifier Request Message as specified in [RFC1928], section 3.

The VER field MUST be set to 0x05.

The number of methods and method identifiers (see [RFC1928], section 3) supported by the client MUST be specified in the NMETHODS and METHODS fields<63>. The following methods SHOULD be specified in the message<64>:

0x00 (NO AUTHENTICATION REQUIRED)

0x02 (USERNAME and PASSWORD)

The ConnectionEstablishment timer and NetworkReceiveIO timer SHOULD both be started.

The client MUST establish a TCP connection to the proxy identified with ProxyServerHostName and ProxyServerPort and send the SOCKS Version Identifier Request.

3.9.4.2 Closing a SOCKS Connection

The client MUST close the SOCKS connection by sending a graceful TCP disconnect.

The ConnectionState then transitions into the ‘Closed’ state. The connection state variables SHOULD be discarded.

Copyright © 2014 Microsoft Corporation.

TCP client or server connections are closed using either a Graceful Close (TCP FIN Flag set to 1) or Abortive Close (TCP Reset Flag set to 1) mechanism. Closing the connection gracefully provides clients and servers with the benefit of never having to resend unacknowledged payload data. Abortive Closed connections are efficient in connection tear down but can require the client or server to resend payload data which has not been acknowledged at the TCP level. HTTP Encapsulation of SSTP Protocols relies on SSTP to ensure that any unacknowledged payload data is later resent. The choice of connection close mechanism is one of efficiency (efficient connection tear down verses efficient byte transmission). This document does not mandate that implementations adhere to one approach or another. Implementations can choose to close HTTP Encapsulated Connections either gracefully or abortively based on the implementation's requirements.

3.9.4.3 Sending Application Data

The SOCKS connection MUST be in the 'Established' state to send application data. If the connection is still being established, the client MUST buffer the data.

If the connection is in 'Established' state, the client sends application data over the connection, no SOCKS protocol messages.

The NetworkReceiveIO timer MUST be restarted.

3.9.5 SOCKS Client Message Processing Events and Sequencing Rules

Data received while the ConnectionState is not 'Established' is part of the SOCKS connection negotiation handshake as specified in section 3.9.5.1. Data received while the ConnectionState is 'Established' is handled as specified in section 3.9.5.2.

3.9.5.1 SOCKS Connection Negotiation Processing

The SOCKS response message type is dependent on the SOCKS ConnectionState. When data is received and the ConnectionState is 'Negotiating', the data is handled as specified in section 3.9.5.1.1. Data received when the ConnectionState is 'Connecting' is handled as specified in section 3.9.5.1.3.

3.9.5.1.1 Version Identifier Response

ConnectionState MUST be "Negotiating". Otherwise, it is a protocol error and the SOCKS connection MUST be closed, as specified in section 3.9.4.2.

The client verifies that the response message **VER** field number equals the client SOCKS version number, "0x05". If the versions are not equal, it is a protocol error, and the SOCKS connection MUST be closed, as specified in section 3.9.4.2.

If the response message **METHODS** field contains the method "0xFF" (NO ACCEPTABLE METHOD), it means that the server does not support any authentication methods supported by the client. The client MUST close the SOCKS connection, as specified in section 3.9.4.2.

If the response message **METHODS** field is "0x00" (NO AUTHENTICATION REQUIRED), the client sets the ConnectionState to "Negotiated" and continues as specified in section 3.9.5.1.2 to send the SOCKS connect request.

If the response message **METHODS** field is "0x01" (GSSAPI), the client MAY perform GSSAPI authentication negotiation, as specified in [RFC1961].<65>

If the response message **METHODS** field is "0x02" (USERNAME and PASSWORD), the client MUST perform **USERNAME** and **PASSWORD** authentication negotiation, as specified in [RFC1929].

Copyright © 2014 Microsoft Corporation.

After successfully completing the proxy authentication, the **ConnectionState** MUST transition to the "Negotiated" state.

The client MUST send a SOCKS connect request message, as specified in section 3.9.5.1.2.

3.9.5.1.2 Connect Request

The **ConnectionState** MUST be "Negotiated". Otherwise, it is a protocol error and the SOCKS connection MUST be closed, as specified in section 3.9.4.2.

The client sets **ConnectionState** to "Connecting".

The client constructs the SOCKS connect request as specified in [RFC1928], section 4.

The client sets the **DEST.ADDR** field with the value of the **ServerHost** state variable and the **DEST.PORT** field with the value of the **ServerPort** state variable.

The client MUST set "CONNECT [0x01]" as the CMD identifier.

The client SHOULD set "DOMAINNAME [0x03]" as the ATYP identifier.

The client sends a connect request.

3.9.5.1.3 Connect Response

The client validates the Connect Request as specified in [RFC1928], section 6.

A Reply field code of 0x00 indicates success. All other reply codes MUST be treated as SOCKS connection handshake failures. See [RFC1928], section 6, for a list of all Reply codes. In all error cases, the SOCKS connection MUST be closed (see section 3.9.4.2).

The KeepAlive timer and NetworkReceiveIO timer SHOULD both be started. The ConnectionEstablishment timer SHOULD be stopped and no further timer expiration processing is performed.

The SOCKS ConnectionState variable MUST transition to the 'Established' state.

If there is any buffered data to send, the client MUST send it, as specified in section 3.9.4.3.

3.9.5.2 Application Data Processing

The SOCKS connection MUST be in the "Established" state to receive application data.

If the connection state is not "Established", it is a protocol error. The data MUST be discarded and the connection closed.

The server receives application data over the connection. There are no additional SOCKS protocol messages. The application data is passed to the application layer for processing.

The **NetworkReceiveIO** timer MUST be stopped after each transport receive operation.

3.9.6 SOCKS Client Timer Events

3.9.6.1 ConnectionEstablishment Timer Event

The ConnectionEstablishment Event fires when the ConnectionEstablishment timer for a given SOCKs connection expires before the connection can be established. If this timer expires before the

Copyright © 2014 Microsoft Corporation.

SOCKs connection enters the 'Established' state, the TCP connection for the SOCKs connection SHOULD be closed by the client.

3.9.6.2 NetworkReceiveIO Timer Event

The NetworkReceiveIO Timer Event fires when a network receive does not complete within the specified amount of time. If the NetworkReceiveIO Timer Event triggers, the client SHOULD close the SOCKs connection, and then MAY retry the connection immediately.

3.9.6.3 KeepAlive Timer Event

A KeepAlive Event SHOULD trigger an encapsulated protocol message such as an SSTP_NOOP command to be sent across the wire<66>. Well behaved connections will see this event every KeepAlive interval. All SOCKS connections SHOULD use KeepAlive timers. Well behaved connections will see this event every KeepAlive interval.

3.9.7 SOCKS Client Other Local Events

On transport disconnect events all connection state information SHOULD be discarded.

The client SHOULD let the application layer decide whether to wait before establishing a new SOCKs connection to the target server again, or use a different encapsulation protocol to establish a connection to the server.

3.10 SOCKS Encapsulation of SSTP Protocol Server Details

From the server's standpoint, the SOCKs proxy only sends SSTP protocol traffic. SOCKs proxies connect to servers on the SSTP well known port 2492/TCP. The server does not participate in the SOCKs handshake between the client and the SOCKs proxy. All messages that flow between the client and server are standard SSTP messages, with no SOCKs protocol messages. See SSTP [MS-GRVSSTP] for server processing rules.

3.10.1 SOCKS Server Abstract Data Model

None.

3.10.2 SOCKS Server Timers

None.

3.10.3 SOCKS Server Initialization

None.

3.10.4 SOCKS Server Higher-Layer Triggered Events

None.

3.10.5 SOCKS Server Message Processing Events and Sequencing Rules

None.

Copyright © 2014 Microsoft Corporation.

3.10.6 SOCKS Server Timer Events

None.

3.10.7 SOCKS Server Other Local Events

None.

Copyright © 2014 Microsoft Corporation.

4 Protocol Examples

This section explains the sequence and structure of the messages required to successfully create an HTTP Encapsulation of SSTP connection to the point where the client and the server can exchange Application-Data (see section 2.2.1.1.4) with each other. In addition, this section also includes Secure Tunnel, SOCKS proxy and NTLM ProxyAuthentication examples.

In the HTTP encapsulation examples, the data sent and received on the wire is displayed between "-----Message START-----" and "-----Message END-----" tokens for readability. These tokens are not part of the protocol. The CRLF token in the HTTP headers is replaced with a new line to make the message more readable. An empty line indicates an additional CRLF token.

4.1 HTTP LongLived Encapsulation Examples

HTTP LongLived encapsulation is a full duplex virtual connection that consists of two individual TCP Connections made to the server on port 80. In the following examples, the connections to the server are named GET and POST. The POST connection is used to send data to the relay and the GET connection is used to request data from the relay. The following diagram shows the process used to establish a connection using HTTP LongLived encapsulation.

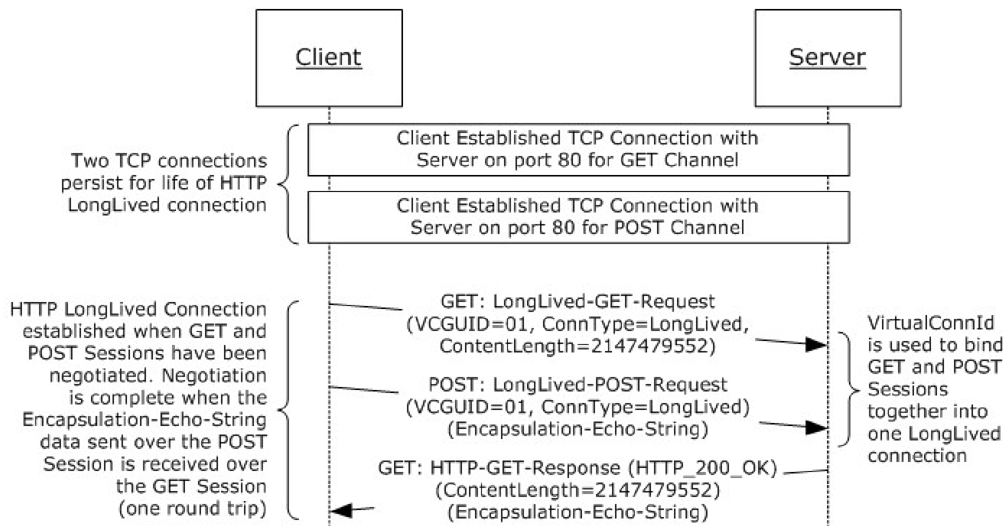

Figure 22: LongLived Encapsulation connection establishment

The following are examples of the HTTP messages exchanged to create a virtual LongLived Connection:

LongLived-GET-Request:

```
----------------------------------Message START ----------------------------------
GET
/2.0/server.domain.net/hczn5kctbrpxfgkgxzqs6zmkp9uwvswszvs6f72,ConnType=LongLived,ContentLeng
th=2147479552 HTTP/1.0
Accept: */*
Content-Type: application/octet-stream
User-Agent: Mozilla/4.0 (compatible; MSIE 5.5; Win32)
Host: 10.150.1.226
Pragma: no-cache
```


Copyright © 2014 Microsoft Corporation.

```
Cache-Control: no-cache
Expires: 0
Cache-Control: max-age=0

---------------------------------Message END -----------------------------------
```

LongLived-POST-Request:

```
---------------------------------Message START ---------------------------------
POST /2.0/server.domain.net/hczn5kctbrpxfgkgxzqs6zmkp9uwvswszvs6f72,ConnType=LongLived
HTTP/1.0
Accept: */*
Content-Type: application/octet-stream
User-Agent: Mozilla/4.0 (compatible; MSIE 5.5; Win32)
UserAgent: server.domain.net
Content-Length: 2147479552
Pragma: no-cache
Cache-Control: no-cache
Expires: 0
Cache-Control: max-age=0

GroovePing: 1.0,Ping

---------------------------------Message END -----------------------------------
```

LongLived-GET-Response:

```
---------------------------------Message START ---------------------------------
HTTP/1.0 200 OK
Date: Wed, 26 Dec 2007 20:31:28 GMT
Server: Groove-Relay/12.0
Connection: Keep-Alive
Content-Length: 2147479552

GroovePing: 1.0,Ping

---------------------------------Message END -----------------------------------
```

Note that the initial response is received only on the GET connection. The HTTP return code of 200 indicates the successful creation of a virtual LongLived Connection.

After the virtual LongLived is established, the client and server can exchange application data as shown in the following figure.

Copyright © 2014 Microsoft Corporation.

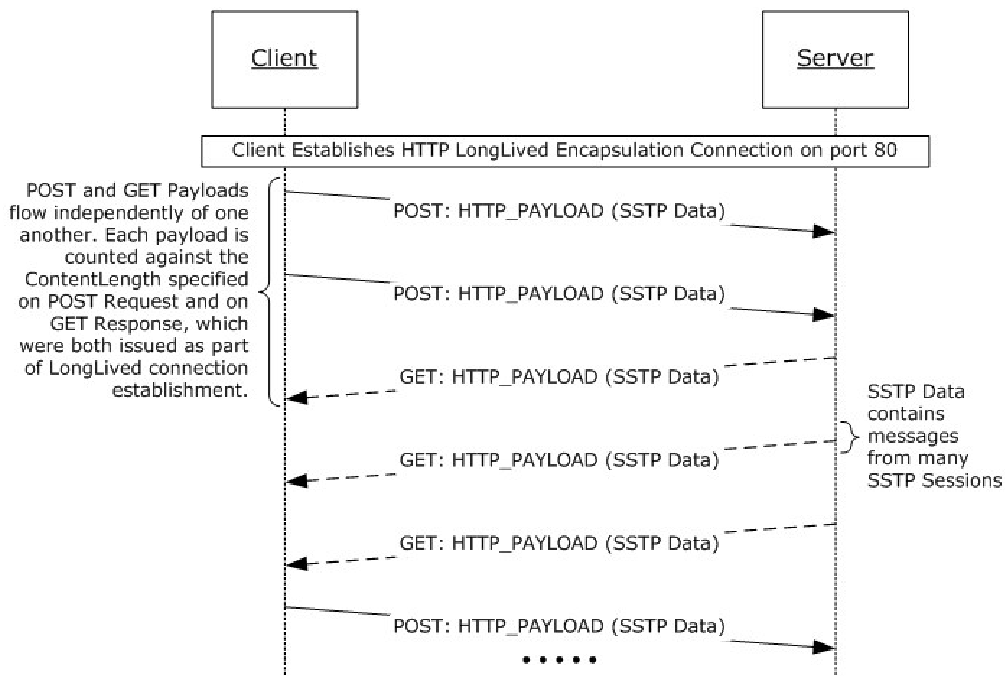

Figure 23: LongLived Encapsulation application data exchange

4.2 HTTP KeepAlive Encapsulation Examples

HTTP KeepAlive encapsulation client creates a full duplex virtual connection that consists of two individual TCP connections. In the following examples, the connections to the server are named GET and POST. The POST connection is used to send data to the server; the GET connection is used to request data from the server. The following figure shows how the KeepAlive encapsulation connection is established.

Copyright © 2014 Microsoft Corporation.

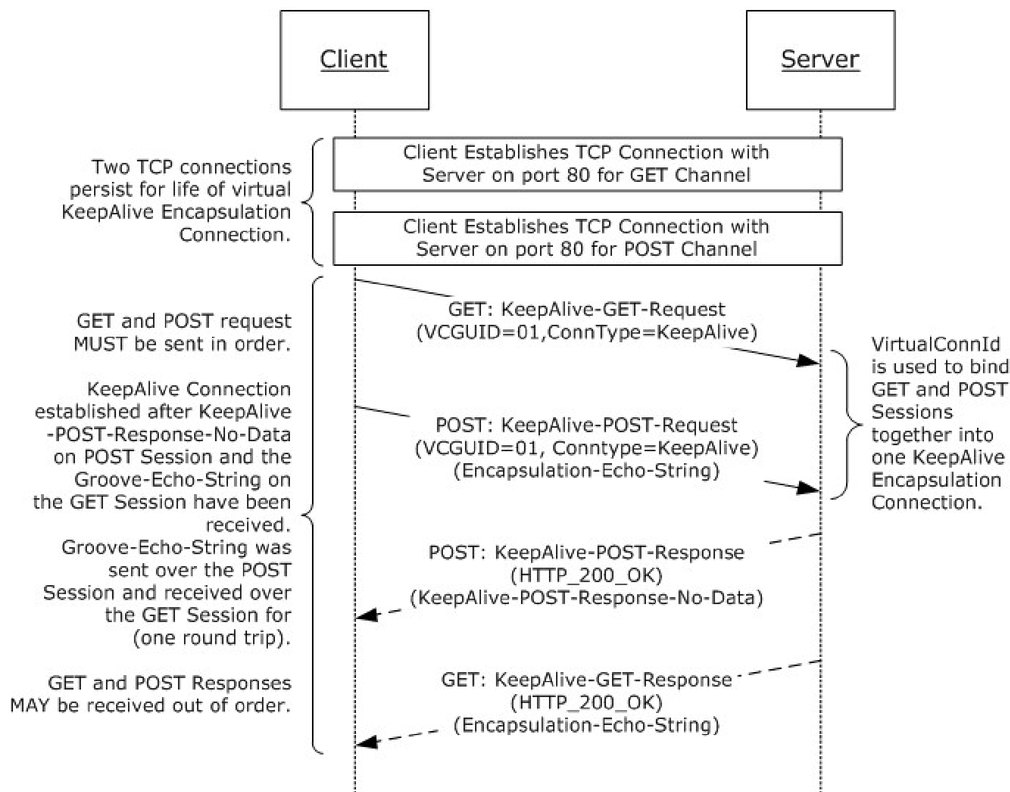

Figure 24: KeepAlive Encapsulation connection establishment

The following are the messages exchanged to initialize a KeepAlive connection:

KeepAlive-GET-Request:

```
---------------------------------Message START ---------------------------------
GET /2.0/server.domain.net/kicxp8rrgwqdwhf7c6xsgbagmcdnxm9phtvbj5a,ConnType=KeepAlive
HTTP/1.0
Accept: */*
Content-Type: application/octet-stream
User-Agent: Mozilla/4.0 (compatible; MSIE 5.5; Win32)
Host: 10.150.1.226
Pragma: no-cache
Cache-Control: no-cache
Expires: 0
Connection: Keep-Alive
Cache-Control: max-age=0

---------------------------------Message END -----------------------------------
```

KeepAlive-POST-Request:

```
---------------------------------Message START ---------------------------------
POST /2.0/server.domain.net/kicxp8rrgwqdwhf7c6xsgbagmcdnxm9phtvbj5a,ConnType=KeepAlive
HTTP/1.0
Accept: */*
```


Copyright © 2014 Microsoft Corporation.

```
Content-Type: application/octet-stream
User-Agent: Mozilla/4.0 (compatible; MSIE 5.5; Win32)
UserAgent: server.domain.net
Connection: Keep-Alive
Content-Length: 22
Pragma: no-cache
Cache-Control: no-cache
Expires: 0
Cache-Control: max-age=0

GroovePing: 1.0,Ping

----------------------------------Message END ------------------------------------
```

KeepAlive-POST-Response:

```
----------------------------------Message START ----------------------------------
HTTP/1.0 200 OK
Date: Wed, 26 Dec 2007 19:50:26 GMT
Server: Groove-Relay/12.0
Connection: Keep-Alive
Content-Length: 15

<HTML></HTML>

----------------------------------Message END ------------------------------------
```

KeepAlive-GET-Response:

```
----------------------------------Message START ----------------------------------
HTTP/1.0 200 OK
Date: Wed, 26 Dec 2007 19:50:26 GMT
Server: Groove-Relay/12.0
Connection: Keep-Alive
Content-Length: 22

GroovePing: 1.0,Ping

----------------------------------Message END ------------------------------------
```

For a KeepAlive connection, each request results in a response from the server. The HTTP return codes of 200 on both responses indicates the successful creation of both HTTP connections. The GroovePing message is sent on the POST Request and received on the GET Response to complete the establishment of the virtual KeepAlive connection.

The following figure shows the message flow that is used to send and receive application data between protocol client and protocol server.

Copyright © 2014 Microsoft Corporation.

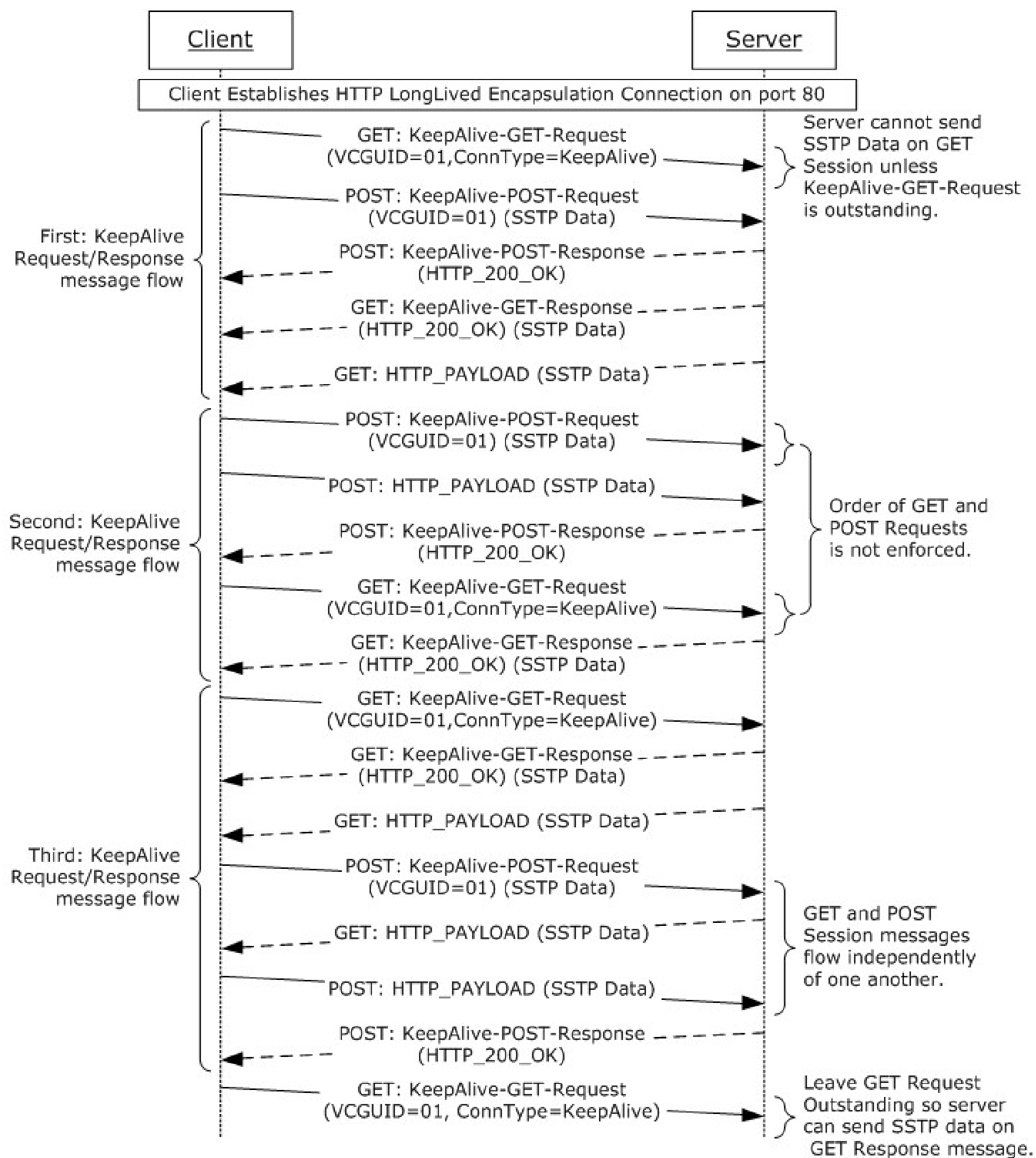

Figure 25: HTTP KeepAlive Encapsulation application data exchange

The following are examples of the messages that are exchanged to send and receive application data between the client and server.

KeepAlive-GET-Request:

```
----------------------------------Message START ----------------------------------
GET /2.0/server.domain.net/kicxp8rrgwqdwhf7c6xsgbagmcdnxm9phtvbj5a,ConnType=KeepAlive
HTTP/1.0
Accept: */*
Content-Type: application/octet-stream
User-Agent: Mozilla/4.0 (compatible; MSIE 5.5; Win32)
Host: 10.150.1.226
Pragma: no-cache
```


Copyright © 2014 Microsoft Corporation.

```
Cache-Control: no-cache
Expires: 0
Connection: Keep-Alive
Cache-Control: max-age=0

---------------------------------Message END ------------------------------------
```

KeepAlive-POST-Request:

```
---------------------------------Message START ----------------------------------
POST /2.0/server.domain.net/kicxp8rrgwqdwhf7c6xsgbagmcdnxm9phtvbj5a,ConnType=KeepAlive
HTTP/1.0
Accept: */*
Content-Type: application/octet-stream
User-Agent: Mozilla/4.0 (compatible; MSIE 5.5; Win32)
UserAgent: server.domain.net
Connection: Keep-Alive
Content-Length: 188
Pragma: no-cache
Cache-Control: no-cache
Expires: 0
Cache-Control: max-age=0

Application-Data
---------------------------------Message END ------------------------------------
```

KeepAlive-GET-Response:

```
---------------------------------Message START ----------------------------------
HTTP/1.0 200 OK
Date: Wed, 26 Dec 2007 19:50:26 GMT
Server: Groove-Relay/12.0
Connection: Keep-Alive
Content-Length: 169

Application-Data
---------------------------------Message END ------------------------------------
```

KeepAlive-POST-Response:

```
---------------------------------Message START ----------------------------------
HTTP/1.0 200 OK
Date: Wed, 26 Dec 2007 19:50:26 GMT
Server: Groove-Relay/12.0
Connection: Keep-Alive
Content-Length: 0

---------------------------------Message END ------------------------------------
```

KeepAlive-GET-Request:

```
---------------------------------Message START ----------------------------------
GET /2.0/server.domain.net/g5u55h8rrgwqdwhf7c6xsgbagmcdnxm9pht2kop41a,ConnType=KeepAlive
HTTP/1.0
```


Copyright © 2014 Microsoft Corporation.

```
Accept: */*
Content-Type: application/octet-stream
User-Agent: Mozilla/4.0 (compatible; MSIE 5.5; Win32)
Host: 10.150.1.226
Pragma: no-cache
Cache-Control: no-cache
Expires: 0
Connection: Keep-Alive
Cache-Control: max-age=0

---------------------------------Message END -----------------------------------
```

4.3 HTTP Polling Encapsulation Examples

HTTP Polling encapsulation uses a series of TCP connections carrying HTTP POST requests and responses. Each POST request-response pair forms a logical half duplex connection for transmitting POST data in the POST entity bodies. This logical connection is used to send and receive application data between the client and server.

The following figure shows the establishment of the virtual polling connection.

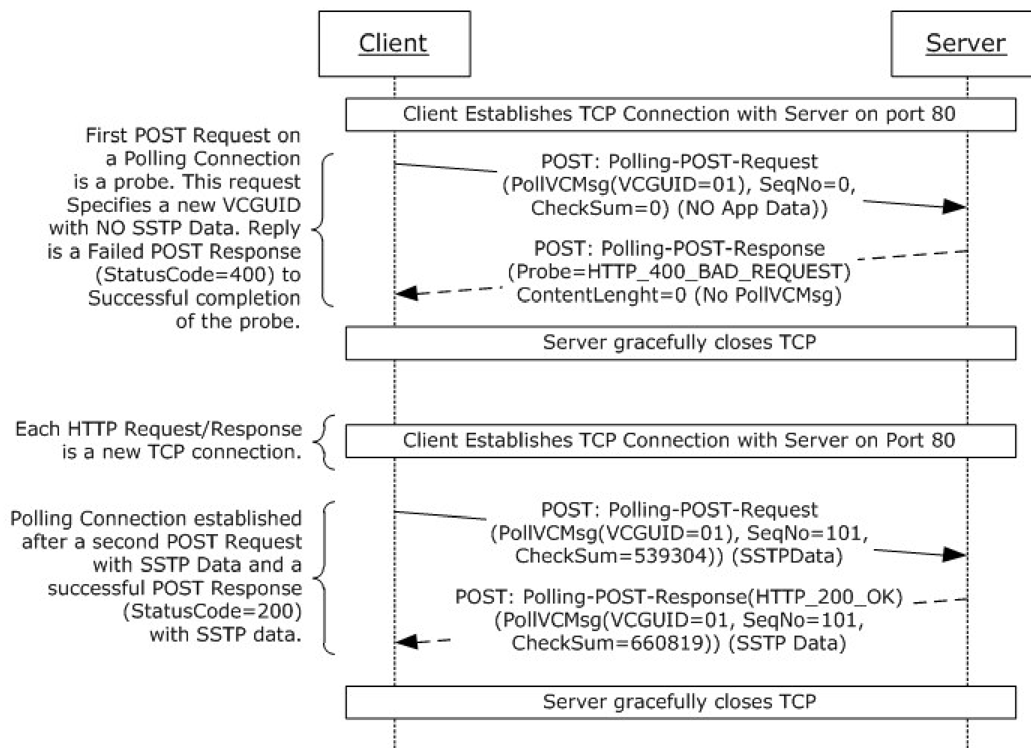

Figure 26: Polling connection handshake

Polling-POST-Request:

```
---------------------------------Message START ---------------------------------
POST / HTTP/1.0
Accept: */*
```


Copyright © 2014 Microsoft Corporation.

```
Content-Type: application/octet-stream
User-Agent: Mozilla/4.0 (compatible; MSIE 5.5; Win32)
Host: 10.150.1.226
Content-Length: 79
Pragma: no-cache
Cache-Control: no-cache
Expires: 0
Cache-Control: max-age=0

Polling-Request-Entity-Body-1
----------------------------------Message END ------------------------------------
```

Polling-Request-Entity-Body-1:

```
Offset                 HEX Details                         ASCII Details
----  -----------------------------------------------  ------------------
0000  31 2e 32 00 67 72 6f 6f 76 65 44 4e 53 3a 2f 2f  1.2.grooveDNS://
0010  73 65 72 76 65 72 30 31 2e 72 65 6c 61 79 2e 6e  server01.relay.n
0020  65 74 00 6d 33 75 37 6d 35 65 76 36 69 7a 39 68  et.m3u7m5ev6iz9h
0030  6a 36 6d 78 39 37 73 34 6b 64 72 6e 6b 38 6b 68  j6mx97s4kdrnk8kh
0040  61 6a 76 62 33 62 77 6e 62 61 00 30 00 30 00     ajvb3bwnba.0.0.
```

Polling-POST-Response:

```
----------------------------------Message START ----------------------------------
HTTP/1.0 400 Bad Request
Date: Wed, 26 Dec 2007 19:01:55 GMT
Server: Groove-Relay/12.0
Connection: Keep-Alive
Content-Length: 0

----------------------------------Message END ------------------------------------
```

The server gracefully closes the connection.

The client again creates a TCP connection with the server on port 80:

Polling-POST-Request:

```
----------------------------------Message START ----------------------------------
 POST / HTTP/1.0
Accept: */*
Content-Type: application/octet-stream
User-Agent: Mozilla/4.0 (compatible; MSIE 5.5; Win32)
Host: 10.150.1.226
Content-Length: 272
Pragma: no-cache
Cache-Control: no-cache
Expires: 0
Cache-Control: max-age=0

Polling-Request-Entity-Body-2
----------------------------------Message END ------------------------------------
```


Copyright © 2014 Microsoft Corporation.

Polling-Request-Entity-Body-2:

```
Offset                HEX Details                            ASCII Details
----  -----------------------------------------------  ------------------
0000  31 2e 32 00 67 72 6f 6f 76 65 44 4e 53 3a 2f 2f  1.2.grooveDNS://
0010  73 65 72 76 65 72 30 31 2e 72 65 6c 61 79 2e 6e  server01.relay.n
0020  65 74 00 6d 33 75 37 6d 35 65 76 36 69 7a 39 68  et.m3u7m5ev6iz9h
0030  6a 36 6d 78 39 37 73 34 6b 64 72 6e 6b 38 6b 68  j6mx97s4kdrnk8kh
0040  61 6a 76 62 33 62 77 6e 62 61 00 30 00 35 33 39  ajvb3bwnba.0.539
0050  33 30 34 00 01 bc 00 01 05 00 67 72 6f 6f 76 65  304.groove
0060  44 4e 53 3a 2f 2f 73 65 72 76 65 72 30 31 2e 72  DNS://server01.r
0070  65 6c 61 79 2e 6e 65 74 00 01 64 70 70 3a 2f 2f  elay.net.dpp://
0080  2f 6d 6b 38 6b 35 64 61 70 37 6e 66 62 6e 69 33  /mk8k5dap7nfbni3
0090  63 74 77 79 62 6d 65 77 32 68 32 73 67 69 33 66  ctwybmew2h2sgi3f
00a0  68 33 67 6a 35 69 75 69 00 4d 00 01 03 01 18 00  h3gj5iui.M.
00b0  e9 4c b5 b2 b7 df 00 c9 65 ed 36 ea ed 61 26 1d .L.e.6.a&.
00c0  a8 a0 a2 cb 5d 1b 83 66 14 00 98 90 f1 20 14 f5 .].f.
00d0  31 cf 1d 03 93 09 0d ee 8e 70 8d 93 21 8c 18 00  1.p.!.
00e0  59 0f 17 da 46 1a 3d 1a ae 5b 86 99 93 45 59 9f  Y.F.=.[.EY.
00f0  41 c5 7e c4 07 f5 9e ed 47 72 6f 6f 76 65 20 43  A.~.Groove C
0100  6c 69 65 6e 74 20 34 2e 32 20 32 36 32 33 00 00  lient 4.2 2623.
```

Polling-POST-Response:

```
----------------------------------Message START -----------------------------------
/1.0 200 OK
Date: Wed, 26 Dec 2007 19:01:55 GMT
Server: Groove-Relay/12.0
Connection: Keep-Alive
Content-Length: 261

Polling-Response-Entity-Body-2
----------------------------------Message END -------------------------------------
```

Polling-Response-Entity-Body-1

```
Offset                HEX Details                            ASCII Details
----  -----------------------------------------------  ------------------
0000  31 2e 32 00 67 72 6f 6f 76 65 44 4e 53 3a 2f 2f  1.2.grooveDNS://
0010  73 65 72 76 65 72 30 31 2e 72 65 6c 61 79 2e 6e  server01.relay.n
0020  65 74 00 6d 33 75 37 6d 35 65 76 36 69 7a 39 68  et.m3u7m5ev6iz9h
0030  6a 36 6d 78 39 37 73 34 6b 64 72 6e 6b 38 6b 68  j6mx97s4kdrnk8kh
0040  61 6a 76 62 33 62 77 6e 62 61 00 30 00 36 36 30  ajvb3bwnba.0.660
0050  38 31 39 00 31 32 30 2c 35 2c 33 00 02 a9 00 01  819.120,5,3.
0060  05 00 67 00 01 03 02 18 00 69 06 19 ae 3b ad 0b .g.i.;.
0070  7a 5a 7a 67 e0 a2 90 41 65 28 5b f6 d2 e7 8e b4  zZzg.Ae([.
0080  f6 14 00 9d 05 4d 7d b0 a0 ed 48 1e d7 05 ba f0 .M}.H.
0090  24 a0 d6 be 02 2b d1 18 00 00 e8 f4 ab 6c b1 f0  $.+.l.
00a0  c0 e9 2d c8 0b 2d 91 5e 07 4c 9a 24 34 b7 f1 b8 .-.-.^.L.$4.
00b0  10 18 00 bc 51 27 8e 48 f5 d6 73 9a 53 32 3f 68 .Q'.H.s.S2?h
00c0  93 b6 c5 1d 55 4f e2 42 51 21 13 03 47 72 6f 6f .UO.BQ!.Groo
00d0  76 65 20 52 65 6c 61 79 20 31 32 2e 30 20 31 34  ve Relay 12.0 14
00e0  30 37 00 00 01 67 72 6f 6f 76 65 44 4e 53 3a 2f  07.grooveDNS:/
00f0  2f 73 65 72 76 65 72 30 31 2e 72 65 6c 61 79 2e  /server01.relay.
0100  6e 65 74 00 00                                   net.
```


Copyright © 2014 Microsoft Corporation.

The following diagram shows the exchange of messages for data that is fragmented by TCP or by the application itself.

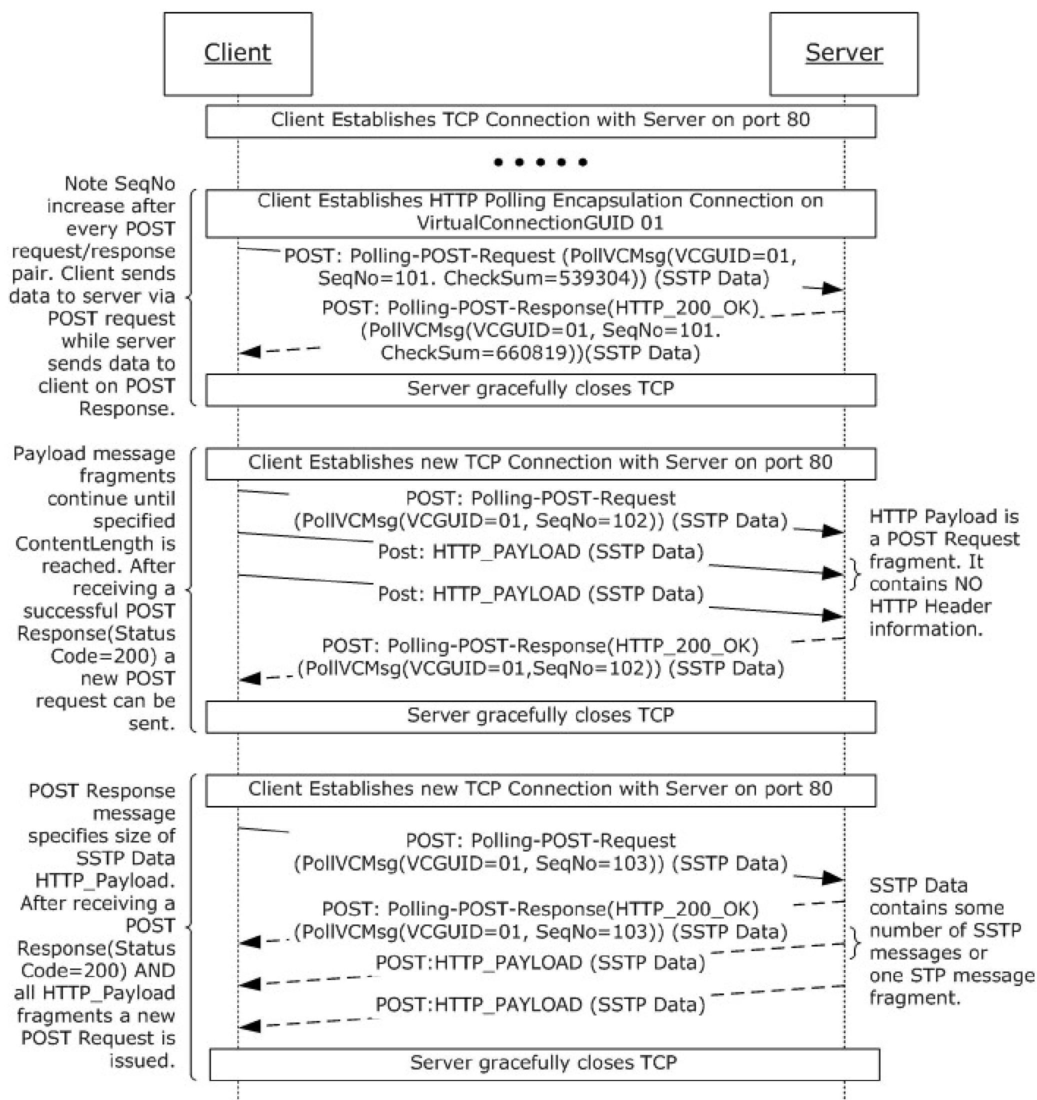

Figure 27: Polling connection data exchange

The following diagram provides an example of a polling request and response sequence in which the client sends data to the server but the server has no data to send in return.

Copyright © 2014 Microsoft Corporation.

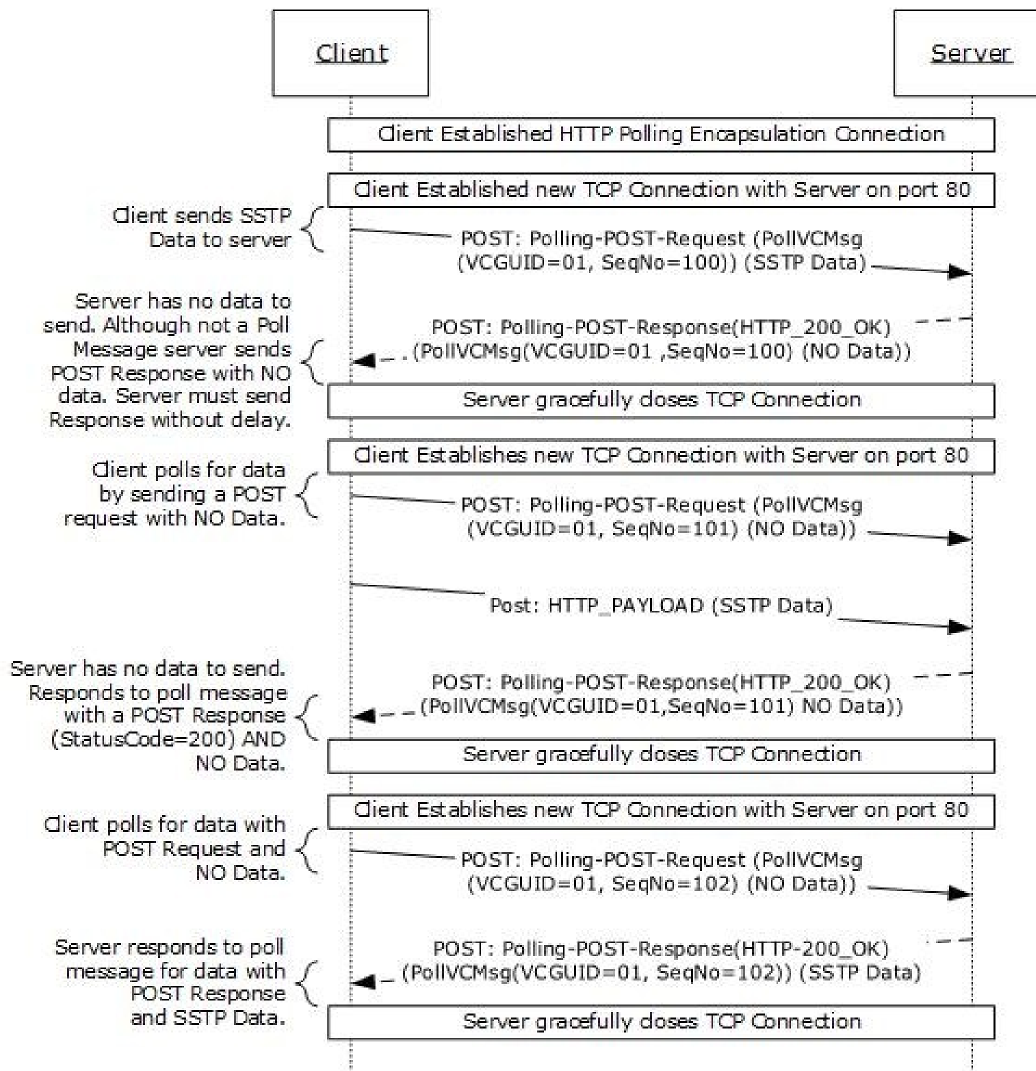

Figure 28: Polling connection with client polling server for data

The following provides an example of a polling request and response for the same scenario in which the client sends data to the server but the server has no data to send to the client. In all of the following examples, the client creates a TCP connection to the server on port 80 for each request and the server closes the connection after each response.

Polling-POST-Request:

```
---------------------------------Message START ---------------------------------
POST / HTTP/1.0
Accept: */*
Content-Type: application/octet-stream
User-Agent: Mozilla/4.0 (compatible; MSIE 5.5; Win32)
Host: 10.150.1.226
Content-Length: 1087
Pragma: no-cache
Cache-Control: no-cache
```


Copyright © 2014 Microsoft Corporation.

```
Expires: 0
Cache-Control: max-age=0

Polling-Request-Entity-Body-3
----------------------------------Message END ------------------------------------
```

Polling-Request-Entity-Body-3:

```
Offset                 HEX Details                          ASCII Details
----  -----------------------------------------------  ------------------
0000  31 2e 32 00 67 72 6f 6f 76 65 44 4e 53 3a 2f 2f  1.2.grooveDNS://
0010  73 65 72 76 65 72 30 31 2e 72 65 6c 61 79 2e 6e  server01.relay.n
0020  65 74 00 61 35 73 32 66 6a 38 71 35 35 63 78 6e  et.a5s2fj8q55cxn
0030  65 32 76 34 77 72 34 38 61 64 39 63 69 66 66 73  e2v4wr48ad9ciffs
0040  7a 6e 7a 71 39 61 70 63 7a 69 00 33 37 00 33 33  znzq9apczi.37.33
0050  35 36 31 33 34 30 00 0e a0 00 21 00 00 00 4d 49  561340.!.MI
0060  4d 45 2d 56 65 72 73 69 6f 6e 3a 20 31 2e 30 20  ME-Version: 1.0
0070  28 47 72 6f 6f 76 65 20 32 29 0d 0a 43 6f 6e 74  (Groove 2).Cont
0080  65 6e 74 2d 54 79 70 65 3a 20 6d 75 6c 74 69 70  ent-Type: multip
0090  61 72 74 2f 72 65 6c 61 74 65 64 3b 20 62 6f 75  art/related; bou
00a0  6e 64 61 72 79 3d 22 3c 3c 5b 5b 26 26 26 5d 5d  ndary="<<[[&&&]]
00b0  3e 3e 22 0d 0a 3c 3c 5b 5b 26 26 26 5d 5d 3e 3e  >>".<<[[&&&]]>>
00c0  0d 0a 43 6f 6e 74 65 6e 74 2d 54 79 70 65 3a 20 .Content-Type:
        ..................................................................
0410  81 72 04 82 0b 83 82 0e 01 84 83 3f 04 83 53 83 .r..?.S.
0420  83 59 01 01 01 0d 0a 2d 2d 3c 3c 5b 5b 26 26 26 .Y.--<<[[&&&
0430  5d 5d 3e 3e 2d 2d 0d 0a 0f 07 00 21 00 00 00     ]]>>--.!.
```

Polling-POST-Response:

```
----------------------------------Message START ----------------------------------
HTTP/1.0 200 OK
Date: Wed, 26 Dec 2007 19:02:10 GMT
Server: Groove-Relay/12.0
Connection: Keep-Alive
Content-Length: 88

Polling-Response-Entity-Body-3
----------------------------------Message END ------------------------------------
```

Polling-Response-Entity-Body-3:

```
Offset                 HEX Details                          ASCII Details
----  -----------------------------------------------  ------------------
0000  31 2e 32 00 67 72 6f 6f 76 65 44 4e 53 3a 2f 2f  1.2.grooveDNS://
0010  73 65 72 76 65 72 30 31 2e 72 65 6c 61 79 2e 6e  server01.relay.n
0020  65 74 00 61 35 73 32 66 6a 38 71 35 35 63 78 6e  et.a5s2fj8q55cxn
0030  65 32 76 34 77 72 34 38 61 64 39 63 69 66 66 73  e2v4wr48ad9ciffs
0040  7a 6e 7a 71 39 61 70 63 7a 69 00 33 37 00 30 00  znzq9apczi.37.0.
0050  31 32 30 2c 35 2c 33 00                          120,5,3.
```


Copyright © 2014 Microsoft Corporation.

The following Polling request/response example shows the messages exchanged when the client has no data to send to the server but the server has some data to send to the client.

```
Polling-POST-Request:
---------------------------------Message START ---------------------------------
POST / HTTP/1.0
Accept: */*
Content-Type: application/octet-stream
User-Agent: Mozilla/4.0 (compatible; MSIE 5.5; Win32)
Host: 10.150.1.226
Content-Length: 80
Pragma: no-cache
Cache-Control: no-cache
Expires: 0
Cache-Control: max-age=0

Polling-Request-Entity-Body-4
---------------------------------Message END -----------------------------------
```

Polling-Request-Entity-Body-4:

```
Offset                 HEX Details                          ASCII Details
----  -----------------------------------------------  ------------------
0000  31 2e 32 00 67 72 6f 6f 76 65 44 4e 53 3a 2f 2f  1.2.grooveDNS://
0010  73 65 72 76 65 72 30 31 2e 72 65 6c 61 79 2e 6e  server01.relay.n
0020  65 74 00 61 35 73 32 66 6a 38 71 35 35 63 78 6e  et.a5s2fj8q55cxn
0030  65 32 76 34 77 72 34 38 61 64 39 63 69 66 66 73  e2v4wr48ad9ciffs
0040  7a 6e 7a 71 39 61 70 63 7a 69 00 33 38 00 30 00  znzq9apczi.38.0.
```

Polling-POST-Response:

```
---------------------------------Message START ---------------------------------
HTTP/1.0 200 OK
Date: Wed, 26 Dec 2007 19:02:10 GMT
Server: Groove-Relay/12.0
Connection: Keep-Alive
Content-Length: 96

Polling-Response-Entity-Body-4
---------------------------------Message END -----------------------------------
```

Polling-Response-Entity-Body-4:

```
Offset                 HEX Details                          ASCII Details
----  -----------------------------------------------  ------------------
0000  31 2e 32 00 67 72 6f 6f 76 65 44 4e 53 3a 2f 2f  1.2.grooveDNS://
0010  73 65 72 76 65 72 30 31 2e 72 65 6c 61 79 2e 6e  server01.relay.n
0020  65 74 00 61 35 73 32 66 6a 38 71 35 35 63 78 6e  et.a5s2fj8q55cxn
0030  65 32 76 34 77 72 34 38 61 64 39 63 69 66 66 73  e2v4wr48ad9ciffs
0040  7a 6e 7a 71 39 61 70 63 7a 69 00 33 38 00 36 32  znzq9apczi.38.62
0050  00 31 32 30 2c 35 2c 33 00 10 07 00 01 00 00 00 .120,5,3.
```


Copyright © 2014 Microsoft Corporation.

The following Polling request/response shows the messages when the server and the client do not have any data to send to each other.

Polling-POST-Request:

```
---------------------------------Message START ---------------------------------
POST / HTTP/1.0
Accept: */*
Content-Type: application/octet-stream
User-Agent: Mozilla/4.0 (compatible; MSIE 5.5; Win32)
Host: 10.150.1.226
Content-Length: 80
Pragma: no-cache
Cache-Control: no-cache
Expires: 0
Cache-Control: max-age=0

Polling-Request-Entity-Body-5
---------------------------------Message END -----------------------------------
```

Polling-Request-Entity-Body-5:

```
Offset                 HEX Details                        ASCII Details
----  -----------------------------------------------  ------------------
0000  31 2e 32 00 67 72 6f 6f 76 65 44 4e 53 3a 2f 2f  1.2.grooveDNS://
0010  73 65 72 76 65 72 30 31 2e 72 65 6c 61 79 2e 6e  server01.relay.n
0020  65 74 00 61 35 73 32 66 6a 38 71 35 35 63 78 6e  et.a5s2fj8q55cxn
0030  65 32 76 34 77 72 34 38 61 64 39 63 69 66 66 73  e2v4wr48ad9ciffs
0040  7a 6e 7a 71 39 61 70 63 7a 69 00 33 39 00 30 00  znzq9apczi.39.0.
```

Polling-POST-Response:

```
---------------------------------Message START ---------------------------------
HTTP/1.0 200 OK
Date: Wed, 26 Dec 2007 19:02:10 GMT
Server: Groove-Relay/12.0
Connection: Keep-Alive
Content-Length: 88

Polling-Response-Entity-Body-5
---------------------------------Message END -----------------------------------
```

Polling-Response-Entity-Body-5:

```
Offset                 HEX Details                        ASCII Details
----  -----------------------------------------------  ------------------
0000  31 2e 32 00 67 72 6f 6f 76 65 44 4e 53 3a 2f 2f  1.2.grooveDNS://
0010  73 65 72 76 65 72 30 31 2e 72 65 6c 61 79 2e 6e  server01.relay.n
0020  65 74 00 61 35 73 32 66 6a 38 71 35 35 63 78 6e  et.a5s2fj8q55cxn
0030  65 32 76 34 77 72 34 38 61 64 39 63 69 66 66 73  e2v4wr48ad9ciffs
0040  7a 6e 7a 71 39 61 70 63 7a 69 00 33 39 00 30 00  znzq9apczi.39.0.
0050  31 32 30 2c 35 2c 33 00                          120,5,3.
```


Copyright © 2014 Microsoft Corporation.

4.4 Secure Tunnel Proxy Protocol Examples

The Secure Tunnel Proxy Protocol [TCPPROXY] relies on an HTTP proxy to create a connection to the server. Once the Secure Tunnel proxy creates a connection to the server, the Application-Data (see section 2.2.1.1.4) can be exchanged transparently through the proxy.

Section 2.2.5 includes examples of messages exchanged between the client and the Secure Tunnel proxy. After the Secure Tunnel proxy successfully creates a connection with the server, application data can be exchanged as shown in the following diagram.

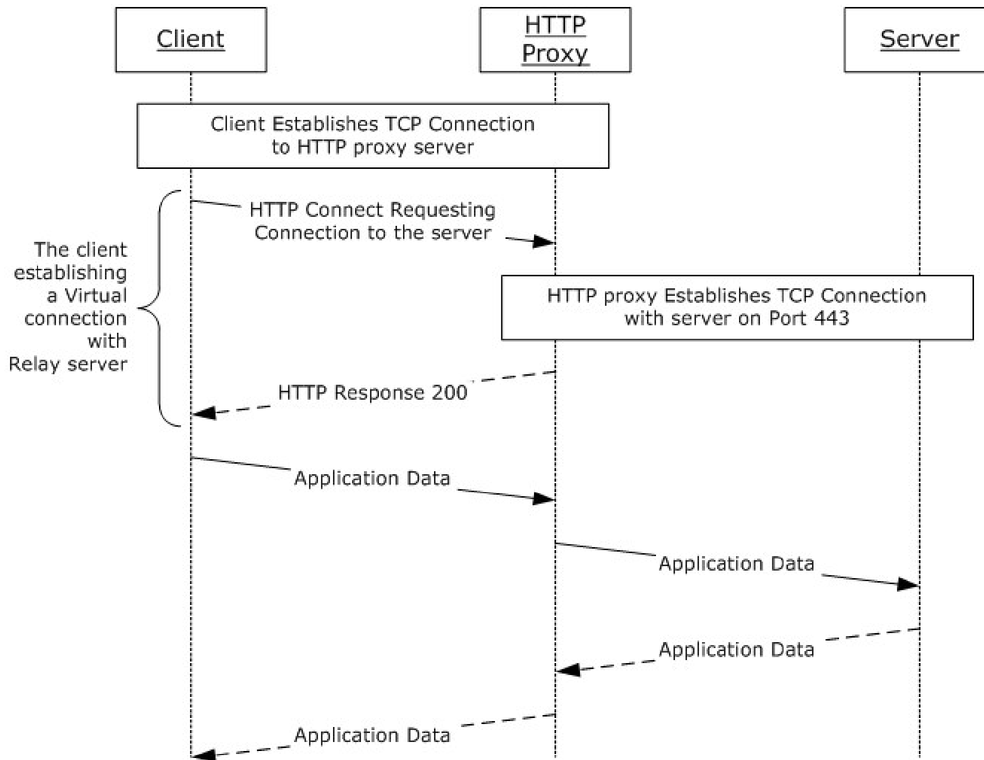

Figure 29: Secure tunnel proxy message flow

4.5 SOCKS Proxy

Section 2.2.6 includes examples of messages exchanged between the client and the SOCKS[RFC1928] proxy. After the SOCKS proxy successfully creates a connection with the relay, the Application-Data (see section 2.2.1.1.4) can be exchanged transparently to the SOCKS proxy as graphically represented in the following diagram.

Copyright © 2014 Microsoft Corporation.

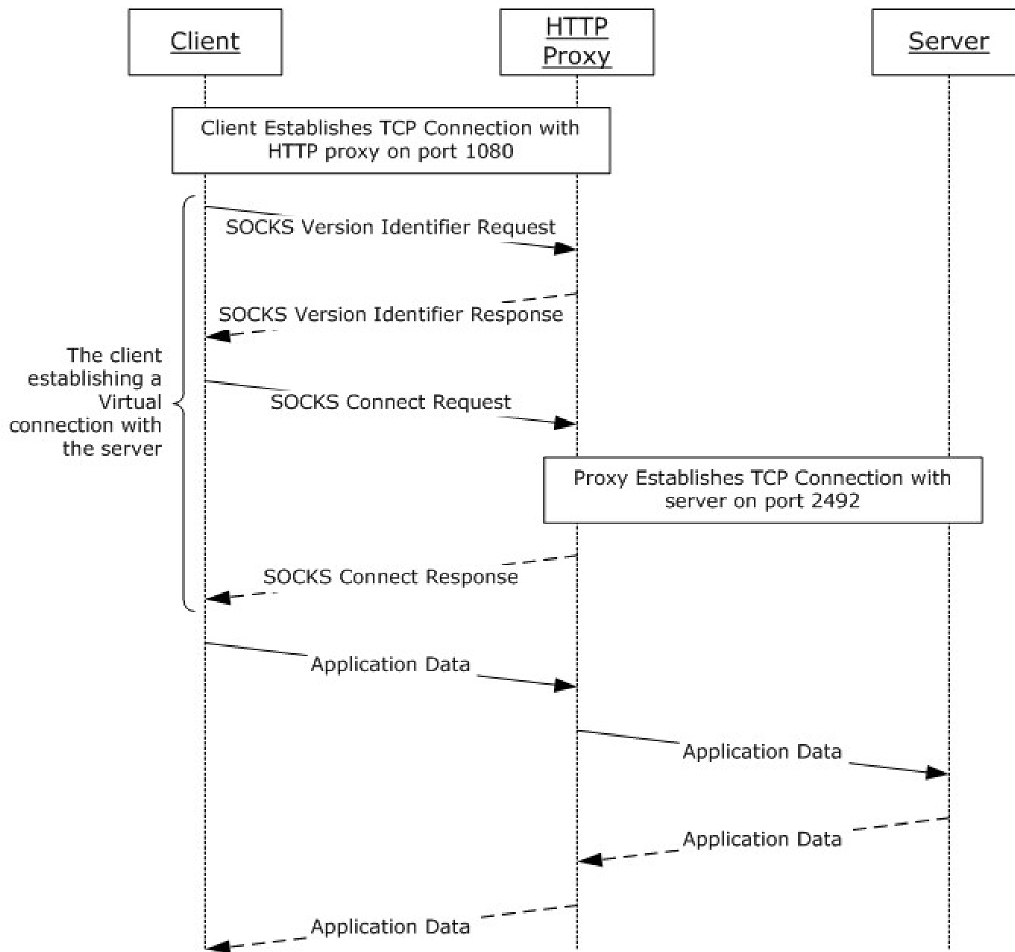

Figure 30: Client to relay message flow through a SOCKS proxy

4.6 Proxy Authentication using NTLM Example

The following example illustrates the sequence of messages exchanged to communicate through a NTLM enabled proxy. These examples use the Secure Tunnel proxy to enable the NTLM authentication.

Copyright © 2014 Microsoft Corporation.

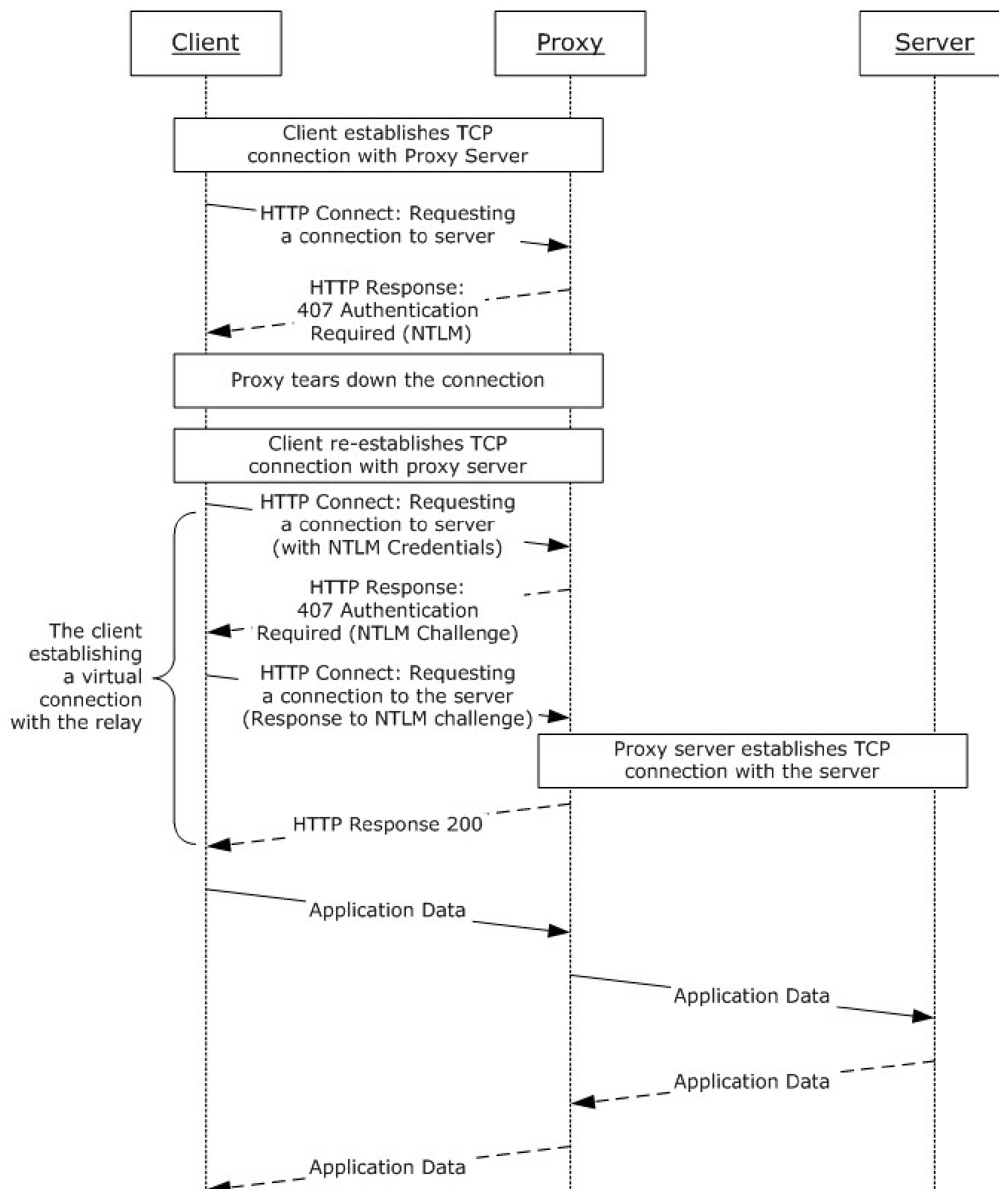

Figure 31: Client NTLM authentication example

The following is an example of the messages exchanged between the client and the Secure Tunnel Proxy to create a connection between the client and the server.

The client creates a TCP connection to the Secure Tunnel proxy and requests a connection to the server using the following message:

```
---------------------------------Message START ---------------------------------
CONNECT server.domain.net:443 HTTP/1.0
User-Agent:Mozilla/4.0 (compatible; MSIE 5.5; Win32)
proxy-Connection: Keep-Alive
Pragma: no-cache
```


Copyright © 2014 Microsoft Corporation.

```
---------------------------------Message END -----------------------------------
```

The Secure Tunnel proxy responds with the following "Access Required" message and tears down the connection gracefully:

```
---------------------------------Message START ---------------------------------
HTTP/1.1 407 ProxyAuthentication Required ( The ISA Server requires authorization to fulfill
the request. Access to the Web proxy service is denied. )
Via: 1.1 SPIRIT1B
proxy-Authenticate: Negotiate
proxy-Authenticate: Kerberos
proxy-Authenticate: NTLM
Connection: close
proxy-Connection: close
Pragma: no-cache
Cache-Control: no-cache
Content-Type: text/html
Content-Length: 701

---------------------------------Message END ---------------------------------------------
------
```

The client again connects to the Secure Tunnel proxy and sends the following message with authentication information:

```
---------------------------------Message START ---------------------------------
CONNECT server.domain.net:443 HTTP/1.0
User-Agent:Mozilla/4.0 (compatible; MSIE 5.5; Win32)
proxy-Connection: Keep-Alive
Pragma: no-cache
proxy-Authorization: NTLM
TlRMTVNTUAABAAAAt7II4gkACQAxAAAACQAJACgAAAAFASgKAAAAD0xBQlNNT0tFM1dPUktHUk9VUA==

---------------------------------Message END -----------------------------------
```

The proxy responds with the following message indicating the denied access and an authentication challenge for the client:

```
---------------------------------Message START ---------------------------------
HTTP/1.1 407 ProxyAuthentication Required ( Access is denied. )
Via: 1.1 SPIRIT1B
proxy-Authenticate: NTLM
TlRMTVNTUAACAAAAEAAQADgAAAA1goriluCDYHcYI/sAAAAAAAAAAFQAVABIAAAABQLODgAAAA9TAFAASQBSAEkAVAAxA
EIAAgAQAFMAUABJAFIASQBUADEAQgABABAAUwBQAEkAUgBJAFQAMQBCAAQAEABzAHAAaQByAGkAdAAxAGIAAwAQAHMAcA
BpAHIAaQB0ADEAYgAAAAAA
Connection: Keep-Alive
proxy-Connection: Keep-Alive
Pragma: no-cache
Cache-Control: no-cache
Content-Type: text/html
Content-Length: 0

---------------------------------Message END -----------------------------------
```


Copyright © 2014 Microsoft Corporation.

The client again requests a connection to the server and includes the response to the authentication challenge:

```
----------------------------------Message START ----------------------------------
CONNECT server.domain.net:443 HTTP/1.0
User-Agent:Mozilla/4.0 (compatible; MSIE 5.5; Win32)
proxy-Connection: Keep-Alive
Pragma: no-cache
proxy-Authorization: NTLM
TlRMTVNTUAADAAAAGAAYAHIAAAAYABgAigAAABIAEgBIAAAABgAGAFoAAAASABIAYAAAABAAEACiAAAANYKI4gUBKAoAA
AAPTABBAEIAUwBNAE8ASwBFADMAXwBxAGEATABBAEIAUwBNAE8ASwBFADMA0NKq8HYYhj8AAAAAAAAAAAAAAAAAAAAAOI
iih3mR+AkyM4r99sy1mdFonCu2ILODro1WTTrJ4b4JcXEzUBA2Ig==

----------------------------------Message END ------------------------------------
```

Upon successful proxy authentication, the Secure Tunnel proxy responds with the following message indicating successful authentication and establishment of a connection to the server:

```
----------------------------------Message START ----------------------------------
HTTP/1.1 200 Connection established
Via: 1.1 SPIRIT1B

----------------------------------Message END ------------------------------------
```

The application data can be exchanged after the NTLM authentication is finished and the Secure Tunnel proxy successfully creates the connection to the server.

Copyright © 2014 Microsoft Corporation.

5 Security

5.1 Security Considerations for Implementers

This section is meant to inform the implementers and users of the security shortcomings in the HTTP Encapsulation of SSTP Protocol. The discussion in this section includes the security considerations and suggests ways to reduce the security risks. These suggestions are not to be treated as the definitive solutions to the security issues revealed.

The HTTP Encapsulation of SSTP Protocol implementers and investigators need to take into account the Security Considerations specified in [RFC2616], section 15, [RFC1945], section 12 and [MS-GRVSSTP] section 5. This section defines the security threats that are specific to the HTTP Encapsulation of SSTP Protocol.

The HTTP Encapsulation of SSTP Protocol does not provide any authentication for the client; the consumers of this protocol have the responsibility to provide authentication. The SSTP Security Protocol [MS-GRVSSTPS] does provide initial end-to-end authentication when used with the SSTP Protocol [MS-GRVSSTP].

For HTTP Polling encapsulation, it is possible for an attacker to bypass the authentication service provided by the SSTP Security Protocol. The attacker could use the information in the HTTP headers and the Polling-Virtual-Connection-Message to generate a request and receive information from the server or insert information destined for other clients. The consumers of the SSTP Protocol need to attempt to provide protection against such threats, for example, by encrypting data or employing other security mechanisms.

5.2 Index of Security Parameters

None.

Copyright © 2014 Microsoft Corporation.

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 Office 2010 suites
- Microsoft Office Groove 2007
- Microsoft Office Groove Server 2007
- Microsoft Groove Server 2010
- Microsoft SharePoint Workspace 2010

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: The Office Groove 2007 clients support both the Secure Tunnel and SOCKS tunneling protocol, but the SharePoint Workspace 2010 clients support only the Secure Tunnel protocol. The SharePoint Workspace 2010 clients do not support the SOCKS protocol.

<2> Section 2.2.1.2.3: The Office Groove 2007 and SharePoint Workspace 2010 clients specify the following User-Agent product token value string:

"Mozilla/4.0 (compatible; MSIE 5.5; Win32)"

Implementers are advised to provide an implementation specific string as their product token value string.

<3> Section 2.2.1.2.8: The Office Groove 2007 and SharePoint Workspace 2010 implementations do not follow the common usage form for multiple cache-directives. Each cache-directive is specified using a separate Cache-Control Header. This syntax is used in place of the more common form of a single Cache-Control Header with a comma separated list of cache-directives.

<4> Section 2.2.1.3.2: The Office Groove Server 2007 relay server specifies the following Server header server product name and version token value as the HTTP Response Server header string:

"Groove-Relay/12.0"

The Groove Server 2010 relay server specifies the following Server header server product name and version token value as the HTTP Response Server header string:

"Groove-Relay/14.0"

Implementers are advised to provide an implementation-specific server product name and version string to identify their implementation.

<5> Section 2.2.2.1.1.3: The Office Groove 2007 and SharePoint Workspace 2010 implementations use 0x7ffff000 (2147479552 decimal) octets as the LongLived-Content-Length value. If

Copyright © 2014 Microsoft Corporation.

implementers choose to increase this length, they are advised to do extensive testing with a wide variety of proxies to determine if the new length is within the proxies' acceptable limits.

<6> Section 3.1: If only port 80 is allowed through the firewall, the Office Groove 2007 and SharePoint Workspace 2010 clients always use the LongLived Encapsulation Protocol. The implementer could choose to use KeepAlive or Polling Encapsulation as their preferred protocol.

<7> Section 3.1: The Office Groove 2007 and SharePoint Workspace 2010 client implementation HTTP based encapsulation protocols (LongLived, KeepAlive, Polling) use only the HTTP proxy.

<8> Section 3.1.1.2: Firewalls are generally transparent to clients. Clients need no configuration information to use a firewall, as firewalls perform routing at Layer 3 (Network Layer) of the OSI model [ISO/IEC 7498-1:1994].

Proxies are not transparent to clients. Clients need to be configured with the proxy connection information (FQDN, PORT, and Protocol) to be able to establish a connection with a proxy. The Office Groove 2007 and SharePoint Workspace 2010 clients use the proxy configuration information from a browser. The Office Groove 2007 and SharePoint Workspace 2010 clients can also access proxy auth-configuration or PAC files to get this information. In this case all that is needed is the proxy FQDN and Port Number. Therefore Office Groove 2007 and SharePoint Workspace 2010 clients do not actually store the proxy configuration persistently.

<9> Section 3.1.2.3: The Office Groove 2007 and SharePoint Workspace 2010 implementations of SSTP [MS-GRVSSTP] assists the HTTP encapsulation protocols by implementing KeepAlive semantics. SSTP's KeepAlive semantics are implemented using an SSTP_NOOP command. SSTP_NOOP commands are used to send SSTP ACKs over SSTP Connections. If there are no SSTP ACKs to send, SSTP sends an SSTP_NOOP command with a ACK count of 0. SSTP_NOOPs are sent at 45 second intervals. The SSTP default KeepAlive value of 5 minutes is overridden when SSTP is encapsulated. If HTTP Encapsulation of SSTP protocols are used to encapsulate non-SSTP data, then these non-SSTP protocols need to implement their own KeepAlive mechanisms, as the HTTP encapsulation protocols provide no KeepAlive semantics of their own. KeepAlive requests are important to HTTP encapsulation protocols used with proxy connections. Proxies can implement various timers to close idle proxy connections. Some firewalls and proxy implementations do not distinguish between proxy and non-proxy connections. Therefore the recommended behavior is that encapsulated protocols always send a KeepAlive message when used with HTTP Encapsulation.

<10> Section 3.1.2.3: The Office Groove 2007 and SharePoint Workspace 2010 clients' HTTP Encapsulation implementation adjusts the SSTP protocol layer implementation's default KeepAlive timer value. The HTTP Encapsulation layer overrides the SSTP default KeepAlive timer value, changing the default value of 5 minutes to 45 seconds. The default SSTP KeepAlive timer is modified to increase the frequency of the KeepAlive messages to help ensure that proxies do not treat the connections as idle and close them.

<11> Section 3.1.4.1: The Office Groove 2007 and SharePoint Workspace 2010 clients' connect sequence without proxies configured is as follows. The client attempts to create a direct connection using the following protocols in the specified order. If any attempt succeeds, the direct connection to the target server is established. If all connection attempts fail, then the connection attempt to the target server fails. The Office Groove 2007 and SharePoint Workspace 2010 clients attempt connections in the following order:

1. SSTP 2492/TCP connecting

2. SSTP 443/TCP connecting

3. HTTP LongLived Encapsulation 80/TCP connecting

Copyright © 2014 Microsoft Corporation.

The Office Groove 2007 and SharePoint Workspace 2010 clients' connect sequence with proxies configured is as follows. The client attempts to create a proxy connection using the following protocols in the specified order. If any attempt succeeds, the connection to the target server is established. If all connection attempts fail, then the connection attempt to the target server fails. The Office Groove 2007 and SharePoint Workspace 2010 clients attempt connections in the following order:

1. Secure Tunnel Encapsulation 443/TCP connecting
2. SOCKS Encapsulation 1080/TCP connecting (Office Groove 2007 clients only; SharePoint Workspace 2010 clients do not support SOCKS Encapsulation.)
3. HTTP LongLived Encapsulation 80/TCP connecting
4. HTTP KeepAlive Encapsulation 80/TCP connecting
5. HTTP Polling Encapsulation 80/TCP connecting

<12> Section 3.1.4.1.2: The Office Groove 2007 and SharePoint Workspace 2010 clients support the following HTTP Proxy Authentication schemes: **basic authentication scheme** and NTLM.

<13> Section 3.1.4.3: The Office Groove 2007 and SharePoint Workspace 2010 clients behave differently than recommended when the LongLived Encapsulation GET session maximum content length limit is reached at the server. The behavior works as expected when the client reaches the LongLived-Content-Length limit on the POST session. Following is the behavior when the server reaches the LongLived-Content-Length limit:

Recommended behavior:

The server closes the GET sessions. The client then detects the close connection request from the server and closes the virtual LongLived connection. The client then establishes a new LongLived virtual connection using a new Virtual-Connection-GUID. The client and server then begin sending and receiving data using the new LongLived connection.

Actual behavior:

The server closes the GET session. The client ignores the TCP disconnect request.

The GET session data flow stops. POST session traffic continues until the encapsulated protocol (SSTP) blocks waiting for GET session messages, at which point the LongLived connection hangs. The connection eventually times out and disconnects. The KeepAlive timer eventually causes the virtual connection to close. The client then establishes a new LongLived virtual connection using a new Virtual-Connection-GUID and starts to exchange SSTP commands.

<14> Section 3.1.6.3: The Office Groove 2007 and SharePoint Workspace 2010 implementations of SSTP [MS-GRVSSTP] assists the HTTP encapsulation protocols by implementing KeepAlive semantics. SSTP's KeepAlive semantics are implemented using an SSTP_NOOP command. SSTP_NOOP commands are used to send SSTP ACKs over SSTP Connections. If there are no SSTP ACKs to send, SSTP sends an SSTP_NOOP command with a ACK count of 0. SSTP_NOOPs are sent at 45 second intervals. The SSTP default KeepAlive value of 5 minutes is overridden when SSTP is encapsulated. If HTTP Encapsulation of SSTP protocols is used to encapsulate non-SSTP data, then these non-SSTP protocols need to implement their own KeepAlive mechanisms, as the HTTP encapsulation protocols provide no KeepAlive semantics of their own. KeepAlive requests are important to HTTP encapsulation protocols used with proxy connections. Proxies can implement various timers to close idle proxy connections. Some firewalls and proxy implementations do not distinguish between proxy and non-proxy connections. Therefore the recommend behavior is that encapsulated protocols always send a KeepAlive message when used with HTTP Encapsulation.

Copyright © 2014 Microsoft Corporation.

<15> Section 3.2.2.3: The Office Groove Server 2007 and Groove Server 2010 implementations of SSTP [MS-GRVSSTP] assist the HTTP encapsulation protocols by implementing KeepAlive semantics. SSTP's KeepAlive semantics are implemented using an SSTP_NOOP command. SSTP_NOOP commands are used to send SSTP ACKs over SSTP Connections. If there are no SSTP ACKs to send, SSTP sends an SSTP_NOOP command with a ACK count of 0. SSTP_NOOPs are sent at 45 second intervals. The SSTP default KeepAlive value of 5 minutes is overridden when SSTP is encapsulated. If HTTP Encapsulation of SSTP protocols is used to encapsulate non-SSTP data, then these non-SSTP protocols need to implement their own KeepAlive mechanisms, as the HTTP encapsulation protocols provide no KeepAlive semantics of their own. KeepAlive requests are important to HTTP encapsulation protocols used with proxy connections. Proxies can implement various timers to close idle proxy connections. Some firewalls and proxy implementations do not distinguish between proxy and non-proxy connections. Therefore the recommended behavior is that encapsulated protocols always send a KeepAlive message when used with HTTP Encapsulation.

<16> Section 3.2.3.1: The Office Groove Server 2007 and Groove Server 2010 servers use a special purpose HTTP 1.0 protocol stack. Except for the subset of HTTP Request-Headers and Response-Headers specified in this document, all other HTTP Headers will be ignored as specified in the HTTP 1.0 [RFC1945] sections 5.2, 6.2 and 7.1.

<17> Section 3.2.4.2: The Office Groove Server 2007 and Groove Server 2010 servers behave differently than recommended when the LongLived Encapsulation GET session maximum content length limit is reached at the server. The behavior works as expected when the client reaches the LongLived-Content-Length limit on the POST session. Following is the behavior when the server reaches the LongLived-Content-Length limit:

Recommended behavior:

The server closes the GET sessions. The client then detects the close connection request from the server and close the virtual LongLived connection. The client then establishes a new LongLived virtual connection using a new Virtual-Connection-GUID. The client and server then begin sending and receiving data using the new LongLived connection.

Actual behavior:

The server closes the GET session. The client ignores the TCP disconnect request.

The GET session data flow stops. POST session traffic continues until the encapsulated protocol (SSTP) blocks waiting for GET session messages, at which point the LongLived connection hangs. The connection eventually times out and disconnects. The KeepAlive timer eventually causes the virtual connection to close. The client then establishes a new LongLived virtual connection using a new Virtual-Connection-GUID and starts to exchange SSTP commands.

<18> Section 3.2.5.1.1: Office Groove Server 2007 and Groove Server 2010 do not check the protocol version of the encapsulation protocols.

<19> Section 3.2.5.1.1: The Office Groove Server 2007 and Groove Server 2010 relay servers do not validate that the URI contains a Relay-Server-Name that equals the local server name. The implementer is recommended to validate this field to ensure that the message was routed to the intended server.

<20> Section 3.2.5.2.1: Office Groove Server 2007 and Groove Server 2010 do not check the protocol version of the encapsulation protocols.

<21> Section 3.2.5.2.1: The Office Groove Server 2007 and Groove Server 2010 relay servers do not validate that the URI contains a Relay-Server-Name that equals the local server name. The implementer is recommended to validate this field to ensure that the message was routed to the intended server.

Copyright © 2014 Microsoft Corporation.

<22> Section 3.2.5.2.2: Office Groove Server 2007 and Groove Server 2010 use an implementation defined internal buffer size of 32768 octets.

<23> Section 3.2.6.3: The Office Groove Server 2007 and Groove Server 2010 implementations of SSTP [MS-GRVSSTP] assist the HTTP encapsulation protocols by implementing KeepAlive semantics. SSTP's KeepAlive semantics are implemented using an SSTP_NOOP command. SSTP_NOOP commands are used to send SSTP ACKs over SSTP Connections. If there are no SSTP ACKs to send, SSTP sends an SSTP_NOOP command with a ACK count of 0. SSTP_NOOPs are sent at 45 second intervals. The SSTP default KeepAlive value of 5 minutes is overridden when SSTP is encapsulated. If HTTP Encapsulation of SSTP protocols is used to encapsulate non-SSTP data, then these non-SSTP protocols need to implement their own KeepAlive mechanisms, as the HTTP encapsulation protocols provide no KeepAlive semantics of their own. KeepAlive requests are important to HTTP encapsulation protocols used with proxy connections. Proxies can implement various timers to close idle proxy connections. Some firewalls and proxy implementations do not distinguish between proxy and non-proxy connections. Therefore the recommended behavior is that encapsulated protocols always send a KeepAlive message when used with HTTP Encapsulation.

<24> Section 3.3.1.2: Firewalls are generally transparent to clients. Clients need no configuration information to use a firewall, as firewalls perform routing at Layer 3 (Network Layer) of the OSI model [ISO/IEC 7498-1:1994].

Proxies are not transparent to clients. Clients need to be configured with the proxy connection information (FQDN, PORT, Protocol) to be able to establish a connection with a proxy. The Office Groove 2007 and SharePoint Workspace 2010 clients use the proxy configuration information from a browser. The Office Groove 2007 and SharePoint Workspace 2010 clients can also access proxy auth-configuration or PAC files to get this information. In this case all that is needed is the proxy FQDN and Port Number. Therefore the Office Groove 2007 and SharePoint Workspace 2010 clients do not actually store the proxy configuration persistently.

<25> Section 3.3.2.4: The Office Groove 2007 and SharePoint Workspace 2010 implementations of SSTP [MS-GRVSSTP] assist the HTTP encapsulation protocols by implementing KeepAlive semantics. SSTP's KeepAlive semantics are implemented using an SSTP_NOOP command. SSTP_NOOP commands are used to send SSTP ACKs over SSTP Connections. If there are no SSTP ACKs to send, SSTP sends an SSTP_NOOP command with a ACK count of 0. SSTP_NOOPs are sent at 45 second intervals. The SSTP default KeepAlive value of 5 minutes is overridden when SSTP is encapsulated. If HTTP Encapsulation of SSTP protocols is used to encapsulate non-SSTP data, then these non-SSTP protocols need to implement their own KeepAlive mechanisms, as the HTTP encapsulation protocols provide no KeepAlive semantics of their own. KeepAlive requests are important to HTTP encapsulation protocols used with proxy connections. Proxies can implement various timers to close idle proxy connections. Some firewalls and proxy implementations do not distinguish between proxy and non-proxy connections. Therefore the recommend behavior is that encapsulated protocols always send a KeepAlive message when used with HTTP Encapsulation.

<26> Section 3.3.4.1: Firewalls are generally transparent to clients. Clients need no configuration information to use a firewall, as firewalls perform routing at Layer 3 (Network Layer) of the OSI model [ISO/IEC 7498-1:1994].

Proxies are not transparent to clients. Clients need to be configured with the proxy connection information (FQDN, PORT, Protocol) to be able to establish a connection with a proxy. The Office Groove 2007 and SharePoint Workspace 2010 clients use the proxy configuration information from a browser. The Office Groove 2007 and SharePoint Workspace 2010 clients can also access proxy auth-configuration or PAC files to get this information. In this case all that is needed is the proxy FQDN and Port Number. Therefore the Office Groove 2007 and SharePoint Workspace 2010 clients do not actually store the proxy configuration persistently.

Copyright © 2014 Microsoft Corporation.

<27> Section 3.3.6.4: The Office Groove 2007 and SharePoint Workspace 2010 implementations of SSTP, as specified in [MS-GRVSSTP], assist the HTTP encapsulation protocols by implementing KeepAlive semantics. SSTP's KeepAlive semantics are implemented using an SSTP_NOOP command. SSTP_NOOP commands are used to send SSTP ACKs over SSTP Connections. If there are no SSTP ACKs to send, SSTP sends an SSTP_NOOP command with a ACK count of 0. SSTP_NOOPs are sent at 45 second intervals. The SSTP default KeepAlive value of 5 minutes is overridden when SSTP is encapsulated. If HTTP Encapsulation of SSTP protocols is used to encapsulate non-SSTP data, then these non-SSTP protocols need to implement their own KeepAlive mechanisms, as the HTTP encapsulation protocols provide no KeepAlive semantics of their own. KeepAlive requests are important to HTTP encapsulation protocols used with proxy connections. Proxies can implement various timers to close idle proxy connections. Some firewalls and proxy implementations do not distinguish between proxy and non-proxy connections. Therefore the recommend behavior is that encapsulated protocols always send a KeepAlive message when used with HTTP Encapsulation.

<28> Section 3.4.2.3: The Office Groove Server 2007 and Groove Server 2010 implementations of SSTP, as specified in [MS-GRVSSTP], assist the HTTP encapsulation protocols by implementing KeepAlive semantics. SSTP's KeepAlive semantics are implemented using an SSTP_NOOP command. SSTP_NOOP commands are used to send SSTP ACKs over SSTP Connections. If there are no SSTP ACKs to send, SSTP sends an SSTP_NOOP command with a ACK count of 0. SSTP_NOOPs are sent at 45 second intervals. The SSTP default KeepAlive value of 5 minutes is overridden when SSTP is encapsulated. If HTTP Encapsulation of SSTP protocols is used to encapsulate non-SSTP data, then these non-SSTP protocols need to implement their own KeepAlive mechanisms, as the HTTP encapsulation protocols provide no KeepAlive semantics of their own. KeepAlive requests are important to HTTP encapsulation protocols used with proxy connections. Proxies can implement various timers to close idle proxy connections. Some firewalls and proxy implementations do not distinguish between proxy and non-proxy connections. Therefore the recommended behavior is that encapsulated protocols always send a KeepAlive message when used with HTTP Encapsulation.

<29> Section 3.4.3.1: The Office Groove Server 2007 and Groove Server 2010 servers use a special purpose HTTP 1.0 protocol stack. Except for the subset of HTTP Request-Headers and Response-Headers specified in this document, all other HTTP Headers will be ignored as specified in the HTTP 1.0 [RFC1945] sections 5.2, 6.2 and 7.1.

<30> Section 3.4.5.1.1: Office Groove Server 2007 and Groove Server 2010 do not check the protocol version of the encapsulation protocols.

<31> Section 3.4.5.1.1: The Office Groove Server 2007 and Groove Server 2010 relay servers do not validate that the URI contains a Relay-Server-Name that equals the local server name. The implementer is recommended to validate this field to ensure that the message was routed to the intended server.

<32> Section 3.4.5.1.1.2: Office Groove Server 2007 and Groove Server 2010 do not check the protocol version of the encapsulation protocols.

<33> Section 3.4.5.1.2: Office Groove Server 2007 and Groove Server 2010 do not check the protocol version of the encapsulation protocols.

<34> Section 3.4.5.1.2: The Office Groove Server 2007 and Groove Server 2010 relay servers do not validate that the URI contains a Relay-Server-Name that equals the local server name. The implementer is recommended to validate this field to ensure that the message was routed to the intended server.

<35> Section 3.4.5.2: Office Groove Server 2007 and Groove Server 2010 do not check the protocol version of the encapsulation protocols.

Copyright © 2014 Microsoft Corporation.

<36> Section 3.4.5.2: The Office Groove Server 2007 and Groove Server 2010 relay servers do not validate that the URI contains a Relay-Server-Name that equals the local server name. The implementer is recommended to validate this field to ensure that the message was routed to the intended server.

<37> Section 3.4.6.3: The Office Groove Server 2007 and Groove Server 2010 implementations of SSTP [MS-GRVSSTP] assist the HTTP encapsulation protocols by implementing KeepAlive semantics. SSTP's KeepAlive semantics are implemented using an SSTP_NOOP command. SSTP_NOOP commands are used to send SSTP ACKs over SSTP Connections. If there are no SSTP ACKs to send, SSTP sends an SSTP_NOOP command with a ACK count of 0. SSTP_NOOPs are sent at 45 second intervals. The SSTP default KeepAlive value of 5 minutes is overridden when SSTP is encapsulated. If HTTP Encapsulation of SSTP protocols is used to encapsulate non-SSTP data, then these non-SSTP protocols need to implement their own KeepAlive mechanisms, as the HTTP encapsulation protocols provide no KeepAlive semantics of their own. KeepAlive requests are important to HTTP encapsulation protocols used with proxy connections. Proxies can implement various timers to close idle proxy connections. Some firewalls and proxy implementations do not distinguish between proxy and non-proxy connections. Therefore the recommend behavior is that encapsulated protocols always send a KeepAlive message when used with HTTP Encapsulation.

<38> Section 3.5.1.1: To interoperate with Office Groove 2007 and SharePoint Workspace 2010, clients and servers need to support the Polling Connection idle connection back off algorithm. This value is sent by servers to clients on every POST response. The recommended behavior is that client implementations refresh the idle connection back off values on a per request/response basis.

<39> Section 3.5.1.1: To interoperate with Office Groove 2007 and SharePoint Workspace 2010, clients and servers need to support the Polling Connection idle connection back off algorithm. This value is sent by servers to clients on every POST response. The recommended behavior is that client implementations refresh the idle connection back off values on a per request/response basis.

<40> Section 3.5.1.1: To interoperate with Office Groove 2007 and SharePoint Workspace 2010, clients and servers need to support the Polling Connection idle connection back off algorithm. This value is sent by servers to clients on every POST response. The recommended behavior is that client implementations refresh the idle connection back off values on a per request/response basis.

<41> Section 3.5.1.2: Firewalls are generally transparent to clients. Clients need no configuration information to use a firewall, as firewalls perform routing at Layer 3 (Network Layer) of the OSI model, as specified in [ISO/IEC 7498-1:1994].

Proxies are not transparent to clients. Clients need to be configured with the proxy connection information (FQDN, PORT, Protocol) to be able to establish a connection with a proxy. The Office Groove 2007 and SharePoint Workspace 2010 clients use the proxy configuration information from a browser. The Office Groove 2007 and SharePoint Workspace 2010 clients can also access proxy auth-configuration or PAC files to get this information. In this case all that is needed is the proxy FQDN and Port Number. Therefore, the Office Groove 2007 and SharePoint Workspace 2010 clients do not actually store the proxy configuration persistently.

<42> Section 3.5.2.3: The default Polling-Virtual-Connection-Response-Message values sent by the server to the client on a Polling-POST-Response were empirically derived using many firewall and proxy vendor implementations.

In the Office Groove Server 2007 and Groove Server 2010 implementations, the default Poll Timer values used by Polling connections for polling servers for application data are:

Default server specified MaxPollInterval value is 120 seconds

Default server specified MinPollInterval value is 5 seconds

Copyright © 2014 Microsoft Corporation.

Default server specified PollRepetitions value is 3 iterations

The maximum MaxPollInterval value's upper limit is determined by limits imposed by firewall and proxies on the maximum idle time for connections. Once the poll interval exceeds a proxy's maximum idle timer value, the connection will be automatically closed by the proxy.

<43> Section 3.5.4.1: Firewalls are generally transparent to clients. Clients need no configuration information to use a firewall, as firewalls perform routing at Layer 3 (Network Layer) of the OSI model [ISO/IEC 7498-1:1994].

Proxies are not transparent to clients. Clients need to be configured with the proxy connection information (FQDN, PORT, Protocol) to be able to establish a connection with a proxy. The Office Groove 2007 and SharePoint Workspace 2010 clients use the proxy configuration information from a browser. The Office Groove 2007 and SharePoint Workspace 2010 clients can also access proxy auth-configuration or PAC files to get this information. In this case all that is needed is the proxy FQDN and Port Number. Therefore the Office Groove 2007 and SharePoint Workspace 2010 clients do not actually store the proxy configuration persistently.

<44> Section 3.5.5.1.1.1: Office Groove 2007 and SharePoint Workspace 2010 do not check the protocol version of the encapsulation protocols.

<45> Section 3.5.5.1.1.1: The Office Groove Server 2007 and Groove Server 2010 relay servers do not validate that the URI contains a Relay-Server-Name that equals the local server name. The implementer is recommended to validate this field to ensure that the message was routed to the intended server.

<46> Section 3.5.5.1.1.2: Office Groove 2007 and SharePoint Workspace 2010 do not check the protocol version of the encapsulation protocols.

<47> Section 3.5.5.1.1.2: The Office Groove Server 2007 and Groove Server 2010 relay servers do not validate that the URI contains a Relay-Server-Name that equals the local server name. The implementer is recommended to validate this field to ensure that the message was routed to the intended server.

<48> Section 3.6.3.1: The Office Groove Server 2007 and Groove Server 2010 servers use a special purpose HTTP 1.0 protocol stack. Except for the subset of HTTP Request-Headers and Response-Headers specified in this document, all other HTTP Headers will be ignored as specified in the HTTP 1.0 [RFC1945] sections 5.2, 6.2 and 7.1.

<49> Section 3.6.5: Office Groove Server 2007 and Groove Server 2010 do not check the protocol version of the encapsulation protocols.

<50> Section 3.6.5: The Office Groove Server 2007 and Groove Server 2010 relay servers do not validate that the URI contains a Relay-Server-Name that equals the local server name. The implementer is recommended to validate this field to ensure that the message was routed to the intended server.

<51> Section 3.6.5.1: Office Groove Server 2007 and Groove Server 2010 do not check the Sequence Number on the Initial Handshake Request of the Polling Encapsulation protocol.

<52> Section 3.6.5.2.1: The Office Groove Server 2007 and Groove Server 2010 relay servers do not use load balancing algorithms to control the client's poll timer interval (see section 3.5.2.3). The PollingMinRepetitionInterval, PollingMinRepetitionInterval and PollingRepetitionCount state variable values are set during application initialization. The default values are specified in <21>.

Copyright © 2014 Microsoft Corporation.

<53> Section 3.7.1.2: Firewalls are generally transparent to clients. Clients need no configuration information to use a firewall, as firewalls perform routing at Layer 3 (Network Layer) of the OSI model, as specified in [ISO/IEC 7498-1:1994].

Proxies are not transparent to clients. Clients need to be configured with the proxy connection information (FQDN, PORT, Protocol) to be able to establish a connection with a proxy. The Office Groove 2007 and SharePoint Workspace 2010 clients use the proxy configuration information from a browser. The Office Groove 2007 and SharePoint Workspace 2010 clients can also access proxy auth-configuration or PAC files to get this information. In this case all that is needed is the proxy FQDN and Port Number. Therefore the Office Groove 2007 and SharePoint Workspace 2010 clients do not actually store the proxy configuration persistently.

<54> Section 3.7.2.3: The Office Groove 2007 and SharePoint Workspace 2010 implementations of SSTP, as specified in [MS-GRVSSTP], assist the HTTP encapsulation protocols by implementing KeepAlive semantics. SSTP's KeepAlive semantics are implemented using an SSTP_NOOP command. SSTP_NOOP commands are used to send SSTP ACKs over SSTP Connections. If there are no SSTP ACKs to send, SSTP sends an SSTP_NOOP command with a ACK count of 0. SSTP_NOOPs are sent at 45 second intervals. The SSTP default KeepAlive value of 5 minutes is overridden when SSTP is encapsulated. If HTTP Encapsulation of SSTP protocols is used to encapsulate non-SSTP data, then these non-SSTP protocols need to implement their own KeepAlive mechanisms, as the HTTP encapsulation protocols provide no KeepAlive semantics of their own. KeepAlive requests are important to HTTP encapsulation protocols used with proxy connections. Proxies can implement various timers to close idle proxy connections. Some firewalls and proxy implementations do not distinguish between proxy and non-proxy connections. Therefore the recommended behavior is that encapsulated protocols always send a KeepAlive message when used with HTTP Encapsulation.

<55> Section 3.7.4.1: Firewalls are generally transparent to clients. Clients need no configuration information to use a firewall, as firewalls perform routing at Layer 3 (Network Layer) of the OSI model [ISO/IEC 7498-1:1994].

Proxies are not transparent to clients. Clients need to be configured with the proxy connection information (FQDN, PORT, Protocol) to be able to establish a connection with a proxy. The Office Groove 2007 and SharePoint Workspace 2010 clients use the proxy configuration information from a browser. The Office Groove 2007 and SharePoint Workspace 2010 clients can also access proxy auth-configuration or PAC files to get this information. In this case all that is needed is the proxy FQDN and Port Number. Therefore the Office Groove 2007 and SharePoint Workspace 2010 clients do not actually store the proxy configuration persistently.

<56> Section 3.7.6.3: The Office Groove 2007 and SharePoint Workspace 2010 implementations of SSTP, as specified in [MS-GRVSSTP], assist the HTTP encapsulation protocols by implementing KeepAlive semantics. SSTP's KeepAlive semantics are implemented using an SSTP_NOOP command. SSTP_NOOP commands are used to send SSTP ACKs over SSTP Connections. If there are no SSTP ACKs to send, SSTP sends an SSTP_NOOP command with a ACK count of 0. SSTP_NOOPs are sent at 45 second intervals. The SSTP default KeepAlive value of 5 minutes is overridden when SSTP is encapsulated. If HTTP Encapsulation of SSTP protocols is used to encapsulate non-SSTP data, then these non-SSTP protocols need to implement their own KeepAlive mechanisms, as the HTTP encapsulation protocols provide no KeepAlive semantics of their own. KeepAlive requests are important to HTTP encapsulation protocols used with proxy connections. Proxies can implement various timers to close idle proxy connections. Some firewalls and proxy implementations do not distinguish between proxy and non-proxy connections. Therefore the recommend behavior is that encapsulated protocols always send a KeepAlive message when used with HTTP Encapsulation.

<57> Section 3.8.2.1: The Office Groove Server 2007 and Groove Server 2010 implementations of SSTP, as specified in [MS-GRVSSTP], assist the HTTP encapsulation protocols by implementing KeepAlive semantics. SSTP's KeepAlive semantics are implemented using an SSTP_NOOP command. SSTP_NOOP commands are used to send SSTP ACKs over SSTP Connections. If there are no SSTP

Copyright © 2014 Microsoft Corporation.

ACKs to send, SSTP sends an SSTP_NOOP command with a ACK count of 0. SSTP_NOOPs are sent at 45 second intervals. The SSTP default KeepAlive value of 5 minutes is overridden when SSTP is encapsulated. If HTTP Encapsulation of SSTP protocols is used to encapsulate non-SSTP data, then these non-SSTP protocols need to implement their own KeepAlive mechanisms, as the HTTP encapsulation protocols provide no KeepAlive semantics of their own. KeepAlive requests are important to HTTP encapsulation protocols used with proxy connections. Proxies can implement various timers to close idle proxy connections. Some firewalls and proxy implementations do not distinguish between proxy and non-proxy connections. Therefore, the recommend behavior is that encapsulated protocols always send a KeepAlive message when used with HTTP Encapsulation.

<58> Section 3.8.2.1: The Office Groove 2007 and SharePoint Workspace 2010 clients' HTTP Encapsulation implementation adjusts the SSTP protocol layer implementation's default KeepAlive timer value. The HTTP Encapsulation layer overrides the SSTP default KeepAlive timer value, changing the default value of 5 minutes to 45 seconds. The default SSTP KeepAlive timer is modified to increase the frequency of the KeepAlive messages to help ensure that proxies do not treat the connections as idle and close them.

<59> Section 3.9.1.2: This proxy configuration information applies to Office Groove 2007 and SharePoint Workspace 2010.

<60> Section 3.9.1.2: Firewalls are generally transparent to clients. Clients need no configuration information to use a firewall, as firewalls perform routing at Layer 3 (Network Layer) of the OSI model [ISO/IEC 7498-1:1994].

Proxies are not transparent to clients. Clients need to be configured with the proxy connection information (FQDN, PORT, Protocol) to be able to establish a connection with a proxy. The Office Groove 2007 and SharePoint Workspace 2010 clients use the proxy configuration information from a browser. The Office Groove 2007 and SharePoint Workspace 2010 clients can also access proxy auth-configuration or PAC files to get this information. In this case all that is needed is the proxy FQDN and Port Number. Therefore the Office Groove 2007 and SharePoint Workspace 2010 clients do not actually store the proxy configuration persistently.

<61> Section 3.9.2.3: The Office Groove 2007 and SharePoint Workspace 2010 implementations of SSTP as specified in [MS-GRVSSTP], assist the HTTP encapsulation protocols by implementing KeepAlive semantics. SSTP's KeepAlive semantics are implemented using an SSTP_NOOP command. SSTP_NOOP commands are used to send SSTP ACKs over SSTP Connections. If there are no SSTP ACKs to send, SSTP sends an SSTP_NOOP command with a ACK count of 0. SSTP_NOOPs are sent at 45 second intervals. The SSTP default KeepAlive value of 5 minutes is overridden when SSTP is encapsulated. If HTTP Encapsulation of SSTP protocols is used to encapsulate non-SSTP data, then these non-SSTP protocols need to implement their own KeepAlive mechanisms, as the HTTP encapsulation protocols provide no KeepAlive semantics of their own. KeepAlive requests are important to HTTP encapsulation protocols used with proxy connections. Proxies can implement various timers to close idle proxy connections. Some firewalls and proxy implementations do not distinguish between proxy and non-proxy connections. Therefore the recommend behavior is that encapsulated protocols always send a KeepAlive message when used with HTTP Encapsulation.

<62> Section 3.9.4.1: Firewalls are generally transparent to clients. Clients need no configuration information to use a firewall, as firewalls perform routing at Layer 3 (Network Layer) of the OSI model [ISO/IEC 7498-1:1994].

Proxies are not transparent to clients. Clients need to be configured with the proxy connection information (FQDN, PORT, Protocol) to be able to establish a connection with a proxy. The Office Groove 2007 and SharePoint Workspace 2010 clients use the proxy configuration information from a browser. The Office Groove 2007 and SharePoint Workspace 2010 clients can also access proxy auth-configuration or PAC files to get this information. In this case all that is needed is the proxy

Copyright © 2014 Microsoft Corporation.

FQDN and Port Number. Therefore the Office Groove 2007 and SharePoint Workspace 2010 clients do not actually store the proxy configuration persistently.

<63> Section 3.9.4.1.1: The Office Groove 2007 and SharePoint Workspace 2010 clients support two SOCKS authentication methods: 0x00 (NO AUTHENTICATION REQUIRED) and 0x02 (USERNAME/PASSWORD). These two methods are represented by a sequence of two hex octets: [00 02]. The SharePoint Workspace 2010 client will not support the SOCKS protocol.

<64> Section 3.9.4.1.1: The Office Groove 2007 and SharePoint Workspace 2010 clients do not support GSSAPI.

<65> Section 3.9.5.1.1: The Office Groove 2007 and SharePoint Workspace 2010 clients do not support GSSAPI.

<66> Section 3.9.6.3: The Office Groove 2007 and SharePoint Workspace 2010 implementations of SSTP, as specified in [MS-GRVSSTP], assist the HTTP encapsulation protocols by implementing KeepAlive semantics. SSTP's KeepAlive semantics are implemented using an SSTP_NOOP command. SSTP_NOOP commands are used to send SSTP ACKs over SSTP Connections. If there are no SSTP ACKs to send, SSTP sends an SSTP_NOOP command with a ACK count of 0. SSTP_NOOPs are sent at 45 second intervals. The SSTP default KeepAlive value of 5 minutes is overridden when SSTP is encapsulated. If HTTP Encapsulation of SSTP protocols is used to encapsulate non-SSTP data, then these non-SSTP protocols need to implement their own KeepAlive mechanisms, as the HTTP encapsulation protocols provide no KeepAlive semantics of their own. KeepAlive requests are important to HTTP encapsulation protocols used with proxy connections. Proxies can implement various timers to close idle proxy connections. Some firewalls and proxy implementations do not distinguish between proxy and non-proxy connections. Therefore, the recommend behavior is that encapsulated protocols always send a KeepAlive message when used with HTTP Encapsulation.

Copyright © 2014 Microsoft Corporation.

7 Change Tracking

No table of changes is available. The document is either new or has had no changes since its last release.

Copyright © 2014 Microsoft Corporation.

8 Index

A

Abstract data model
client (section 3.1.1 55, section 3.3.1 71, section 3.5.1 94, section 3.7.1 112, section 3.9.1 119)
KeepAlive encapsulation protocol client 71
KeepAlive encapsulation protocol server 86
LongLived encapsulation protocol client 55
LongLived encapsulation protocol server 65
Polling encapsulation protocol client 94
Polling encapsulation protocol server 107
Secure Tunnel encapsulation protocol client 112
server (section 3.2.1 65, section 3.4.1 86, section 3.6.1 107, section 3.8.1 118, section 3.10.1 125)
SOCKS encapsulation of SSTP protocol client 119
Applicability 28

C

Capability negotiation 28
Change tracking 159
Client
abstract data model (section 3.1.1 55, section 3.3.1 71, section 3.5.1 94, section 3.7.1 112, section 3.9.1 119)
message processing (section 3.1.5 62, section 3.7.5 116, section 3.9.5 123)
other local events (section 3.1.7 65, section 3.3.7 84, section 3.5.7 107, section 3.7.7 118, section 3.9.7 125)
overview (section 3.1 55, section 3.3 71, section 3.5 94, section 3.7 112, section 3.9 119)
sequencing rules (section 3.1.5 62, section 3.7.5 116, section 3.9.5 123)
Client - KeepAlive encapsulation protocol
closing a KeepAlive connection 77
closing a KeepAlive GET session 78
closing a KeepAlive POST session 77
connection state information 73
ConnectionEstablishment timer event 84
establishing a KeepAlive encapsulation connection 76
GetNetworkReceiveIO timer event 84
KeepAlive timer event 84
KeepAlive-GET-Response processing 81
KeepAlive-POST-Response processing 79
overview 71
PostNetworkReceiveIO timer event 84
proxy state information 74
re-opening a KeepAlive GET session 78
re-opening a KeepAlive POST session 78
sending a KeepAlive-GET-Request 83
sending application data 78
Client - KeepAlive encapsulation protocol - abstract data model 71
Client - KeepAlive encapsulation protocol - initialization 75
Client - KeepAlive encapsulation protocol - local events 84
re-opening the POST session after a transport disconnect 84
Client - LongLived encapsulation protocol
closing a LongLived connection 61
connection state information 57
ConnectionEstablishment timer (section 3.1.2.1 59, section 3.3.2.1 75)
ConnectionEstablishment timer event 64
establishing a LongLived encapsulation connection 59
GetNetworkReceiveIO timer 75
KeepAlive timer (section 3.1.2.3 59, section 3.3.2.4 75)
KeepAlive timer event 65
NetworkReceiveIO timer 59
NetworkReceiveIO timer event 65
overview 55
PostNetworkReceiveIO timer 75
proxy state information 58
receiving data on the GET session 63
receiving data on the POST session 62
sending application data 62
Client - LongLived encapsulation protocol - abstract data model 55
Client - LongLived encapsulation protocol - initialization 59
Client - LongLived encapsulation protocol - local events 65
Client - Polling encapsulation protocol
client state information 98
closing a Polling connection 101
connection state information 96
ConnectionEstablishment timer 98
ConnectionEstablishment timer event 106
establishing a Polling encapsulation connection 100
NetworkReceiveIO timer 98
NetworkReceiveIO timer event 107
overview 94
polling encapsulation timer 99
Polling encapsulation timer event 107
Polling-POST-Response processing 103
proxy state information 98
sending application data 101
Client - Polling encapsulation protocol - abstract data model 94
Client - Polling encapsulation protocol - initialization 99
Client - Polling encapsulation protocol - local events 107
Client - Secure Tunnel encapsulation protocol
closing a Secure Tunnel connection 116
connection state information 113
ConnectionEstablishment timer 114
ConnectionEstablishment timer event 118
establishing a Secure Tunnel encapsulation connection 115

Copyright © 2014 Microsoft Corporation.

KeepAlive timer 115
KeepAlive timer event 118
NetworkReceiveIO timer 114
NetworkReceiveIO timer event 118
overview 112
proxy state information 114
sending application data 116
Client - Secure Tunnel encapsulation protocol - abstract data model 112
Client - Secure Tunnel encapsulation protocol - initialization 115
Secure Tunnel listener endpoints 115
timers started 115
Client - Secure Tunnel encapsulation protocol - local events 118
Client - Secure Tunnel encapsulation protocol - message processing 116
Client - Secure Tunnel encapsulation protocol - sequencing rules 116
Client - SOCKS encapsulation of SSTP protocol
closing a SOCKS connection 122
connection state information 120
ConnectionEstablishment timer 121
ConnectionEstablishment timer event 124
establishing a SOCKS encapsulation connection 122
KeepAlive timer 121
KeepAlive timer event 125
NetworkReceiveIO timer 121
NetworkReceiveIO timer event 125
proxy state information 121
sending application data 123
Client - SOCKS encapsulation of SSTP protocol - abstract data model 119
Client - SOCKS encapsulation of SSTP protocol - initialization 122
Client - SOCKS encapsulation of SSTP protocol - local events 125
Client - SOCKS encapsulation of SSTP protocol - message processing 123
Client - SOCKS encapsulation of SSTP protocol - sequencing rules 123
Client - SOCKS encapsulation of SSTP protocol client
overview 119
Client state information
Polling encapsulation protocol client 98
Closing a KeepAlive connection event
KeepAlive encapsulation protocol client 77
KeepAlive encapsulation protocol server 88
Closing a KeepAlive GET session event
KeepAlive encapsulation protocol client 78
Closing a KeepAlive POST session event
KeepAlive encapsulation protocol client 77
Closing a LongLived connection event
LongLived encapsulation protocol client 61
LongLived encapsulation protocol server 66
Closing a Polling connection event
Polling encapsulation protocol client 101
Polling encapsulation protocol server 108
Closing a Polling session event
Polling encapsulation protocol server 108
Closing a POST session event
KeepAlive encapsulation protocol server 88
Closing a Secure Tunnel connection event
Secure Tunnel encapsulation protocol client 116
Closing a SOCKS connection event
SOCKS encapsulation of SSTP protocol client 122
Common HTTP data types
encapsulation data types - Application-Data 31
encapsulation data types - Encapsulation-Echo-String 31
encapsulation data types - Relay-Server-Name 31
encapsulation data types - Server-User-Agent 32
encapsulation data types - Virtual-Connection-GUID 30
HTTP response headers 35
Request-Header 32
Response-Code-And-Reason-Phrase 36
Connection state information
KeepAlive encapsulation protocol client 73
LongLived encapsulation protocol client 57
Polling encapsulation protocol client 96
Secure Tunnel encapsulation protocol client 113
SOCKS encapsulation of SSTP protocol client 120
ConnectionEstablishment timer event - KeepAlive encapsulation protocol client 84
ConnectionEstablishment timer event - KeepAlive encapsulation protocol server 93
ConnectionEstablishment timer event - LongLived encapsulation protocol client 64
ConnectionEstablishment timer event - LongLived encapsulation protocol server 70
ConnectionEstablishment timer event - Polling encapsulation protocol client 106
ConnectionEstablishment timer event - Polling encapsulation protocol server 112
ConnectionEstablishment timer event - Secure Tunnel encapsulation protocol client 118
ConnectionEstablishment timer event - SOCKS encapsulation of SSTP protocol client 124
ConnectionEstablishment timer- KeepAlive encapsulation protocol server 87
ConnectionEstablishment timer- LongLived encapsulation protocol client (section 3.1.2.1 59, section 3.3.2.1 75)
ConnectionEstablishment timer- LongLived encapsulation protocol server 66
ConnectionEstablishment timer- Polling encapsulation protocol client 98
ConnectionEstablishment timer- Polling encapsulation protocol server 108
ConnectionEstablishment timer- Secure Tunnel encapsulation protocol client 114
ConnectionEstablishment timer- SOCKS encapsulation of SSTP protocol client 121

D

Data model - abstract
client (section 3.1.1 55, section 3.3.1 71, section 3.5.1 94, section 3.7.1 112, section 3.9.1 119)
KeepAlive encapsulation protocol client 71

Copyright © 2014 Microsoft Corporation.

KeepAlive encapsulation protocol server 86
LongLived encapsulation protocol client 55
LongLived encapsulation protocol server 65
Polling encapsulation protocol client 94
Polling encapsulation protocol server 107
Secure Tunnel encapsulation protocol client 112
server (section 3.2.1 65, section 3.4.1 86, section 3.6.1 107, section 3.8.1 118, section 3.10.1 125)
SOCKS encapsulation of SSTP protocol client 119

E

Establishing a KeepAlive encapsulation connection event
KeepAlive encapsulation protocol client 76
Establishing a LongLived encapsulation connection event
LongLived encapsulation protocol client 59
Establishing a Polling encapsulation connection event
Polling encapsulation protocol client 100
Establishing a Secure Tunnel encapsulation connection event
Secure Tunnel encapsulation protocol client 115
Establishing a SOCKS encapsulation connection event
SOCKS encapsulation of SSTP protocol client 122
Examples
HTTP KeepAlive encapsulation 129
HTTP LongLived encapsulation 127
HTTP Polling encapsulation 134
overview 127
proxy authentication using NTLM 143
Secure Tunnel proxy protocol 142
SOCKS proxy 142

F

Fields - vendor-extensible 29

G

GetNetworkReceiveIO timer event - KeepAlive encapsulation protocol client 84
GetNetworkReceiveIO timer- LongLived encapsulation protocol client 75
Glossary 13

H

Higher-layer triggered events
server (section 3.8.4 119, section 3.10.4 125)
HTTP encapsulation protocols
HTTP KeepAlive encapsulation connections 20
HTTP LongLived encapsulation connections 19
HTTP Polling encapsulation connections 22
HTTP KeepAlive encapsulation example 129
HTTP LongLived encapsulation example 127
HTTP Polling encapsulation example 134

I

IdleConnection timer event - KeepAlive encapsulation protocol server 93
IdleConnection timer- KeepAlive encapsulation protocol server 87
Implementer - security considerations 147
Index of security parameters 147
Informative references 14
Initialization
server 125
Initialization - KeepAlive encapsulation protocol client 75
Initialization - KeepAlive encapsulation protocol server 87
KeepAlive listener 88
Initialization - LongLived encapsulation protocol client 59
Initialization - LongLived encapsulation protocol server 66
LongLived listener 66
Initialization - Polling encapsulation protocol client 99
Initialization - Polling encapsulation protocol server 108
Polling encapsulation listener 108
Initialization - Secure Tunnel encapsulation of SSTP protocol server
Secure Tunnel encapsulation listener 119
Initialization - Secure Tunnel encapsulation protocol client 115
Secure Tunnel listener endpoints 115
timers started 115
Initialization - SOCKS encapsulation of SSTP protocol client 122
Introduction 12

K

KeepAlive encapsulation
KeepAlive-GET-Request 42
KeepAlive-GET-Response 45
KeepAlive-POST-Request 44
KeepAlive-POST-Response 46
KeepAlive encapsulation client
overview 71
KeepAlive encapsulation server
connection state information 87
KeepAlive timer event - KeepAlive encapsulation protocol client 84
KeepAlive timer event - KeepAlive encapsulation protocol server 94
KeepAlive timer event - LongLived encapsulation protocol client 65
KeepAlive timer event - LongLived encapsulation protocol server 71
KeepAlive timer event - Secure Tunnel encapsulation protocol client 118
KeepAlive timer event - SOCKS encapsulation of SSTP protocol client 125
KeepAlive timer- KeepAlive encapsulation protocol server 87
KeepAlive timer- LongLived encapsulation protocol client (section 3.1.2.3 59, section 3.3.2.4 75)

Copyright © 2014 Microsoft Corporation.

KeepAlive timer- LongLived encapsulation protocol server 66
KeepAlive timer- Secure Tunnel encapsulation protocol client 115
KeepAlive timer- SOCKS encapsulation of SSTP protocol client 121

L

Local events - KeepAlive encapsulation protocol
  client 84
  re-opening the POST session after a transport disconnect 84
Local events - KeepAlive encapsulation protocol server 94
Local events - LongLived encapsulation protocol client 65
Local events - LongLived encapsulation protocol server 71
Local events - Polling encapsulation protocol client 107
Local events - Polling encapsulation protocol server 112
Local events - Secure Tunnel encapsulation protocol client 118
Local events - SOCKS encapsulation of SSTP protocol client 125
LongLived encapsulation
  LongLived-GET-Request 36
  LongLived-GET-Response 40
  LongLived-POST-Request 38
  LongLived-POST-Response 42
LongLived encapsulation client
  overview 55
LongLived encapsulation server
  connection state information 65

M

Message processing
  client (section 3.1.5 62, section 3.7.5 116, section 3.9.5 123)
  server (section 3.2.5 67, section 3.6.5 109, section 3.10.5 125)
Message processing - KeepAlive encapsulation protocol client
  KeepAlive-GET-Response processing 81
  KeepAlive-POST-Response processing 79
  sending a KeepAlive-GET-Request 83
Message processing - KeepAlive encapsulation protocol server
  GET session processing 89
  POST session processing 91
Message processing - LongLived encapsulation protocol client
  receiving data on the GET session 63
  receiving data on the POST session 62
Message processing - LongLived encapsulation protocol server 67
  GET session processing 67
  POST session processing 69
Message processing - Polling encapsulation protocol client
  Polling-POST-Response processing 103
Message processing - Polling encapsulation protocol server 109
  receiving a Polling-POST-request (after handshake) 111
  receiving a Polling-POST-request (initial handshake request) 109
  receiving a Polling-POST-request (last handshake request) 110
Message processing - Secure Tunnel encapsulation protocol client 116
  application data processing 117
  HTTP response processing 116
Message processing - SOCKS encapsulation of SSTP protocol client 123
  application data processing 124
  SOCKS connection negotiation processing 123
Messages
  Secure Tunnel Proxy 51
  SOCKS Encapsulation 52
  syntax 30
  syntax - Secure Tunnel Proxy 51
  syntax - SOCKS encapsulation 52
  transport 30

N

NetworkReceiveIO timer event - LongLived encapsulation protocol client 65
NetworkReceiveIO timer event - LongLived encapsulation protocol server 71
NetworkReceiveIO timer event - Polling encapsulation protocol client 107
NetworkReceiveIO timer event - Secure Tunnel encapsulation protocol client 118
NetworkReceiveIO timer event - SOCKS encapsulation of SSTP protocol client 125
NetworkReceiveIO timer- LongLived encapsulation protocol client 59
NetworkReceiveIO timer- LongLived encapsulation protocol server 66
NetworkReceiveIO timer- Polling encapsulation protocol client 98
NetworkReceiveIO timer- Secure Tunnel encapsulation protocol client 114
NetworkReceiveIO timer- SOCKS encapsulation of SSTP protocol client 121
Normative references 14

O

Other local events
  client (section 3.1.7 65, section 3.3.7 84, section 3.5.7 107, section 3.7.7 118, section 3.9.7 125)
  server (section 3.2.7 71, section 3.4.7 94, section 3.6.7 112, section 3.8.7 119, section 3.10.7 126)
Overview (synopsis) 15
  HTTP encapsulation protocols 18

Copyright © 2014 Microsoft Corporation.

performance considerations 26
Secure Tunnel connections 24
SOCKS connections 26

P

Parameters - security index 147
Polling encapsulation
Polling-POST-Request 47
Polling-POST-Response 50
Polling encapsulation client
overview 94
Polling encapsulation server
connection state information 107
Polling encapsulation timer event - Polling encapsulation protocol client 107
Polling encapsulation timer event - Polling encapsulation protocol server 112
Polling encapsulation timer- Polling encapsulation protocol client 99
PostNetworkReceiveIO timer event - KeepAlive encapsulation protocol client 84
PostNetworkReceiveIO timer- LongLived encapsulation protocol client 75
Preconditions 28
Prerequisites 28
Product behavior 148
proxy authentication using NTLM example 143
Proxy state information
KeepAlive encapsulation protocol client 74
LongLived encapsulation protocol client 58
Polling encapsulation protocol client 98
Secure Tunnel encapsulation protocol client 114
SOCKS encapsulation of SSTP protocol client 121

R

References 13
informative 14
normative 14
Relationship to other protocols 27
Re-opening a KeepAlive GET session event
KeepAlive encapsulation protocol client 78
Re-opening a KeepAlive POST session event
KeepAlive encapsulation protocol client 78

S

Secure Tunnel encapsulation client
overview 112
Secure Tunnel encapsulation of SSTP server
overview 118
Secure Tunnel Proxy message 51
Secure Tunnel proxy protocol example 142
Security
authentication of clients 147
implementer considerations 147
overview 147
parameter index 147
Sending application data
KeepAlive encapsulation protocol client 78
KeepAlive encapsulation protocol server 88
LongLived encapsulation protocol client 62
LongLived encapsulation protocol server 67
Polling encapsulation protocol client 101
Polling encapsulation protocol server 109
Secure Tunnel encapsulation protocol client 116
SOCKS encapsulation of SSTP protocol client 123
Sequencing rules
client (section 3.1.5 62, section 3.7.5 116, section 3.9.5 123)
server (section 3.2.5 67, section 3.6.5 109, section 3.10.5 125)
Sequencing rules - LongLived encapsulation protocol server 67
Sequencing rules - Polling encapsulation protocol server 109
Sequencing rules - Secure Tunnel encapsulation protocol client 116
Sequencing rules - SOCKS encapsulation of SSTP protocol client 123
Server
abstract data model (section 3.2.1 65, section 3.4.1 86, section 3.6.1 107, section 3.8.1 118, section 3.10.1 125)
higher-layer triggered events (section 3.8.4 119, section 3.10.4 125)
initialization 125
message processing (section 3.2.5 67, section 3.6.5 109, section 3.10.5 125)
other local events (section 3.2.7 71, section 3.4.7 94, section 3.6.7 112, section 3.8.7 119, section 3.10.7 126)
overview (section 3.8 118, section 3.10 125)
sequencing rules (section 3.2.5 67, section 3.6.5 109, section 3.10.5 125)
timer events (section 3.8.6 119, section 3.10.6 126)
timers (section 3.8.2 118, section 3.10.2 125)
Server - KeepAlive encapsulation protocol
closing a KeepAlive connection 88
closing a POST session 88
connection state information 87
ConnectionEstablishment timer 87
ConnectionEstablishment timer event 93
GET session processing 89
IdleConnection timer 87
IdleConnection timer event 93
KeepAlive timer 87
KeepAlive timer event 94
POST session processing 91
sending application data 88
Server - KeepAlive encapsulation protocol - abstract data model 86
Server - KeepAlive encapsulation protocol - initialization 87
KeepAlive listener 88
Server - KeepAlive encapsulation protocol - local events 94
Server - LongLived encapsulation protocol
closing a LongLived connection 66
connection state information 65
ConnectionEstablishment timer 66

Copyright © 2014 Microsoft Corporation.

ConnectionEstablishment timer event 70
GET session processing 67
KeepAlive timer 66
KeepAlive timer event 71
NetworkReceiveIO timer 66
NetworkReceiveIO timer event 71
POST session processing 69
sending application data 67
Server - LongLived encapsulation protocol - abstract data model 65
Server - LongLived encapsulation protocol - initialization 66
LongLived listener 66
Server - LongLived encapsulation protocol - local events 71
Server - LongLived encapsulation protocol - message processing 67
Server - LongLived encapsulation protocol - sequencing rules 67
Server - Polling encapsulation protocol
closing a Polling connection 108
closing a Polling session 108
connection state information 107
ConnectionEstablishment timer 108
ConnectionEstablishment timer event 112
Polling encapsulation timer event 112
receiving a Polling-POST-request (after handshake) 111
receiving a Polling-POST-request (initial handshake request) 109
receiving a Polling-POST-request (last handshake request) 110
sending application data 109
Server - Polling encapsulation protocol - abstract data model 107
Server - Polling encapsulation protocol - initialization 108
Polling encapsulation listener 108
Server - Polling encapsulation protocol - local events 112
Server - Polling encapsulation protocol - message processing 109
Server - Polling encapsulation protocol - sequencing rules 109
Server - Secure Tunnel encapsulation of SSTP protocol
overview 118
STTP KeepAlive timer 118
Server - Secure Tunnel encapsulation of SSTP protocol - initialization
Secure Tunnel encapsulation listener 119
Server - Secure Tunnel encapsulation of SSTP protocol - timers 118
Server - SOCKS encapsulation of SSTP protocol
overview 125
SOCKS Encapsulation message 52
SOCKS encapsulation of SSTP protocol client
overview 119
SOCKS encapsulation of SSTP protocol server
overview 125
SOCKS proxy example 142
Standards assignments 29
STTP KeepAlive timer- Secure Tunnel encapsulation of SSTP protocol server 118
Syntax 30

T

Timer events
server (section 3.8.6 119, section 3.10.6 126)
Timers
server (section 3.8.2 118, section 3.10.2 125)
Timers - Secure Tunnel encapsulation of SSTP protocol server 118
Tracking changes 159
Transport 30
Triggered events - higher-layer
server (section 3.8.4 119, section 3.10.4 125)
Triggered events - KeepAlive encapsulation protocol client
closing a KeepAlive connection 77
closing a KeepAlive GET session 78
closing a KeepAlive POST session 77
establishing a KeepAlive encapsulation connection 76
re-opening a KeepAlive GET session 78
re-opening a KeepAlive POST session 78
sending application data 78
Triggered events - KeepAlive encapsulation protocol server
closing a KeepAlive connection 88
closing a POST session 88
sending application data 88
Triggered events - LongLived encapsulation protocol client
closing a LongLived connection 61
establishing a LongLived encapsulation connection 59
sending application data 62
Triggered events - LongLived encapsulation protocol server
closing a LongLived connection 66
sending application data 67
Triggered events - Polling encapsulation protocol client
closing a Polling connection 101
establishing a Polling encapsulation connection 100
sending application data 101
Triggered events - Polling encapsulation protocol server
closing a Polling connection 108
closing a Polling session 108
sending application data 109
Triggered events - Secure Tunnel encapsulation protocol client
closing a Secure Tunnel connection 116
establishing a Secure Tunnel encapsulation connection 115
sending application data 116
Triggered events - SOCKS encapsulation of SSTP protocol client
closing a SOCKS connection 122

Copyright © 2014 Microsoft Corporation.

establishing a SOCKS encapsulation connection 122
sending application data 123

V

Vendor-extensible fields 29
Versioning 28

Copyright © 2014 Microsoft Corporation.