[MS-RPCH]:
Remote Procedure Call Over HTTP Protocol Specification

Revision Summary

Date	Revision History	Revision Class	Comments
10/22/2006	0.01		MCPP Milestone 1 Initial Availability
01/19/2007	1.0		MCPP Milestone 1
03/02/2007	1.1		Monthly release
04/03/2007	1.2		Monthly release
05/11/2007	1.3		Monthly release

Date	Revision History	Revision Class	Comments
06/01/2007	1.3.1	Editorial	Revised and edited the technical content.
07/03/2007	1.3.2	Editorial	Revised and edited the technical content.
07/20/2007	1.3.3	Editorial	Revised and edited the technical content.
08/10/2007	1.3.4	Editorial	Revised and edited the technical content.
09/28/2007	1.3.5	Editorial	Revised and edited the technical content.
10/23/2007	1.3.6	Editorial	Revised and edited the technical content.
11/30/2007	1.3.7	Editorial	Revised and edited the technical content.
01/25/2008	1.3.8	Editorial	Revised and edited the technical content.

Table of Contents

1 Introduction

This document specifies the use of HTTP or HTTPS as a transport for the **Remote Procedure Call (RPC)** protocol, as specified in [C706] and extended as specified in [MS-RPCE]. The document builds upon and relies heavily upon the [C706] and [MS-RPCE] specifications, and readers must be familiar with their terms and concepts.

The Remote Procedure Call (RPC) Over HTTP Protocol tunnels RPC network traffic from an RPC **client** to an RPC **server** through a network agent referred to as an **RPC over HTTP proxy**. The protocol is applicable to network topologies where the use of an HTTP or HTTPS-based transport is necessary, for example to traverse an application firewall, and the application or computer systems communicating over the topology require the use of the Remote Procedure Call (RPC) protocol.

1.1 Glossary

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

Binary Large Object (BLOB)
Certificate
Channel Lifetime
Channel Recycling
Client
Dynamic Endpoint
Echo Request
Echo Response
Endpoint
HTTP Client
HTTP Proxy
HTTP Server
IN Channel
IN Channel Recycling
Inbound
Inbound Proxy
Mixed Proxy
OUT Channel
OUT Channel Recycling
Outbound
PDU Stream
Plug a Channel
Plugged Channel Mode
Predecessor Channel
Predecessor Inbound Proxy
Predecessor Outbound Proxy
Protocol Data Unit (PDU)
Protocol Dialect
Proxy
Receive Window
Replacement Channel
Remote Procedure Call (RPC)
RPC over HTTP Proxy
RPC PDU
RPC Protocol Sequence
RPC Transport
RTS Cookie
RTS PDU

Server
Successor Channel
Successor Inbound Proxy
Successor Outbound Proxy
Universally Unique Identifier (UUID) or Globally Unique Identifier (GUID)
Unplug a Channel
Unplugged Channel Mode
URI
Virtual Connection
Virtual IN Channel
Virtual OUT Channel
Well-Known Endpoint

The following terms are specific to this document:

Expired: An **IN channel** or **OUT channel** state in which the maximum content length has been reached or exceeded and can no longer accept any **PDU** awaiting transmission. See also **channel lifetime**. For more details, see section 2.1.2.1.

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

1.2 References

1.2.1 Normative References

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

[C706] The Open Group, "DCE 1.1: Remote Procedure Call", C706, August 1997, http://www.opengroup.org/public/pubs/catalog/c706.htm

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

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

[MS-EERR] Microsoft Corporation, "ExtendedError Remote Data Structure", January 2007.

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

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

[MS-RPCE] Microsoft Corporation, "Remote Procedure Call Protocol Extensions", January 2007.

[NETBEUI] IBM Corporation, "LAN Technical Reference: 802.2 and NetBIOS APIs", 1986, http://publibz.boulder.ibm.com/cgi-bin/bookmgr_OS390/BOOKS/BK8P7001/CCONTENTS

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

[RFC1001] Network Working Group, "Protocol Standard for a NetBIOS Service on a TCP/UDP Transport: Concepts and Methods", RFC 1001, March 1987, http://www.ietf.org/rfc/rfc1001.txt

[RFC1002] Network Working Group, "Protocol Standard for a NetBIOS Service on a TCP/UDP Transport: Detailed Specifications", RFC 1002, March 1987, http://www.ietf.org/rfc/rfc1002.txt

[RFC1034] Mockapetris, P., "Domain Names–Concepts and Facilities", RFC 1034, November 1987, http://www.ietf.org/rfc/rfc1034.txt

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

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

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

[RFC2617] Franks, J., Hallam-Baker, P., Hostetler, J., Lawrence, S., Leach, P., Luotonen, A., and Stewart, L., "HTTP Authentication: Basic and Digest Access Authentication", RFC 2617, June 1999, http://www.ietf.org/rfc/rfc2617.txt

[RFC2818] Rescorla, E., "HTTP Over TLS", RFC 2818, May 2000, http://www.ietf.org/rfc/rfc2818.txt

[RFC3513] Hinden, R. and Deering, S., "Internet Protocol Version 6 (IPv6) Addressing Architecture", RFC 3513, April 2003, http://www.ietf.org/rfc/rfc3513.txt

[RFC3548] Josefsson, S., Ed., "The Base16, Base32, and Base64 Data Encodings", RFC 3548, July 2003, http://www.ietf.org/rfc/rfc3548.txt

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

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

1.2.2 Informative References

[MSDN-RPCHTTP] Microsoft Corporation, "Remote Procedure Calls Using RPC Over HTTP", http://msdn2.microsoft.com/en-us/library/aa375384.aspx

[RFC3280] Housley, R., Polk, W., Ford, W., and Solo, D., "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile", RFC 3280, April 2002, http://www.ietf.org/rfc/rfc3280.txt

1.3 Protocol Overview (Synopsis)

This section presents an overview of the following:

- The provisions in this protocol that enable the use of HTTP as a transport.
- The roles and dialects comprising the protocol.
- The encoding of Remote Procedure Call (RPC) **Protocol Data Units (PDUs)** within HTTP requests and responses.

1.3.1 Extensions to HTTP Functionality

Each connection-oriented transport must meet the requirements as specified in [MS-RPCE] section 2.1.1. This specification incorporates the following provisions to meet those requirements using the hypertext transfer protocol (HTTP) [RFC2616].

- Duplex communications using virtual channels.
- Stream semantics through incrementally sending contents from the message body.
- Unlimited data stream using a sequence of HTTP requests or HTTP responses instead of using chunked transfer encoding ([RFC2617] section 3.6.1).

1.3.2 Roles and Dialects

This protocol defines the role of a Remote Procedure Call (RPC) over HTTP proxy that may be deployed to relay network traffic between a client and a server residing on networks separated by a firewall, through which HTTP or HTTPS traffic is permitted to flow.

This protocol has two main **Protocol Dialects** called RPC over HTTP v1 and RPC over HTTP v2. There are different roles defined for each dialect.

RPC over HTTP v1 defines the roles of a client, a server, and an RPC over HTTP proxy, called a **mixed proxy** in this document. The following diagram shows the different roles and their relationships.

Figure 1: RPC over HTTP v1 roles

RPC over HTTP v2 works in a more complex topology and defines the roles of a client, server, **inbound** RPC over HTTP proxy, and **outbound proxy** RPC over HTTP proxy. RPC over HTTP v2 proxies do not have fixed roles. They can act as inbound or outbound proxies depending on the protocol sequence in which they participate. The following diagram shows the different roles and their interactions.

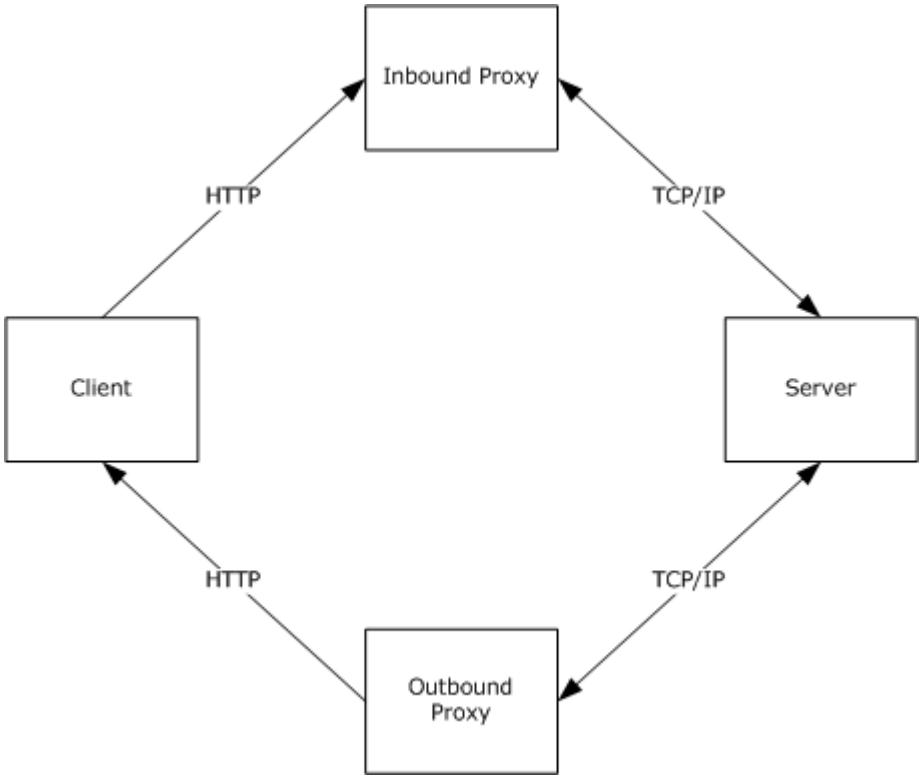


Figure 2: RPC over HTTP v2 roles

The roles defined herein are preserved even when the **inbound proxy** and outbound proxy roles run on the same network node. However, this protocol does not assume that the inbound proxy and outbound proxy reside on the same network node. Load balancing and clustering technologies, among others, may cause the inbound proxy and outbound proxy to run on different network nodes.<1>

An RPC over HTTP proxy that only supports RPC over HTTP v2 cannot interoperate with an RPC over HTTP v1 Client or an RPC over HTTP v1 server.

The differences between RPC over HTTP v1 and v2 fall into three main categories, based on the following:

- The RPC over HTTP PDUs and RPC over HTTP PDU's location.
- The **Proxy** roles.
- The mapping of RPC and RPC over HTTP PDUs to HTTP requests.

1.3.3 High-Level Overview

The Remote Procedure Call (RPC) protocol transmits RPC Protocol Data Units (PDUs) between RPC clients and RPC servers [C706] and extended [MS-RPCE]. At a very high level, this protocol functions as an **RPC transport** and relays (tunnels) these PDUs to the server using HTTP (or HTTPS) and TCP/IP as specified in section 1.4.

This protocol takes an **RPC PDU** generated [C706] and extended [MS-RPCE] on either an RPC client or an RPC server and transfers it to the other side, to the RPC server for the RPC client and to the

RPC client for the RPC server, using a network agent called an RPC over HTTP proxy. All traffic MUST go through an RPC over HTTP proxy.

The most common deployment configuration, even though it is not a requirement for this protocol, is for the client to be separated from the RPC over HTTP proxy by a wide area network (WAN) such as the Internet where the network traffic for this protocol travels over HTTP or HTTPS. The RPC over HTTP proxy and the RPC server are usually connected through a local area network (LAN) where the network traffic for this protocol travels over TCP/IP.

The RPC PDUs are conceptually viewed by this protocol as an ordered sequence or stream of PDUs that can travel from RPC client to RPC server or from RPC server to RPC client. This protocol does not modify or consume RPC PDUs. The only exception to this rule is when using HTTPS and RPC over HTTP v2. In this case, RPC PDUs will be encrypted at the **HTTP Client** and decrypted at the inbound or outbound proxy when traveling between an HTTP Client and an inbound proxy or outbound proxy.

This protocol inserts its own PDUs into the RPC **PDU stream** and routes the resulting stream of PDUs over HTTP requests and responses or TCP/IP connections as defined throughout this document. Using ABNF notation as specified in [RFC4234], the definition of the resulting stream of RPC and RPC over HTTP PDUs outside the protocol sequences specified in section 3 of this document is as follows:

1*[(1*(RPC over HTTP PDU))*(RPC PDU)]

The following diagram illustrates this definition.

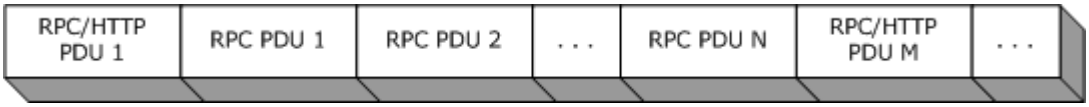


Figure 3: RPC over HTTP PDU stream

An example PDU stream is provided in section 4.

In addition to specifying how the PDUs are ordered and mapped to the underlying transport, the RPC over HTTP v2 dialect of this protocol specifies how:

- An implementation must map an unbounded number of PDUs from a stream onto a number of HTTP requests and responses, each of which is bounded by its content length. This is done through a process called **Channel Recycling**, specified in section 3.2.
- An implementation should prevent HTTP requests and responses that are used by this protocol from being timed out as idle by network agents. This is done by sending PDUs in a process called "pinging," as specified in section 3.2. The same pinging process is used to detect whether the other party is still running and reachable through the network.

1.4 Relationship to Other Protocols

The RPC Over HTTP Protocol is used in conjunction with the Remote Procedure Call (RPC) Protocol Extensions [MS-RPCE] and relies on HTTP 1.0 and keep-alive connections from HTTP 1.1 [RFC2616]. It also relies on HTTPS [RFC2818] for data protection services. The following diagram illustrates the protocol layering for this protocol on the client.

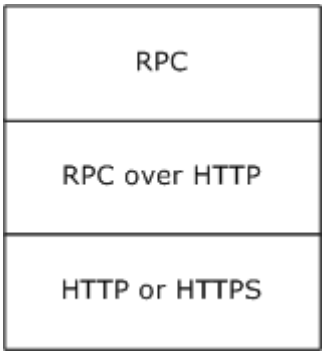


Figure 4: Protocol layering on the client

For RPC over HTTP, the mixed, inbound, and outbound proxies use the following protocol layering for their client-facing part.

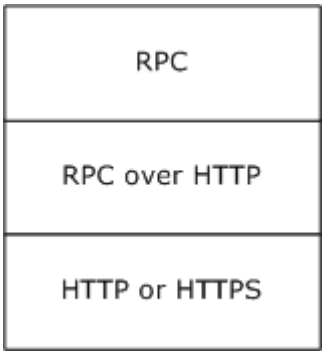


Figure 5: Protocol layering on client-facing proxy

For the server-facing part of the mixed, inbound, and outbound proxy, the protocol layering is as shown in the following diagram.

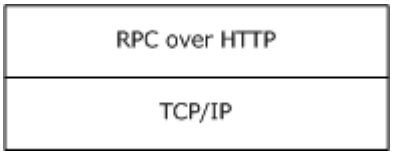


Figure 6: Protocol layering on server-facing proxy

The server uses the following protocol layering.

Figure 7: Protocol layering on server

A consequence of this protocol layering is that an RPC client using RPC over TCP (ncacn_ip_tcp) predecessor **RPC protocol sequence** cannot interoperate with an RPC server using RPC over HTTP (ncacn_http) RPC Protocol Sequence, and vice versa.

RPC over HTTP v1 can run on HTTP only. RPC over HTTP v2 can run either over HTTP or HTTPS. The decision on whether to use HTTP or HTTPS is made by the client based on information provided by higher-layer protocols.

RPC over HTTP v2 makes use of the ExtendedError Remote Protocol Extensions as specified in [MS-EERR] to transmit error information.

1.5 Prerequisites/Preconditions

If HTTPS transport is used, a **certificate** must be deployed on the inbound and outbound proxies.

The RPC Over HTTP Protocol does not define any means for activating a server or proxy and thus the server and all proxies must be fully initialized and listening before the RPC Over HTTP Protocol can start operating. The server must be listening on a well-known or **dynamic endpoint**. RPC over HTTP proxies must listen in an implementation-specific way on the **URIs** specified in sections 3.1.2.1 and 3.2.3.3.

1.6 Applicability Statement

The RPC Over HTTP Protocol is applicable to scenarios where an RPC client needs to communicate with an RPC server, and due to network constraints (for example, topology, firewalls, protocols, and so on) an HTTP transport must be used.

This protocol is also applicable when data is received from the Internet or other public networks and additional protection for the RPC server is required. RPC over HTTP is generally not applicable in

cases where a single RPC method call will be executed with little data exchanged by the RPC client and the RPC server. The reason is that the additional security provisions of this protocol and the additional synchronization required by inbound and outbound proxies introduce significant overhead on the initial connection establishment. Once a connection is established, RPC/HTTP is very efficient in transmitting data between RPC clients and RPC servers.

RPC over HTTP v1 is superseded by RPC over HTTP v2 and should not be used unless maintaining backward compatibility with RPC over HTTP v1 is a requirement.<2> RPC over HTTP v1 has weak security, poor compatibility with existing HTTP infrastructure, and deviates from RPC connection-oriented protocol requirements ([MS-RPCE] section 2.1.1). More specifically, RPC over HTTP v1 does not meet the second bullet requirement in [MS-RPCE] section 2.1.1 by failing to maintain a reliable communication session. RPC over HTTP v1 fails to keep the communication session opened if the network agents deem the communication session idle.

1.7 Versioning and Capability Negotiation

- **Supported Transports:** The RPC Over HTTP Protocol can run on top of HTTP 1.0 or HTTPS. Remote Procedure Call (RPC) over HTTP v2 requires HTTP 1.1 connection keep-alive support. Details are provided in section 2.1. For historical reasons related to how this protocol has evolved, some HTTP requests and HTTP responses are versioned as 1.0 and some are versioned as 1.1. When not specified explicitly in this specification, version 1.1 should be assumed to be the default.
- **Protocol Versions:** This protocol supports the following explicit protocol dialects: "RPC over HTTP v1" and "RPC over HTTP v2." These protocol dialects are defined in section 1.3. RPC over HTTP v2 supports versioning within RPC over HTTP v2 as defined in section 2.2.3.5.7. RPC over HTTP v1 has no support for versioning.
- **Security and Authentication Methods:** This protocol relies on the security provided by HTTPS and HTTP Basic, or NTLM authentication [RFC2617], and acts as a pass-through for the security provided by RPC. The RPC Over HTTP Protocol does not have security and authentication provisions of its own.
- **Capability Negotiation:** This protocol negotiates one of its two protocol dialects, RPC over HTTP v1 and RPC over HTTP v2, by trying to first establish a connection using RPC over HTTP v2 and, if this fails, falling back to RPC over HTTP v1. The negotiation between RPC over HTTP v1 and RPC over HTTP v2 is defined in section 3.

1.8 Vendor-Extensible Fields

The RPC Over HTTP Protocol does not include vendor-extensible fields. However, this protocol builds on top of HTTP (or HTTPS), which allows vendors to add new HTTP headers [RFC2616]. This protocol also allows vendors to add HTTP headers, but it ignores all such headers.

1.9 Standards Assignments

Parameter	Value	Reference
RPC over HTTP **endpoint** mapper TCP port	593	As specified in [IANAPORT].

2 Messages

This section defines how the RPC Over HTTP Protocol maps over lower-layer protocols, and it defines the syntax for the messages used by this protocol.

The message syntax in this document uses the notation and conventions as specified in [RFC2616] section 2. The parsing constructs: OCTET, CHAR, UPALPHA, LOALPHA, ALPHA, DIGIT, CTL, CR, LF, SP, HT, CRLF, LWS, TEXT, and HEX used in this document are the same as those specified in [RFC2616] section 2.2.

2.1 Transport

Both Remote Procedure Call (RPC) over HTTP v1 and RPC over HTTP v2 start their transport mapping process from a stream of RPC and RPC over HTTP Protocol Data Units (PDUs) that need to be mapped to one or more HTTP or HTTPS requests and TCP/IP connections. Both protocol dialects also share the following characteristics.

- An endpoint mapper with a **Well-Known Endpoint** of 593.
- An RPC protocol identifier of 0x1F.
- An RPC network address for the RPC server provided by a higher layer that MUST be an IPv4 or IPv6 address.
- The RPC endpoint for the RPC server MUST be a TCP/IP port number.
- The predecessor RPC Protocol sequence is "ncacn_http".
- RPC network options provided by higher layers that:
  - MUST contain a valid IPv4 or IPv6 address for the HTTP server.<3>
  - MAY contain an **HTTP Proxy**.<4>

2.1.1 RPC over HTTP v1 Transport

This section defines the mapping of the Remote Procedure Call (RPC) over HTTP v1 Protocol Dialect over lower-layer protocols. From a high-level perspective, this protocol uses a single, custom HTTP request between the client and the mixed proxy and all RPC PDUs are mapped as **Binary Large Objects (BLOBs)** in the message body of this request. The sections that immediately follow specify in detail how this is done.

2.1.1.1 Client to Mixed Proxy Traffic

Remote Procedure Call (RPC) over HTTP v1 MUST use HTTP between the client and the mixed proxy. It MUST use a single HTTP request to map both inbound and **outbound** traffic to the server. The HTTP request MUST be initiated from the client and MUST be received by an **HTTP Server** that runs on the mixed proxy. The address of the HTTP Server is provided by a higher-layer protocol as specified in the previous section. RPC over HTTP v1 MUST use port 80 for the HTTP traffic.

The following subsections define the syntax of the HTTP requests and HTTP responses used by this protocol, and how RPC PDUs are mapped into an HTTP request or an HTTP response.

2.1.1.1.1 RPC Connect Request

The Remote Procedure Call (RPC) connect request is an HTTP request that MUST have the following HTTP header fields.

Method: MUST be set to RPC_CONNECT.

Pragma: MUST be set to the string "No-cache".

Protocol: Clients MUST set this to 1.1. Proxies SHOULD ignore this header field.

URL: The server name and port MUST be encoded in this field as specified in section 2.2.2 of this document.

User-Agent: MUST be set to the string "RPC".

Message Body: MUST be composed as specified in section 2.1.1.1.3.

This request does not use the Content-Type and Content-Length header fields. It also does not use transfer coding or specify a MIME type.

2.1.1.1.2 RPC Connect Response

The Remote Procedure Call (RPC) connect response is an HTTP response that MUST have the following HTTP header fields.

Status Line: [RFC2616] section 6.1 specifies that the status line be composed of three nonspace subfields. The three subfields MUST be set to the following values.

- **HTTP-Version:** MUST be the string "HTTP/1.0"
- **Reason-Phrase:** MUST be the string "Success"
- **Status-Code:** MUST be the string "200"

Message Body: Must be composed as specified in section 2.1.1.1.4 of this document.

2.1.1.1.3 Inbound PDU Stream

Inbound Protocol Data Units (PDUs) from the PDU stream MUST be encoded as BLOBs in the message body of the Remote Procedure Call (RPC) connect request. The first inbound PDU MUST start from the beginning of the message body of the RPC connect request and each subsequent PDU from the PDU stream MUST be placed as a BLOB immediately after the previous PDU in the RPC connect request without any delimiters. The following diagram defines the layout of the PDUs in the message body of the RPC connect request.

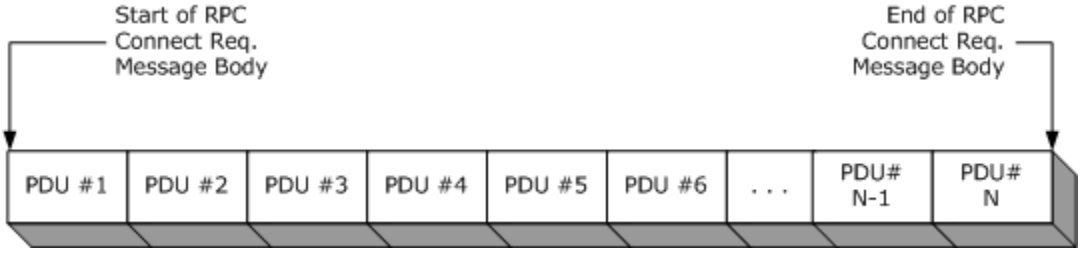


Figure 8: Inbound connect request PDU stream

Each PDU encoded as a BLOB contains its length inside the PDU as specified in [C706] part 4, "RPC PDU Encodings", and thus no delimiters are necessary between the BLOBs. For RPC over HTTP v1, the implementation of the underlying HTTP transport MUST be capable of the following:

- Duplex communication.
- Sending a potentially unbounded number of PDUs in the message body of the RPC connect request while at the same time receiving a potentially unbounded number of PDUs in the message body of the RPC connect response. This protocol specifically allows for sending and receiving a potentially unbounded number of PDUs in the message body of the RPC connect request.

The PDUs are sent in the message body as they are generated for **unplugged channel mode**. In this mode, PDU N MUST be sent as soon as it is generated and will not wait for PDU N+1 to be generated.

2.1.1.1.4 Outbound PDU Stream

Outbound Protocol Data Units (PDUs) from the PDU stream MUST be encoded as BLOBs in the message body of the Remote Procedure Call (RPC) connect response. The first PDU in the RPC connect response MUST start from the beginning of the message body of the RPC connect response, and each subsequent PDU from the PDU stream MUST be placed as a BLOB immediately after the previous PDU in the RPC connect response without any delimiters. The following diagram defines the layout of the PDUs in the message body of the RPC connect response.

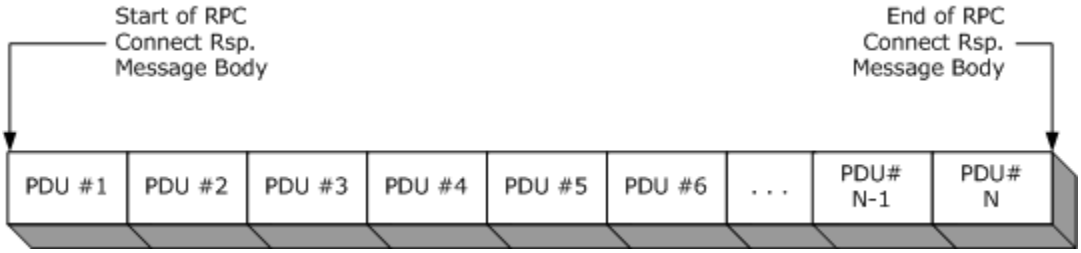


Figure 9: Outbound RPC connect response PDU stream

Each PDU encoded as a BLOB contains its length inside the PDU as specified in [C706] Part 4, "RPC PDU Encodings," and thus no delimiters are necessary between the BLOBs.

For RPC over HTTP v1, the implementation of the underlying HTTP transport MUST be capable of the following:

- Duplex communication.
- Sending a potentially unbounded number of PDUs in the message body of the RPC connect request, while at the same time receiving a potentially unbounded number of PDUs in the message body of the RPC connect response.

The PDUs are sent in the message body as they are generated for unplugged channel mode. In this mode, PDU N MUST be sent as soon as it is generated and will not wait for PDU N+1 to be generated.

2.1.1.2 Mixed Proxy to Server Traffic

Remote Procedure Call (RPC) over HTTP v1 uses TCP/IP between the mixed proxy and the server. The TCP connection MUST be initiated by the mixed proxy. The server name and port to be used for

setting up the TCP connection MUST be extracted from the URI of the HTTP request as specified in section 2.1.1.1. Once the connection is established, the mixed proxy and the server MUST use this connection for transmission of all the PDUs in the PDU stream.

2.1.1.2.1 Legacy Server Response

A server MUST send the ASCII string "ncacn_http/1.0" to the mixed proxy as soon as the TCP connection from the mixed proxy to the server is established. This string literal is called the legacy server response.

2.1.2 RPC over HTTP v2 Transport

This section defines the mapping of the Remote Procedure Call (RPC) over HTTP v2 Protocol Dialect over lower-layer protocols. From a high-level perspective, in its steady state this protocol uses a pair of custom HTTP requests from the client to the inbound proxy and from the client to the outbound proxy. All inbound RPC PDUs are mapped as BLOBs in the message body of the custom request to the inbound proxy and all outbound RPC PDUs are mapped as BLOBs in the message body of the custom request to the outbound proxy. The sections that immediately follow specify in detail how this is done.

2.1.2.1 Client to Inbound or Outbound Proxy

Remote Procedure Call (RPC) over HTTP v2 MUST operate either on top of HTTP or on top of HTTPS. It requires HTTP 1.0 plus connection keep-alive support from HTTP 1.1. Mapping to both protocols happens identically. In this section, mapping is defined only on HTTP, but the same rules apply for HTTPS.<5>

If instructed by a higher-level protocol in an implementation-specific way, implementations of this protocol MUST require the HTTP implementation on the client to authenticate to the HTTP Server running on the inbound proxy or outbound proxy using basic authentication for HTTP [RFC2617] or NTLM authentication for HTTP.

The higher-level protocol MUST provide, in an implementation-specific way, either credentials in the form of user name/password or a client-side certificate. Implementations of this protocol MUST NOT process the credentials or authentication information. Such processing typically happens entirely inside implementations of lower protocol layers.<6>

The same mapping MUST be applied for both the inbound proxy and the outbound proxy traffic. A client implementation SHOULD instruct the implementation of the HTTP protocol on which it runs to use an implementation-specific but reasonable time-out value for all requests.<7>

RPC over HTTP v2 MUST always use a pair of HTTP requests to build a **virtual connection**. The HTTP requests MUST be initiated by the client and received by the inbound proxy or outbound proxy.

Both HTTP requests have implementation-specific content length as defined in the following sections. The address of the HTTP Server is provided by a higher-layer protocol. RPC over HTTP v2 always uses port 80 for HTTP traffic and port 443 for HTTPS traffic.

The next few sections describe the HTTP requests and responses used by RPC over HTTP v2, and the mapping of the PDU stream on top of these requests. The general syntax and meaning of each of the HTTP header fields are specified in [RFC2616]. The next few sections only define the use of a given header field when this protocol uses the field in a more specific or different meaning than the one specified in [RFC2616]. This protocol entirely preserves the syntax and semantics of any HTTP header field not explicitly mentioned here.

2.1.2.1.1 IN Channel Request

The **IN channel** request is an HTTP request [RFC2616]. The header fields of that HTTP request are as follows:

Method: MUST be the "RPC_IN_DATA" string.

Accept: Clients SHOULD set this to "application/rpc" string literal. Inbound proxies MUST ignore this header field.

Cache-Control: Clients MUST set this to "no-cache". Inbound proxies MUST ignore this header field.

Connection: Clients MUST set this to "Keep-Alive". Inbound proxies MUST ignore this header field.

Content-Length: MUST be in the inclusive range of 128 KB to 2 GB.<8>

Host: Clients MUST set this to the server name of the inbound proxy ([RFC2616] section 14.23 "Host"). Inbound proxies SHOULD ignore this header field.

Pragma Directives

- Clients MUST add a "No-cache" pragma directive as specified in [RFC2616] section 14.32. Inbound proxies MUST ignore this directive.
- Optional pragma directive that, if present, MUST be defined to have the format "Pragma:MinConnTimeout=T" where T MUST be a decimal string representation of the minimum connection time out, in seconds, to be used for this IN channel. The time out MUST be in the inclusive range of 120 to 14400 seconds.
- Optional pragma directive that, if present, MUST be defined to have the format "Pragma:ResourceTypeUuid=R" where R is a **Universally Unique Identifier (UUID)** formatted as a string ([C706] Appendix A "Universal Unique Identifier"). This pragma specifies the resource type UUID for this channel. For more details on resource type UUID, see section 3.2.3.1.
- Optional pragma directive that, if present, MUST be defined to have the format "Pragma:SessionId=S" where S is a UUID formatted as a string ([C706] Appendix A "Universal Unique Identifier"). This pragma specifies the session ID for this channel. For more details on session ID, see section 3.2.3.1.

Protocol: Clients SHOULD set this to 1.0. Inbound proxies SHOULD ignore this header field.

URL: The server name and port MUST be encoded in this field. For details on how the encoding MUST be done, see section 2.2.2.

User-Agent: Clients SHOULD set this to the "MSRPC" string literal. Inbound proxies SHOULD ignore this header field.

Message Body: For details on how the message body of an IN channel request MUST be created, see section 2.1.2.1.7.

2.1.2.1.2 OUT Channel Request

The **OUT channel** request is an HTTP request [RFC2616]. The header fields of that HTTP request are as follows:

Method: MUST be set to the "RPC_OUT_DATA" string.

Accept: Clients SHOULD set this to "application/rpc" string literal. Outbound proxies MUST ignore this header field.

Cache-Control: Clients MUST set this to "no-cache". Outbound proxies MUST ignore this header field.

Connection: Clients MUST set this to "Keep-Alive". Outbound proxies MUST ignore this header field.

Content-Length: MUST be set to 76 for non-replacement OUT channels and set to 120 for **replacement OUT channels**.

Host: Clients MUST set this to the server name of the Outbound Proxy ([RFC2616] section 14.23, "Host"). Outbound proxies SHOULD ignore this header field.

Pragma Directives

- Clients MUST add a "No-cache" pragma directive as specified in [RFC2616] section 14.32. Outbound proxies MUST ignore this directive.
- Optional pragma directive that, if present, MUST be defined to have the format "Pragma:MinConnTimeout=T" where T MUST be a decimal string representation of the minimum connection timeout, in seconds, to be used for this IN channel. The timeout MUST be in the inclusive range of 120 to 14,400 seconds.
- Optional pragma directive that, if present, MUST be defined to have the format "Pragma:ResourceTypeUuid=R" where R MUST be a Universally Unique Identifier (UUID) formatted as a string ([C706] Appendix A "Universal Unique Identifier"). This pragma specifies the resource type UUID for this channel. For more details on resource type UUID, see section 3.2.3.1.
- Optional pragma directive that, if present, MUST be defined to have the format "Pragma:SessionId=S" where S MUST be a UUID formatted as a string ([C706] Appendix A "Universal Unique Identifier"). This pragma specifies the session ID for this channel. For more details on session ID, see section 3.2.3.1.

Protocol: Clients SHOULD set this to 1.0. Outbound proxies SHOULD ignore this header field.

URL: The server name and port are encoded in this field. For information on how the encoding is done, see section 2.2.2 of this document.

User-Agent: Clients SHOULD set this to the "MSRPC" string literal. Outbound proxies SHOULD ignore this header field.

Message Body: For the definition of how the message body of an OUT channel request MUST be created see section 2.1.2.1.8 of this document.

2.1.2.1.3 IN Channel Response

The IN channel response is an HTTP response [RFC2616]. It is used only in error conditions on the remote procedure call (RPC) over HTTP Proxy. The HTTP header fields and message body syntax where different [RFC2616] are as follows:

Status Line: [RFC2616] section 6.1 specifies that the status line be composed of three nonspace subfields:

HTTP-Version: SHOULD be the character sequence HTTP/1.0.

Reason-Phrase MUST be in the following form:

```
reason_phrase = "RPC Error: " RPC_Error [ee_info]
       RPC_Error = 1*HEX
       ee_info = ", EEInfo: " EncodedEEInfo
```

RPC_Error: MUST be interpreted as a hexadecimal representation of an error code. The error code MUST be an implementation-specific value between 0x0 and 0xFFFFFFFF. The error code MUST NOT be one of the error codes specified in [MS-RPCE] section 3.3.3.5.1.<9>

ee_info: Is part of the reason-phrase and MUST be present if error information is available to the inbound proxy. The behavior of the inbound proxy is defined in section 3.2.3.5.10.

EncodedEEInfo: MUST be a base64-encoded Binary Large Object (BLOB). The base64 encoding MUST be as specified in [RFC3548]. The content of the BLOB is specified in [MS-EERR]. The BLOB MUST continue until the CRLF delimiter at the end of the status line.

The total length of the reason-phrase line MUST NOT exceed 1024 bytes.

Status-Code: MUST be the character sequence 503.

MessageBody: The message body MUST be in the following format:

```
message_body = ["RPC EEInfo:" EncodedEEInfo]
```

EncodedEEInfo: MUST be a base64-encoded BLOB. The base64 encoding MUST be as specified in [RFC3548]. The content of the BLOB is specified in [MS-EERR]. The BLOB MUST continue until the CRLF delimiter at the end of the message body.

2.1.2.1.4 OUT Channel Response

The OUT channel response is sent in both success and failure cases. In success case, the header fields of the HTTP response to the OUT channel request are as follows:

Content-Length: MUST be set to an implementation-specific value in the inclusive range of 128 KB to 2 GB.<10>

Content-Type: MUST be set to the string literal "application/rpc".

Status Line: [RFC2616] section 6.1 specifies that the status line be composed of three nonspace subfields:

- **HTTP-Version:** MUST be the character sequence HTTP/1.1.
- **Status-Code:** MUST be the character sequence 200.
- **Reason-Phrase:** MUST be the character sequence OK.

In a failure case, the format of the OUT channel response is the same as the IN channel response as defined in section 2.1.2.1.3 of this document.

2.1.2.1.5 Echo Request

An Echo Request is used in the proxy discovery protocol sequence as specified in section 3.2.1.4.3.1. The header fields for an Echo Request are:

Method: MUST be set to either the "RPC_IN_DATA" or "RPC_OUT_DATA" string. Both are valid. The client SHOULD use "RPC_IN_DATA" when it is sending an Echo Request as part of a protocol sequence associated with IN channels, and SHOULD use "RPC_OUT_DATA" when it is sending an Echo Request as part of a protocol sequence associated with OUT channels. If the client sends "RPC_IN_DATA" in this field, the proxy MUST act as inbound proxy. If the client sends "RPC_OUT_DATA" in this field, the proxy MUST act as outbound proxy.

Accept: Clients SHOULD set this to the "application/rpc" string literal. Inbound and outbound proxies MUST ignore this header field.

Cache-Control: Clients MUST set this to "no-cache". Inbound and Outbound proxies MUST ignore this header field.

Connection: Clients SHOULD set this to Keep-Alive. Inbound and outbound proxies MUST ignore this header field.

Content-Length: Clients MUST set this header field to a value in the inclusive range of 0 to 0x10.<11>

Host: Clients MUST set this to the server name of the Inbound or Outbound Proxies as specified in [RFC2616] section 14.23, "Host". Inbound and outbound proxies SHOULD ignore this header field.

Pragma Directives:

- Clients MUST add a "No-cache" pragma directive as specified in [RFC2616] section 14.32. Inbound and outbound proxies MUST ignore this directive.

Protocol: Clients SHOULD set this to 1.0. Inbound and outbound proxies SHOULD ignore this header field.

URL: The server name and port are encoded in this field. For information on how the encoding is done, see section 2.2.2.

User-Agent: Clients SHOULD set this to the "MSRPC" string literal. Inbound and outbound proxies SHOULD ignore this header field.

Message Body: Clients MAY set the message body to random content they choose as specified in [RFC2616].<12> Inbound and outbound proxies MUST ignore the message body.

2.1.2.1.6 Echo Response

An Echo Response is used in the proxy discovery protocol sequence as specified in section 3.2.1.4.3.1. This response is sent by an inbound or outbound proxy as an HTTP response to the echo HTTP request. The same Echo Response is sent by both inbound and outbound proxies.

The header fields of the HTTP response are as follows:

Connection: Inbound and outbound proxies SHOULD set this to Keep-Alive. Clients MUST ignore this header field.

Content-Length: Inbound and outbound proxies MUST set this field to 20. Clients MUST ignore this header field.

Content-Type: Inbound and outbound proxies MUST set this header field to the string literal "application/rpc". Clients SHOULD ignore this header field.

Status Line:[RFC2616] section 6.1 specifies that the status line to be composed of three nonspace subfields:

- **HTTP-Version:** The HTTP protocol version of the HTTP Server. This protocol does not require any particular HTTP version. Any HTTP version that is 1.0 or higher SHOULD be accepted by implementations of this protocol.
- **Reason-Phrase:** MUST be OK.
- **Status-Code:** MUST be 200.

This protocol SHOULD always respond with **status-code** of 200 and **reason-phrase** of "OK". It is not a requirement of this protocol for implementations to use the **status-code** field to indicate errors, though implementations MAY do so.<13>

Message Body: Inbound and outbound proxies put in the message body the Echo Response RTS packet described in section 2.2.4.48 and encoded as a Binary Large Object (BLOB).

2.1.2.1.7 Inbound PDU Stream

Inbound Protocol Data Units (PDUs)) from the PDU stream MUST be encoded as Binary Large Object (BLOB) in the message body of the IN channel. The first PDU in the IN channel MUST start from the beginning of the message body of the IN channel and each subsequent PDU from the PDU stream MUST be placed as a BLOB immediately after the previous PDU in the IN channel without any delimiters. The following diagram describes the layout of the PDUs in the message body of the IN channel:

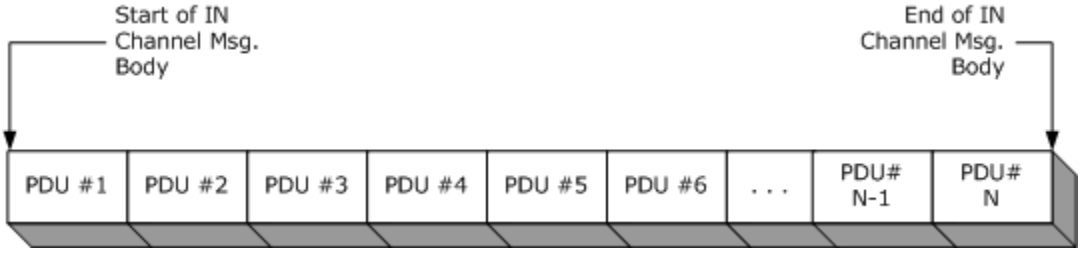


Figure 10: IN channel message PDU stream

Each PDU is encoded as a variable-sized BLOB containing its length inside the PDU; therefore no delimiters are necessary between the BLOBs. The length of the Remote Procedure Call (RPC) PDUs as specified in [C706] Part 4, "RPC PDU Encodings". The length of the **RTS PDUs** is defined in section 2.2.3.6 of this document. An IN channel contains a variable number of PDUs and the PDUs themselves may have variant sizes. An IN channel MUST NOT contain more PDUs than can fit in its maximum content length as indicated by the Content-Length header. If there is not enough space on an IN channel for another PDU from the PDU stream, the IN channel is considered **expired** and MUST NOT be used by the client anymore. A successor IN channel MUST be established. For more details on how the client manages the **channel lifetime**, see section 3.2.2.

The PDUs MUST be sent in the message body as they are generated: PDU N MUST be sent as soon as it is generated and MUST NOT wait for PDU N+1 to be generated.

By using the message body of the IN channel to transmit PDUs over HTTP/HTTPS, this protocol obtains a half-duplex channel for a limited number of bytes that provides reliable, in-order, at-most-once delivery semantics between a client and inbound proxy.

2.1.2.1.8 Outbound PDU Stream

Outbound Protocol Data Units (PDUs) from the PDU stream MUST be encoded as Binary Large Objects (BLOBs) in the message body of the OUT channel. The first PDU in the OUT channel MUST start from the beginning of the message body of the OUT channel and each subsequent PDU from the PDU stream MUST be placed as a BLOB immediately after the previous PDU in the OUT channel without any delimiters. The following diagram describes the layout of the PDUs in the message body of the OUT channel.

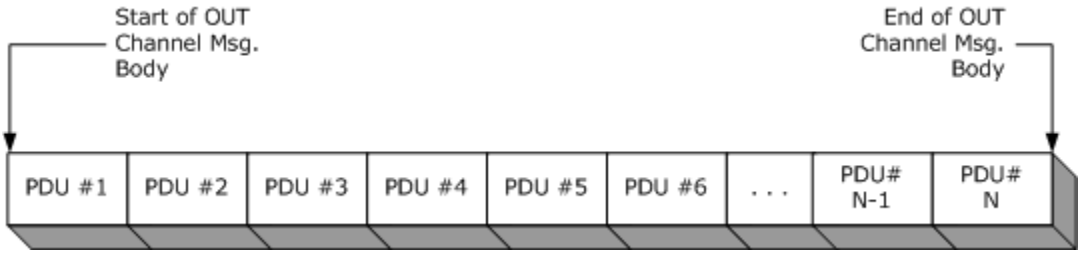


Figure 11: OUT channel message PDU stream

Each PDU encoded as a BLOB contains its length inside the PDU and thus no delimiters are necessary between the BLOBs. The length of the Remote Procedure Call (RPC) PDUs is defined in "RPC PDU Encodings" in Part 4 of [C706]. The length of the RTS PDUs is defined in section 2.2.3.6.

An OUT channel contains a variable number of PDUs and the PDUs themselves may have variable sizes. An OUT channel MUST NOT contain more PDUs than can fit in its maximum content length as indicated by the Content-Length header. If there is not enough space on an OUT channel for another PDU from the PDU stream, the OUT channel is considered channel lifetime and MUST NOT be used by the server anymore. A successor OUT channel MUST be established. How the server manages the channel lifetime is specified in section 3.2.5.

The PDUs are sent in the message body as they are generated. PDU N will be sent as soon as it is generated and will not wait for PDU N+1 to be generated.

By using the message body of the OUT channel to transmit PDUs over HTTP/HTTPS, this protocol obtains a half-duplex channel for a limited number of bytes that provides reliable, in-order, at-most-once delivery semantics between a client and inbound proxy.

2.1.2.2 Inbound or Outbound Proxy to Server

Remote procedure call (RPC) over HTTP v2 uses TCP/IP between the inbound or outbound proxy and the server. The same mapping is applied for both the inbound and the outbound proxy.

The TCP connection is initiated by the inbound or outbound proxy. The server name and port to be used for setting up the TCP connection are extracted from the URL of the HTTP request as specified in section 2.1.1.1. Once the connection is established, the inbound proxy or outbound proxy and the server use this connection for transmission of all the Protocol Data Units (PDUs) of the PDU stream.

By using a TCP/IP connection between the inbound or outbound proxy and the server, implementations of this protocol obtain a full-duplex channel for an unlimited number of bytes that provides reliable, in-order, at-most-once delivery semantics.

2.1.2.2.1 Legacy Server Response

A server SHOULD send the string literal "ncacn_http/1.0" to the inbound or outbound proxy as soon as the TCP connection from the inbound or outbound proxy to the server is established. This string literal is called the legacy server response.

2.2 Message Syntax

This section defines the message syntax for the messages and Protocol Data Units (PDUs) used by this protocol. First it specifies the conventions and some common data structures used in multiple messages. Then it defines the rules for combining the common data structures, and finally it defines the PDUs for this protocol.

2.2.1 Common Conventions

All data structures described in this section share certain the following common characteristics.

- All numeric fields MUST be encoded using little-endian byte ordering.
- Alignment for all data structures except the Uniformed Resource Identifier (URI) MUST be four bytes.

All structures in this section except the URI are used for Remote Procedure Call (RPC) over HTTP v2 only.

2.2.2 URI Encoding

The format of the Uniformed Resource Identifier (URI) header field of the HTTP request has a special interpretation in this protocol. As specified in [RFC2616], the URI is to be of the following form.

```
http_URL = "http:" "//" host [ ":" port ] [ abs_path
[ "?" query ]]
```

This protocol defines that abs_path MUST be present and have the following form.

```
nocert_path = "/rpc/proxy.dll"
withcert_path = "/rpcwithcert/rpcproxy.dll"

abs_path = nocert_path | withcert_path
```

The form matching withcert_path MUST be used whenever the HTTP Client authenticates to the HTTP Server using a client-side certificate. The form matching nocert_path MUST be used in all other cases.<14>

This protocol specifies that query MUST be present and MUST be of the following form.

```
query = server_name ":" server_port
```

The inbound proxy or outbound proxy uses the query string to establish a connection to a Remote Procedure Call (RPC) over the HTTP Server, as specified in sections 3.2.3.5.3 and 3.2.4.5.3.

```
server_name = DNS_Name | IP_literal_address |
IPv6_literal_address | NetBIOS_Name
server_port = 1*6(DIGIT)
```

The length of server_name MUST be less than 1,024 characters.

DNS_Name: MUST be a DNS name as specified in [RFC1034] section 3. IP_literal_address MUST be the string representation of an IP literal address as specified in [RFC1123] section 2.1.

IPv6_literal_address: MUST be the string representation of an IPv6 literal address as specified in [RFC3513] section 2.2.

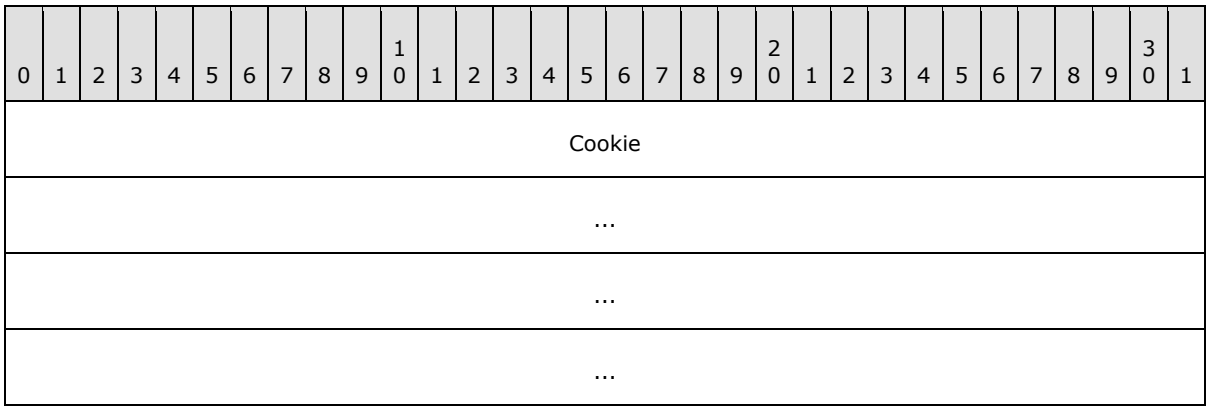
NetBIOS_Name: MUST be a NetBIOS name. For more details about NetBIOS, refer to [NETBEUI], [RFC1001], and [RFC1002].

2.2.3 Common Data Structures

This section defines several common data structures and values used by the RPC Over HTTP Protocol. They are used in multiple Protocol Data Units (PDUs). The PDUs themselves are defined in section 2.2.4. The common conventions for the messages are defined in section 2.2.1.

2.2.3.1 RTS Cookie

The RTS Cookie is a token exchanged between parties in a RPC Over HTTP Protocol sequence and is used to name objects and abstractions as defined throughout this document. This section defines the encoding for an **RTS cookie**.



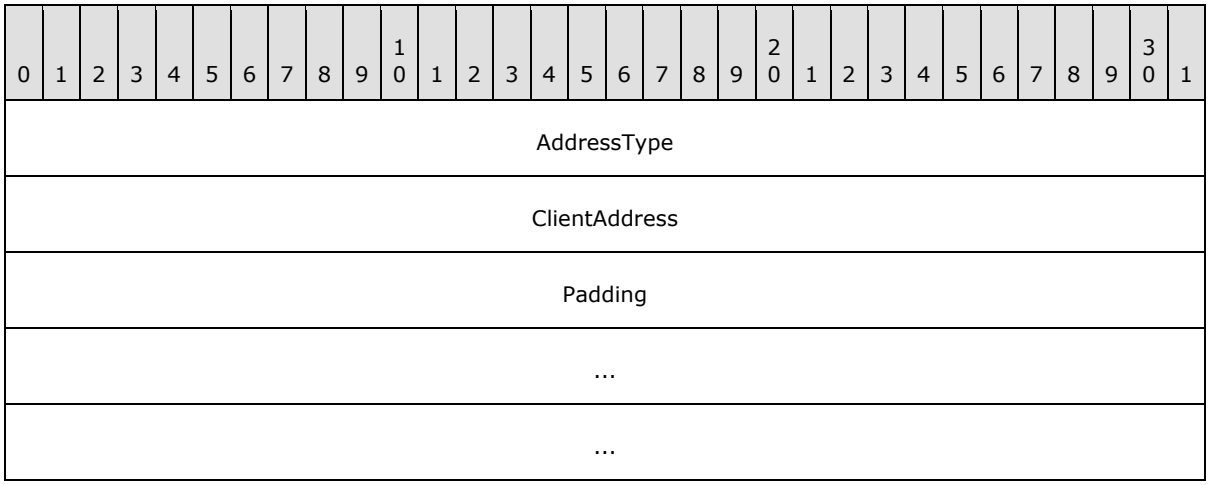
The value chosen for an RTS cookie SHOULD be a 16-byte cryptographically strong random number. It has the same uniqueness requirements as a Universally Unique Identifier (UUID), and implementations MAY use a UUID as the RTS cookie.<15>

2.2.3.2 Client Address

The client address data structure is used to transmit the IP address of a client to a proxy or a server. It has two basic formats, IPv4 and IPv6, as described in sections 2.2.3.2.1 and 2.2.3.2.2.

2.2.3.2.1 Client Address - IPv4

The client address data structure is used to transmit the IP address of a client to a proxy or a server. The encoding of the client address for the IPv4 format is as follows:



AddressType (4 bytes): This MUST be set to the value 0 to indicate IPv4 format.

ClientAddress (4 bytes): This MUST contain the IPv4 address of the client in little-endian byte order.

Padding (12 bytes): Senders SHOULD set all bytes in this field to the value 0x00. Receivers MUST ignore this field.

2.2.3.2.2 Client Address - IPv6

The client address data structure is used to transmit the IP address of a client to a proxy or a server. The encoding of the client address for the IPv6 format is as follows:

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
AddressType																															
ClientAddress																															
...																															
...																															
...																															
Padding																															
...																															
...																															

AddressType (4 bytes): This MUST be the value 1 to indicate IPv6 format.

ClientAddress (16 bytes): This MUST contain the IPv6 address of the client in little-endian byte order.

Padding (12 bytes): Senders SHOULD set all bytes in this field to the value 0x00. Receivers MUST ignore this.

2.2.3.3 Forward Destinations

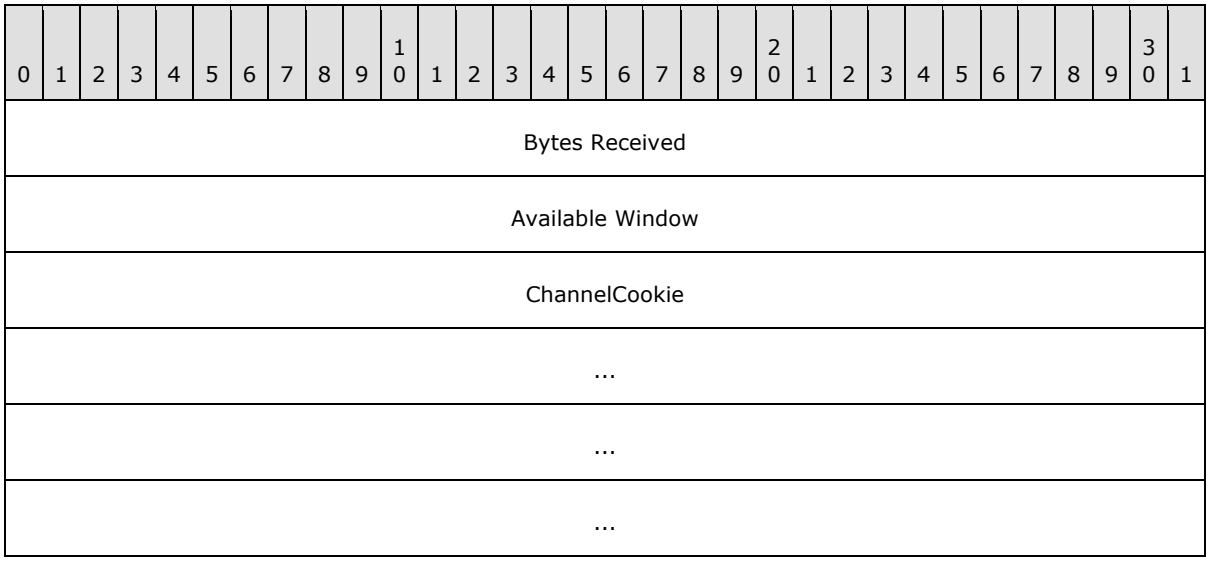
The forward destination enumeration specifies the target of a forwarded Protocol Data Unit (PDU) as per the following table.

Constant/value	Description
FDClient 0x00000000	Forward to client.
FDInProxy 0x00000001	Forward to inbound proxy.
FDServer 0x00000002	Forward to server.
FDOutProxy 0x00000003	Forward to outbound proxy.

If a PDU is forwarded, the party that originally created the PDU is called the originator of the PDU and the party that sends the PDU to the next hop in the forwarding chain is called the sender of the PDU. For a definition of the processing rules related to PDU forwarding, see section 3.2.1.4.2.

2.2.3.4 Flow Control Acknowledgment

The Flow Control Acknowledgment data structure is embedded in a packet performing some sort of flow control acknowledgment for traffic received. The encoding of this data structure is as follows:



Bytes Received (4 bytes): The number of bytes received at the time the flow control acknowledgment was issued. For a definition of the processing rules related to flow control acknowledgment, see section 3.2.1.1. This value MUST be in the inclusive range of 0 to the channel lifetime denoted by the channel cookie field.

Available Window (4 bytes): The number of bytes available in the **receive window** of the originator of this Protocol Data Unit (PDU).

ChannelCookie (16 bytes): An RTS cookie that uniquely identifies the channel for which the traffic received is being acknowledged (see section 2.2.3.1).

2.2.3.5 RTS Commands

The RTS protocol data units (PDUs) contain a series of commands. This section defines the valid RTS commands. Section 2.2.3.6 defines how the commands are ordered in a PDU.

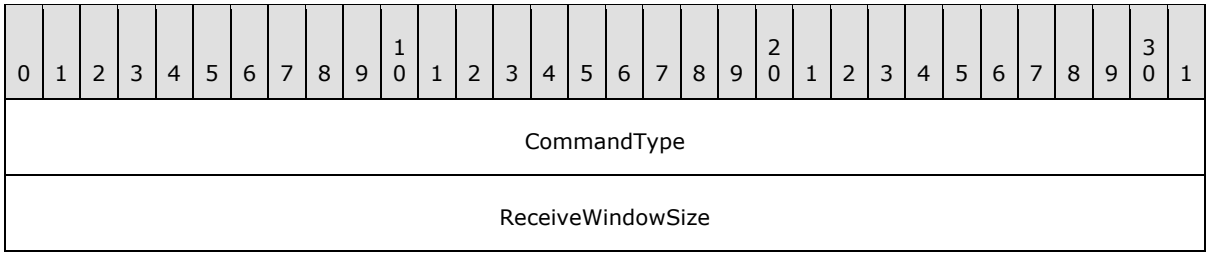
The type of each command in an RTS PDU is identified by a numeric value. Each command is used in one or more RTS PDUs as defined in sections 2.2.4.2 through 2.2.4.50. Section 3.2 defines when each RTS PDU is used, who sends it, and who receives it. The following table specifies the meaning and numeric value of each command type.

Value	Meaning
ReceiveWindowSize (0x00000000)	Command communicating the size of the receive window.
FlowControlAck (0x00000001)	Command carrying acknowledgment for traffic received.
ConnectionTimeout (0x00000002)	Command specifying the configured connection time out.

Value	Meaning
Cookie (0x00000003)	Command carrying an RTS cookie.
ChannelLifetime (0x00000004)	Command specifying the channel lifetime.
ClientKeepalive (0x00000005)	Command carrying desired interval for sending keep-alive PDUs.
Version (0x00000006)	Command carrying the remote procedure call (RPC) over HTTP v2 version number for the sender of the PDU that contains this command.
Empty (0x00000007)	Empty command.
Padding (0x00000008)	Variable-size command used to pad the size of an RTS PDU to a desired size.
NegativeANCE (0x00000009)	Command indicating that a **successor channel** was not established successfully.
ANCE (0x0000000A)	Command indicating that a successor channel was established successfully.
ClientAddress (0x0000000B)	Command that carries the client IP address. The IP address is encoded as specified in section 2.2.3.2. Regardless of who sends this PDU, the address MUST be interpreted to be the address of the client.
AssociationGroupId (0x0000000C)	Command that carries the client association group ID as specified in section 2.2.3.5.13. Regardless of who sends this PDU, the association group ID MUST be interpreted to be that of the client.
Destination (0x0000000D)	Command that carries the destination to which a PDU MUST be forwarded.
PingTrafficSentNotify (0x0000000E)	Command that carries the number of bytes sent by the outbound proxy to the client as part of ping traffic.

2.2.3.5.1 Receive Window Size

The Receive Window Size command specifies the size of the receive window of a party. The party from which the receive window originated is specified in the section for the RTS protocol data unit (PDU) that contains this command. The structure of the command is as follows:

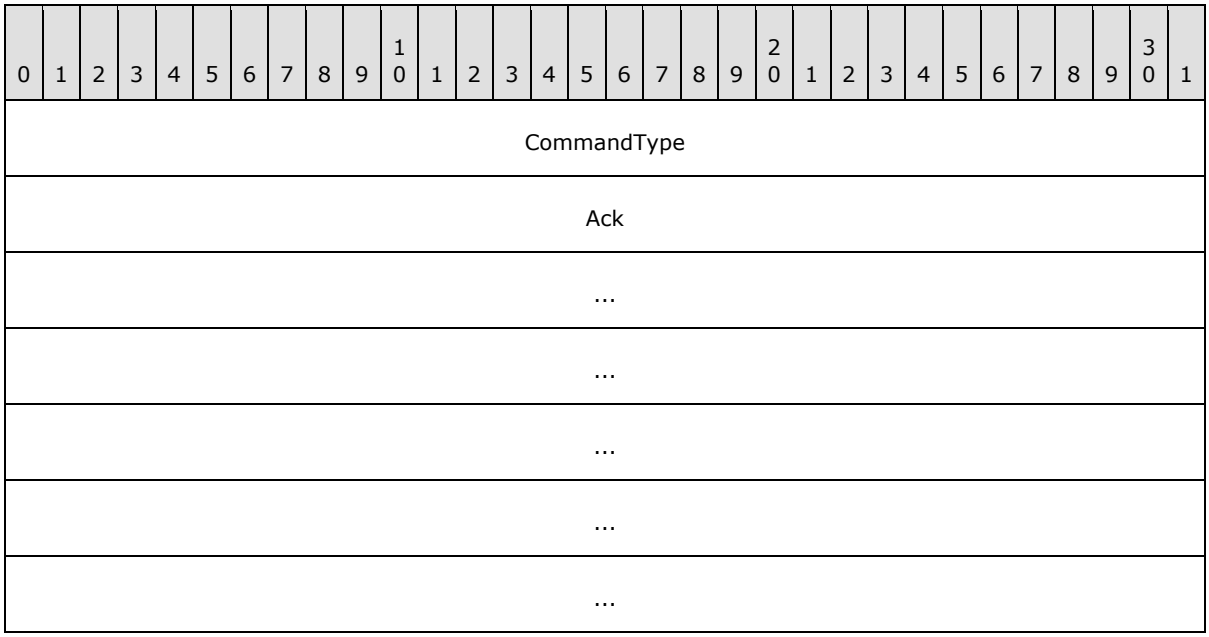


CommandType (4 bytes): This MUST be the value ReceiveWindowSize (0x00000000).

ReceiveWindowSize (4 bytes): The size of the receive window, in bytes. It MUST be in the inclusive range of 8 KB to 256 KB. The receive window MUST be greater than or equal to the PDU fragment size transmitted in the bind/bind_ack packets at the Remote Procedure Call (RPC) layer ([C706] section 12.4).<16>

2.2.3.5.2 Flow Control Acknowledgment

The Flow Control Acknowledgment command specifies acknowledgment for traffic received. The structure of the command is as follows:

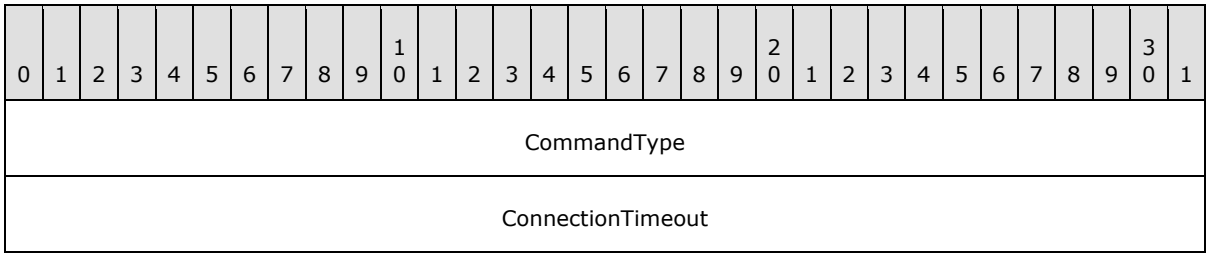


CommandType (4 bytes): MUST be the value FlowControlAck (0x00000001).

Ack (24 bytes): This MUST be a flow control acknowledgment structure as defined in section 2.2.3.4.

2.2.3.5.3 Connection Timeout

The Connection Timeout command specifies the desired frequency for sending keep-alive Protocol Data Units (PDUs) generated by this protocol as defined in section 3.2. The party from which the connection time out originated is specified in the section for the RTS PDU that contains this command.

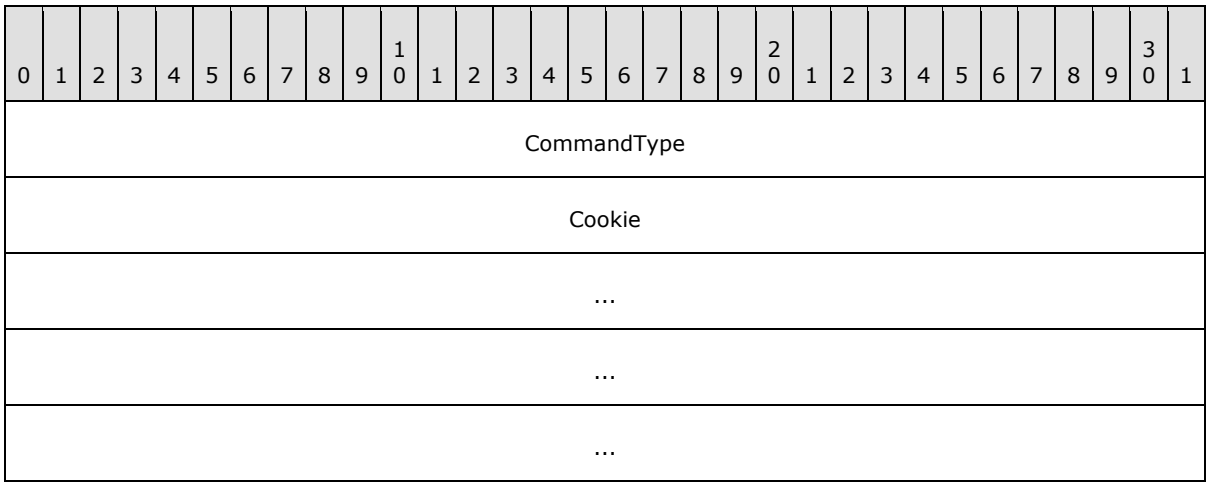


CommandType (4 bytes): This MUST be the value ConnectionTimeout (0x00000002).

ConnectionTimeout (4 bytes): This MUST be the integer value for the client keep-alive that this connection is configured to use in milliseconds. The value MUST be in the inclusive range of 120,000 to 14,400,000 milliseconds.

2.2.3.5.4 Cookie

The Cookie command specifies an RTS cookie. The meaning of the RTS cookie is inferred from its position in the command sequence as specified in section 2.2.4 and the context established by the protocol sequence as defined in section 3.2.

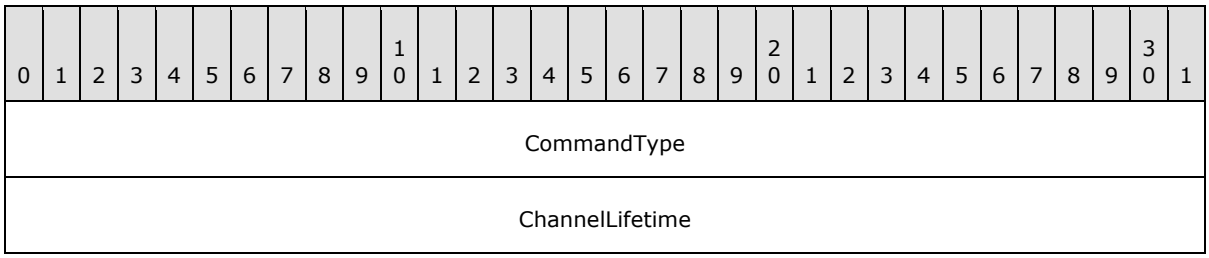


CommandType (4 bytes): This MUST be the value Cookie (0x00000003).

Cookie (16 bytes): This MUST contain an RTS cookie, which is specified in 2.2.3.1.

2.2.3.5.5 Channel Lifetime

The Channel Lifetime command specifies the channel lifetime. The party from which the channel lifetime originated is specified in the sections that define the RTS protocol data unit (PDU) that contains this command.

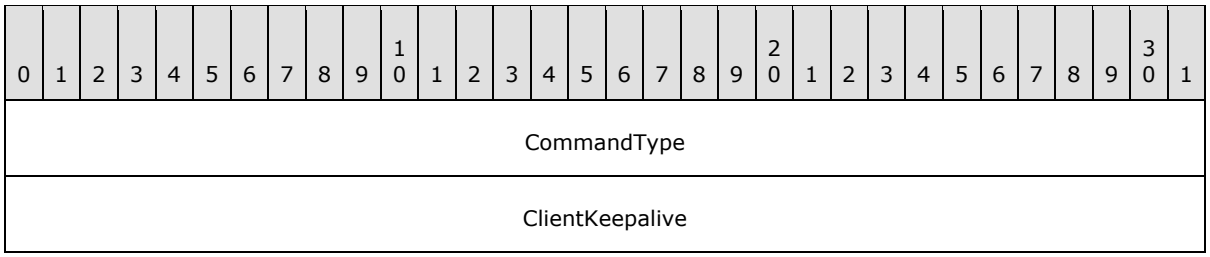


CommandType (4 bytes): This MUST be the value ChannelLifetime (0x00000004).

ChannelLifetime (4 bytes): The channel lifetime, in bytes. This value MUST be in the inclusive range of 128 KB to 2 GB.<17>

2.2.3.5.6 Client Keepalive

The Client Keep-Alive command carries the desired interval for sending keep-alive Protocol Data Units (PDUs) on behalf of the client whose usage is defined in section 3.2. The party from which the client keep-alive originated is specified in the sections that define the RTS PDU that contains this command.

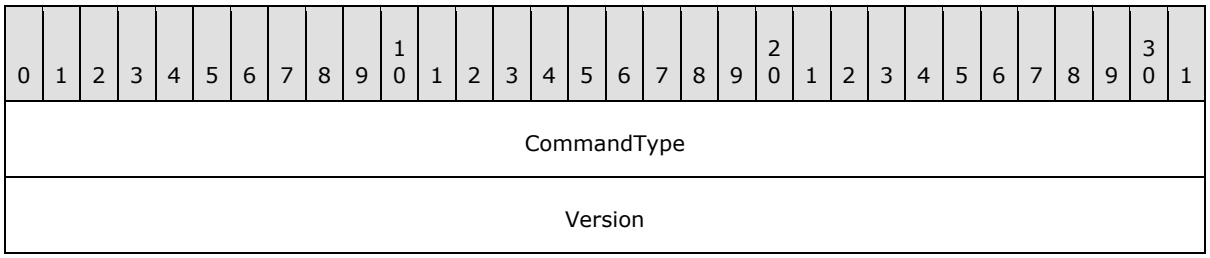


CommandType (4 bytes): This MUST be the value ClientKeepalive (0x00000005).

ClientKeepalive (4 bytes): An unsigned integer that specifies the keep-alive interval, in milliseconds, that this connection is configured to use. This value MUST be 0 or in the inclusive range of 60,000 to 4,294,967,295. If it is 0, it MUST be interpreted as 300,000.

2.2.3.5.7 Version

The Version command specifies an RPC over HTTP v2 version number. This version number allows versioning within RPC over HTTP v2. Version information MUST be interpreted to refer to the sender of the Protocol Data Unit (PDU).

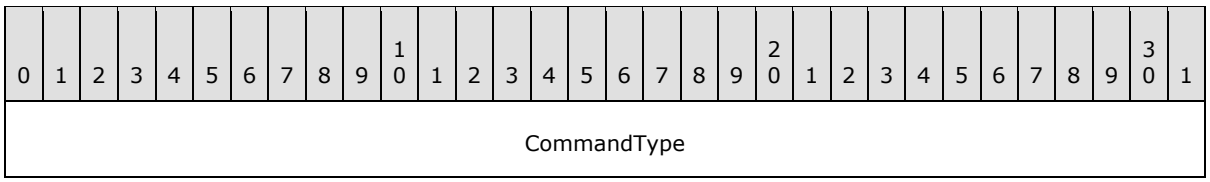


CommandType (4 bytes): This MUST be the value Version (0x00000006).

Version (4 bytes): An unsigned integer that specifies the version of RPC over HTTP v2 that the sender of the PDU will use. Implementation of this protocol SHOULD set this to 1 on sending and MUST ignore it on receiving.

2.2.3.5.8 Empty

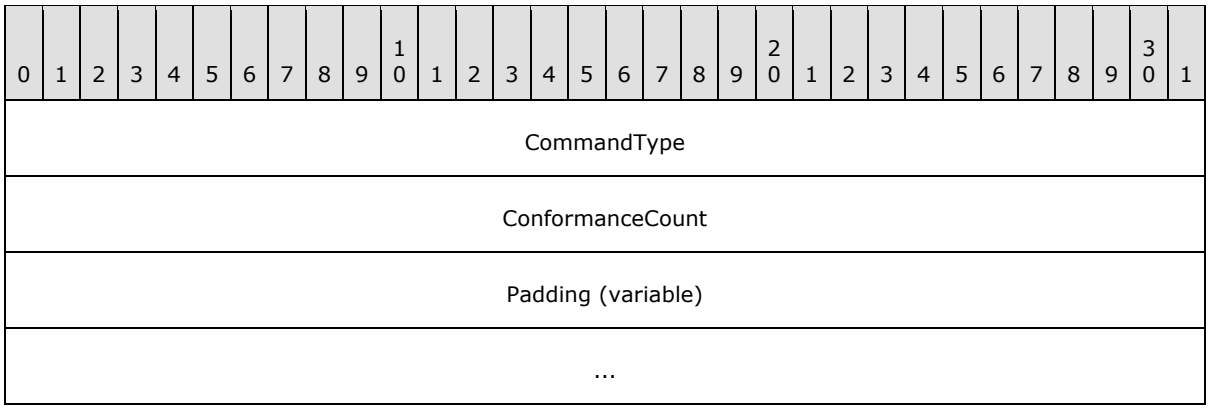
The Empty command specifies an empty command with no contents. Its meaning is context-specific and is defined in section 3.2.



CommandType (4 bytes): This MUST be the value Empty (0x00000007).

2.2.3.5.9 Padding

The Padding command is a variable-size command that may be used to pad the size of an RTS protocol data unit (PDU) to a desired size, as specified in section 2.2.4.45.



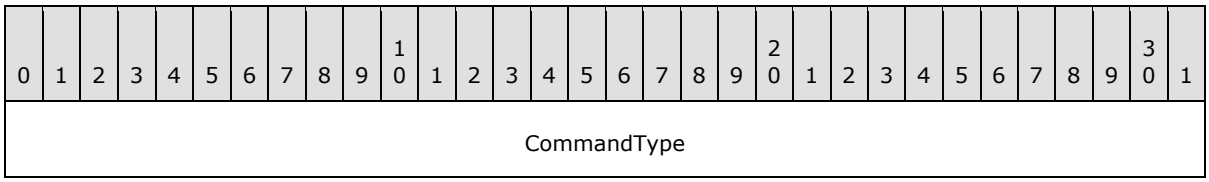
CommandType (4 bytes): This MUST be the value Padding (0x00000008).

ConformanceCount (4 bytes): The size of the padding field, in bytes. It MUST be in the inclusive range of 0 to 0xFFFF.

Padding (variable): An array of padding bytes that is **ConformanceCount** bytes long. Protocol implementations SHOULD initialize padding bytes to zero on sending and MUST ignore them on receiving.

2.2.3.5.10 NegativeANCE

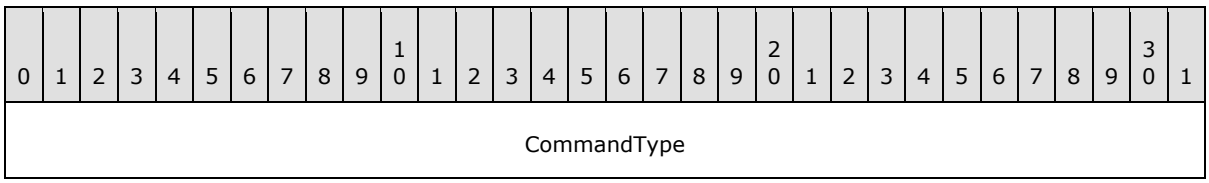
The NegativeANCE command specifies that a successor channel was not established successfully. The meaning of the command is context-specific and is defined in section 3.2.



CommandType (4 bytes): This MUST be the value NegativeANCE (0x00000009).

2.2.3.5.11 ANCE

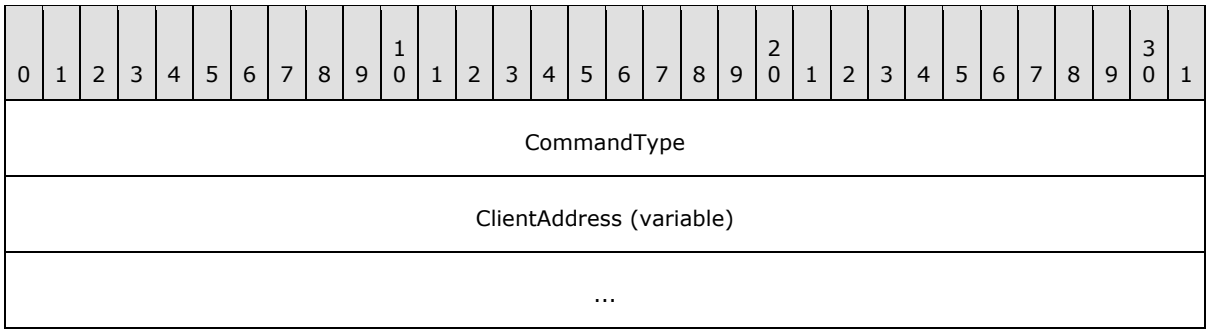
The ANCE command specifies that a successor channel was established successfully. The meaning of the command is context-specific and is defined in section 3.2.



CommandType (4 bytes): This MUST be the value ANCE (0x0000000A).

2.2.3.5.12 Client Address

The Client Address command specifies the IP address of the client. Regardless of who sends this Protocol Data Unit (PDU), the address MUST be interpreted to be the address of the client.

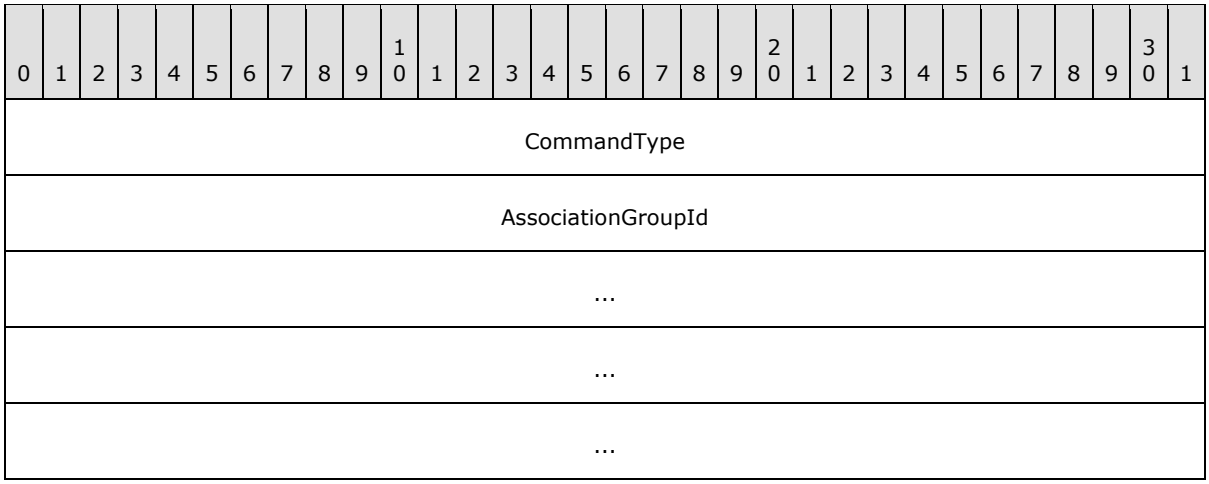


CommandType (4 bytes): This MUST be the value ClientAddress (0x0000000B).

ClientAddress (variable): This MUST contain the address of the client and is encoded as defined in section 2.2.3.2.

2.2.3.5.13 AssociationGroupId

The AssociationGroupId command specifies the client association group ID. For each association on the Remote Procedure Call (RPC) protocol level, the client MUST associate exactly one RTS cookie when the RPC protocol sequence is ncacn_http and this RTS cookie is sent with this command. For more information on association and association group ID, see [C706] and [MS-RPCE] section 3.3.1.1.1. Regardless of who sends this PDU, the association group ID MUST be interpreted to be that of the client.

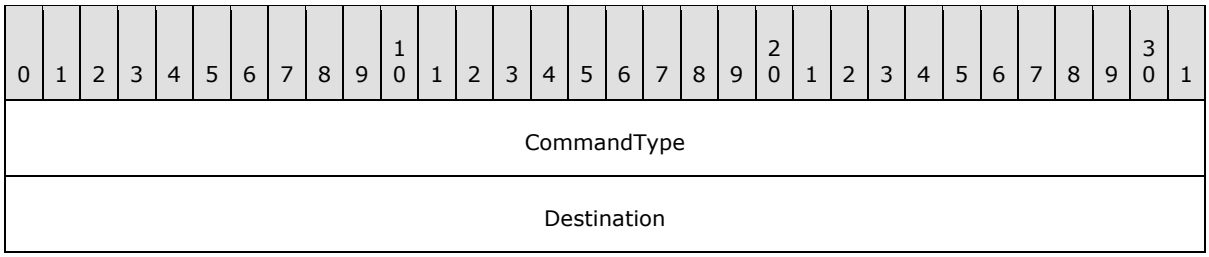


CommandType (4 bytes): This MUST be the value AssociationGroupId (0x0000000C).

AssociationGroupId (16 bytes): This field MUST be encoded as an RTS cookie that the client generated for this association as explained in this section. It is encoded as defined in section 2.2.3.1.

2.2.3.5.14 Destination

The Destination command specifies the destination to which a Protocol Data Unit (PDU) that carries this command MUST be forwarded.

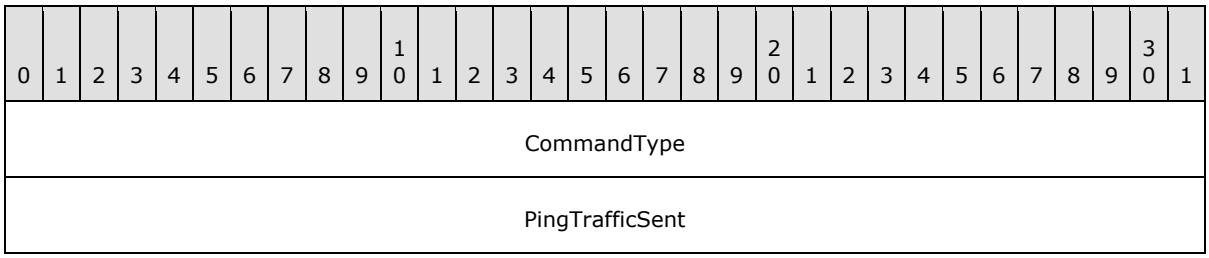


CommandType (4 bytes): This MUST be the value Destination (0x0000000D).

Destination (4 bytes): This MUST be one of the values defined in section 2.2.3.3. For more details about PDU forwarding, see section 3.2.1.4.2.

2.2.3.5.15 PingTrafficSentNotify

The PingTrafficSentNotify command specifies the number of bytes sent by the outbound proxy to the client as part of ping traffic. It is sent from an outbound proxy to the server and notifies the server that the outbound proxy has sent the specified number of bytes to the client as part of pinging the client.



CommandType (4 bytes): This MUST be the value PingTrafficSentNotify (0x0000000E).

PingTrafficSent (4 bytes): This MUST be the number of bytes sent by the outbound proxy. Servers SHOULD impose an implementation-specific reasonable upper bound on this value.<18>

2.2.3.6 RTS PDU Structure

The RTS protocol data unit (PDU) MUST be composed of exactly one header and one or more RTS commands defined in section 2.2.3.5 in the RTS PDU body. The following diagram illustrates the structure.

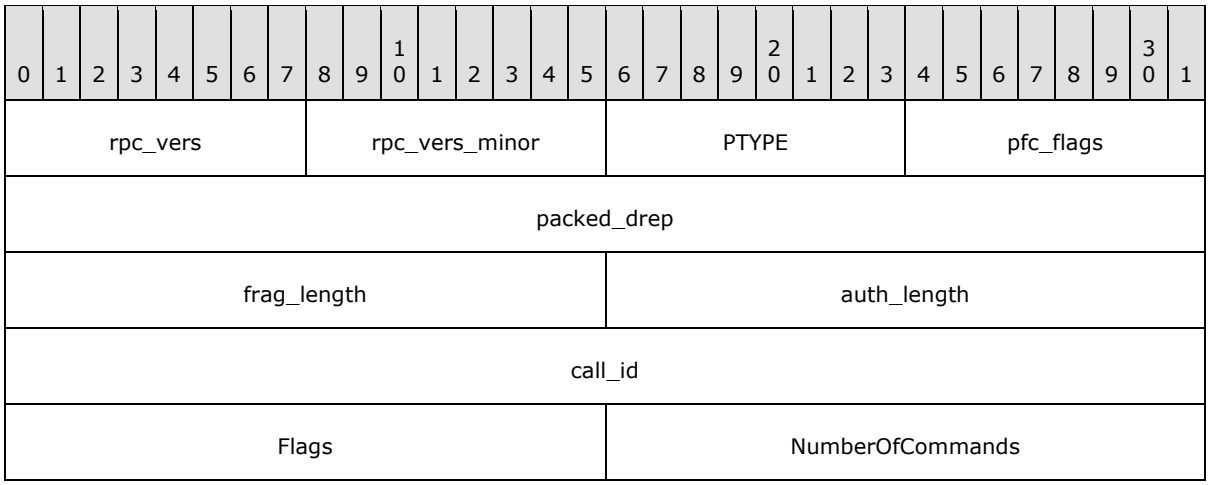
Figure 12: RTS PDU structure

2.2.3.6.1 RTS PDU Header

The RTS PDU Header has the same layout as the common header of the connection-oriented Remote Procedure Call (RPC) Protocol Data Unit (PDU) as specified in [C706] section 12.6.1, with a few additional requirements around the contents of the header fields. The additional requirements are as follows:

- All fields MUST use little-endian byte order.
- Fragmentation MUST NOT occur for an RTS PDU.
- PFC_FIRST_FRAG and PFC_LAST_FRAG MUST be present in all RTS PDUs, and all other PFC flags MUST NOT be present.
- The rpc_vers and rpc_vers_minor fields MUST contain version information as specified in [MS-RPCE] section 1.7.
- PTYPE MUST be set to a value of 20. This field differentiates RTS packets from other RPC packets.
- The packed_drep MUST indicate little-endian integer and floating-pointer byte order, IEEE float-point format representation, and ASCII character format as specified in [C706] section 12.6.
- The auth_length MUST be set to 0.
- The frag_length field MUST reflect the size of the header + the size of all commands, including the variable portion of variable-sized commands.
- The call_id MUST be set to 0 by senders and MUST be 0 on receipt.

This protocol adds two more fields to the RTS PDU header that MUST be present immediately after the common header. The following diagram specifies the header format.



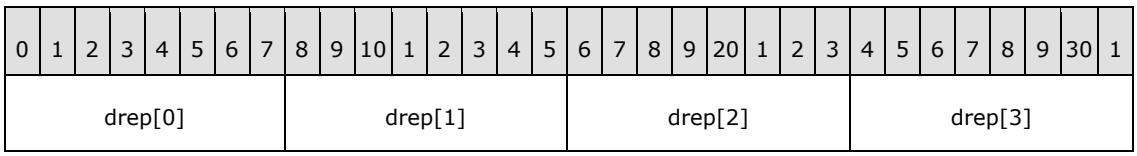
rpc_vers (1 byte): As specified in [C706] section 12.6.1, with additional requirements specified above.

rpc_vers_minor (1 byte): As specified in [C706] section 12.6.1, with additional requirements specified above.

PTYPE (1 byte): As specified in [C706] section 12.6.1, with additional requirements specified above.

pfc_flags (1 byte): As specified in [C706] section 12.6.1, with additional requirements specified above.

packed_drep (4 bytes): As specified in [C706] section 12.6.1, with additional requirements specified above. **packed_drep** takes the following form:



frag_length (2 bytes): As specified in [C706] section 12.6.1, with additional requirements specified above.

auth_length (2 bytes): As specified in [C706] section 12.6.1, with additional requirements specified above.

call_id (4 bytes): As specified in [C706] section 12.6.1, with additional requirements specified above.

Flags (2 bytes): The flags field MUST contain one or more of the following flags. The valid combination of flags for each RTS PDU is defined in section 2.2.4 of this document. The table below is meant to define numeric values for each flag and as an aid in understanding this specification, and to convey the general context in which a given flag is used. Precise definition on what flags MUST be used for each RTS PDU MUST be obtained from the section for the respective RTS PDU in section 2.2.4. An implementation MUST NOT change the flags in the RTS PDU as defined in the respective RTS PDU section within section 2.2.4.

Value	Meaning
RTS_FLAG_NONE 0x0000	No special flags.
RTS_FLAG_PING 0x0001	Proves that the sender is still active, and can also be used to flush the pipeline by the other party.
RTS_FLAG_OTHER_CMD 0x0002	Indicates that the PDU contains a command that cannot be defined by the other flags in this table.
RTS_FLAG_RECYCLE_CHANNEL 0x0004	Streamlines processing of some RTS PDUs associated with recycling a channel.
RTS_FLAG_IN_CHANNEL 0x0008	Streamlines processing of some RTS PDUs associated with IN channel communications.
RTS_FLAG_OUT_CHANNEL 0x0010	Streamlines processing of some RTS PDUs associated with OUT channel.
RTS_FLAG_EOF 0x0020	Indicates that this is the last PDU on an IN channel or OUT channel. Not all channels, however, use this to indicate the last PDU.
RTS_FLAG_ECHO 0x0040	Signifies that this PDU is an **echo request** or response.

NumberOfCommands (2 bytes): An implementation MUST set this field to be equal to the number of commands in the RTS PDU body.

2.2.3.6.2 RTS PDU Body

The RTS protocol data unit (PDU) body MUST be composed of one or more RTS commands. The first command MUST be placed immediately after the RTS PDU header and each subsequent command MUST be placed immediately after the previous command without any padding or delimiters until all commands in the PDU are placed. The order of commands in the RTS PDU body is significant from a protocol perspective, and implementations MUST follow the rules in this document about command ordering as specified in section 2.2.4.

2.2.4 RTS PDUs

This protocol defines specific combinations of PDU commands that are combined into single PDUs. These PDUs are referred to as RTS PDUs and form the basis of routing and control flow in RPC over HTTP v2.

This section defines the syntax of the RPC PDUs using the common structure and command definitions specified earlier in this section.

2.2.4.1 RTS PDUs Naming and Document Conventions

All definitions in this section share some common naming conventions. An RTS PDU can be one of three types. It can be used by a single protocol sequence only; it can be used in more than one protocol sequence; or, it can be used outside a protocol sequence. If the RTS PDU is specific to a single protocol sequence, the name of the PDU is created by using a strict convention that allows for an RTS PDU to be associated quickly with its place in the protocol sequence. The name of the RTS PDU is not reflected on the network and thus has no protocol significance other than making it

easier to find and understand information in this document. The name of this type of RTS PDU follows the format:

```
RTS PDU name = protocol_sequence_name "/" group_name
 group_order
protocol_sequence_name = "CONN" | "IN_R1" | "IN_R2" | "OUT_R1" |
 "OUT_R2"
group_name = "A" | "B" | "C"
group_order = 1*(DIGIT)
```

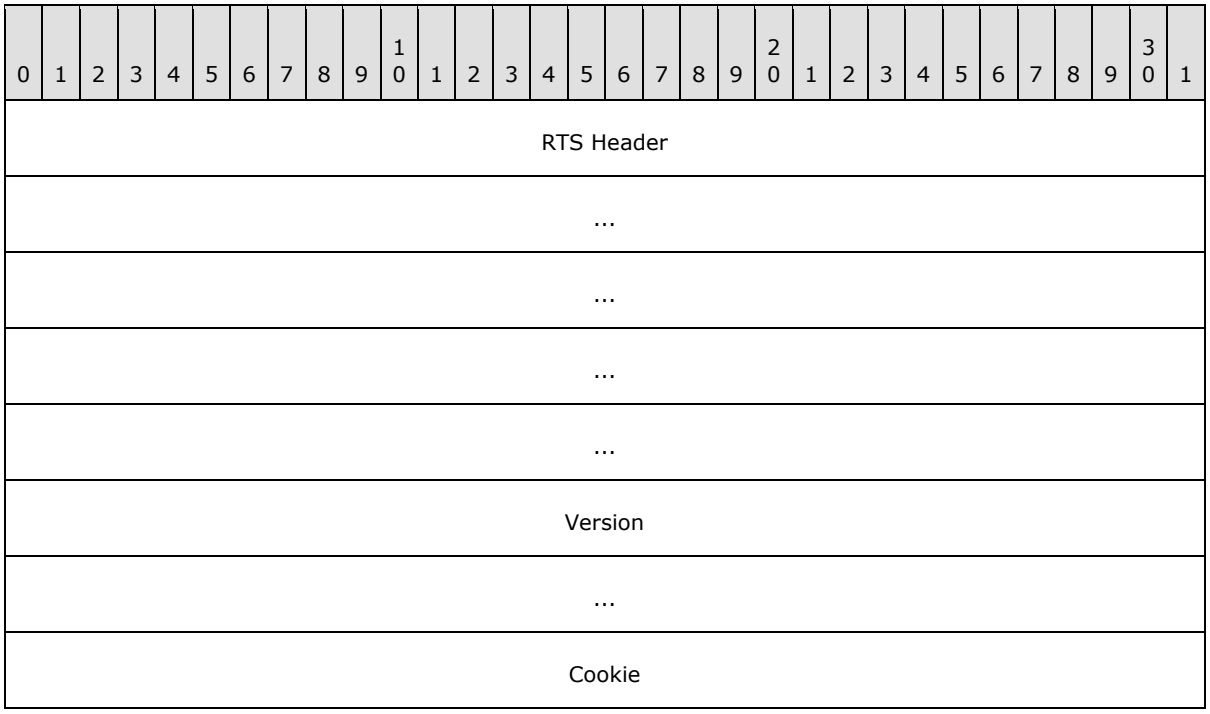
The names of the protocol sequences are given in sections 3.2.1.4.3.2 through 3.2.1.4.3.6 of this document. The group_name is a group of PDUs within the protocol sequence and the name and meaning of the group is defined in the section for the respective protocol sequence. The group order is a number that starts at 1 and is incremented sequentially for each RTS PDU in the group. For example, CONN/A1 is the first RTS PDU from group A from protocol sequence CONN.

If an RTS PDU is used in more than one protocol sequence or is used outside a protocol sequence, the convention defined above is not used. The name of the PDU is descriptive of the meaning of the PDU and is not associated in any way with the protocol sequences in which it is used.

As defined in section 2.2.3.6, an RTS PDU is composed of an RTS PDU header and one or more RTS PDU commands. To facilitate reading this document, the packet diagrams in this section use a lighter shade of grayscale for odd commands and a darker shade of grayscale for even commands of the RTS PDU.

2.2.4.2 CONN/A1 RTS PDU

The CONN/A1 RTS PDU MUST be sent from the client to the outbound proxy on the OUT channel to initiate the establishment of a virtual connection.



...
...
...
...
OUTChannelCookie
...
...
...
...
ReceiveWindowSize
...

RTS Header (20 bytes): See section 2.2.3.6.1. The **Flags** field of RTS Header MUST be the value of RTS_FLAG_NONE. The **NumberOfCommands** field of RTS Header MUST be the value 4.

Version (8 bytes): This MUST be a version command indicating the RPC over HTTP v2 Protocol version as specified in section 2.2.3.5.7.

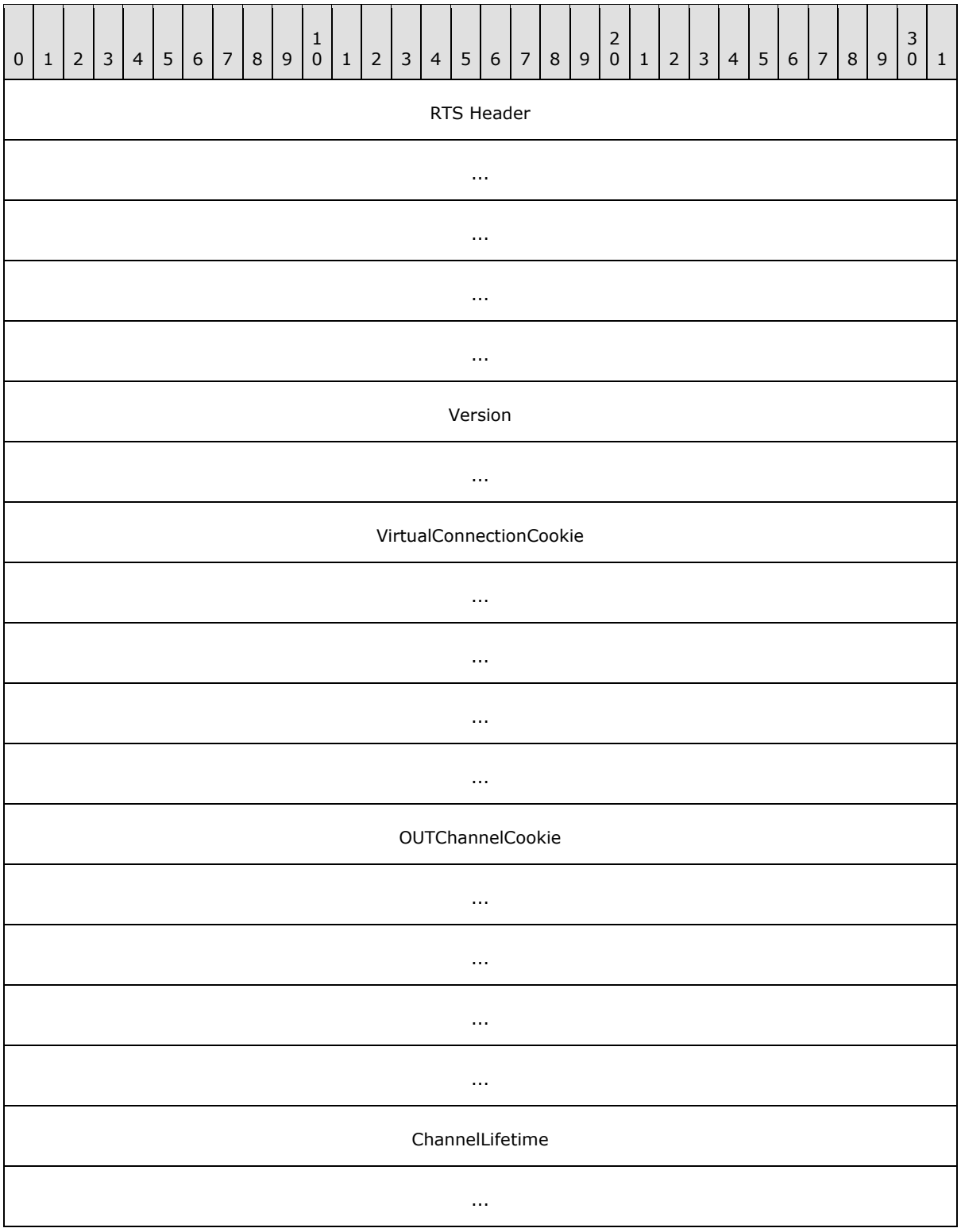
Cookie (20 bytes): MUST be a cookie command identifying the virtual connection that is being established by this protocol sequence. The cookie command format is defined in section 2.2.3.5.4.

OUTChannelCookie (20 bytes): MUST be a cookie command identifying the OUT channel that this protocol sequence is trying to establish. The cookie command format is defined in section 2.2.3.5.4.

ReceiveWindowSize (8 bytes): MUST be a ReceiveWindowSize command containing the size of the receive window for the clientOUT channel. The ReceiveWindowSize command format is defined in section 2.2.3.5.1.

2.2.4.3 CONN/A2 RTS PDU

The CONN/A2 RTS PDU MUST be sent from the outbound proxy to the server on the OUT channel to initiate the establishment of a virtual connection.



ReceiveWindowSize
...

RTS Header (20 bytes): See section 2.2.3.6.1. The **Flags** field of the RTS Header MUST be the value RTS_FLAG_OUT_CHANNEL. The **NumberOfCommands** field of RTS Header MUST be the value 5.

Version (8 bytes): This MUST be a version command containing the lower of the outbound proxy version and the client version reported in CONN/A1 RTS PDU. The format for the RPC over HTTP v2 Protocol version command is defined in section 2.2.3.5.7.

VirtualConnectionCookie (20 bytes): MUST be a cookie command identifying the virtual connection that this protocol sequence is trying to establish. The cookie command format is defined in section 2.2.3.5.4.

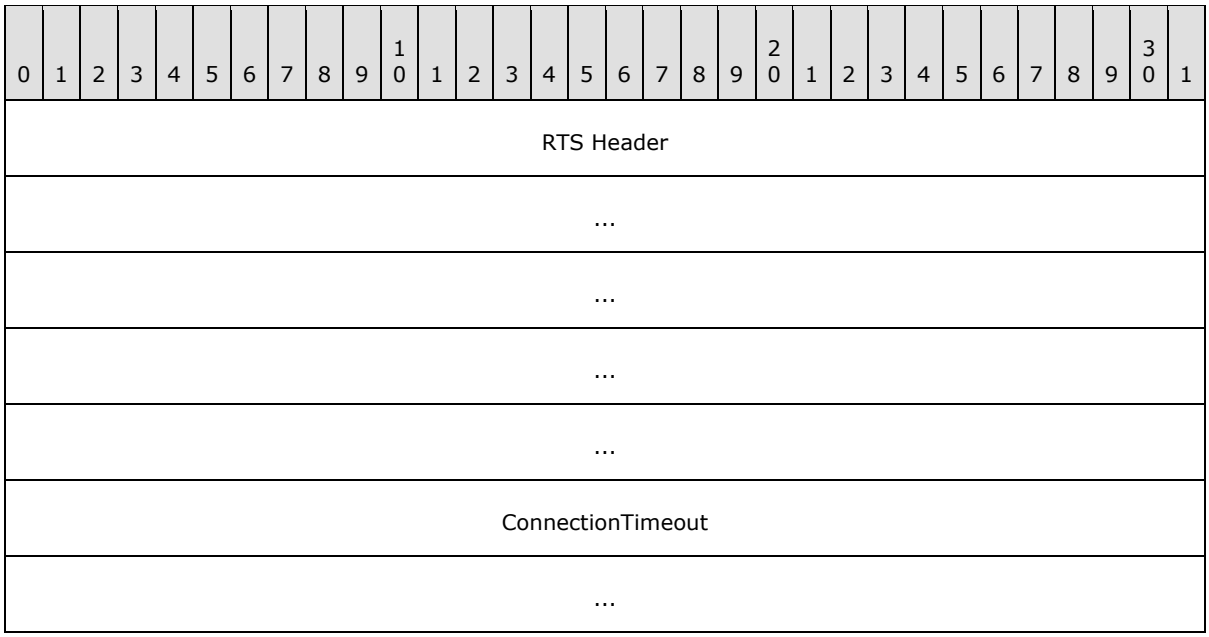
OUTChannelCookie (20 bytes): MUST be a cookie command for the OUT channel that this protocol sequence is trying to establish. The cookie command format is defined in section 2.2.3.5.4.

ChannelLifetime (8 bytes): MUST be a ChannelLifetime command containing the lifetime, in bytes, of the OUT channel from the outbound proxy to the client. The ChannelLifetime command format is defined in section 2.2.3.5.5.

ReceiveWindowSize (8 bytes): MUST be a ReceiveWindowSize command containing the size of the receive window for the OUT channel to the proxy. The ReceiveWindowSize command format is defined in section 2.2.3.5.1.

2.2.4.4 CONN/A3 RTS PDU

The CONN/A3 RTS PDU MUST be sent from the outbound proxy to the client on the OUT channel to continue the establishment of the virtual connection.

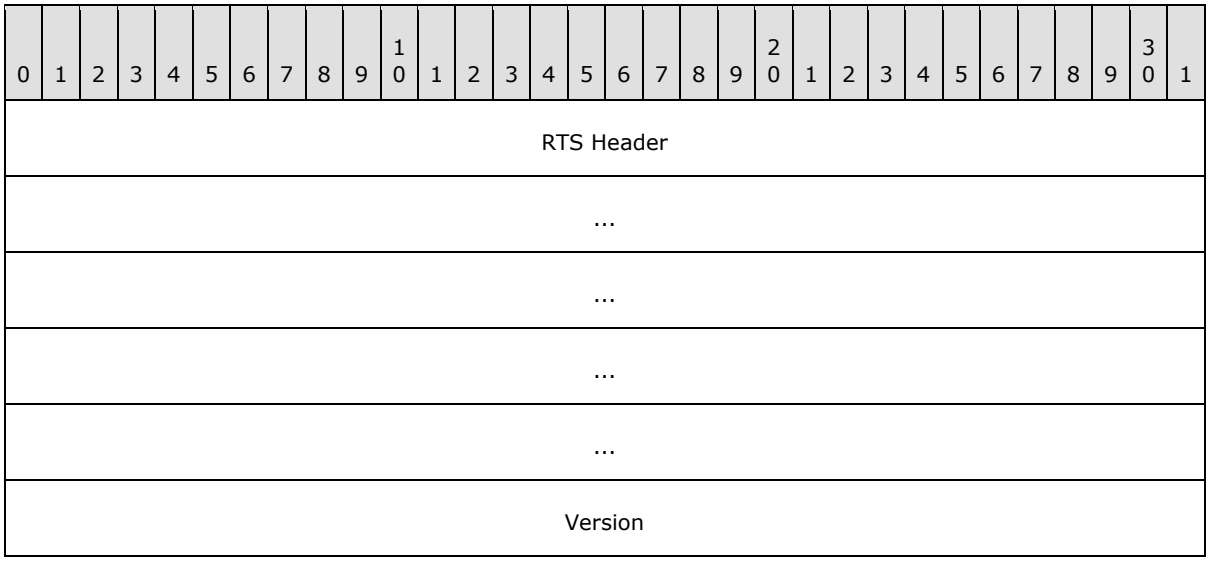


RTS Header (20 bytes): See section 2.2.3.6.1. The **Flags** field of the RTS Header MUST be the value RTS_FLAG_NONE. The **NumberOfCommands** field of RTS Header MUST be the value 1.

ConnectionTimeout (8 bytes): MUST be a ConnectionTimeout command containing the connection time out for the OUT channel between the outbound proxy and the client. The ConnectionTimeout command format is defined in section 2.2.3.5.3.

2.2.4.5 CONN/B1 RTS PDU

The CONN/B1 RTS PDU MUST be sent from the client to the inbound proxy on the IN channel to initiate the establishment of a virtual connection.



...
VirtualConnectionCookie
...
...
...
...
INChannelCookie
...
...
...
...
ChannelLifetime
...
ClientKeepalive
...
AssociationGroupId
...
...
...
...

RTS Header (20 bytes): See section 2.2.3.6.1. The **Flags** field of the RTS Header MUST be the value RTS_FLAG_NONE. The **NumberOfCommands** field of RTS Header MUST be the value 6.

Version (8 bytes): This MUST be a Version command containing the version of RPC over HTTP v2 that the client supports, formatted as specified in section 2.2.3.5.7.

VirtualConnectionCookie (20 bytes): MUST be a cookie command identifying the virtual connection that this protocol sequence is trying to establish. The cookie command format is defined in section 2.2.3.5.4.

INChannelCookie (20 bytes): MUST be a cookie command identifying the IN channel cookie that this protocol sequence is trying to establish. The cookie command format is defined in section 2.2.3.5.4.

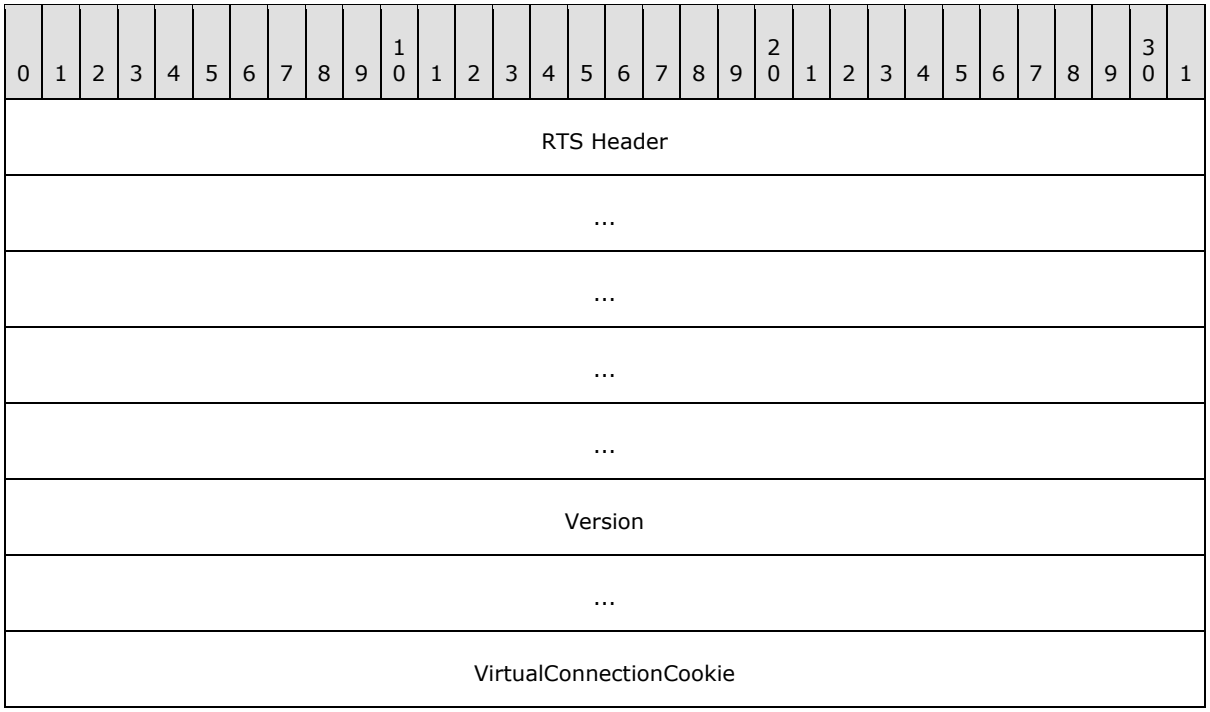
ChannelLifetime (8 bytes): MUST be a ChannelLifetime command containing the lifetime in bytes of the IN channel from the client to the inbound proxy. The ChannelLifetime command format is defined in 2.2.3.5.5. This field is used for troubleshooting only and has no protocol significance. Inbound proxies SHOULD ignore the value of this field.

ClientKeepalive (8 bytes): MUST be a ClientKeepalive command containing the keep-alive interval that the client wants the outbound proxy to use on the OUT channel between the outbound proxy and the server. The ClientKeepalive command format is defined in section 2.2.3.5.6.

AssociationGroupId (20 bytes): MUST be an AssociationGroupId command containing the association group ID for the client. The AssociationGroupId command format is defined in section 2.2.3.5.13.

2.2.4.6 CONN/B2 RTS PDU

The CONN/B2 RTS PDU MUST be sent from the inbound proxy to the server on the IN channel to initiate the establishment of a virtual connection.



...
...
...
...
INChannelCookie
...
...
...
...
ReceiveWindowSize
...
ConnectionTimeout
...
AssociationGroupId
...
...
...
...
ClientAddress (variable)
...

RTS Header (20 bytes): See section 2.2.3.6.1. The **Flags** field of the RTS Header MUST be the value RTS_FLAG_IN_CHANNEL. The **NumberOfCommands** field of RTS Header MUST be the value 7.

Version (8 bytes): This MUST be a Version command containing the lower of the inbound proxy version and the client version reported in CONN/B1 RTS PDU. The format for the RPC over HTTP v2 protocol version command is defined in section 2.2.3.5.7.

VirtualConnectionCookie (20 bytes): MUST be a cookie command for the virtual connection this protocol sequence is trying to establish. The cookie command format is defined in section 2.2.3.5.4.

INChannelCookie (20 bytes): This MUST be a cookie command for the IN channel that this protocol sequence is trying to establish. The cookie command format is defined in section 2.2.3.5.4.

ReceiveWindowSize (8 bytes): This MUST be a ReceiveWindowSize command containing the size of the receive window for the IN channel to the inbound proxy. The ReceiveWindowSize command format is defined in section 2.2.3.5.1.

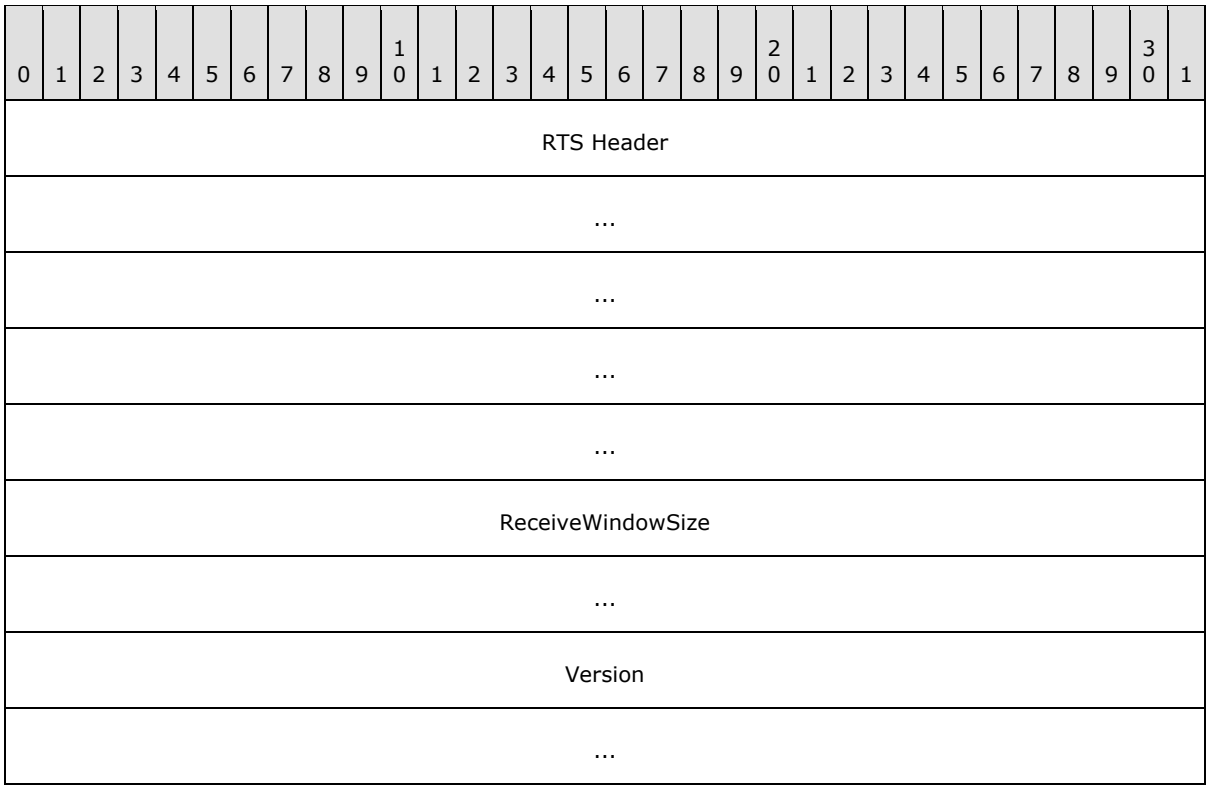
ConnectionTimeout (8 bytes): This MUST be a ConnectionTimeout command containing the connection time out for the IN channel between the inbound proxy and the client. The ConnectionTimeout command format is defined in section 2.2.3.5.3.

AssociationGroupId (20 bytes): This MUST be an AssociationGroupId command containing the association group ID for the client. The AssociationGroupId command format is defined in section 2.2.3.5.13.

ClientAddress (variable): This MUST be a ClientAddress command containing the IP address of the client as seen by the inbound proxy. The ClientAddress command format is defined in section 2.2.3.5.12.

2.2.4.7 CONN/B3 RTS PDU

The CONN/B3 RTS PDU MUST be sent from the server to the inbound proxy on the IN channel to notify it that the establishment of a virtual connection has been completed.



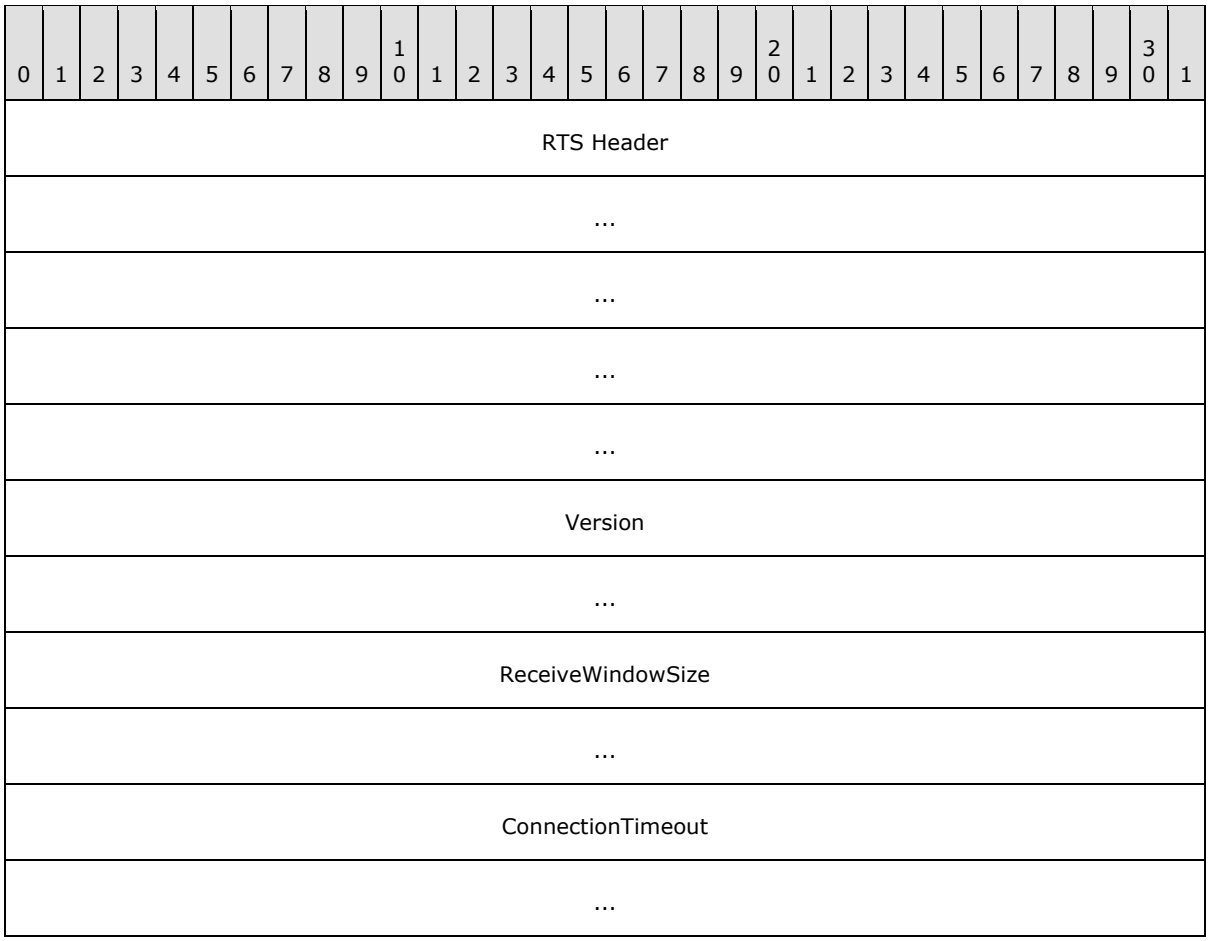
RTS Header (20 bytes): See section 2.2.3.6.1. The **Flags** field of the RTS Header MUST be the value RTS_FLAG_NONE. The **NumberOfCommands** field of RTS Header MUST be the value 2.

ReceiveWindowSize (8 bytes): This MUST be a ReceiveWindowSize command containing the size of the receive window for the server IN channel. The ReceiveWindowSize command format is defined in section 2.2.3.5.1.

Version (8 bytes): This MUST be a Version command containing the lowest of the CONN/B2 RTS PDU (section 2.2.4.6) version, the CONN/A2 RTS PDU (section 2.2.4.3) version, and the server RPC over HTTP v2 version. The format for the RPC over HTTP v2 Protocol version command is defined in section 2.2.3.5.7.

2.2.4.8 CONN/C1 RTS PDU

The CONN/C1 RTS PDU MUST be sent from the server to the outbound proxy on the OUT channel to notify it that the establishment of a virtual connection has been completed.



RTS Header (20 bytes): See section 2.2.3.6.1. The **Flags** field of the RTS Header MUST be the value RTS_FLAG_NONE. The **NumberOfCommands** field of RTS Header MUST be the value 3.

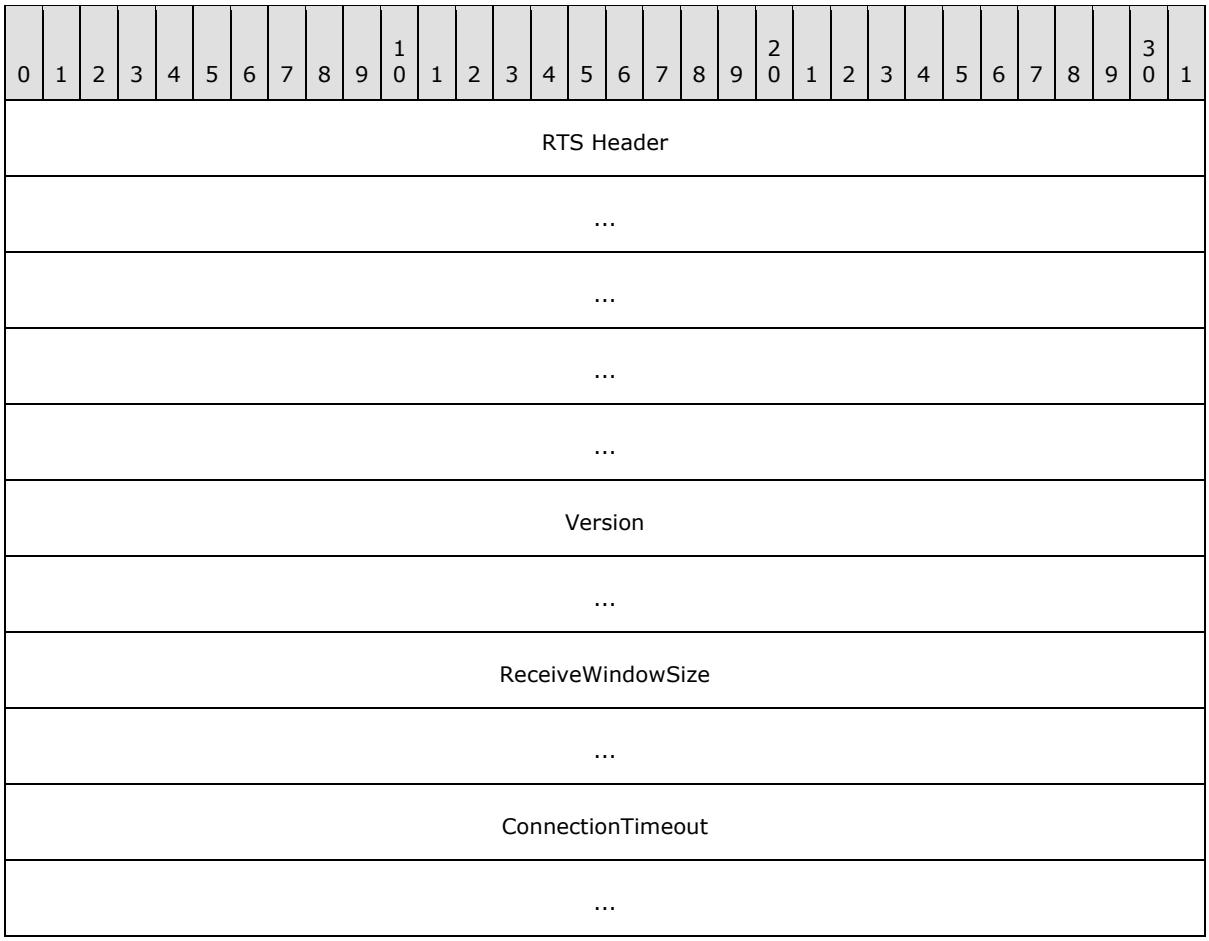
Version (8 bytes): This MUST be a Version command containing the lowest of the CONN/B2 RTS PDU (section 2.2.4.6) version, the CONN/A2 RTS PDU (section 2.2.4.3) version, and the server RPC over HTTP v2 version. The format for the RPC over HTTP v2 protocol version is defined in section 2.2.3.5.7.

ReceiveWindowSize (8 bytes): This MUST be a ReceiveWindowSize command containing the size of the receive window for the IN channel to the inbound proxy. The ReceiveWindowSize command format is defined in section 2.2.3.5.1.

ConnectionTimeout (8 bytes): This MUST be a ConnectionTimeout command containing the connection time out for the IN channel between the inbound proxy and the client. The ConnectionTimeout command format is defined in section 2.2.3.5.3.

2.2.4.9 CONN/C2 RTS PDU

The CONN/C2 RTS PDU MUST be sent from the outbound proxy to the client on the OUT channel to notify it that the establishment of a virtual connection has been completed.



RTS Header (20 bytes): See section 2.2.3.6.1. The **Flags** field of the RTS Header MUST be the value RTS_FLAG_NONE. The **NumberOfCommands** field of RTS Header MUST be the value 3.

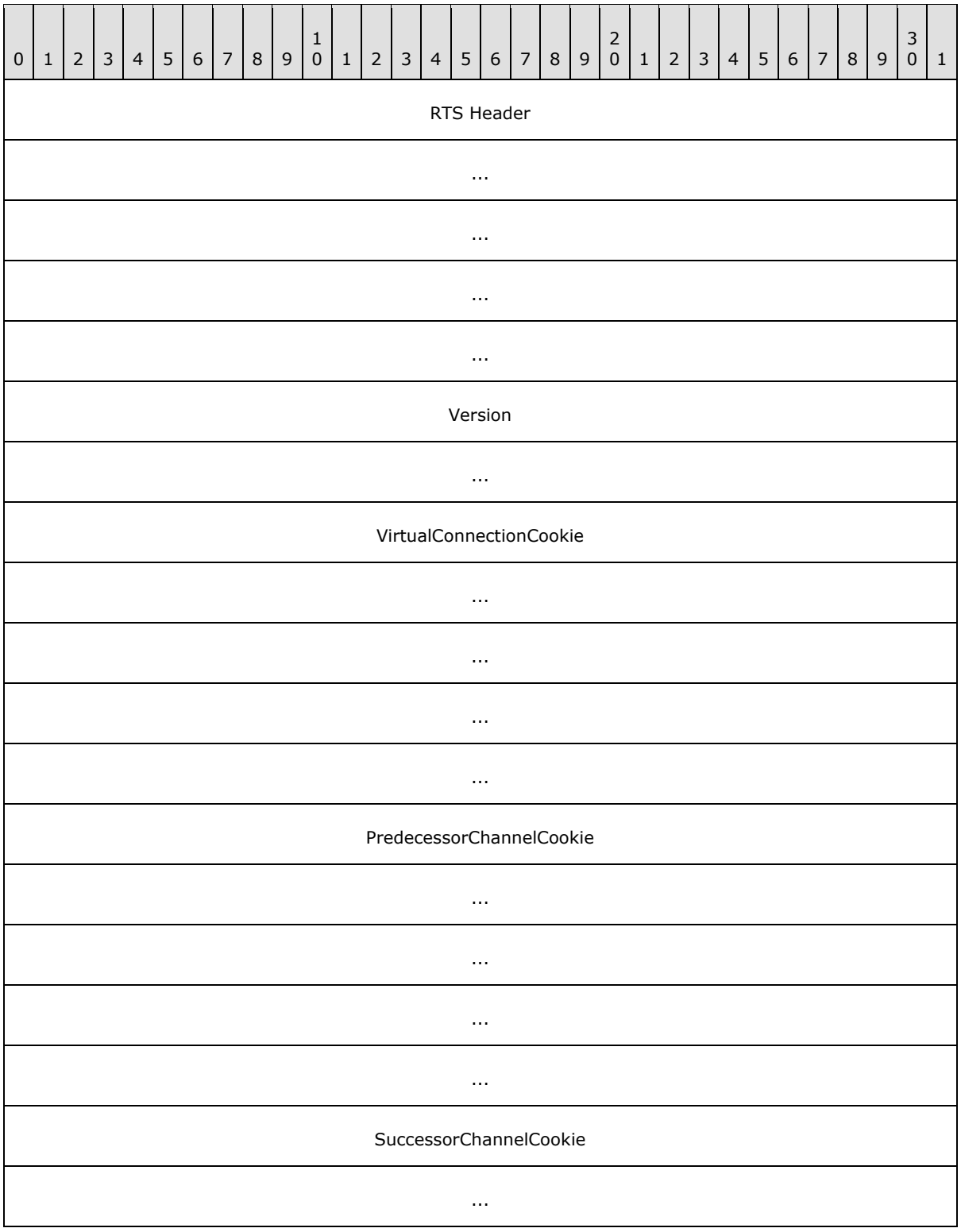
Version (8 bytes): This MUST be a Version command containing the CONN/C1 version. The format of the RPC over HTTP v2 protocol version command is defined in section 2.2.3.5.7.

ReceiveWindowSize (8 bytes): This MUST be a ReceiveWindowSize command containing the size of the receive window for the IN channel to the inbound proxy. The ReceiveWindowSize command format is defined in section 2.2.3.5.1.

ConnectionTimeout (8 bytes): This MUST be a ConnectionTimeout command containing the connection time out for the IN channel between the inbound proxy and the client. The ConnectionTimeout command format is defined in section 2.2.3.5.3.

2.2.4.10 IN_R1/A1 RTS PDU

The IN_R1/A1 RTS PDU MUST be sent from the client to the inbound proxy on a successor instance of an IN channel to initiate the establishment of a successor IN channel.



...
...
...

RTS Header (20 bytes): See section 2.2.3.6.1. The **Flags** field of the RTS Header MUST be the value RTS_FLAG_RECYCLE_CHANNEL. The **NumberOfCommands** field of RTS Header MUST be the value 4.

Version (8 bytes): This MUST be a version command containing the client RPC over HTTP v2 protocol version. The format of the RPC over HTTP v2 Protocol version is defined in section 2.2.3.5.7.

VirtualConnectionCookie (20 bytes): This MUST be a cookie command for the virtual connection that this IN channel belongs to. The cookie command format is defined in section 2.2.3.5.4.

PredecessorChannelCookie (20 bytes): This MUST be a cookie command that is the cookie of the **predecessor IN channel**. The cookie command format is defined in section 2.2.3.5.4.

SuccessorChannelCookie (20 bytes): This MUST be a cookie command identifying the successor IN channel. The cookie command format is defined in section 2.2.3.5.4.

2.2.4.11 IN_R1/A2 RTS PDU

The IN_R1/A2 RTS PDU MUST be sent from the **successor inbound proxy** to the server on the IN channel to initiate the establishment of a successor IN channel.

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
RTS Header																															
...																															
...																															
...																															
...																															
Version																															
...																															
VirtualConnectionCookie																															

...
...
...
...
PredecessorChannelCookie
...
...
...
...
SuccessorChannelCookie
...
...
...
...
InboundProxyReceiveWindowSize
...
InboundProxyConnectionTimeout
...

RTS Header (20 bytes): See section 2.2.3.6.1. The **Flags** field of the RTS Header MUST be the value RTS_FLAG_IN_CHANNEL. The **NumberOfCommands** field of RTS Header MUST be the value 6.

Version (8 bytes): This MUST be a version command containing the lower of the IN_R1/A1 version and the inbound proxy version. The format of the RPC over HTTP v2 Protocol version is defined in section 2.2.3.5.7.

VirtualConnectionCookie (20 bytes): This MUST be a cookie command for the virtual connection this IN channel belongs to. The cookie command format is defined in section 2.2.3.5.4.

PredecessorChannelCookie (20 bytes): This MUST be a cookie command for the predecessor IN channel. The cookie command format is defined in section 2.2.3.5.4.

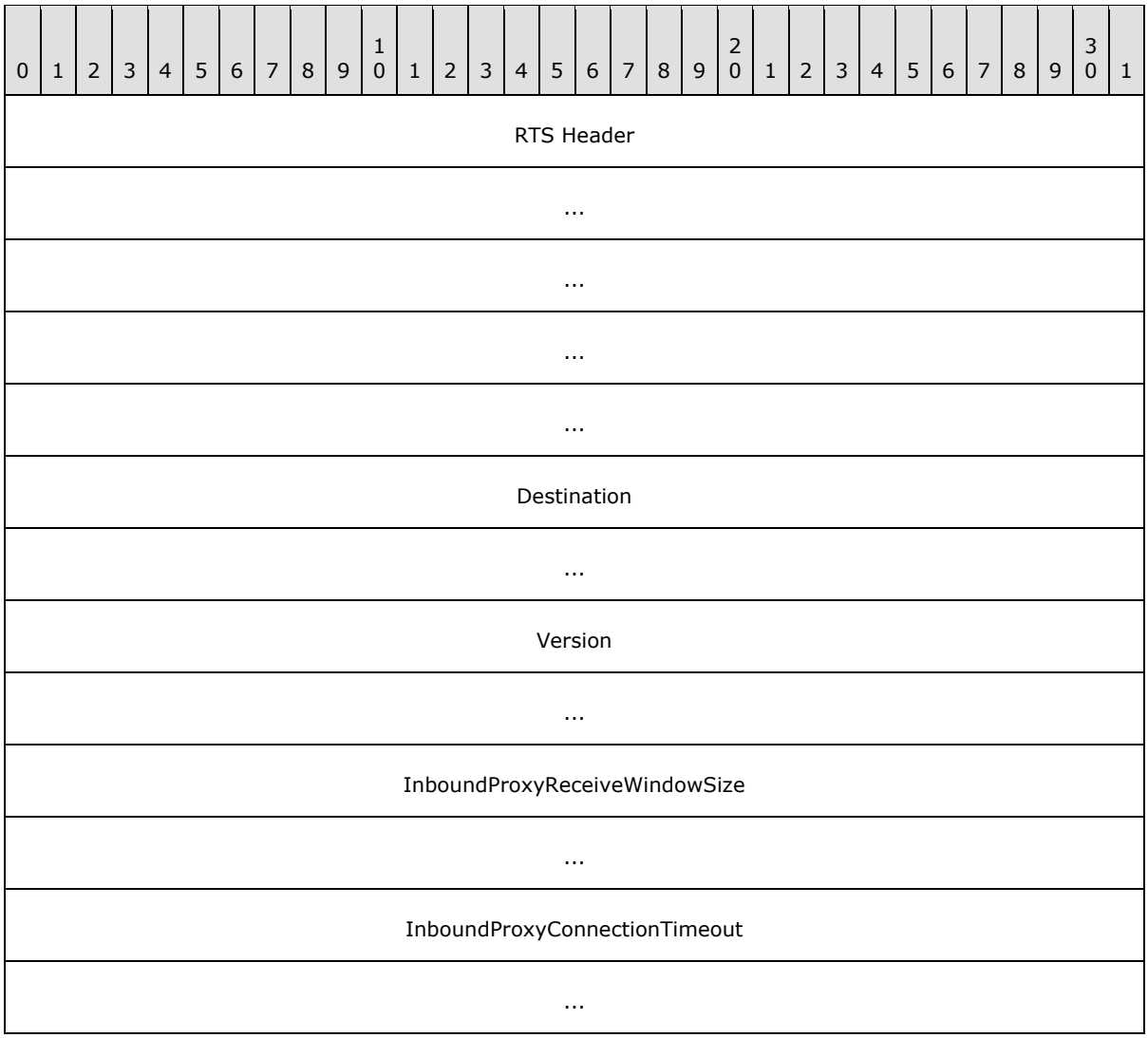
SuccessorChannelCookie (20 bytes): This MUST be a cookie command identifying the successor IN channel. The cookie command format is defined in section 2.2.3.5.4.

InboundProxyReceiveWindowSize (8 bytes): This MUST be a RecieveWindowSize command containing the size of the receive window for the IN channel to the inbound proxy. The ReceiveWindowSize command format is defined in section 2.2.3.5.1.

InboundProxyConnectionTimeout (8 bytes): This MUST be a ConnectionTimeout command specifying the connection time out for the IN channel between the successor inbound proxy and the client. The ConnectionTimeout command format is defined in section 2.2.3.5.3.

2.2.4.12 IN_R1/A3 RTS PDU

The IN_R1/A3 RTS PDU MUST be sent from the server to the outbound proxy on the OUT channel to continue the establishment of a successor IN channel.



RTS Header (20 bytes): See section 2.2.3.6.1. The **Flags** field of the RTS Header MUST be the value RTS_FLAG_NONE. The **NumberOfCommands** field of RTS Header MUST be the value 4.

Destination (8 bytes): This MUST be a destination command. The destination field for the destination command MUST be set to value FDClient as specified in section 2.2.3.3. The destination command format is defined in section 2.2.3.5.14.

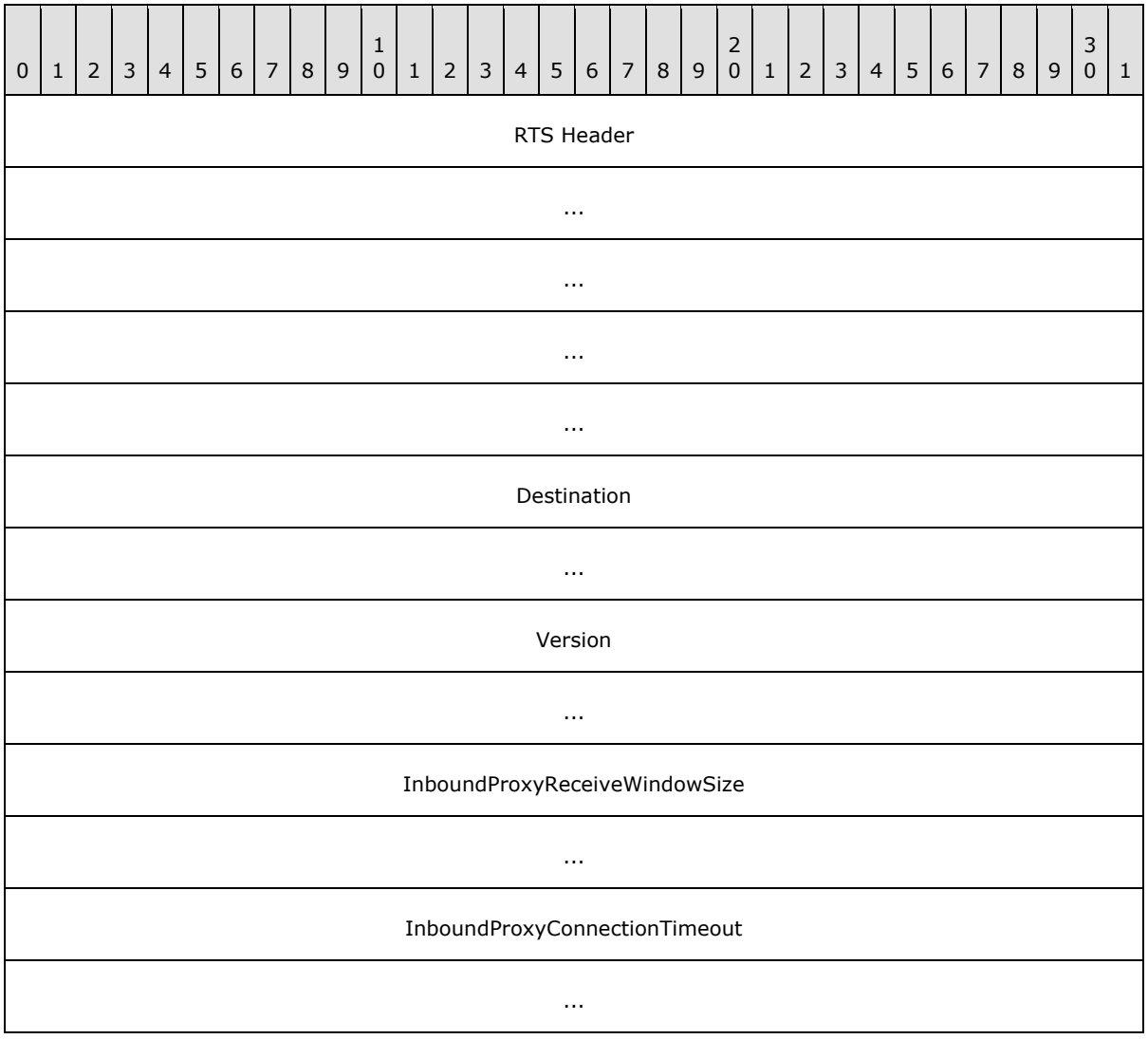
Version (8 bytes): This MUST be a version command specifying the lower of the IN_R1/A2 and the server version. The format of the RPC over HTTP v2 Protocol version is defined in section 2.2.3.5.7.

InboundProxyReceiveWindowSize (8 bytes): This MUST be a ReceiveWindowSize command specifying the size of the receive window for the successor IN channel to the inbound proxy. The ReceiveWindowSize command format is defined in section 2.2.3.5.1.

InboundProxyConnectionTimeout (8 bytes): This MUST be a ConnectionTimeout command specifying the connection time out for the IN channel between the successor inbound proxy and the client. The ConnectionTimeout command format is defined in section 2.2.3.5.3.

2.2.4.13 IN_R1/A4 RTS PDU

The IN_R1/A4 RTS PDU MUST be sent from the outbound proxy to the client on the OUT channel to continue the establishment of a successor IN channel as part of the IN_R1 protocol sequence.



RTS Header (20 bytes): See section 2.2.3.6.1. The **Flags** field of the RTS Header MUST be the value RTS_FLAG_NONE. The **NumberOfCommands** field of RTS Header MUST be the value 4.

Destination (8 bytes): This MUST be a destination command. The destination field for the destination command MUST be set to value FDClient as specified in section 2.2.3.3. The destination command format is defined in section 2.2.3.5.14.

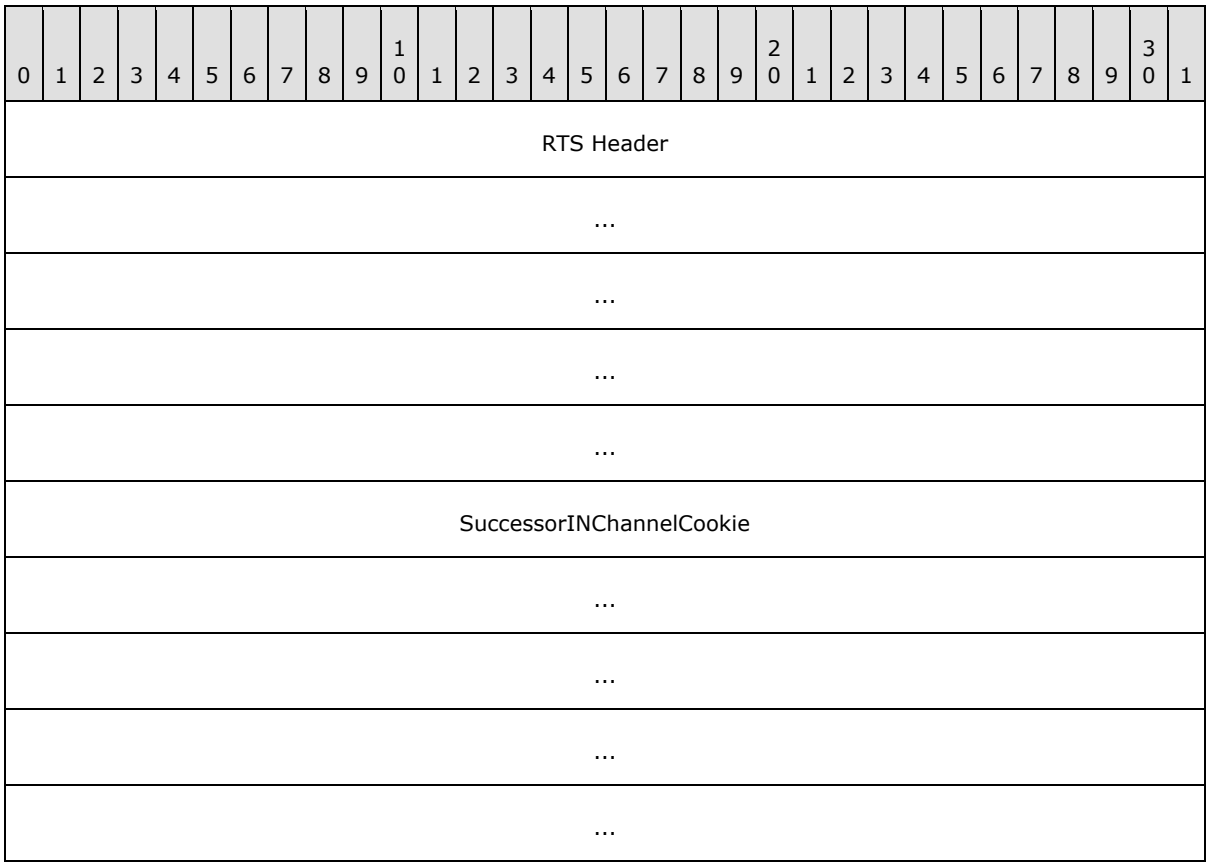
Version (8 bytes): This MUST be a version command specifying the lower of the IN_R1/A2 and the server version. The format of the RPC over HTTP v2 Protocol version is defined in section 2.2.3.5.7.

InboundProxyReceiveWindowSize (8 bytes): This MUST be a ReceiveWindowSize command specifying the size of the receive window for the successor IN channel to the inbound proxy. The WindowSize command format is defined in section 2.2.3.5.1.

InboundProxyConnectionTimeout (8 bytes): This MUST be a ConnectionTimeout command specifying the connection time out for the IN channel between the successor inbound proxy and the client. The ConnectionTimeout command format is defined in section 2.2.3.5.3.

2.2.4.14 IN_R1/A5 RTS PDU

The IN_R1/A5 RTS PDU MUST be sent from the client to the **predecessor inbound proxy** on the predecessor instance of the IN channel to continue the establishment of a successor IN channel.

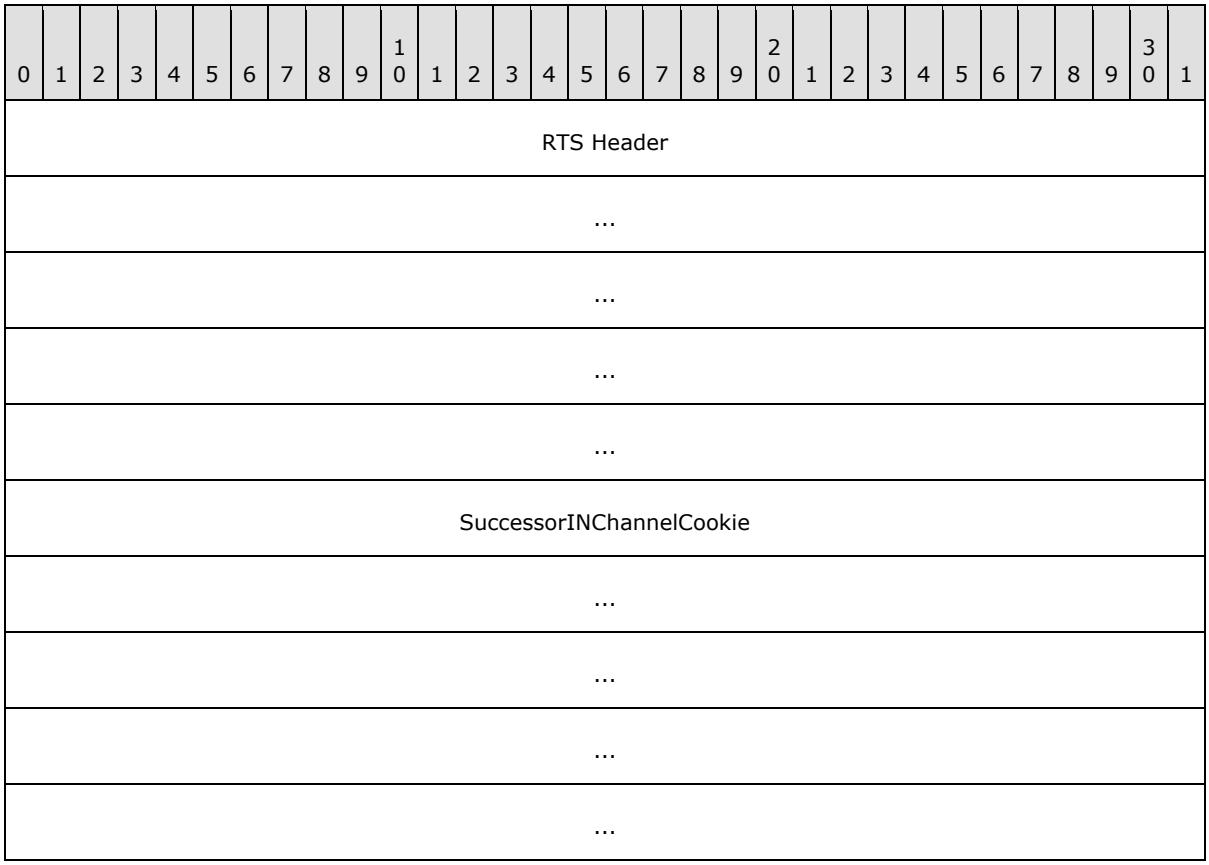


RTS Header (20 bytes): See section 2.2.3.6.1. The **Flags** field of the RTS Header MUST be the value RTS_FLAG_NONE. The **NumberOfCommands** field of RTS Header MUST be the value 1.

SuccessorINChannelCookie (20 bytes): This MUST be a cookie command identifying the successor IN channel cookie. The cookie command format is defined in section 2.2.3.5.4.

2.2.4.15 IN_R1/A6 RTS PDU

The IN_R1/A6 RTS PDU MUST be sent from the predecessor inbound proxy to the server on the predecessor instance of the IN channel to continue the establishment of a successor IN channel.

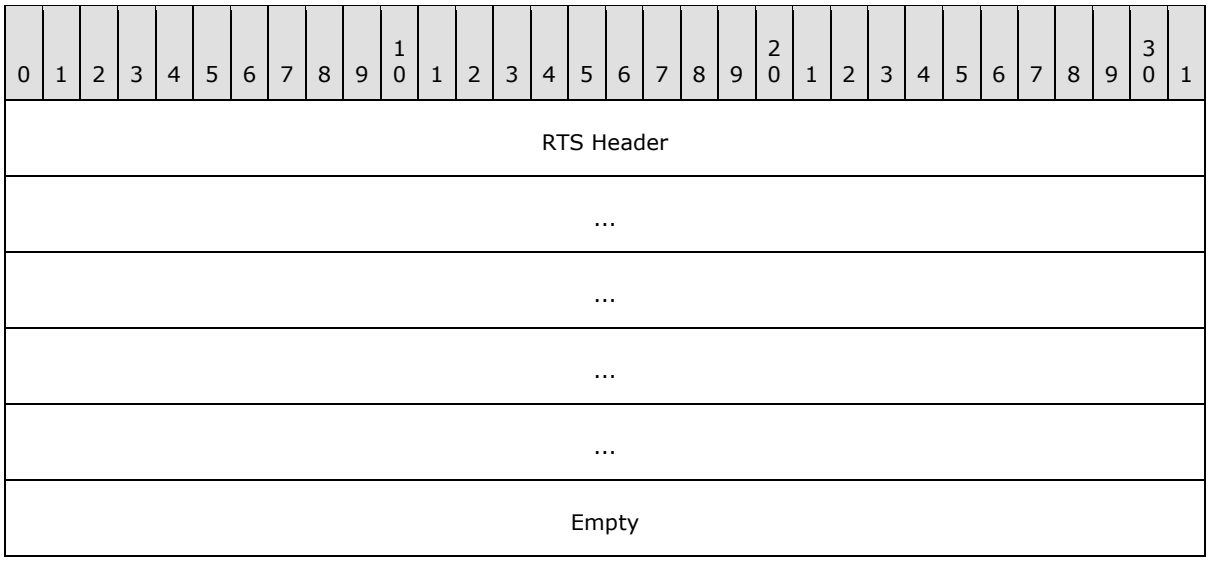


RTS Header (20 bytes): See section 2.2.3.6.1. The **Flags** field of the RTS Header MUST be the value RTS_FLAG_NONE. The **NumberOfCommands** field of RTS Header MUST be the value 1.

SuccessorINChannelCookie (20 bytes): This MUST be a cookie command identifying the successor IN channel cookie. The cookie command format is defined in section 2.2.3.5.4.

2.2.4.16 IN_R1/B1 RTS PDU

The IN_R1/B1 RTS PDU MUST be sent from the predecessor inbound proxy to the server on the predecessor instance of the IN channel to continue the establishment of a successor IN channel.

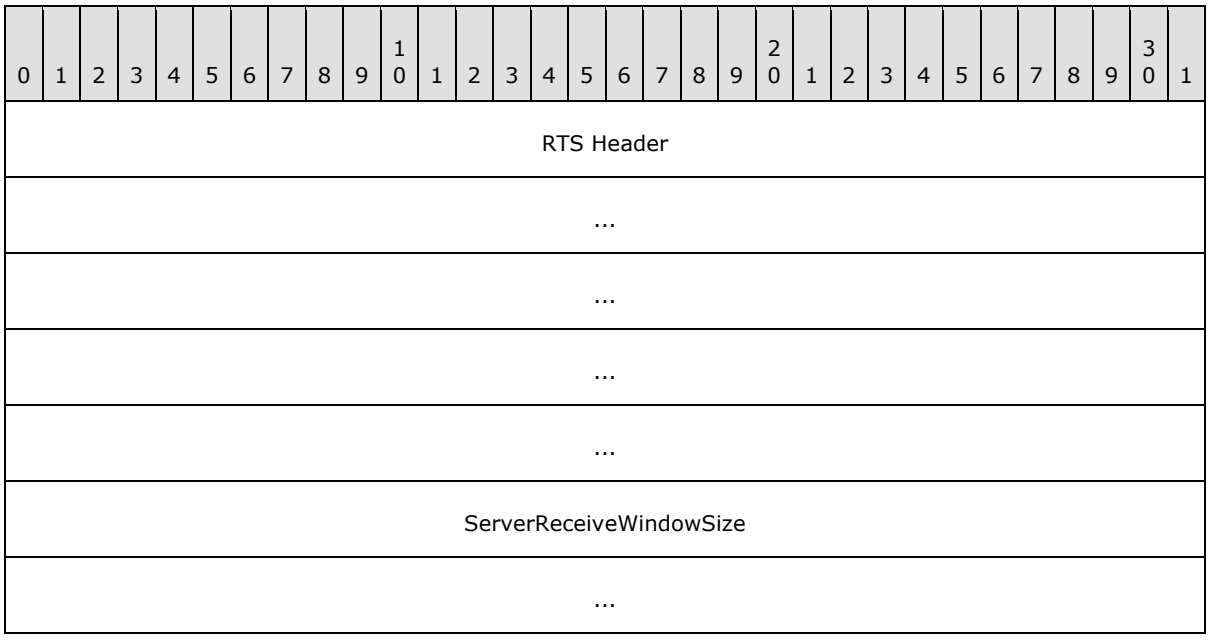


RTS Header (20 bytes): See section 2.2.3.6.1. The **Flags** field of the RTS Header MUST be the value RTS_FLAG_NONE. The **NumberOfCommands** field of RTS Header MUST be the value 1.

Empty (4 bytes): This MUST be an empty command. The format of the empty command is defined in section 2.2.3.5.8.

2.2.4.17 IN_R1/B2 RTS PDU

The IN_R1/B2 RTS PDU MUST be sent from the server to the successor inbound proxy on the successor IN channel to complete the establishment of a successor IN channel.

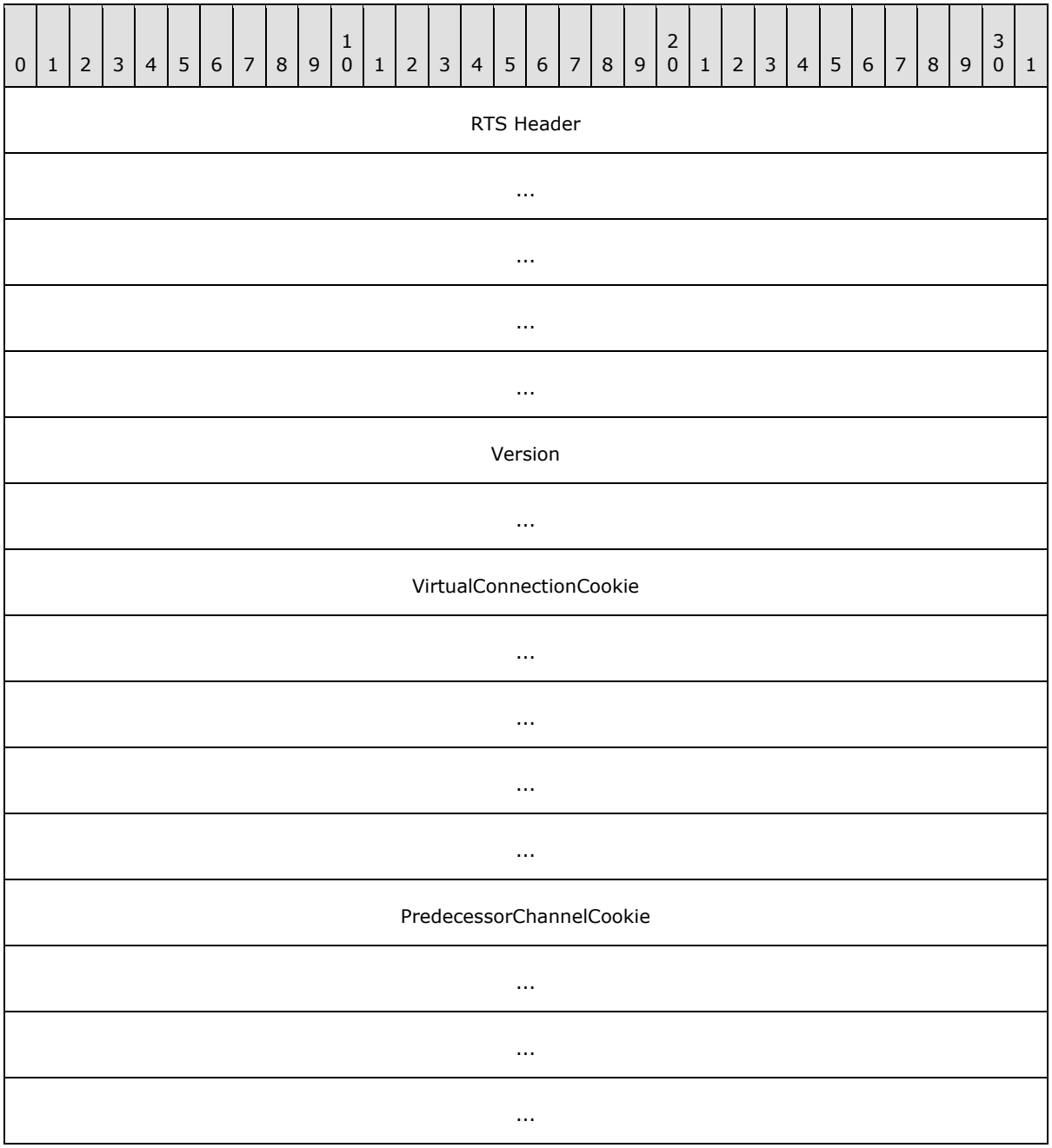


RTS Header (20 bytes): See section 2.2.3.6.1. The **Flags** field of the RTS Header MUST be the value RTS_FLAG_NONE. The **NumberOfCommands** field of RTS Header MUST be the value 1.

ServerReceiveWindowSize (8 bytes): This MUST be a ReceiveWindowSize command specifying the receive window size of the server. The ReceiveWindowSize command format is defined in section 2.2.3.5.1.

2.2.4.18 IN_R2/A1 RTS PDU

The IN_R2/A1 RTS PDU MUST be sent from the client to the inbound proxy on a successor IN channel to initiate the establishment of a successor IN channel.



...
SuccessorChannelCookie
...
...
...
...

RTS Header (20 bytes): See section 2.2.3.6.1. The **Flags** field of the RTS Header MUST be the value RTS_FLAG_RECYCLE_CHANNEL. The **NumberOfCommands** field of RTS Header MUST be the value 4.

Version (8 bytes): This MUST be a version command specifying the client RPC over HTTP v2 protocol version. The format of the RPC over HTTP v2 Protocol version is defined in section 2.2.3.5.7.

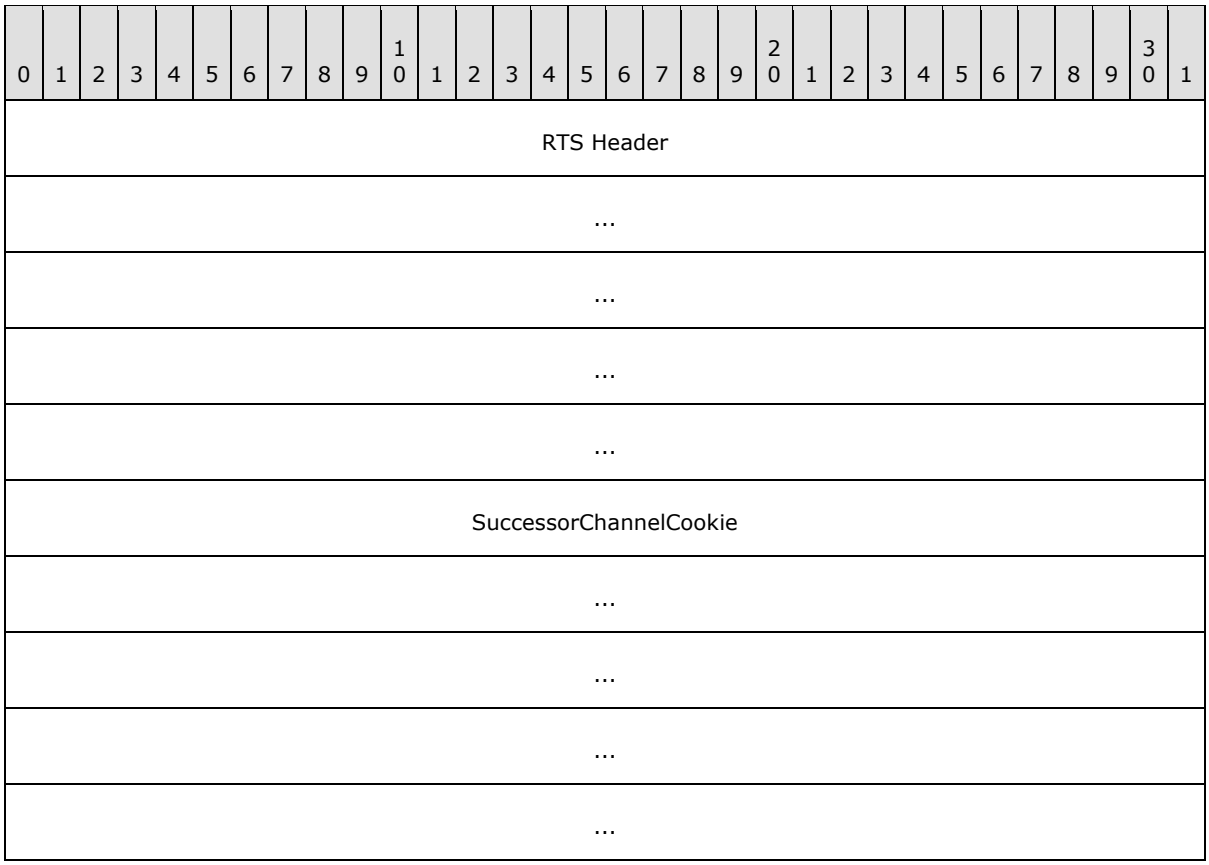
VirtualConnectionCookie (20 bytes): This MUST be a cookie command that is the cookie of the virtual connection to which this IN channel belongs. The cookie command format is defined in section 2.2.3.5.4.

PredecessorChannelCookie (20 bytes): This MUST be a cookie command that is the cookie of the predecessor IN channel. The cookie command format is defined in section 2.2.3.5.4.

SuccessorChannelCookie (20 bytes): This MUST be a cookie command identifying the successor IN channel cookie. The cookie command format is defined in section 2.2.3.5.4.

2.2.4.19 IN_R2/A2 RTS PDU

The IN_R2/A2 RTS PDU MUST be sent from the inbound proxy to the server on the IN channel to continue the establishment of a successor IN channel.

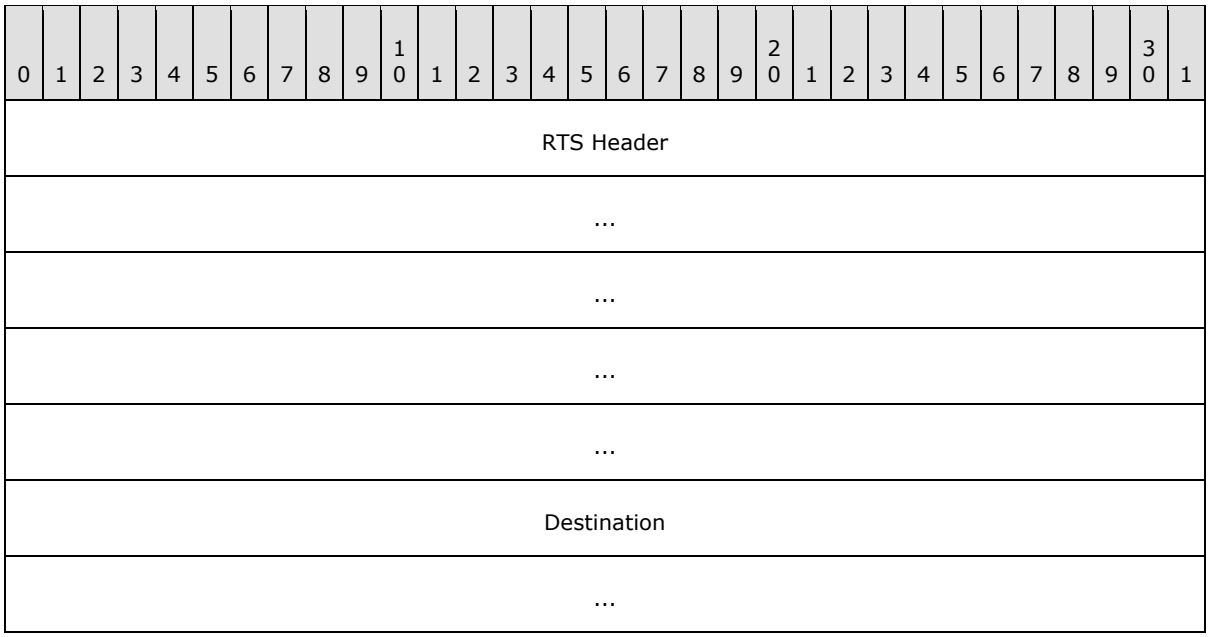


RTS Header (20 bytes): See section 2.2.3.6.1. The **Flags** field of the RTS Header MUST be the value RTS_FLAG_NONE. The **NumberOfCommands** field of RTS Header MUST be the value 1.

SuccessorChannelCookie (20 bytes): This MUST be a cookie command identifying the successor IN channel cookie. The RTS cookie command format is defined in section 2.2.3.5.4.

2.2.4.20 IN_R2/A3 RTS PDU

The IN_R2/A3 RTS PDU MUST be sent from the server to the outbound proxy on the OUT channel to continue the establishment of a successor IN channel.

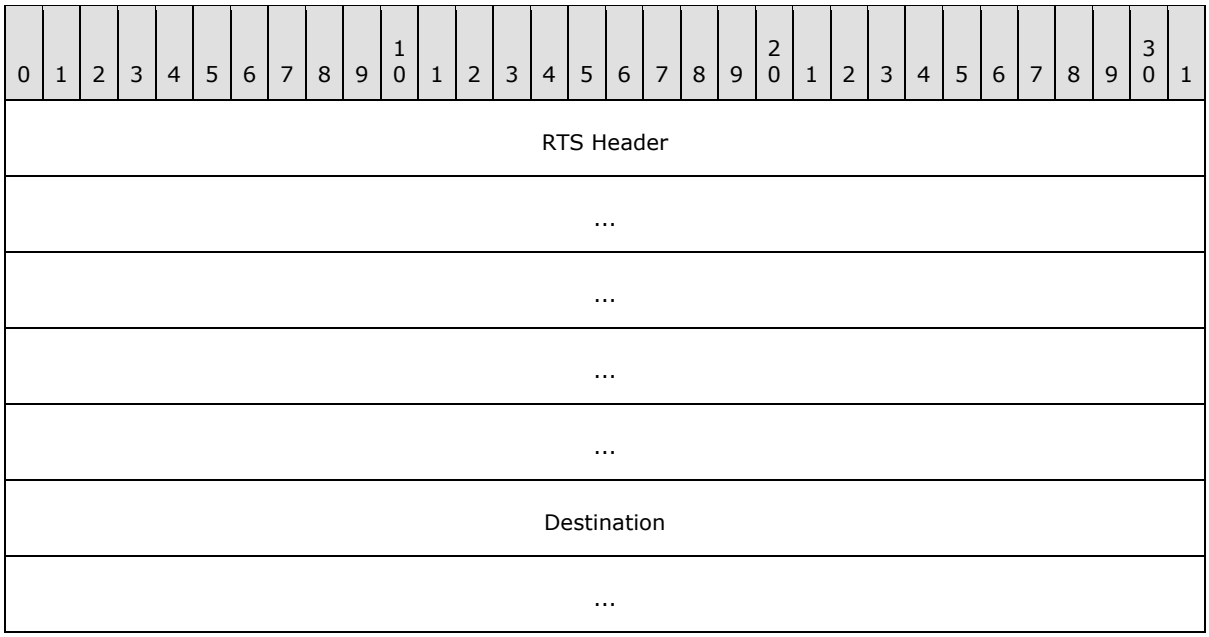


RTS Header (20 bytes): See section 2.2.3.6.1. The **Flags** field of the RTS Header MUST be the value RTS_FLAG_NONE. The **NumberOfCommands** field of RTS Header MUST be the value 1.

Destination (8 bytes): This MUST be a destination command. The destination field for the destination command MUST be set to value FDClient as specified in section 2.2.3.3. The destination command format is defined in section 2.2.3.5.14.

2.2.4.21 IN_R2/A4 RTS PDU

The IN_R2/A4 RTS PDU MUST be sent from the outbound proxy to the client on the OUT channel to continue the establishment of a successor IN channel.

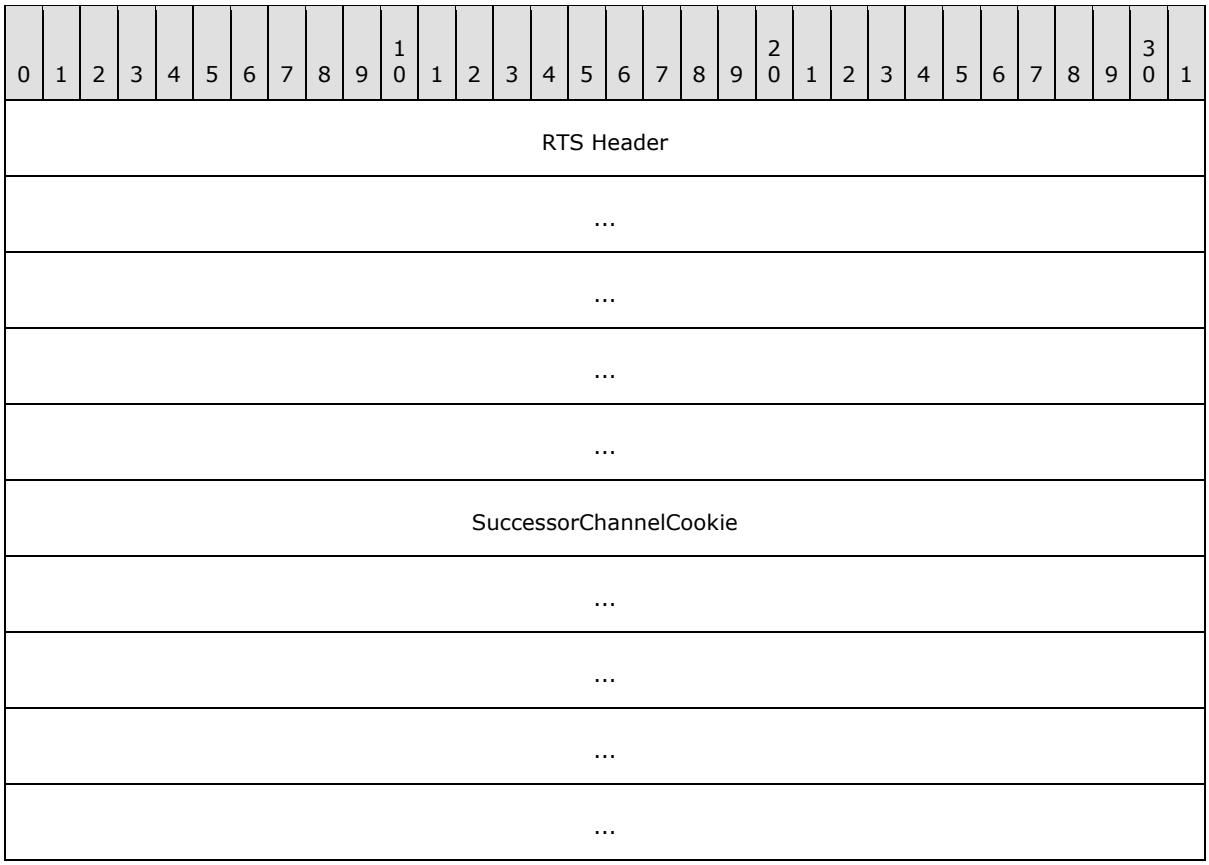


RTS Header (20 bytes): See section 2.2.3.6.1. The **Flags** field of the RTS Header MUST be the value RTS_FLAG_NONE. The **NumberOfCommands** field of RTS Header MUST be the value 1.

Destination (8 bytes): This MUST be a destination command. The destination field for the destination command MUST be set to value FDClient as specified in section 2.2.3.3. The destination command format is defined in section 2.2.3.5.14.

2.2.4.22 IN_R2/A5 RTS PDU

The IN_R2/A5 RTS PDU MUST be sent from the client to the inbound proxy on the predecessor instance of the IN channel to continue the establishment of a successor IN channel.

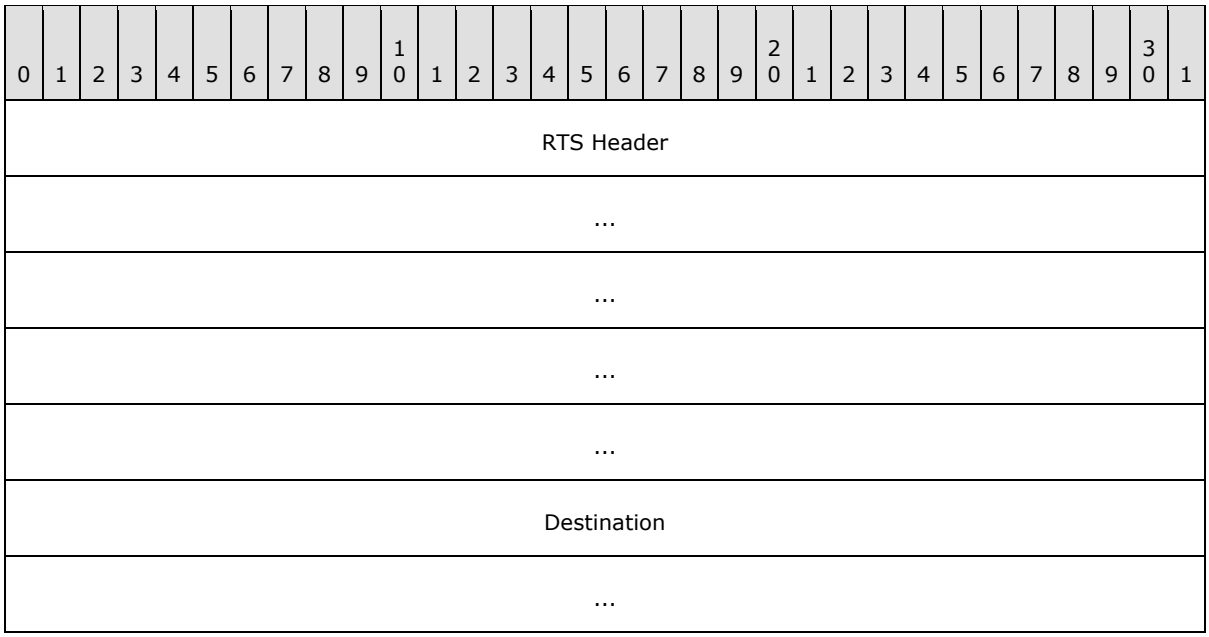


RTS Header (20 bytes): See section 2.2.3.6.1. The **Flags** field of the RTS Header MUST be the value RTS_FLAG_NONE. The **NumberOfCommands** field of RTS Header MUST be the value 1.

SuccessorChannelCookie (20 bytes): This MUST be a cookie command identifying the successor IN channel cookie. The cookie command format is defined in section 2.2.3.5.4.

2.2.4.23 OUT_R1/A1 RTS PDU

The OUT_R1/A1 RTS PDU MUST be sent from the server to the outbound proxy on the OUT channel to initiate the establishment of a successor OUT channel.

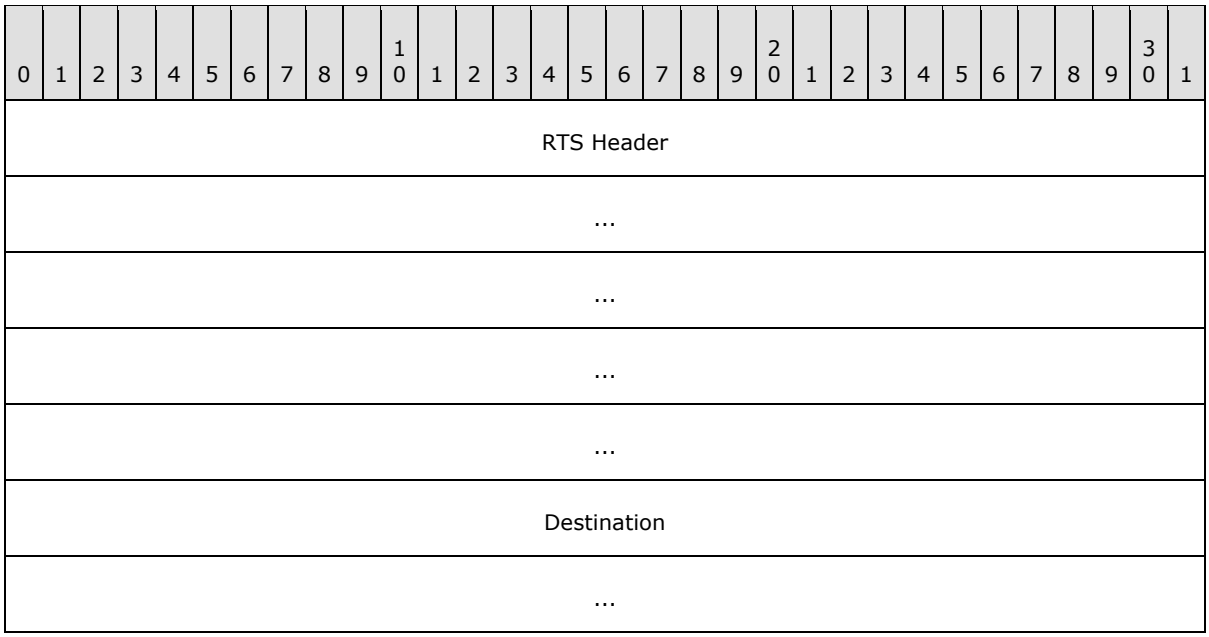


RTS Header (20 bytes): See section 2.2.3.6.1. The **Flags** field of the RTS Header MUST be the value RTS_FLAG_RECYCLE_CHANNEL. The **NumberOfCommands** field of RTS Header MUST be the value 1.

Destination (8 bytes): This MUST be a Destination command. The destination field for the destination command MUST be set to the value FDClient as specified in section 2.2.3.3. The destination command format is defined in section 2.2.3.5.14.

2.2.4.24 OUT_R1/A2 RTS PDU

The OUT_R1/A2 RTS PDU MUST be sent from the outbound proxy to the client on the OUT channel to initiate the establishment of a successor OUT channel.

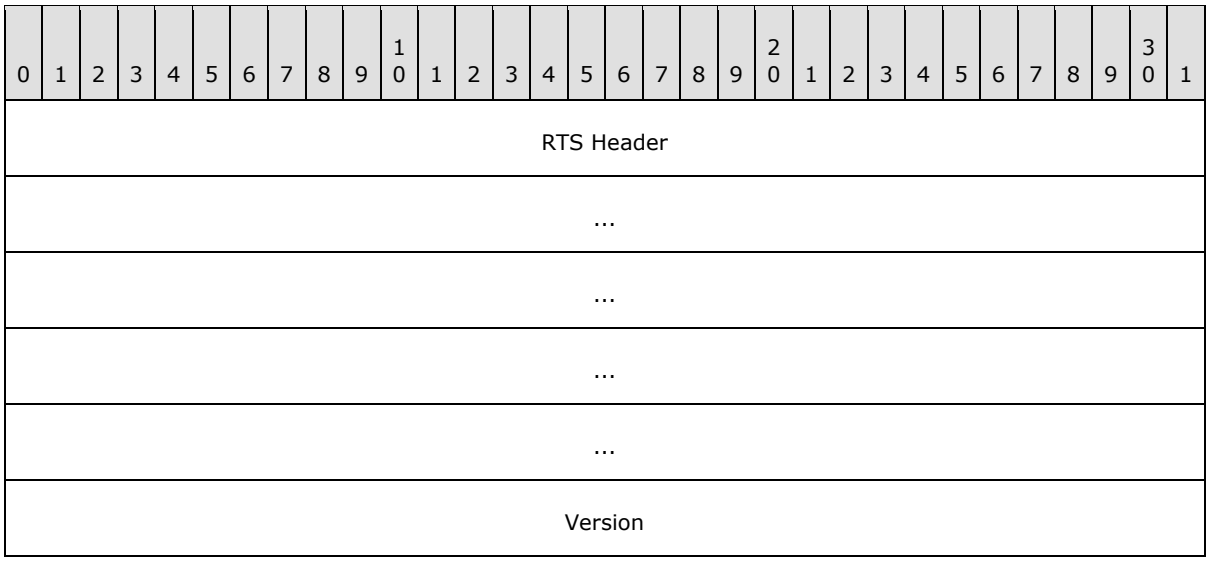


RTS Header (20 bytes): See section 2.2.3.6.1. The **Flags** field of the RTS Header MUST be the value RTS_FLAG_RECYCLE_CHANNEL. The **NumberOfCommands** field of the RTS Header MUST be the value 1.

Destination (8 bytes): This MUST be a Destination command. The destination field for the destination command MUST be set to the value FDClient as specified in section 2.2.3.3. The destination command format is defined in section 2.2.3.5.14.

2.2.4.25 OUT_R1/A3 RTS PDU

The OUT_R1/A3 RTS PDU MUST be sent from the client to the **successor outbound proxy** on the successor OUT channel to initiate the establishment of a successor OUT channel.



...
VirtualConnectionCookie
...
...
...
...
PredecessorChannelCookie
...
...
...
...
SuccessorChannelCookie
...
...
...
...
InboundProxyReceiveWindowSize
...

RTS Header (20 bytes): See section 2.2.3.6.1. The **Flags** field of the RTS Header MUST be the value RTS_FLAG_RECYCLE_CHANNEL. The **NumberOfCommands** field of RTS Header MUST be the value 5.

Version (8 bytes): This MUST be a Version command specifying the client RPC over HTTP v2 protocol version. The format of the RPC over HTTP v2 Protocol version is defined in section 2.2.3.5.7.

VirtualConnectionCookie (20 bytes): This MUST be a Cookie command that is the cookie of the virtual connection that this OUT channel belongs to. The cookie command format is defined in section 2.2.3.5.4.

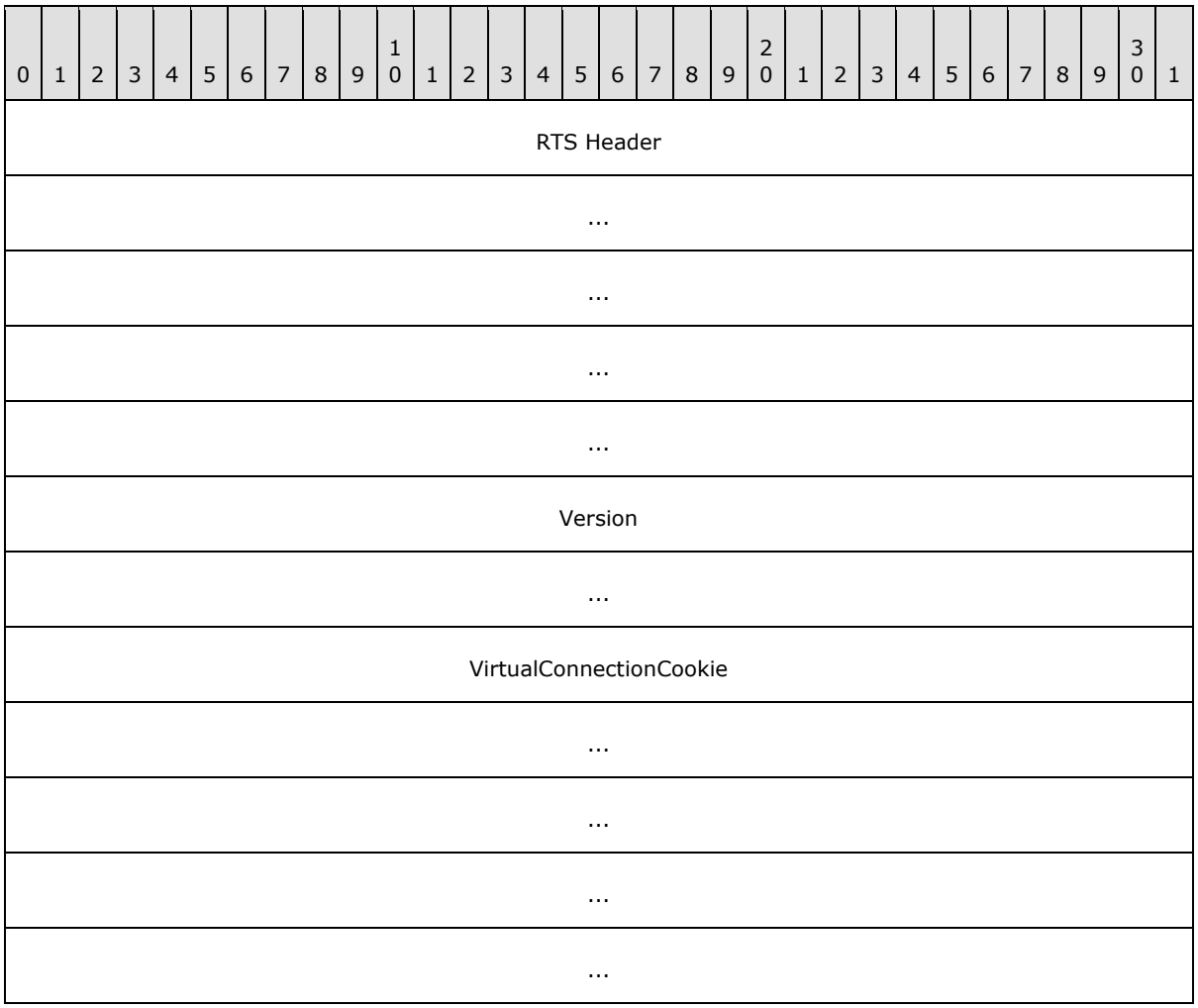
PredecessorChannelCookie (20 bytes): This MUST be a Cookie command that is the cookie of the predecessor OUT channel. The cookie command format is defined in section 2.2.3.5.4.

SuccessorChannelCookie (20 bytes): This MUST be a Cookie command identifying the successor OUT channel cookie. The cookie command format is defined in section 2.2.3.5.4.

InboundProxyReceiveWindowSize (8 bytes): This MUST be a ReceiveWindowSize command specifying the size of the receive window for the client OUT channel. The ReceiveWindowSize command format is defined in section 2.2.3.5.1.

2.2.4.26 OUT_R1/A4 RTS PDU

The OUT_R1/A4 RTS PDU MUST be sent from the successor outbound proxy to the server on the OUT channel to initiate the establishment of a successor OUT channel.



PredecessorChannelCookie
...
...
...
...
SuccessorChannelCookie
...
...
...
...
ChannelLifetime
...
InboundProxyReceiveWindowSize
...
InboundProxyConnectionTimeout
...

RTS Header (20 bytes): See section 2.2.3.6.1. The **Flags** field of the RTS Header MUST be the value RTS_FLAG_RECYCLE_CHANNEL. The **NumberOfCommands** field of RTS Header MUST be the value 7.

Version (8 bytes): This MUST be a Version command specifying the lower of the outbound proxy RPC over HTTP v2 Protocol version and OUT_R1/A3 protocol version. The format of the RPC over HTTP v2 Protocol version is defined in section 2.2.3.5.7.

VirtualConnectionCookie (20 bytes): This MUST be a Cookie command that is the cookie of the virtual connection that this OUT channel belongs to. The cookie command format is defined in section 2.2.3.5.4.

PredecessorChannelCookie (20 bytes): This MUST be a Cookie command that is the cookie of the predecessor OUT channel. The cookie command format is defined in section 2.2.3.5.4.

SuccessorChannelCookie (20 bytes): This MUST be a Cookie command identifying the successor OUT channel cookie. The cookie command format is defined in section 2.2.3.5.4.

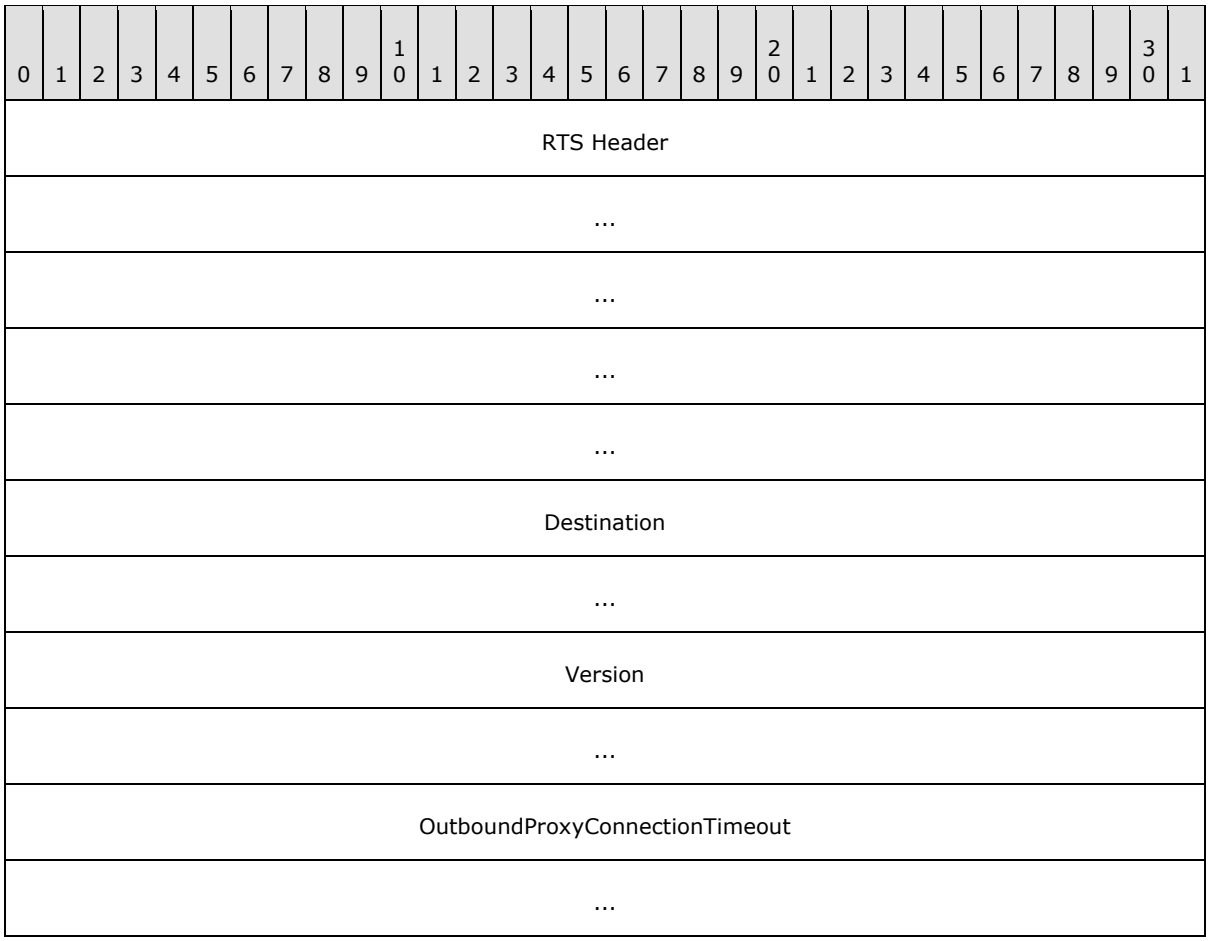
ChannelLifetime (8 bytes): This MUST be a Channel Lifetime command specifying the lifetime in bytes of the OUT channel from the outbound proxy to the client. The ChannelLifetime command format is defined in section 2.2.3.5.5.

InboundProxyReceiveWindowSize (8 bytes): This MUST be a ReceiveWindowSize command specifying the size of the receive window for the successor OUT channel to the outbound proxy. The ReceiveWindowSize command format is defined in section 2.2.3.5.1.

InboundProxyConnectionTimeout (8 bytes): This MUST be a ConnectionTimeout command specifying the connection time out for the OUT channel between the successor outbound proxy and the client. The ConnectionTimeout command format is defined in section 2.2.3.5.3. This command is for troubleshooting purposes only and has no protocol significance. The server SHOULD ignore this value.

2.2.4.27 OUT_R1/A5 RTS PDU

The OUT_R1/A5 RTS PDU MUST be sent from the server to the **predecessor outbound proxy** on the predecessor instance of the OUT channel to continue the establishment of a successor OUT channel.



RTS Header (20 bytes): See section 2.2.3.6.1. The **Flags** field of the RTS Header MUST be the value RTS_FLAG_OUT_CHANNEL. The **NumberOfCommands** field of RTS Header MUST be the value 3.

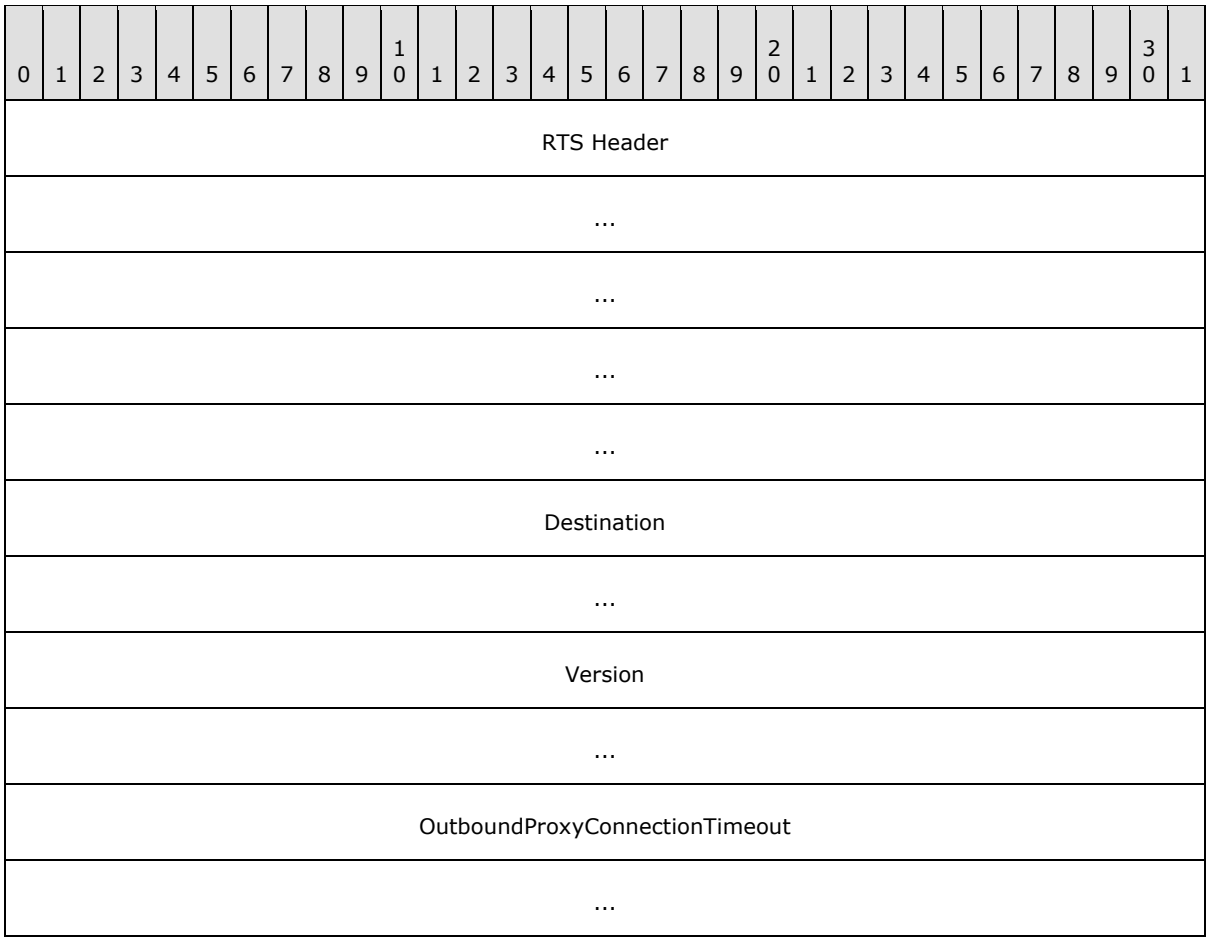
Destination (8 bytes): This MUST be a Destination command. The destination field for the destination command MUST be set to the value FDClient as specified in section 2.2.3.3. The destination command format is defined in section 2.2.3.5.14.

Version (8 bytes): This MUST be a Version command specifying the lower of the server RPC over HTTP v2 Protocol version and OUT_R1/A4 version. The format of the RPC over HTTP v2 Protocol version is defined in section 2.2.3.5.7.

OutboundProxyConnectionTimeout (8 bytes): This MUST be a ConnectionTimeout command specifying the connection time out for the OUT channel between the successor outbound proxy and the client. ConnectionTimeout command format is defined in section 2.2.3.5.3. This command is used for troubleshooting purposes only and has no protocol significance. The predecessor outbound proxy SHOULD ignore this value.

2.2.4.28 OUT_R1/A6 RTS PDU

The OUT_R1/A6 RTS PDU MUST be sent from the predecessor outbound proxy to the client on the OUT channel to continue the establishment of a successor OUT channel.



RTS Header (20 bytes): See section 2.2.3.6.1. The **Flags** field of the RTS Header MUST be the value RTS_FLAG_OUT_CHANNEL. The **NumberOfCommands** field of RTS Header MUST be the value 3.

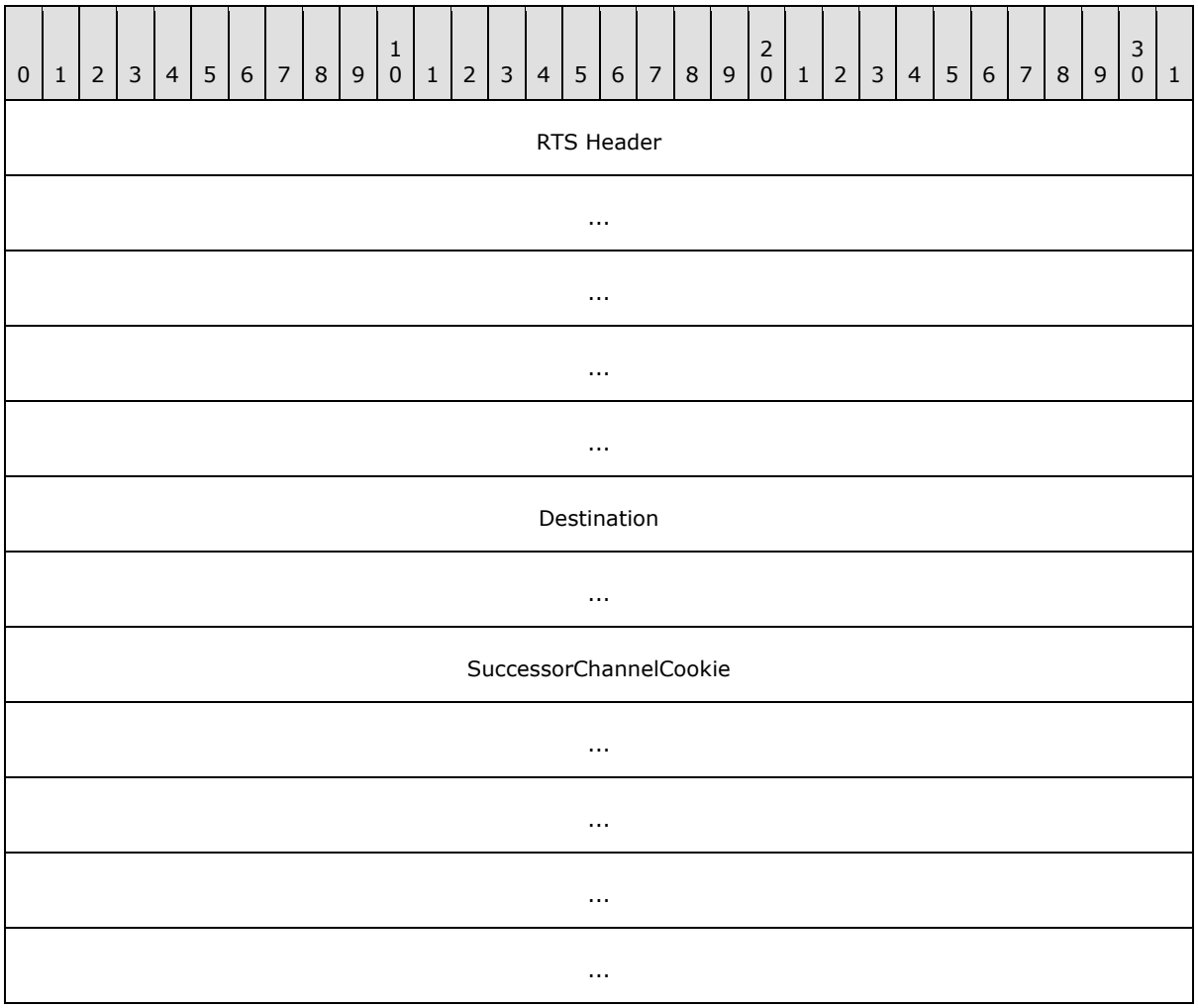
Destination (8 bytes): This MUST be a destination command. The destination field of the destination command MUST be set to value FDClient as specified in section 2.2.3.3. The destination command format is defined in section 2.2.3.5.14.

Version (8 bytes): This MUST be a version command specifying the lower of the server RPC over HTTP v2 Protocol version and OUT_R1/A4 version. The format of the RPC over HTTP v2 Protocol version is defined in section 2.2.3.5.7.

OutboundProxyConnectionTimeout (8 bytes): This MUST be a ConnectionTimeout command specifying the connection time out for the OUT channel between the successor outbound proxy and the client. ConnectionTimeout command format is defined in section 2.2.3.5.3. This command is useful for troubleshooting purposes only and has no protocol significance. The client SHOULD ignore this value.

2.2.4.29 OUT_R1/A7 RTS PDU

The OUT_R1/A7 RTS PDU MUST be sent from the client to the inbound proxy on the IN channel to continue the establishment of a successor OUT channel as part of the OUT_R1 protocol sequence specified in section 3.2.1.4.3.5.



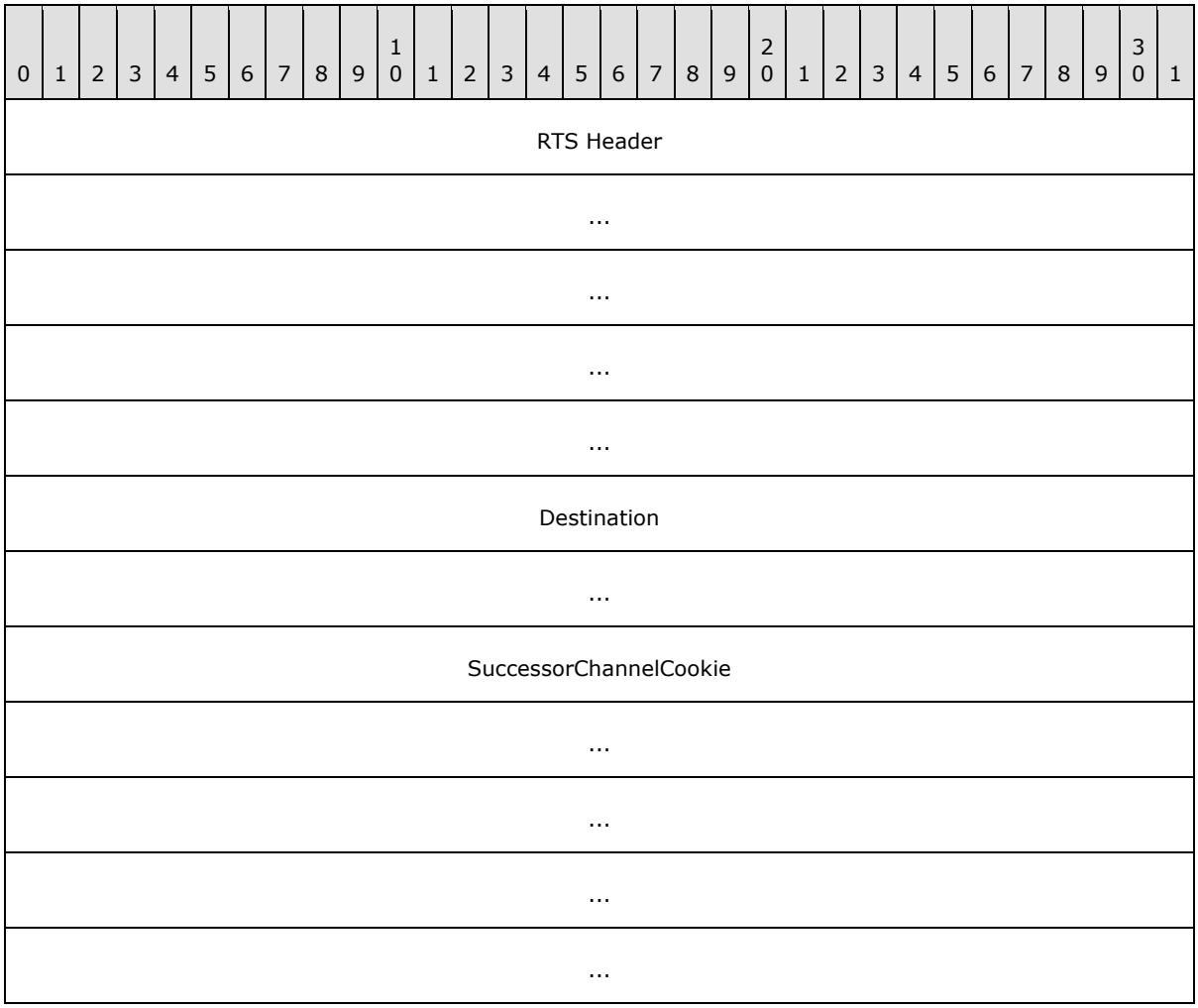
RTS Header (20 bytes): See section 2.2.3.6.1. The **Flags** field of the RTS Header MUST be the value RTS_FLAG_OUT_CHANNEL. The **NumberOfCommands** field of RTS Header MUST be the value 2.

Destination (8 bytes): This MUST be a Destination command. The destination field for the destination command MUST be set to the value FDServer, as specified in section 2.2.3.3. The destination command format is defined in section 2.2.3.5.14.

SuccessorChannelCookie (20 bytes): This MUST be a cookie command identifying the successor OUT channel cookie. The cookie command format is defined in section 2.2.3.5.4.

2.2.4.30 OUT_R1/A8 RTS PDU

The OUT_R1/A8 RTS PDU MUST be sent from the inbound proxy to the server on the IN channel to continue the establishment of a successor OUT channel.



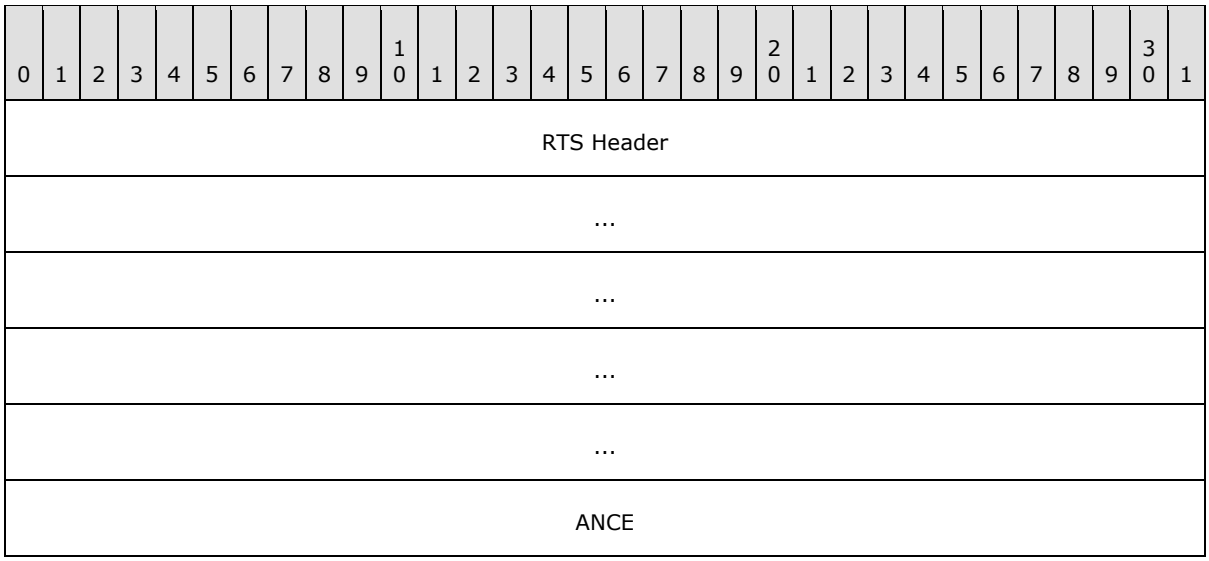
RTS Header (20 bytes): See section 2.2.3.6.1. The **Flags** field of the RTS Header MUST be the value RTS_FLAG_OUT_CHANNEL. The **NumberOfCommands** field of RTS Header MUST be the value 2.

Destination (8 bytes): This MUST be a Destination command. The destination field for the destination command MUST be set to the value FDServer, as specified in section 2.2.3.3. The destination command format is defined in section 2.2.3.5.14.

SuccessorChannelCookie (20 bytes): This MUST be a Cookie command identifying the successor OUT channel cookie. The cookie command format is defined in section 2.2.3.5.4.

2.2.4.31 OUT_R1/A9 RTS PDU

The OUT_R1/A9 RTS PDU MUST be sent from the server to the predecessor outbound proxy to indicate to it that the successor **virtual OUT channel** was established successfully.

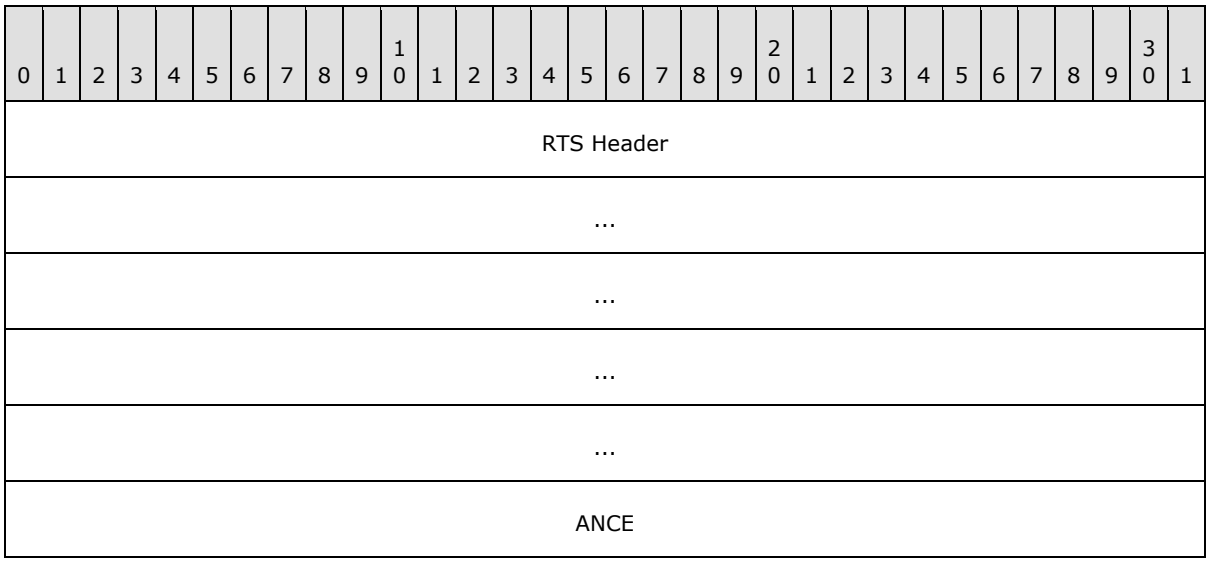


RTS Header (20 bytes): See section 2.2.3.6.1. The **Flags** field of the RTS Header MUST be the value RTS_FLAG_NONE. The **NumberOfCommands** field of RTS Header MUST be the value 1.

ANCE (4 bytes): This MUST be an ANCE command. The format of the ANCE command is defined in section 2.2.3.5.11.

2.2.4.32 OUT_R1/A10 RTS PDU

The OUT_R1/A10 RTS PDU MUST be sent from the predecessor outbound proxy to the client to indicate that the successor virtual OUT channel was established successfully.

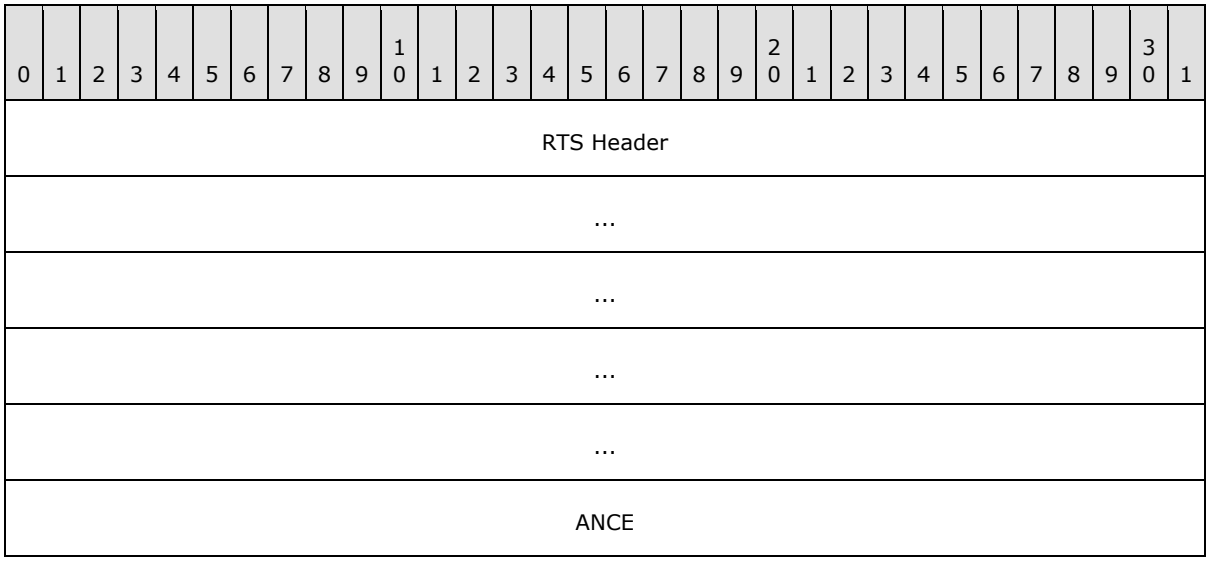


RTS Header (20 bytes): See section 2.2.3.6.1. The **Flags** field of the RTS Header MUST be the value RTS_FLAG_NONE. The **NumberOfCommands** field of RTS Header MUST be the value 1.

ANCE (4 bytes): This MUST be an ANCE command. The format of the ANCE command is defined in section 2.2.3.5.11.

2.2.4.33 OUT_R1/A11 RTS PDU

The OUT_R1/A11 RTS PDU MUST be sent from the client to the successor outbound proxy to indicate to it that the successor virtual OUT channel was established successfully.

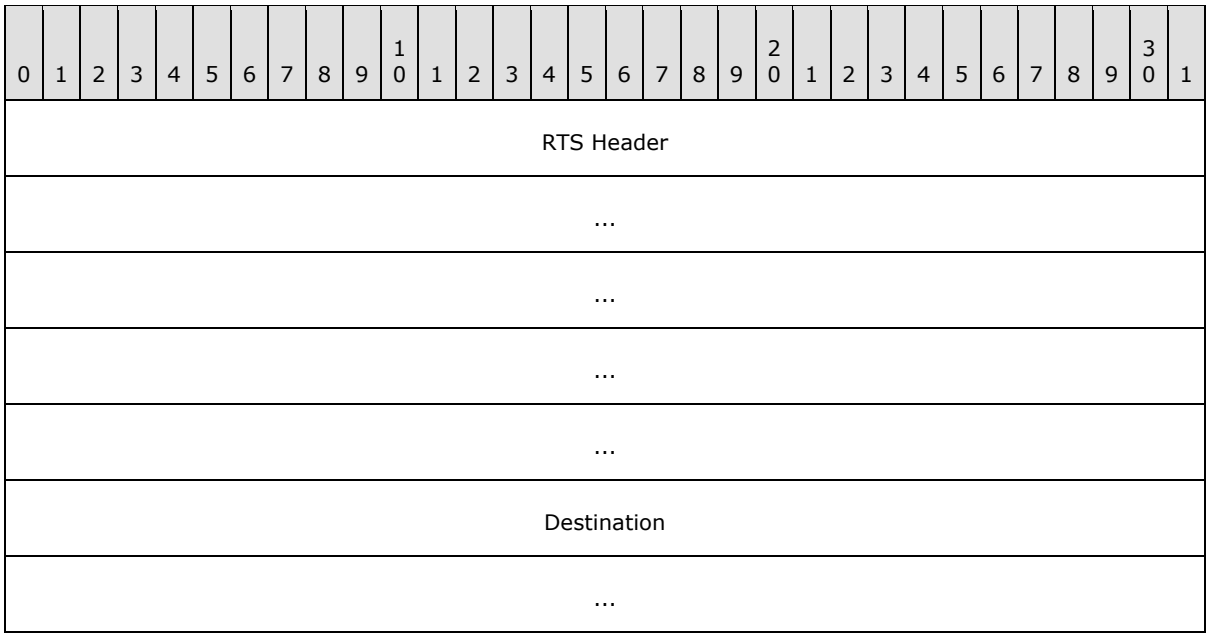


RTS Header (20 bytes): See section 2.2.3.6.1. The **Flags** field of the RTS Header MUST be the value RTS_FLAG_NONE. The **NumberOfCommands** field of RTS Header MUST be the value 1.

ANCE (4 bytes): This MUST be an ANCE command. The format of the ANCE command is defined in section 2.2.3.5.11.

2.2.4.34 OUT_R2/A1 RTS PDU

The OUT_R2/A1 RTS PDU MUST be sent from the server to the outbound proxy on the OUT channel to initiate the establishment of a successor OUT channel.

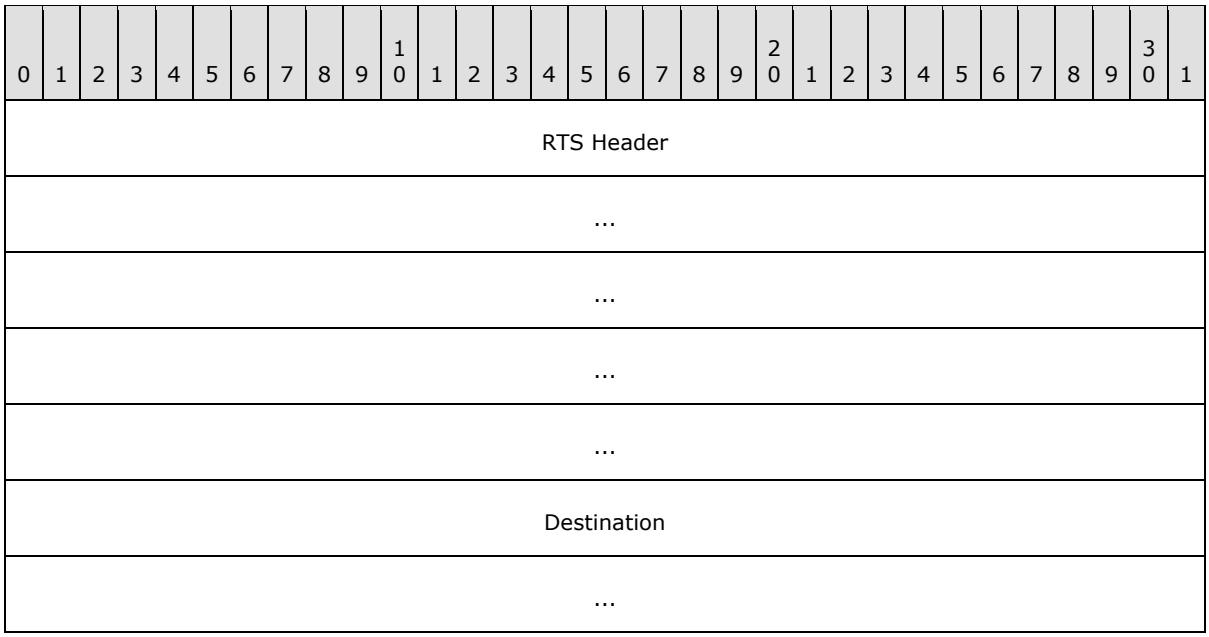


RTS Header (20 bytes): See section 2.2.3.6.1. The **Flags** field of the RTS Header MUST be the value RTS_FLAG_RECYCLE_CHANNEL. The **NumberOfCommands** field of RTS Header MUST be the value 1.

Destination (8 bytes): This MUST be a Destination command. The destination field for the destination command MUST be set to the value FDClient as specified in section 2.2.3.3. The destination command format is defined in section 2.2.3.5.14.

2.2.4.35 OUT_R2/A2 RTS PDU

The OUT_R2/A2 RTS PDU MUST be sent from the outbound proxy to the client on the OUT channel to init OUT channel.

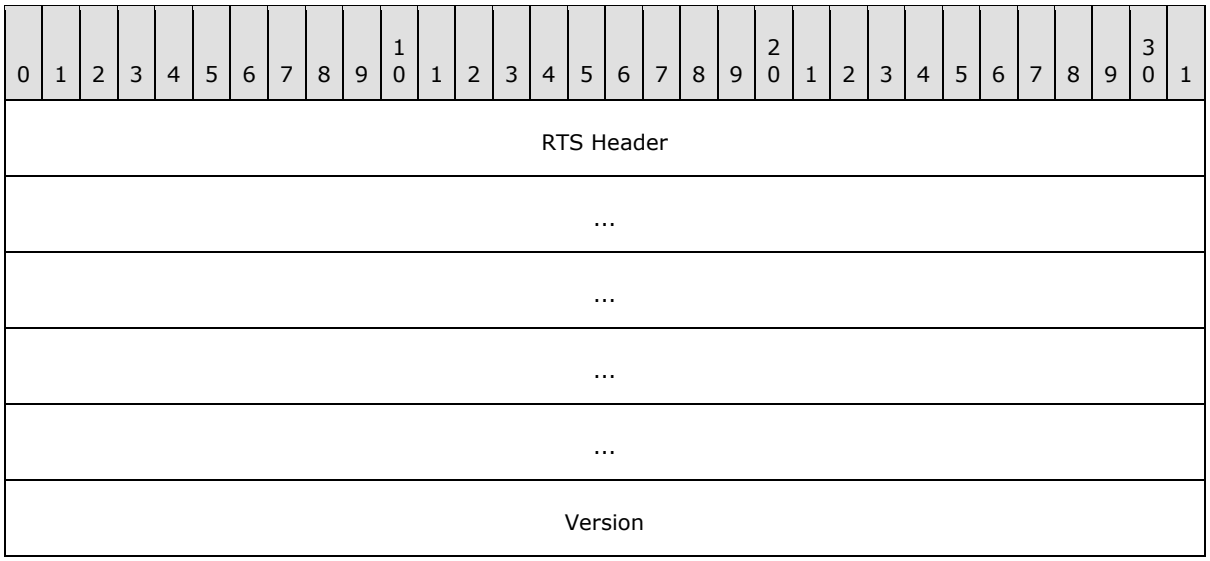


RTS Header (20 bytes): See section 2.2.3.6.1. The **Flags** field of the RTS Header MUST be the value RTS_FLAG_RECYCLE_CHANNEL. The **NumberOfCommands** field of RTS Header MUST be the value 1.

Destination (8 bytes): This MUST be a Destination command. The destination field for the destination command MUST be set to the value FDClient as specified in section 2.2.3.3. The destination command format is defined in section 2.2.3.5.14.

2.2.4.36 OUT_R2/A3 RTS PDU

The OUT_R2/A3 RTS PDU MUST be sent from the client to the successor outbound proxy on the successor OUT channel to continue initiating the establishment of a successor OUT channel.



...
VirtualConnectionCookie
...
...
...
...
PredecessorChannelCookie
...
...
...
...
SuccessorChannelCookie
...
...
...
...
ClientReceiveWindowSize
...

RTS Header (20 bytes): See section 2.2.3.6.1. The **Flags** field of the RTS Header MUST be the value RTS_FLAG_RECYCLE_CHANNEL. The **NumberOfCommands** field of RTS Header MUST be the value 5.

Version (8 bytes): This MUST be a Version command specifying the client RPC over HTTP v2 Protocol version. The format of the RPC over HTTP v2 Protocol version is defined in section 2.2.3.5.7.

VirtualConnectionCookie (20 bytes): This MUST be a Cookie command that contains the cookie of the virtual connection that this OUT channel belongs to. The cookie command format is defined in section 2.2.3.5.4.

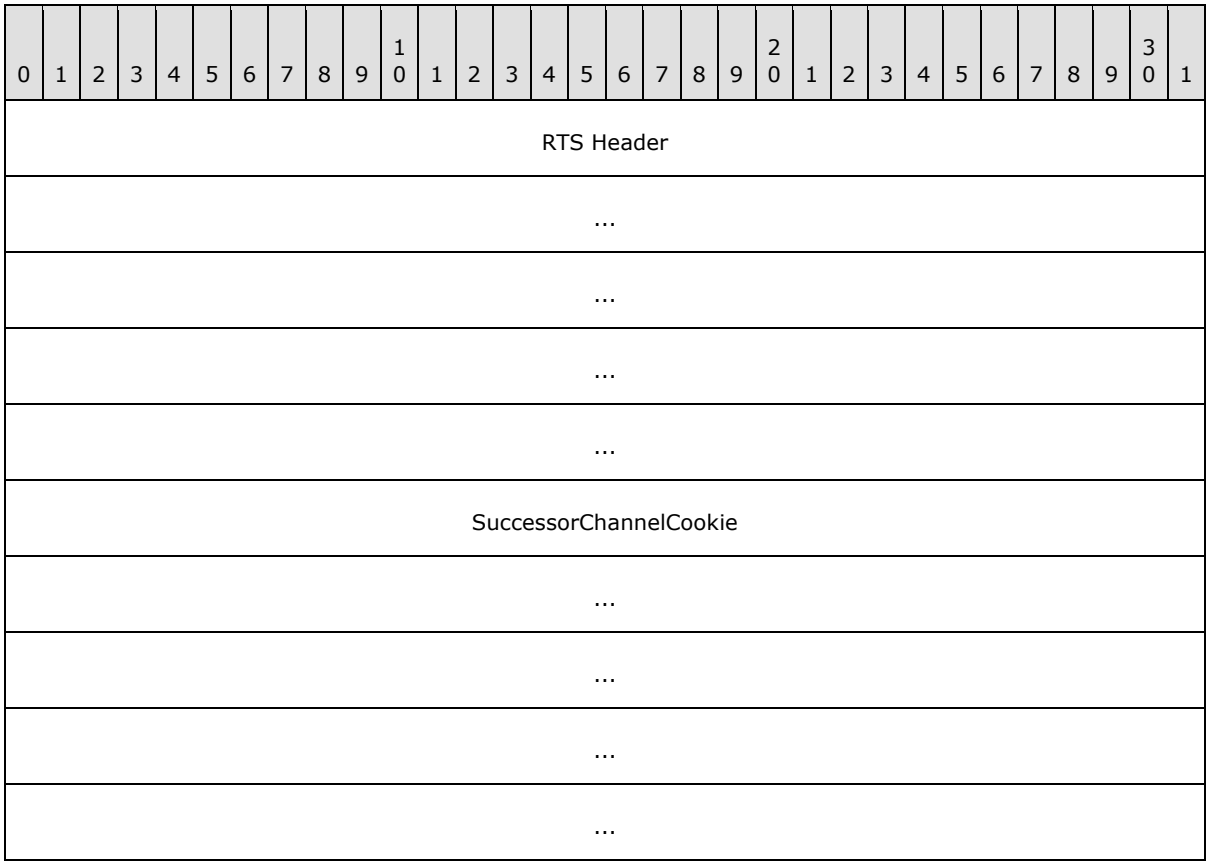
PredecessorChannelCookie (20 bytes): This MUST be a Cookie command that contains the cookie of the predecessor OUT channel. The cookie command format is defined in section 2.2.3.5.4.

SuccessorChannelCookie (20 bytes): This MUST be a Cookie command identifying the successor OUT channel cookie. The cookie command format is defined in section 2.2.3.5.4.

ClientReceiveWindowSize (8 bytes): This MUST be a ReceiveWindowSize command specifying the size of the receive window for the client OUT channel. The ReceiveWindowSize command format is defined in section 2.2.3.5.1.

2.2.4.37 OUT_R2/A4 RTS PDU

The OUT_R2/A4 RTS PDU MUST be sent from the outbound proxy to the server on the OUT channel to continue the establishment of a successor OUT channel.

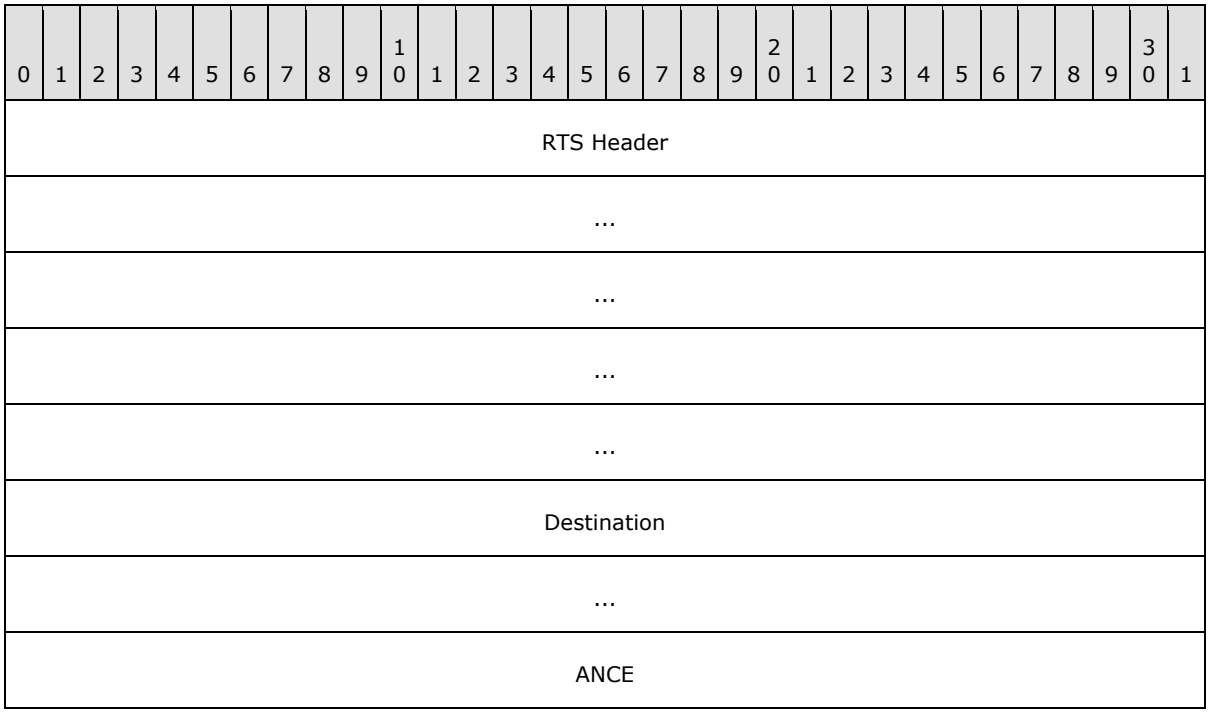


RTS Header (20 bytes): See section 2.2.3.6.1. The **Flags** field of the RTS Header MUST be the value RTS_FLAG_NONE. The **NumberOfCommands** field of RTS Header MUST be the value 1.

SuccessorChannelCookie (20 bytes): This MUST be a Cookie command identifying the successor OUT channel cookie. The cookie command format is defined in section 2.2.3.5.4.

2.2.4.38 OUT_R2/A5 RTS PDU

The OUT_R2/A5 RTS PDU MUST be sent from the server to the outbound proxy to indicate to it that the successor virtual OUT channel was established successfully.



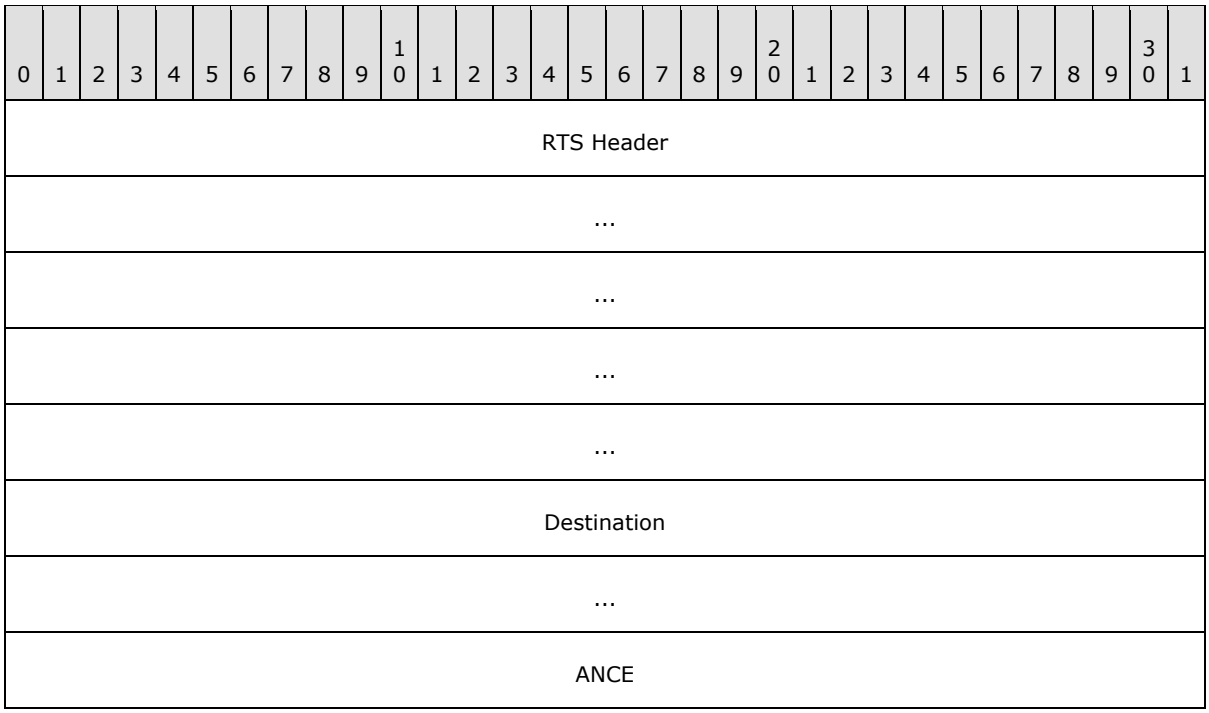
RTS Header (20 bytes): See section 2.2.3.6.1. The **Flags** field of the RTS Header MUST be the value RTS_FLAG_NONE. The **NumberOfCommands** field of RTS Header MUST be the value 2.

Destination (8 bytes): This MUST be a Destination command. The destination field for the destination command MUST be set to the value FDClient, as specified in section 2.2.3.3. The destination command format is defined in section 2.2.3.5.14.

ANCE (4 bytes): This MUST be an ANCE command. The format of the ANCE command is defined in section 2.2.3.5.11.

2.2.4.39 OUT_R2/A6 RTS PDU

The OUT_R2/A6 RTS PDU MUST be forwarded by the outbound proxy to the client as requested in the destination field. It serves the same purpose as OUT_R2/A5.



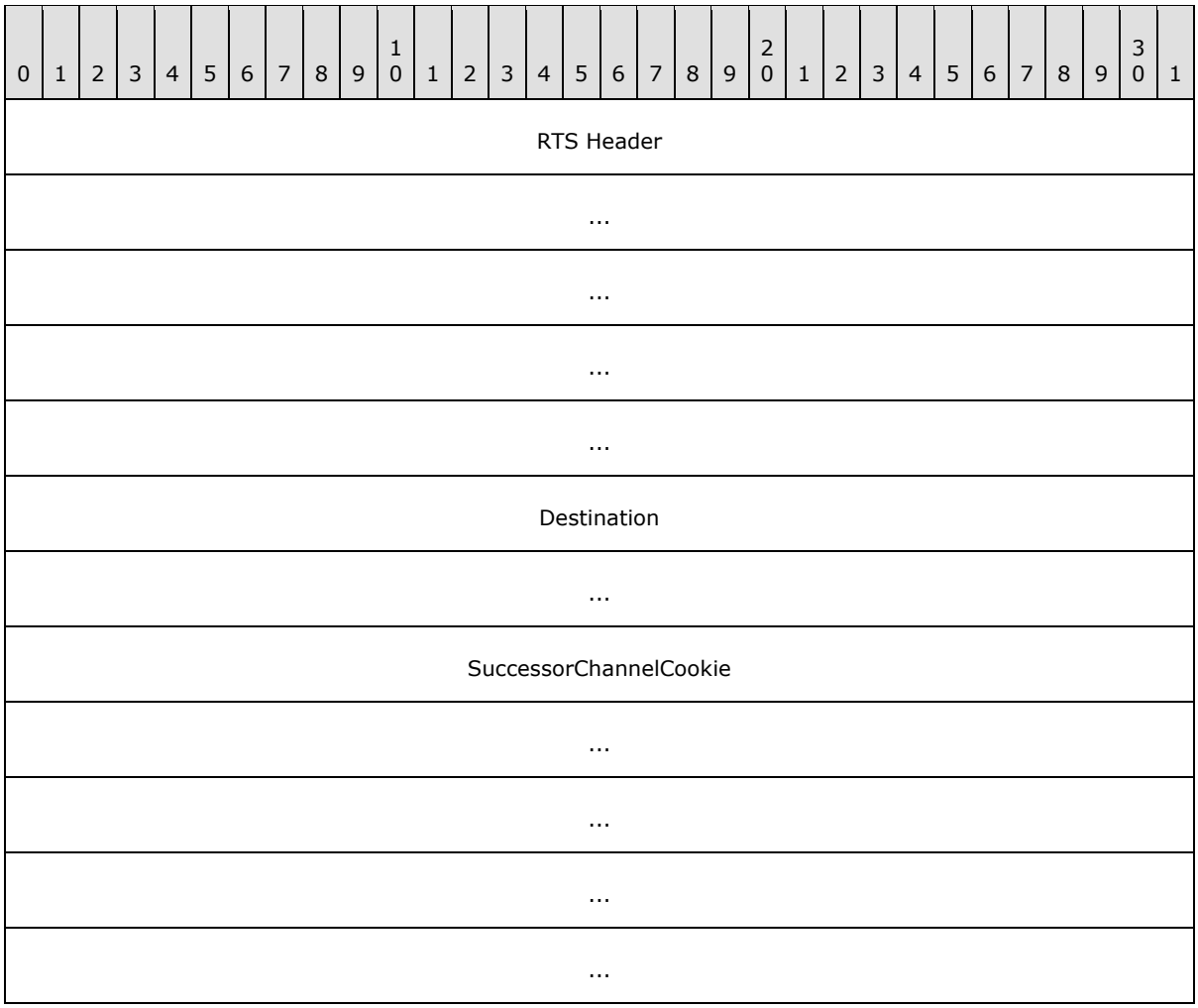
RTS Header (20 bytes): See section 2.2.3.6.1. The **Flags** field of the RTS Header MUST be the value RTS_FLAG_NONE. The **NumberOfCommands** field of RTS Header MUST be the value 2.

Destination (8 bytes): This MUST be a Destination command. The destination field for the destination command MUST be set to the value FDClient, as specified in section 2.2.3.3. The destination command format is defined in section 2.2.3.5.14.

ANCE (4 bytes): This MUST be an ANCE command. The format of the ANCE command is defined in section 2.2.3.5.11.

2.2.4.40 OUT_R2/A7 RTS PDU

The OUT_R2/A7 RTS PDU MUST be sent from the client to the inbound proxy on the IN channel to continue the establishment of a successor OUT channel.



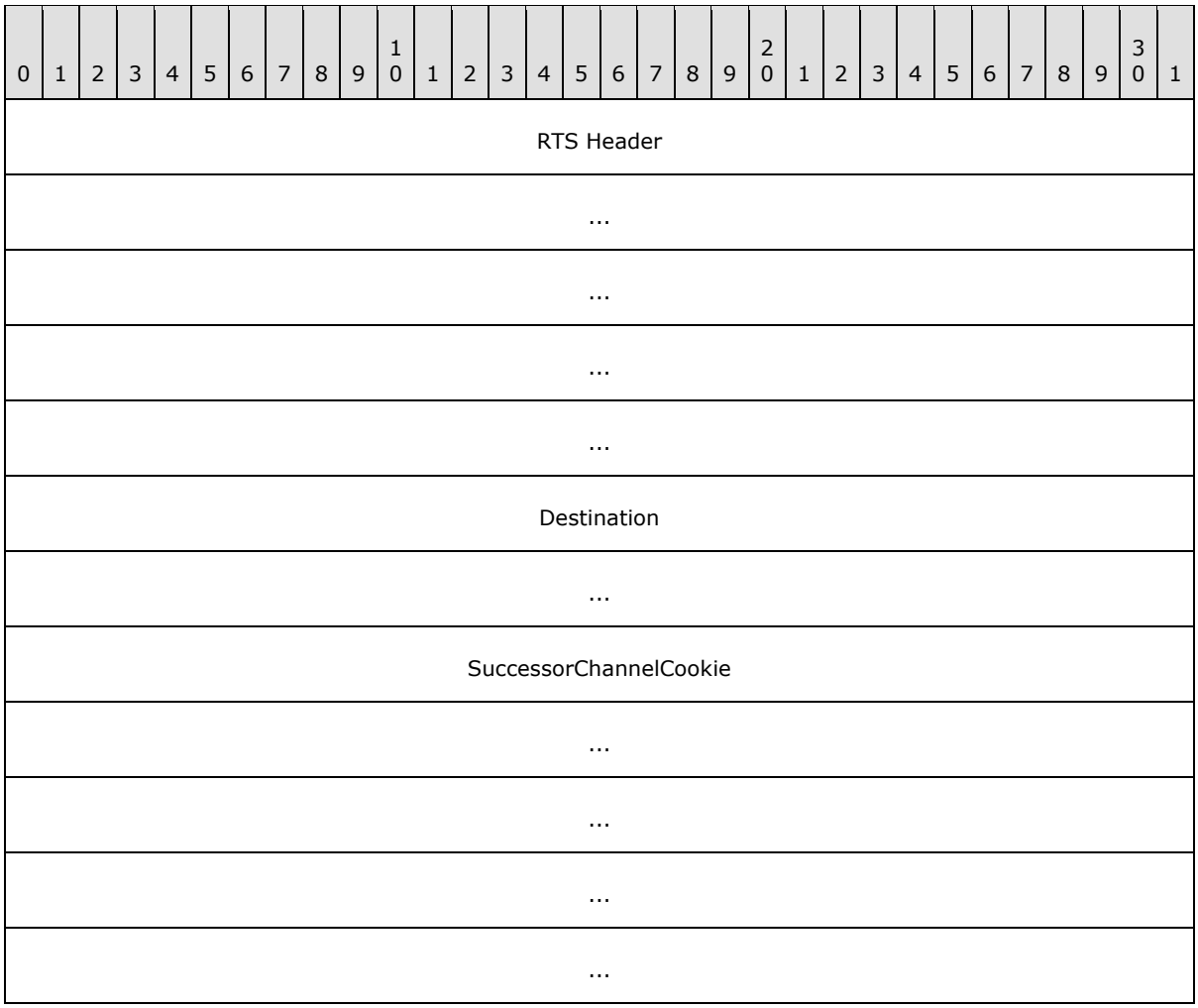
RTS Header (20 bytes): See section 2.2.3.6.1. The **Flags** field of the RTS Header MUST be the value RTS_FLAG_OUT_CHANNEL. The **NumberOfCommands** field of RTS Header MUST be the value 2.

Destination (8 bytes): This MUST be a Destination command. The destination field for the destination command MUST be set to the value FDServer, as specified in section 2.2.3.3. The destination command format is defined in section 2.2.3.5.14.

SuccessorChannelCookie (20 bytes): This MUST be a Cookie command identifying the successor OUT channel cookie. The cookie command format is defined in section 2.2.3.5.4.

2.2.4.41 OUT_R2/A8 RTS PDU

The OUT_R2/A8 RTS PDU MUST be sent from the inbound proxy to the server on the IN channel to continue the establishment of a successor OUT channel.



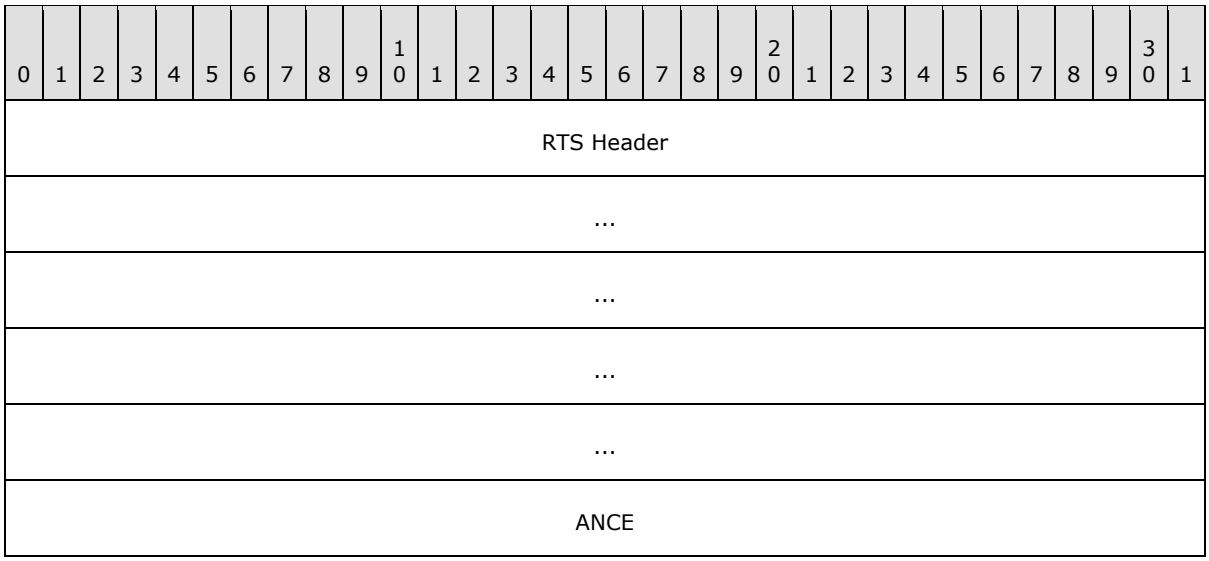
RTS Header (20 bytes): See section 2.2.3.6.1. The **Flags** field of the RTS Header MUST be the value RTS_FLAG_OUT_CHANNEL. The **NumberOfCommands** field of RTS Header MUST be the value 2.

Destination (8 bytes): This MUST be a Destination command. The destination field for the destination command MUST be set to the value FDServer, as specified in section 2.2.3.3. The destination command format is defined in section 2.2.3.5.14.

SuccessorChannelCookie (20 bytes): This MUST be a Cookie command identifying the successor OUT channel cookie. The cookie command format is defined in section 2.2.3.5.4.

2.2.4.42 OUT_R2/B1 RTS PDU

The OUT_R2/B1 RTS PDU MUST be sent from the server to the outbound proxy to indicate to it that the successor **virtual OUT channel** was established successfully.

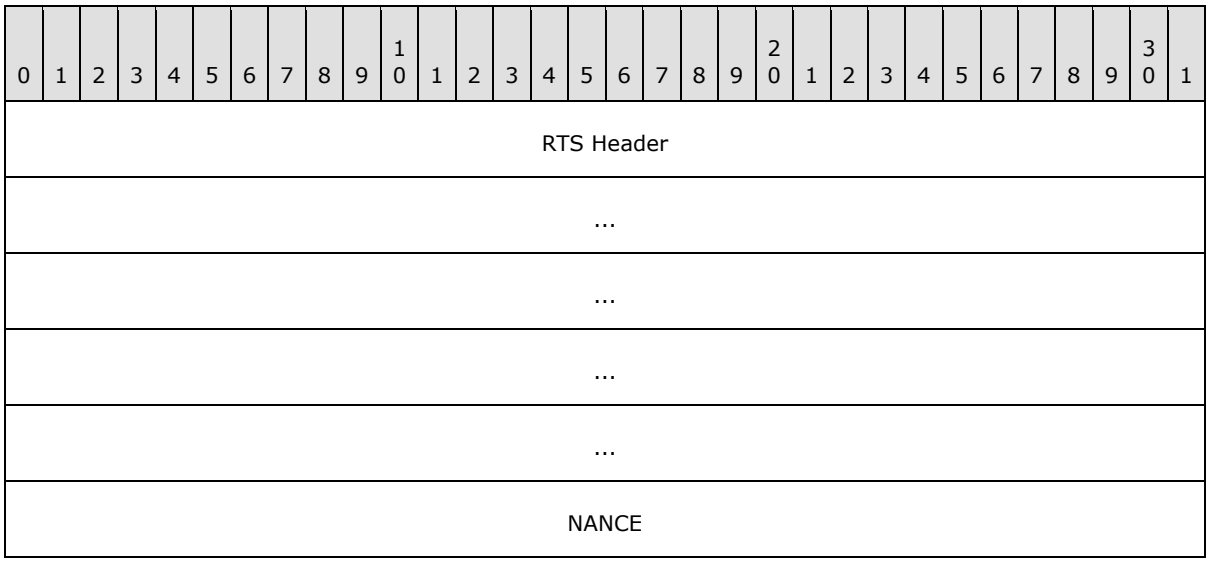


RTS Header (20 bytes): See section 2.2.3.6.1. The **Flags** field of the RTS Header MUST be the value RTS_FLAG_NONE. The **NumberOfCommands** field of RTS Header MUST be the value 1.

ANCE (4 bytes): This MUST be an ANCE command. The format of the ANCE command is defined in section 2.2.3.5.11.

2.2.4.43 OUT_R2/B2 RTS PDU

The OUT_R2/B2 RTS PDU MUST be sent from the server to the outbound proxy to indicate to it that the successor virtual OUT channel was not established successfully.

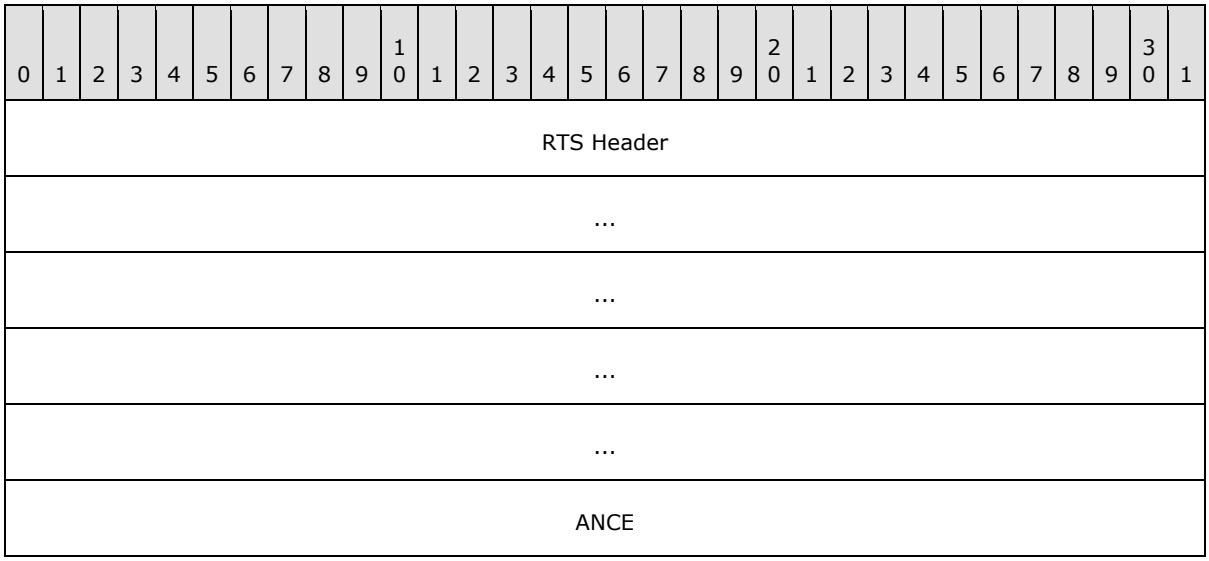


RTS Header (20 bytes): See section 2.2.3.6.1. The **Flags** field of the RTS Header MUST be the value RTS_FLAG_NONE. The **NumberOfCommands** field of RTS Header MUST be the value 1.

NANCE (4 bytes): This MUST be a NANCE command. The format of the NANCE command is defined in section 2.2.3.5.10.

2.2.4.44 OUT_R2/B3 RTS PDU

The OUT_R2/B3 RTS PDU MUST be sent from the outbound proxy to the client to indicate to it that the successor virtual OUT channel was established successfully.

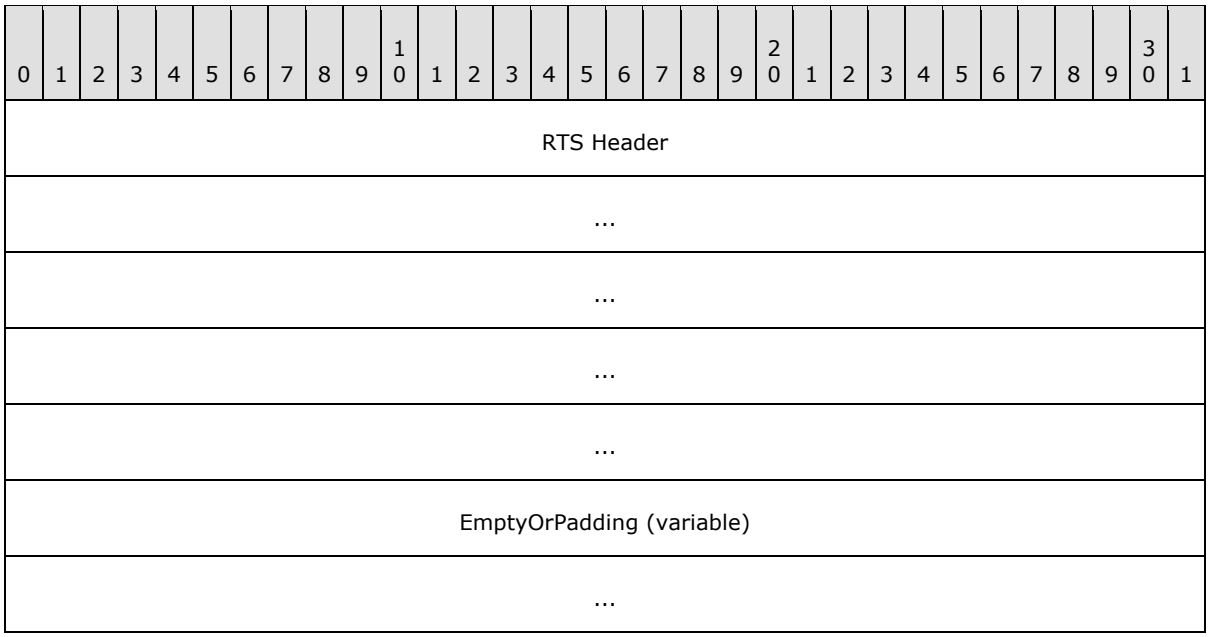


RTS Header (20 bytes): See section 2.2.3.6.1. The **Flags** field of the RTS Header MUST be the value RTS_FLAG_EOF. The **NumberOfCommands** field of RTS Header MUST be the value 1.

ANCE (4 bytes): This MUST be an ANCE command. The format of the ANCE command is defined in section 2.2.3.5.11.

2.2.4.45 OUT_R2/C1 RTS PDU

The OUT_R2/C1 RTS PDU MUST be sent from the client to the outbound proxy as part of the OUT_R2 protocol sequence to fill up the predeclared content length for the OUT channel HTTP request defined in section 2.1.2.1.2.

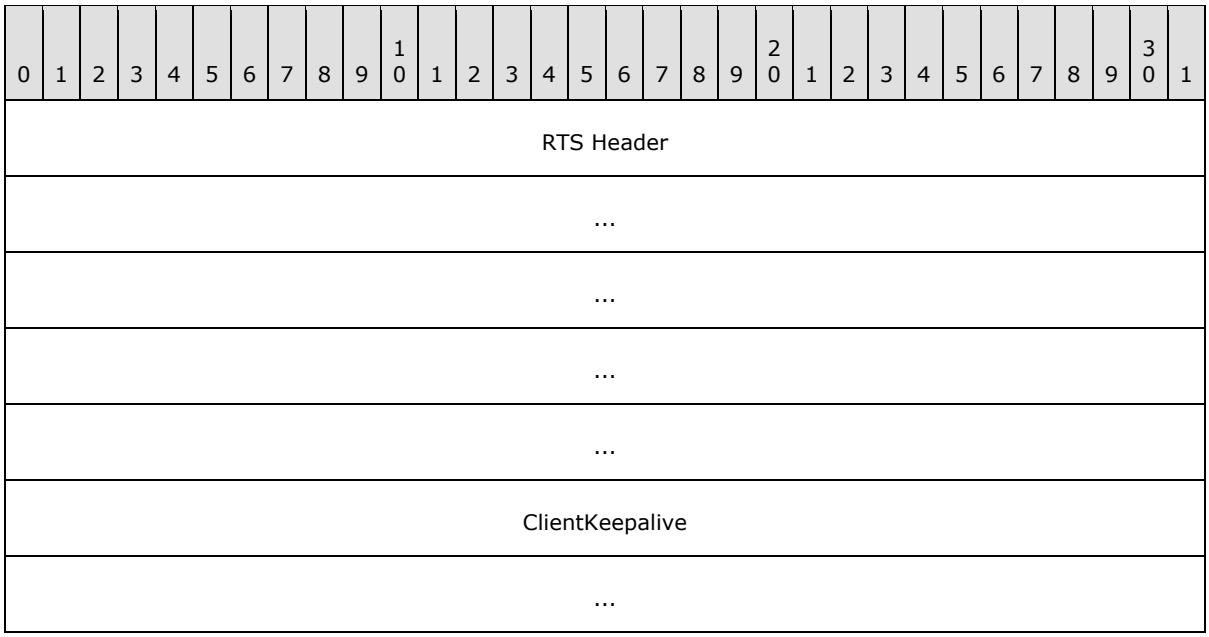


RTS Header (20 bytes): See section 2.2.3.6.1. The **Flags** field of the RTS Header MUST be the value RTS_FLAG_PING. The **NumberOfCommands** field of RTS Header MUST be the value 1.

EmptyOrPadding (variable): This MUST be an empty command or a padding command. This RTS PDU MUST be exactly the same size as OUT_R1/A11. Whichever of the two commands produces the desired PDU size MUST be used. If the padding command is used, the value for the **ConformanceCount** field MUST be chosen such that PDU has a size equal to the size of OUT_R1/A11. The empty command format is defined in section 2.2.3.5.8. The padding command format is defined in section 2.2.3.5.9.

2.2.4.46 Keep-Alive RTS PDU

The Keep-Alive RTS PDU is used outside a protocol sequence to tell the inbound proxy to modify the keep-alive settings on the IN channel between the inbound proxy and the server.

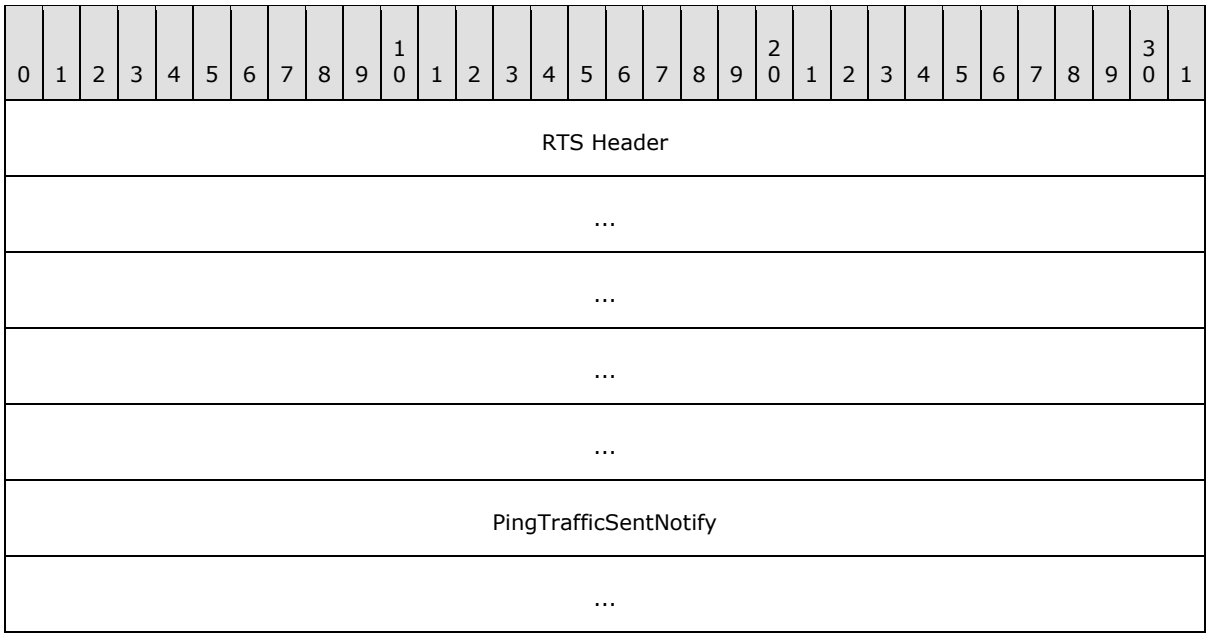


RTS Header (20 bytes): See section 2.2.3.6.1. The **Flags** field of the RTS Header MUST be the value RTS_FLAG_OTHER_CMD. The **NumberOfCommands** field of RTS Header MUST be the value 1.

ClientKeepalive (8 bytes): MUST be a Client Keepalive command specifying the keep-alive interval that the client wants the inbound proxy to use for the IN channel between the inbound proxy and the server. The ClientKeepalive command format is defined in section 2.2.3.5.6.

2.2.4.47 Ping Traffic Sent Notify RTS PDU

The Ping Traffic Sent Notify RTS PDU SHOULD be sent from the outbound proxy to the server on the OUT channel from the server to the outbound proxy, informing the server that the outbound proxy has sent a given number of bytes as ping traffic and the server MUST adjust its OUT channel lifetime. This RTS PDU is sent outside other protocol sequences.

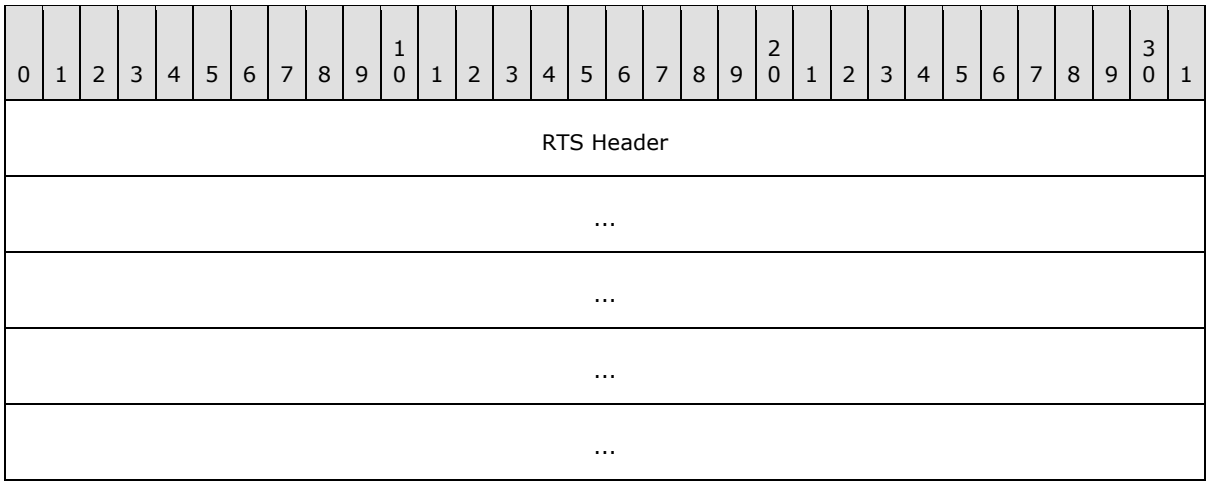


RTS Header (20 bytes): See section 2.2.3.6.1. The **Flags** field of the RTS Header MUST be the value RTS_FLAG_OTHER_CMD. The **NumberOfCommands** field of RTS Header MUST be the value 1.

PingTrafficSentNotify (8 bytes): MUST be a PingTrafficSentNotify command specifying the number of bytes sent by the outbound proxy on the OUT channel between the outbound proxy and the client. The format of the PingTrafficSentNotify command is defined in section 2.2.3.5.15.

2.2.4.48 Echo RTS PDU

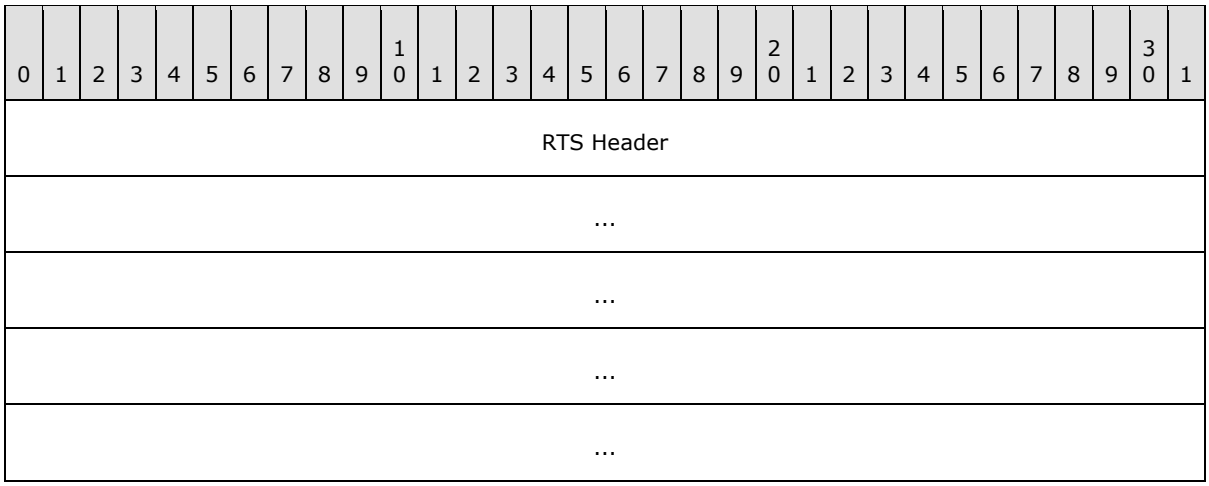
The Echo RTS PDU SHOULD be sent from the inbound or outbound proxy as the message body of the **echo response**message defined in section 2.1.2.1.6.



RTS Header (20 bytes): See section 2.2.3.6.1. The **Flags** field of the RTS Header MUST be the value RTS_FLAG_ECHO. The **NumberOfCommands** field of RTS Header MUST be the value 0.

2.2.4.49 Ping RTS PDU

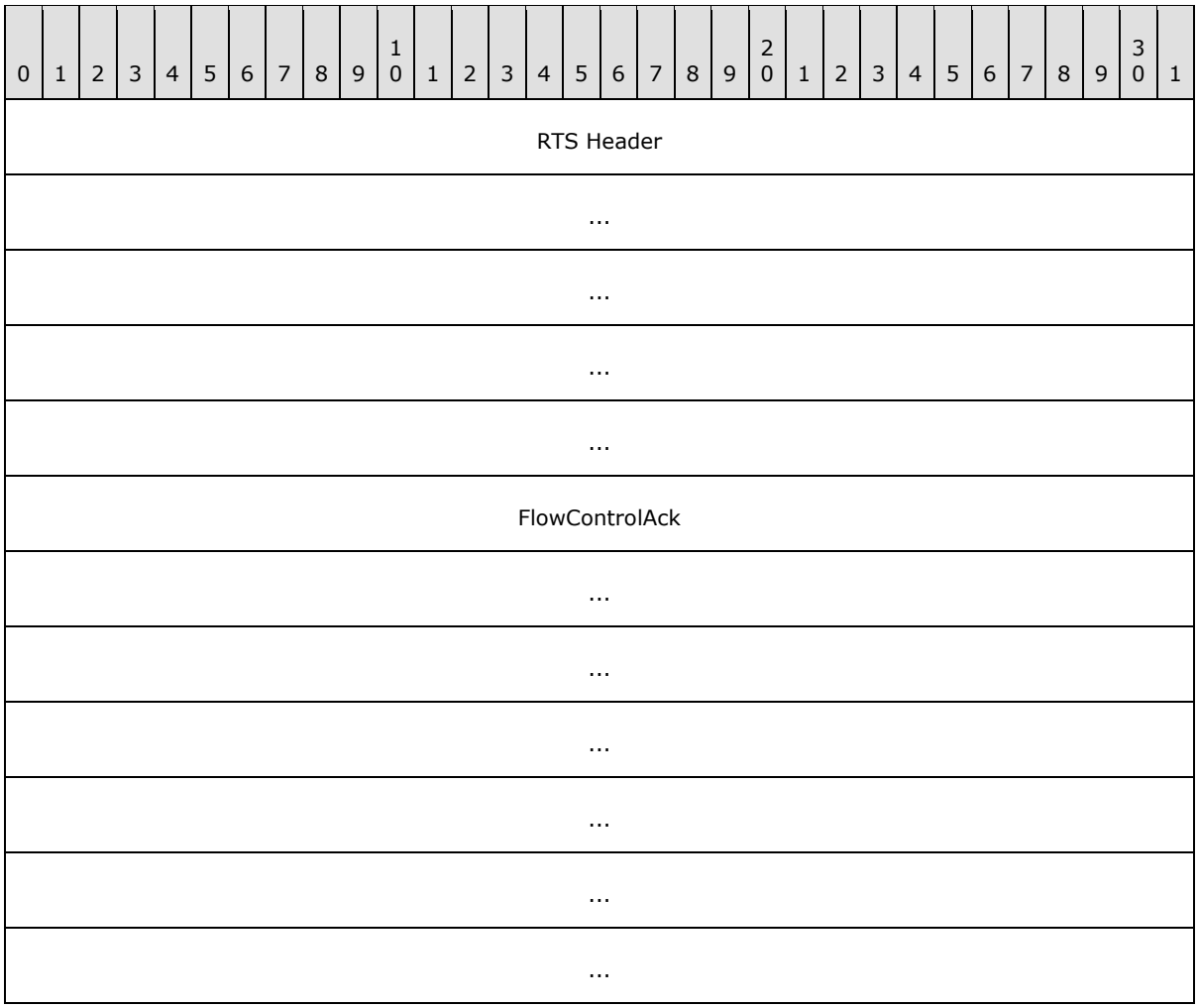
The Ping RTS PDU SHOULD be sent from the client to the inbound proxy and from the outbound proxy to the client. This PDU is sent outside other protocol sequences.



RTS Header (20 bytes): See section 2.2.3.6.1. The **Flags** field of the RTS Header MUST be the value RTS_FLAG_PING. The **NumberOfCommands** field of RTS Header MUST be the value 0.

2.2.4.50 FlowControlAck RTS PDU

The FlowControlAck RTS PDU MUST be sent from any recipient to its sender, and the forwarding rules in section 3.2.1.4.2 MUST be observed. This PDU is sent outside other protocol sequences.

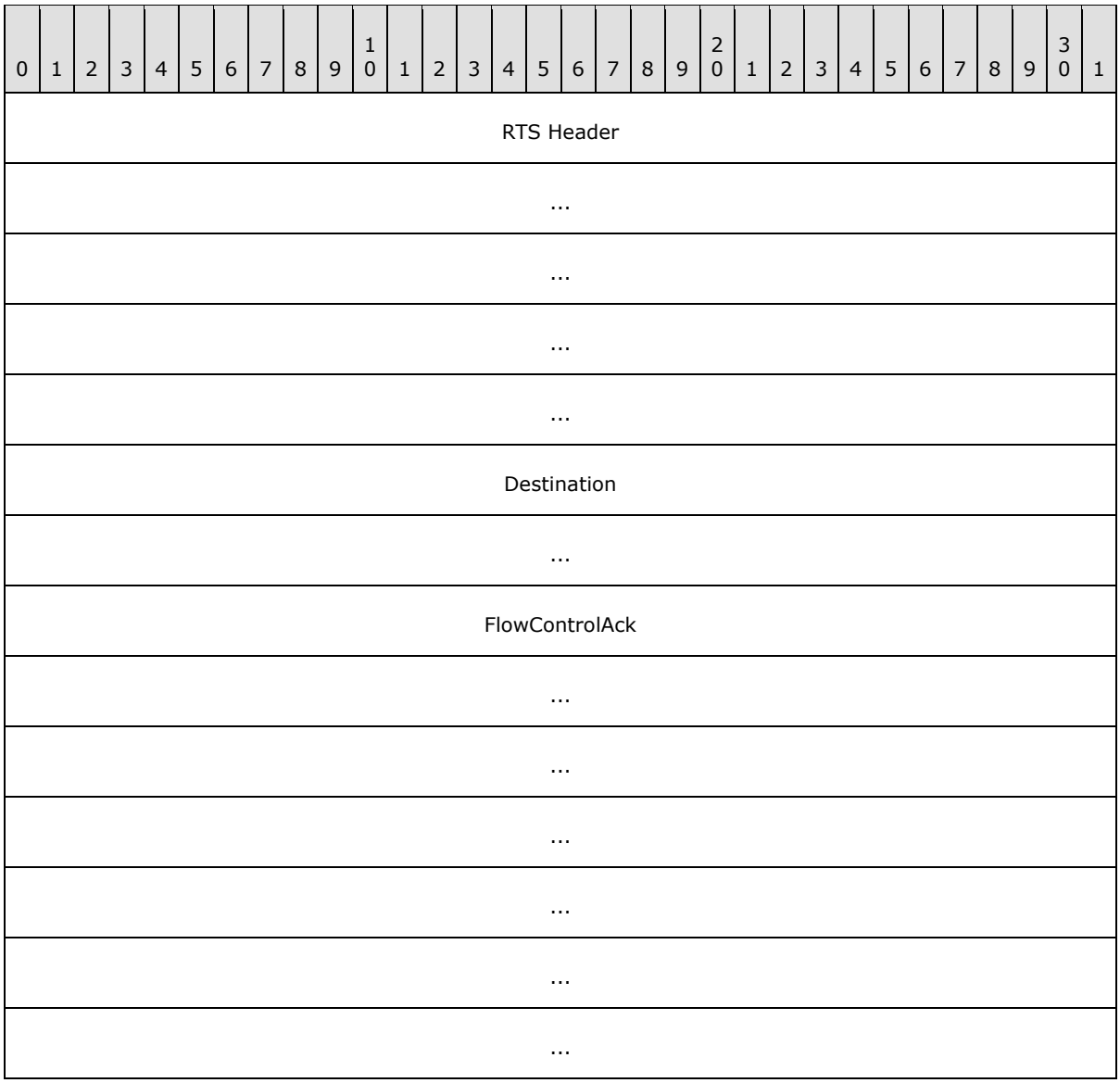


RTS Header (20 bytes): See section 2.2.3.6.1. The **Flags** field of the RTS Header MUST be the value RTS_FLAG_OTHER_CMD. The **NumberOfCommands** field of RTS Header MUST be the value 1.

FlowControlAck (28 bytes): MUST be a Flow Control Acknowledgment command containing the flow control acknowledgment. The format of the FlowControlAck command is defined in section 2.2.3.5.2.

2.2.4.51 FlowControlAckWithDestination RTS PDU

The FlowControlAckWithDestination RTS PDU MUST be sent from any recipient to its sender, and the forwarding rules in section 3.2.1.4.2 MUST be observed. This PDU is sent outside other protocol sequences.



RTS Header (20 bytes): See section 2.2.3.6.1. The **Flags** field of the RTS Header MUST be the value RTS_FLAG_OTHER_CMD. The **NumberOfCommands** field of RTS Header MUST be the value 2.

Destination (8 bytes): This MUST be a destination command. The destination command format is defined in section 2.2.3.5.14.

FlowControlAck (28 bytes): This MUST be a Flow Control Acknowledgment command containing the flow control acknowledgment. The format of the FlowControlAck command is defined in section 2.2.3.5.2.

3 Protocol Details

This section is divided into two parts. The first part defines the protocol roles and processing for RPC over HTTP v1. The second part deals with roles and processing for RPC over HTTP v2. The next paragraph specifies how the roles are assigned.

A client node SHOULD be capable of using both RPC over HTTP v1 and RPC over HTTP v2 Protocol Dialects. A client node SHOULD try to use the RPC over HTTP v2 Protocol Dialect first; if that fails, it SHOULD fall back to the RPC over HTTP v1 Protocol Dialect, unless it has knowledge obtained outside this protocol that RPC over HTTP v1 will not work. In this case, it MUST return an implementation-specific error to a higher-level protocol and not try RPC over HTTP v1.<19>

A server node SHOULD be capable of listening and responding to both RPC over HTTP v1 and RPC over HTTP v2 Protocol Dialects at the same time using the same network address and endpoint. Once a TCP/IP connection to it is established, the server MUST use the first PDU that it receives to determine whether the given TCP/IP connection will be used as part of an RPC over HTTP v1 virtual connection or an RPC over HTTP v2 virtual connection. If the server receives any RTS PDU, it MUST assume that this TCP/IP connection is part of an RPC over HTTP v2 virtual connection. If the first PDU the server receives on a given TCP/IP connection is an RPC PDU and not an RTS PDU, it MUST assume that the TCP connection is part of an RPC over HTTP v1 virtual connection.

All proxies SHOULD be capable of listening and responding to both RPC over HTTP v1 and RPC over HTTP v2 requests at the same time using the same URL. If a proxy receives an HTTP request with an RPC_CONNECT method, it MUST use the RPC over HTTP v1 Protocol Dialectand act as mixed proxy for this particular HTTP request. If it receives an HTTP request with the RPC_IN_DATA method, it MUST use the RPC over HTTP v2 Protocol Dialect, and it MUST act in the inbound proxy role for this particular HTTP request. If a proxy receives an HTTP request with the RPC_OUT_DATA method, it MUST use the RPC over HTTP v2 Protocol Dialect, and it MUST act in the outbound proxy role for this particular HTTP request.

When a proxy receives a message in a Protocol Dialect that it does not implement, it SHOULD process the message exactly as it processes any other message that it does not understand for the Protocol Dialects that it does implement. The processing rules for each Protocol Dialect are specified throughout this section.

3.1 RPC Over HTTP v1 Protocol Details

For all of its roles, RPC over HTTP v1 follows a very simple processing mechanism. Once the connection is established, the protocol acts as a pass-through mechanism where arriving data from the network is passed in an implementation-specific way to the next (higher) protocol layer without processing. Data sent by higher protocol layers is also sent on the network without processing.<20>

Details are given in the following sections.

3.1.1 Client Details

The client adheres to the following state machine.

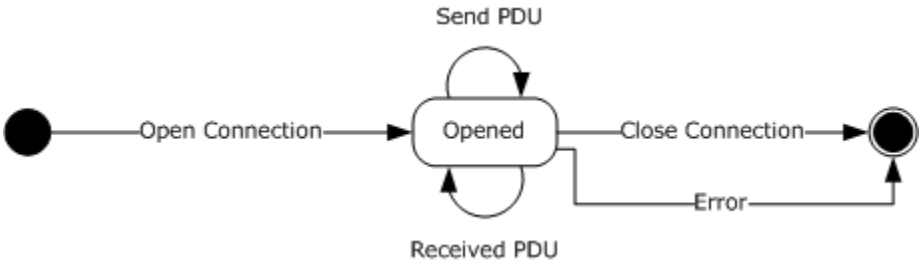


Figure 13: Client state machine

3.1.1.1 Higher-Layer Triggered Events

This section defines the higher-layer triggered events for the RPC over HTTP v1 client. There are three higher-layer triggered events: opening a connection, sending a PDU, and closing a connection.

3.1.1.1.1 Opening a Connection

When an implementation of a higher-level protocol calls an implementation of the RPC Over HTTP Protocol to open a new connection to the server, this protocol MUST perform the following sequence of steps.

- It will create the header of an RPC connect HTTP request as specified in section 2.1.1.1. The server_name component of the URI as defined in section 2.2.2 SHOULD be the network address given to RPC. The endpoint given to RPC will be placed in the server_port component of the URI as defined in section 2.2.2. Thus, the created HTTP request MUST be sent to a mixed proxy whose name is extracted from the network options given to the RPC runtime in an implementation-specific way.
- The client expects to receive an RPC connect response as described in section 2.1.1.1.2. It MUST treat any status code in the inclusive range 200 to 299 as an indication of success. Any other status code MUST be treated as an error and indicated to a higher layer in an implementation-specific way. Once this step is completed, the opening of the connection is considered done.

3.1.1.1.2 Sending a PDU

When an implementation of a higher-level protocol calls an implementation of this protocol to send a PDU to the server, an implementation of this protocol MUST copy the PDU as a BLOB in the message body of the RPC connect request as specified in section 2.1.1.1.3, and MUST send it to the mixed proxy.

3.1.1.1.3 Closing a Connection

When a higher-level protocol calls an implementation of this protocol to close the connection, an implementation of this protocol MUST call to the lower-level protocol to close the connection to the server.

3.1.1.2 Message Processing Events and Sequencing Rules

This section specifies the two message processing events for a client that implements the RPC over HTTP v1 Protocol Dialect. Those events include receiving a PDU and encountering a connection error.

3.1.1.2.1 Receiving a PDU

When an implementation of this protocol receives a PDU, it MUST pass it on to a higher-layer protocol without modifying the contents of the PDU. This happens in an implementation-specific way.<21>

3.1.1.2.2 Encountering a Connection Error

When an implementation of this protocol encounters an error on a connection, it MUST indicate this error to a higher-level protocol in an implementation-specific way and MUST treat the connection as closed.<22>

3.1.2 Mixed Proxy Details

The mixed proxy adheres to the following state machine.

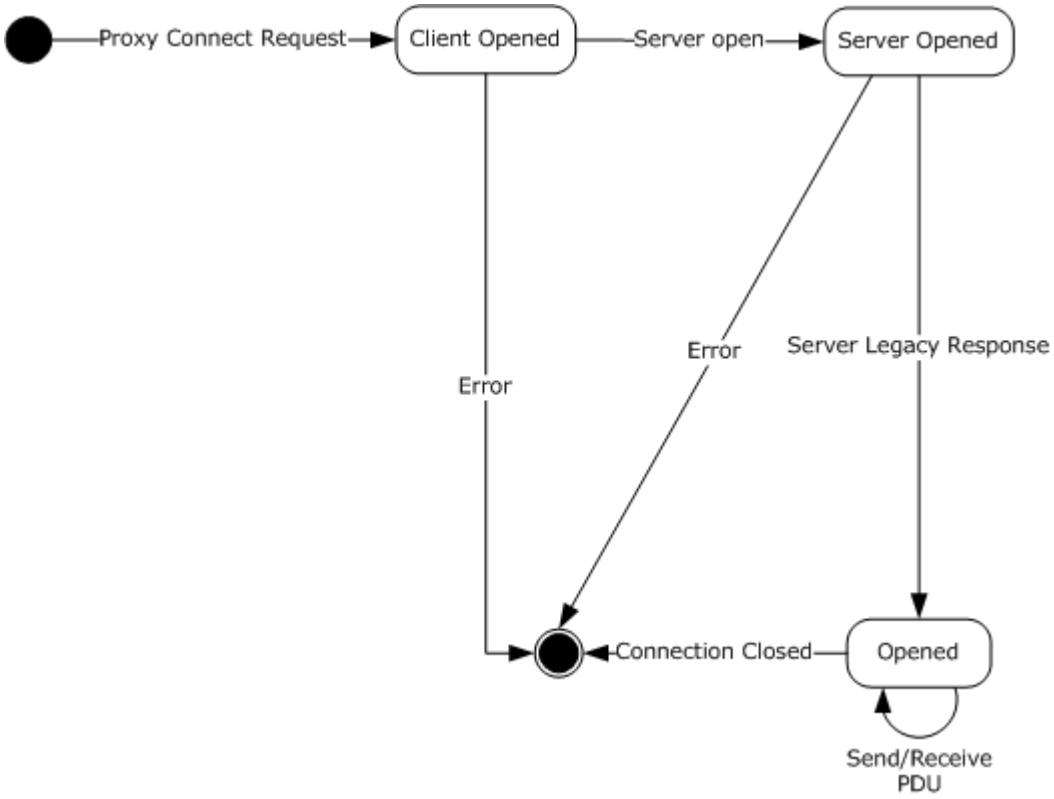


Figure 14: Proxy state machine

3.1.2.1 Initialization

Implementations of this protocol MUST listen on HTTP/HTTPS URL namespace "/rpc/rpcProxy.dll".

3.1.2.2 Message Processing Events and Sequencing Rules

This section specifies the message processing events for a mixed proxy that implements the RPC over HTTP v1 Protocol Dialect. Those events include receiving an RPC connect request, receiving a PDU, and encountering a connection close/connection error.

3.1.2.2.1 RPC Connect Request Received

When a mixed proxy receives an RPC connect request, it MUST retrieve the server name and server port from the URI of the RPC connect request as specified in section 2.2.2. It MUST establish a TCP connection to the server using the server name and port. It then waits for the server legacy response defined in section 2.1.1.2.1. The mixed proxy MUST NOT respond to PDUs received from the client as specified in section 3.1.2.2.2 until a server legacy response is received. When a server legacy response is received, the mixed proxy MUST respond to the client with the header of an RPC connect response as specified in section 2.1.1.1.2. It MUST also be prepared to receive PDUs coming in from the message body of the RPC connect request from the client, as specified in section 2.1.1.1.3, as well as PDUs coming from the server.

3.1.2.2.2 PDU Received

A mixed proxy may receive a PDU from the client or server. If a PDU is received from the client as defined in section 2.1.1.1.3, it MUST forward the PDU to the server. If a PDU is received from the server, it MUST forward it to the client as specified in section 2.1.1.1.4.

3.1.2.2.3 Connection Closed or Connection Error Encountered

Connection close and connection error MUST be handled identically. This section discusses connection close only. A connection close can be initiated by either the client or the server. If a connection close is initiated by the client, the mixed proxy MUST close the connection to the server and transition to the closed state. If a connection close is initiated by the server, the mixed proxy MUST close the connection to the client and transition to the closed state.

3.1.3 Server Details

The server adheres to the following state machine.

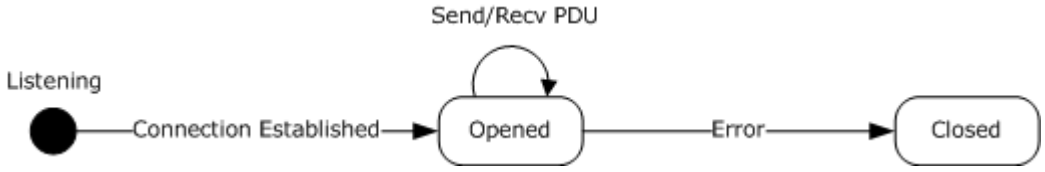


Figure 15: Server state machine

3.1.3.1 Initialization

Implementations of this protocol MUST listen on a TCP endpoint defined by a higher-level protocol.

3.1.3.2 Higher-Layer Triggered Events

This section specifies the processing that MUST occur when a higher-layer protocol sends a PDU on a server that implements the RPC over HTTP v1 Protocol Dialect.

3.1.3.2.1 Sending a PDU

When a higher-layer protocol sends a PDU on a server that implements the RPC over HTTP v1 Protocol Dialect, the PDU MUST be sent to the mixed proxy.

3.1.3.3 Message Processing Events and Sequencing Rules

This section specifies the message processing events for a server that implements the RPC over HTTP v1 Protocol Dialect. These events include establishing the connection, receiving a PDU, and encountering a connection error.

3.1.3.3.1 Establishing a Connection

When a connection to the server is established, the server MUST send a server legacy response as specified in section 2.1.1.2.1 and move to the opened state.

3.1.3.3.2 Receiving a PDU

When an implementation of this protocol receives a PDU, it MUST pass it on to a higher-layer protocol without modifying the contents of the PDU. This happens in an implementation-specific way.<23>

3.1.3.3.3 Encountering a Connection Error

When an implementation of this protocol encounters an error on a connection, it MUST indicate this error to a higher-level protocol in an implementation-specific way and MUST transition to the closed state.<24>

3.2 RPC over HTTP v2 Protocol Details

The client and server do not have fixed roles; each software agent that has an implementation of this protocol may act as a client, as a server, or as both. The role that a given network node assumes is determined by whether the higher-layer protocol initialized it as an RPC client or as an RPC server using the ncacn_http RPC protocol sequence as specified in [MS-RPCE], section 3.

The inbound proxies and outbound proxies are software processes that run on a network node and are usually set up by a network administrator. A single software agent can act as either an inbound proxy, an outbound proxy, or both. A proxy MUST act as an inbound proxy if it gets an IN channel request as defined in section 2.1.2.1.1. It MUST act as an outbound proxy if it gets an OUT channel request as defined in section 2.1.2.1.2. The scope of the role it assumes is for the virtual IN channel or OUT channel. A single network node can act as inbound proxy for a given virtual IN channel and at the same time as an outbound proxy for a given virtual OUT channel.

3.2.1 Common Details

Several processing aspects are either common between all RPC over HTTP v2 Protocol roles or impact multiple roles. They are described in this section.

3.2.1.1 Abstract Data Model

This section specifies the elements of the abstract data model for RPC over HTTP v2. Those elements include the relationship between the different abstractions, receive windows and flow control, and connection time out.

3.2.1.1.1 Virtual Connection, Virtual Channel Hierarchy, and Protocol Variables

Each role specified by this protocol MUST maintain a hierarchical data structure where at most one virtual IN and at most one virtual OUT channel are associated with a virtual connection. The virtual channels that are components of a given virtual connection are defined to belong to this virtual connection. Each virtual connection is identified uniquely between a client, any number of inbound

proxies, any number of outbound proxies, and a server using an RTS cookie known as a virtual connection cookie. All roles defined by this protocol maintain a protocol variable to store the virtual connection cookie for a specific virtual connection. The virtual connection cookie is generated by the client. Other parties acquire it by exchanging one or more PDUs with the client.

Each virtual IN channel is composed of an IN channel between a client and an inbound proxy and a second IN channel between an inbound proxy and a server. Both of these channels are defined to be components of the virtual channel and transitively to be components of the virtual connection. It is also said that they belong to the virtual channel and transitively to the virtual connection.

Each virtual OUT channel is composed of an OUT channel between a client and an outbound proxy and a second OUT channel between an outbound proxy and a server.

Both of these channels are defined to be components of the virtual channel and transitively to be components of the virtual connection. It is also said that they belong to the virtual channel and transitively to the virtual connection.

The relationship is illustrated by the following diagram:

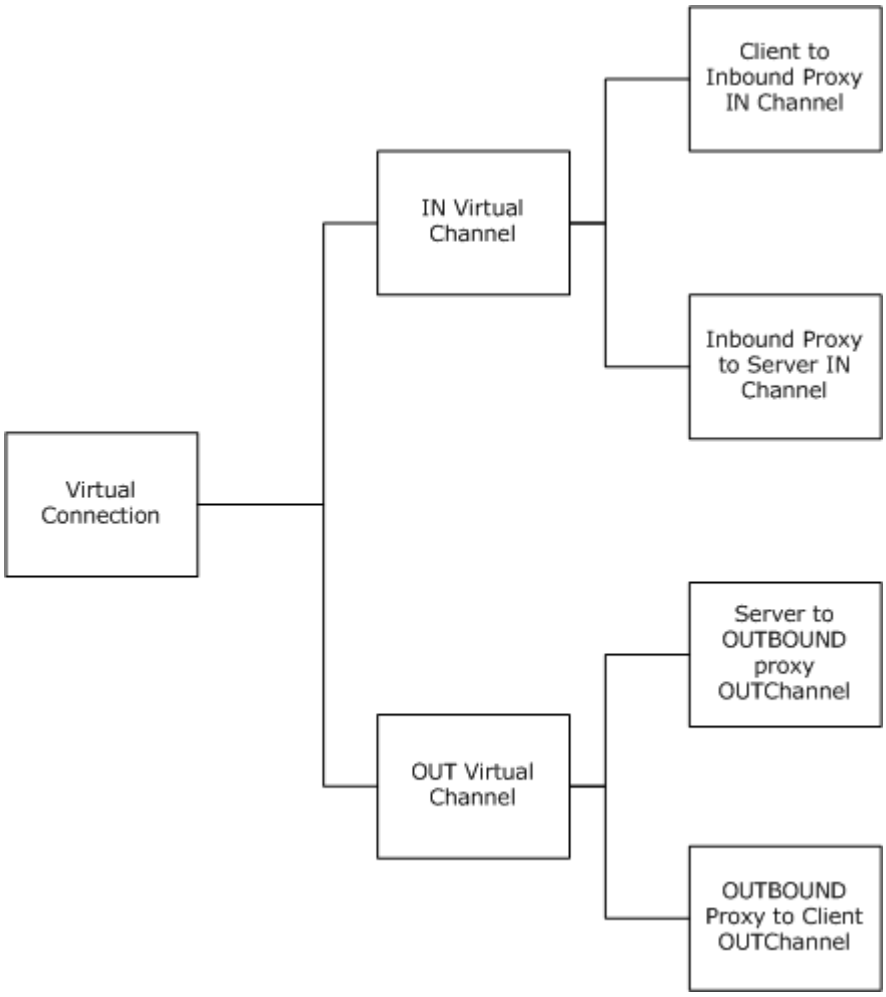


Figure 16: Virtual connection hierarchy

Each IN channel and OUT channel instance is identified uniquely between a client, one or more inbound proxies, one or more outbound proxies, and a server using an RTS cookie known as a "channel cookie".

As specified in sections 2.1.2.1.7 and 2.1.2.1.8, both virtual IN channel and virtual OUT channel are limited to transmitting only a certain number of bytes. For a virtual connection to be capable of sending an unlimited number of bytes, it MUST be able to discard IN channels or OUT channels whose lifetime has channel lifetime and replace them with successor IN channels or OUT channels. The process of discarding a predecessor IN channel or OUT channel and establishing a successor IN channel or OUT channel while ensuring that the reliable, in-order, at-most-once delivery guarantee is maintained is called channel recycling. The successor IN channel or OUT channel is called a predecessor replacement channel. During the recycling process, there is a period of time when both a predecessor channel and a successor channel instance is available. One of these is called the default channel and the other is called the non-default channel. The protocol sequences and message processing rules throughout section 3 specify which channel is the default in each particular case.

3.2.1.1.2 Receive Windows and Flow Control

Each IN channel or OUT channel has two parties, a sender, and a recipient. This section specifies an abstract data model that senders and recipients MUST adhere to in order to implement flow control for this protocol. This protocol specifies that only RPC PDUs are subject to the flow control abstract data model. RTS PDU and the HTTP request and response headers are not subject to flow control. Implementations of this protocol MUST NOT include them when computing any of the variables specified by this abstract data model.

The sections below define the separate protocol variables that are part of the receive windows and flow control data model.

3.2.1.1.2.1 ReceiveWindow

The first element of the abstract data model is the concept of a receive window. A receiver determines what amount of machine memory it is willing to commit to queue PDUs received from the sender. This amount of memory is called a receive window and on the abstract level the receiver MUST treat this data structure as a queue. The receiver SHOULD choose an initial value for the receive window based on an implementation-specific algorithm.<25>

3.2.1.1.2.2 Receiver AvailableWindow

As the receiver queues and releases PDUs in its receive window, it MUST locally keep track of how much space it has left in its receive window. The size of the receive window minus the sum of the size of all RPC PDUs that the receiver queued in this receive window is called space left or space available in the receive window. The protocol variable that contains the size of the space available is called Receiver AvailableWindow. This variable is initialized to be the same size as the ReceiveWindow variable.

3.2.1.1.2.3 Recipient BytesReceived

A third abstract variable that the receiver MUST keep is the total bytes received by it on that IN channel or OUT channel instance. This abstract variable is called BytesReceived and is incremented every time the receiver receives an RPC PDU. This variable is initialized to zero.

3.2.1.1.2.4 Send Queue

In the context of receive windows and flow control, a sender MUST maintain a queue of PDUs that it cannot immediately send for reasons specified in section 3.2.1.3.1.

3.2.1.1.2.5 BytesSent

The sender MUST keep track of the total bytes sent over the IN channel or OUT channel instance it uses to send PDUs. This abstract variable is called BytesSent. This variable is initialized to zero.

3.2.1.1.2.6 Sender AvailableWindow

The sender MUST keep track of the local size in bytes of the available receive window. This variable is called Sender AvailableWindow. This variable is initialized to be the same size as the ReceiveWindow variable on the recipient side.

3.2.1.1.2.7 AvailableWindowAdvertised

This variable MAY be maintained by implementations of this protocol. Implementations of this protocol MAY choose to implement the flow control algorithm without using this variable. In the latter case implementations of this protocol can skip the rest of this section.<26>

If an implementation chooses to maintain this protocol variable, it SHOULD follow the abstract data model in the rest of this section.

As specified in section 3.2.1.3.1, each time a receiver sends a flow control acknowledgment to the sender, it MUST advertise the size of the Receiver AvailableWindow.

This protocol variable keeps track of the value of the Receiver AvailableWindow the last time it advertised it to the sender. This variable is initialized to be the same size as the ReceiveWindow variable.

3.2.1.1.3 Connection Time Out

Network agents handling HTTP traffic often time out connections that are perceived as idle. An implementation of this protocol SHOULD try to prevent virtual connection that are still in use from being timed out by network agents handling the HTTP traffic. If network agents do time out connections perceived as idle, then clients, inbound proxies, and outbound proxies MUST maintain an abstract variable, which is the amount of time that the network agents handling the HTTP traffic are likely to allow an RPC over HTTP channel to remain open and idle. That abstract variable is called ConnectionTimeout.

Implementations of this protocol prevent IN channels and OUT channels that are in use from being timed out by said network agents by sending small packets between the client and the inbound proxy and the outbound proxy and the client. Details on this are provided in the sections for the client or for the outbound proxy, respectively.

3.2.1.2 Initialization

This section specifies the initialization steps that are common between all roles in the RPC over HTTP v2 Protocol Dialect.

3.2.1.2.1 Flow Control and Receive Window Processing

The receiver MUST advertise the size of the receive window using the ReceveWindowsSize RTS command as defined in section 2.2.3.5.1, and the sender MUST initialize its abstract data model from this RTS command. This advertising happens in a way that is specific to each role, and as such is defined in the section for each specific role.

3.2.1.3 Higher-Layer Triggered Events

This section specifies the flow control and receive window processing rules that are common among all roles in the RPC over HTTP v2 Protocol Dialect.

3.2.1.3.1 Flow Control and Receive Window Higher-Layer Triggered Events

3.2.1.3.1.1 Consuming RPC PDUs

Per the abstract data model defined in section 3.2.1.1.2.1, the receive window can be modeled as a queue. On the client and server, the act of releasing an RPC PDU from the receive window by a higher layer is called consuming this RPC PDU. On the inbound and outbound proxies, the act of forwarding an RPC PDU from the receive window to the next hop is also called consuming this RPC PDU. This section defines common processing for when an RPC PDU is consumed.

When the recipient consumes an RPC PDU from the receive window, it recalculates the Receiver AvailableWindow defined in section 3.2.1.1.2.2. If the Receiver AvailableWindow is determined to be greater than an implementation-specific threshold (as defined later in this section), the recipient will send to the sender a FlowControlAck RPC PDU as specified in section 2.2.4.50, indicating in the command the value of the protocol variable BytesReceived on the channel instance, the Receiver AvailableWindow during the time the FlowControlAck RTS PDU was sent, and the channel cookie specified for this channel in section 3.2.1.1.1. The receiver SHOULD choose a threshold value that keeps the number of FlowControlAck RPC PDUs small, while avoiding the sender queuing packets on high latency links.

The AvailableWindowAdvertised variable is updated to the Receiver AvailableWindow that was set in the last FlowControlAck RTS PDU.<27>

3.2.1.3.1.2 Sending RPC PDUs

Each time an RPC PDU is sent, an implementation of this protocol MUST:

- Increment the BytesSent protocol variable by the number of bytes in the RPC PDU sent.
- Decrement the Sender AvailableWindow by the same amount.

The sender MUST NOT send an RPC PDU if it will cause the local Sender AvailableWindow to become negative. In this case, it MUST queue the respective PDU on the send queue instead of sending it, until the Sender AvailableWindow is sufficiently large that sending the RPC PDU will result in a non-negative Sender AvailableWindow.

3.2.1.4 Message Processing Events and Sequencing Rules

This section specifies flow control and receive-window processing rules, PDU forwarding rules, and protocol sequences that are common among all roles in the RPC over HTTP v2 Protocol Dialect.

3.2.1.4.1 Flow Control and Receive Window Processing

This section specifies flow control and receive window processing rules common among all roles in the RPC over HTTP v2 Protocol Dialect.

3.2.1.4.1.1 Receiving RPC PDUs

As it receives RPC PDUs, an implementation of this protocol MUST queue the PDUs in its receive window. As it queues the PDUs, the recipient MUST:

- Decrement Receiver AvailableWindow by the number of bytes in the RPC PDU it queued.
- Increment BytesReceived by the same amount.
- If a protocol implementation implements AvailableWindowAdvertised, decrement it by the same amount.

When the sender receives a FlowControlAck PDU, it MUST use the following formula to recalculate its local copy of the Receiver AvailableWindow variable:

Receiver AvailableWindow = Receiver AvailableWindow_from_ack - (BytesSent - BytesReceived_from_ack)

If the Receiver AvailableWindow becomes negative or becomes greater than the ReceiveWindow advertised by the recipient, a sender SHOULD treat the FlowControlAck PDU as an invalid PDU and process it according to the rules for processing invalid PDUs, as defined in the section for the respective role.

3.2.1.4.1.2 FlowControlAck RTS PDU

All senders of RTS PDUs process flow control acknowledgment RTS PDUs as specified in section 2.2.4.50 identically. An implementation MUST execute the following sequence of steps to process a FlowControlAck RTS PDU in this order:

- A FlowControlAck RTS PDU is received on some channel.
- The **ChannelCookie** field from the FlowControlAck RTS command is compared against the channel cookies for all channels belonging to this virtual connection and a matching channel is selected. If no matching channel can be found, an implementation of this protocol MUST discard the PDU and MUST NOT do any further processing for this PDU.
- Recalculate its local copy of Sender AvailableWindow using the following formula:

  Sender AvailableWindow = Sender AvailableWindow_from_ack - (BytesSent - BytesReceived_from_ack)

- If Sender AvailableWindow becomes larger or equal to the size in bytes of the first RPC PDU in the queue, it MUST send the RPC PDU, update the protocol variables as defined in section 3.2.1.3.1.2, and repeat processing of this step until either there are no RPC PDUs in the queue, or the Sender AvailableWindow becomes smaller than the size in bytes of the first RPC PDU in the queue.

3.2.1.4.2 PDU Forwarding

The RPC over HTTP v2 IN channels and OUT channels that are based on an HTTP or HTTPS transport are half duplex. This means that one party may not be able to send a PDU to another party if the half duplex channel is going in the other direction. To resolve this problem, RPC over HTTP v2 uses

the concept of RTS PDU forwarding. When RTS PDU forwarding is used, a sender MUST mark a PDU as needing forwarding by setting an RTS destination command in the PDU. An implementation of this protocol MUST NOT add a destination command to a RTS PDU that does not have a destination command already. Only RTS PDUs that already have a destination command are subject to forwarding. Once the RTS PDU is marked for forwarding, a sender takes advantage of the fact that it knows that only the IN channel between client and inbound proxy and the OUT channel between the client and the outbound proxy are half duplex and MUST send the RTS PDU to the next hop according to the following table.

Sender	Destination	Next Hop
client	inbound proxy	direct
client	outbound proxy	inbound proxy
client	server	inbound proxy
inbound proxy	outbound proxy	server
inbound proxy	client	server
inbound proxy	server	direct
outbound proxy	inbound proxy	server
outbound proxy	client	direct
outbound proxy	server	direct
server	inbound proxy	direct
server	outbound proxy	direct
server	client	outbound proxy

If a sender has a "direct" value in the next hop column of the routing table, it MUST NOT use forwarding mechanism but instead MUST send the PDU directly.

Upon receiving such a RTS PDU, the receiver MUST forward the PDU to the next hop, which MUST be determined by indexing the table above by its own role as the value of the sender column and the destination as the value of the destination column.

3.2.1.4.3 Protocol Sequences

This section provides some diagrams and explanations that facilitate understanding sections 3.2.2, 3.2.3, 3.2.4, and 3.2.5. It is not intended as a replacement for these sections. The diagrams below illustrate at a high level the flow of RTS PDUs among the different roles during the different protocol sequences. They can be used to put into context the definitions used throughout the rest of the document.

3.2.1.4.3.1 Proxy Use Determination

This protocol sequence is intended to find whether a connection to an inbound or outbound RPC over HTTP v2 proxy can be established directly or whether an HTTP proxy needs to be used. A client implementation of this protocol that has local information allowing it to determine whether an inbound proxy or an outbound proxy can be reached directly or whether an HTTP proxy needs to be used MAY skip this protocol sequence and proceed to the next one.<28>

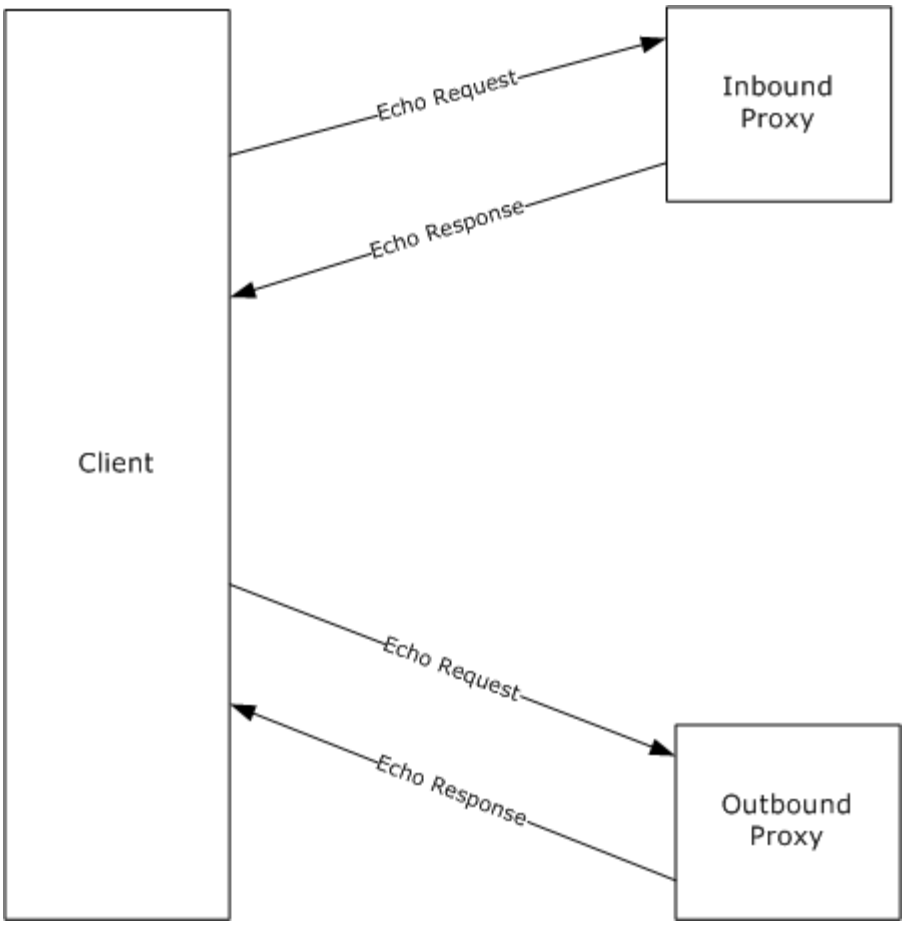


Figure 17: Proxy discovery

The above diagram summarizes the proxy discovery sequence. The client sends an echo request to either an inbound proxy or outbound proxy server. If the proxy server is available, it returns the echo response. The echo request and echo responsemessages are specified in sections 2.1.2.1.5 and 2.1.2.1.6 of this document.

The processing rules for echo request and echo responseare specified in sections 3.2.2.4.1.1 and 3.2.2.5.1 of this document.

3.2.1.4.3.2 Connection Establishment

This protocol sequence illustrates establishing a virtual connection between a client and a server. The name of this sequence is CONN. It has three PDU groups:

Group name	Meaning
A	PDUs sent on the OUT channels that initiate and perform the virtual connection establishment
B	PDUs sent on the IN channel

Group name	Meaning
C	PDUs sent on the OUT channels that complete virtual connection establishment

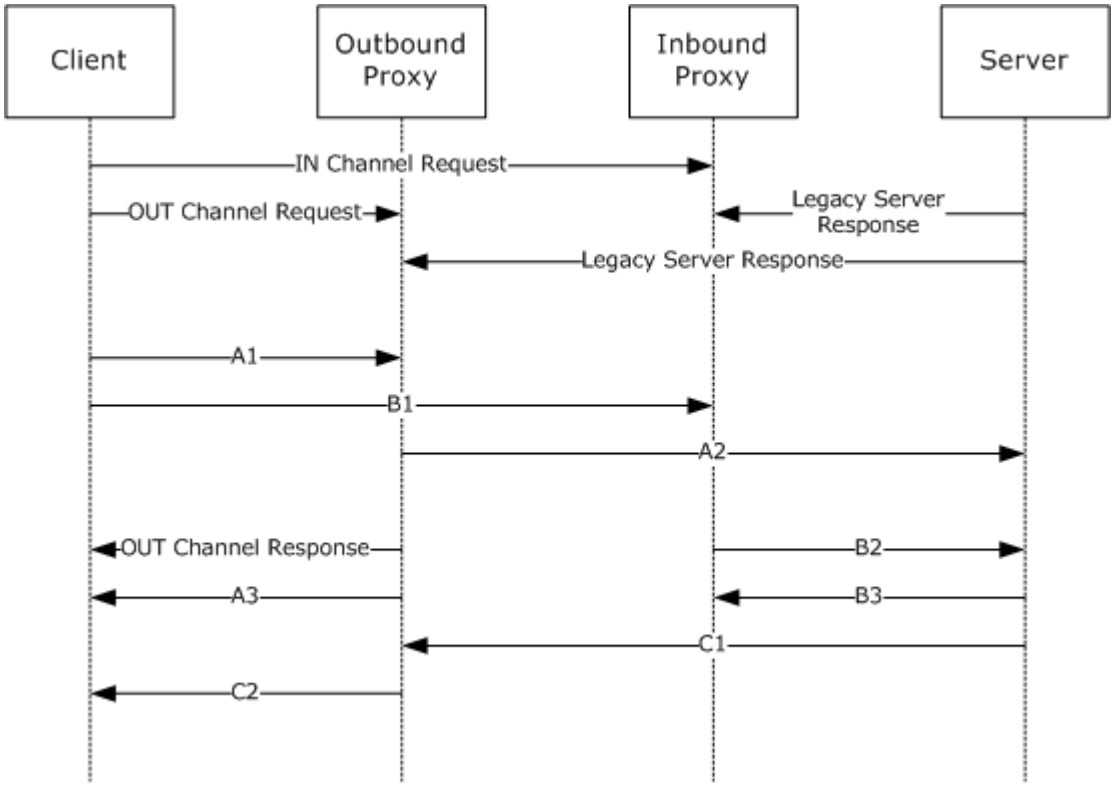


Figure 18: Connection establishment protocol sequence

The references for the PDUs used in this protocol sequence are:

Diagram label	PDU name and reference section
IN channel request	IN channel request (reference IN Channel Request (section 2.1.2.1.1))
OUT channel request	OUT channel request (reference OUT Channel Request (section 2.1.2.1.2))
OUT channel response	OUT channel response (reference Out Channel Response (section 2.1.2.1.4))
Legacy Server Response	Response sent to legacy proxy servers (reference Legacy Server Response (section 2.1.2.2.1))
A1	CONN/A1 RTS PDU (reference CONN/A1 RTS PDU (section 2.2.4.2))
A2	CONN/A2 RTS PDU (reference CONN/A2 RTS PDU (section 2.2.4.3))
A3	CONN/A3 RTS PDU (reference CONN/A3 RTS PDU (section 2.2.4.4))
B1	CONN/B1 RTS PDU (reference CONN/B1 RTS PDU (section 2.2.4.5))

Diagram label	PDU name and reference section
B2	CONN/B2 RTS PDU (reference CONN/B2 RTS PDU (section 2.2.4.6))
B3	CONN/B3 RTS PDU (reference CONN/B3 RTS PDU (section 2.2.4.7))
C1	CONN/C1 RTS PDU (reference CONN/C1 RTS PDU (section 2.2.4.8))
C2	CONN/C2 RTS PDU (reference CONN/C2 RTS PDU (section 2.2.4.9))

The processing rules for this protocol sequence are specified in sections 3.2.2 through 3.2.5 of this document.

Note In an effort to improve readability, the establishments of TCP connections are not shown in the figure.

3.2.1.4.3.3 IN Channel Recycling 1

This protocol sequence illustrates recycling of a virtual IN channel. The name of this sequence is IN_R1. It has two PDU groups:

Group Name	Meaning
A	PDUs that initiate and perform the virtual channel recycling
B	PDUs that complete the virtual channel recycling

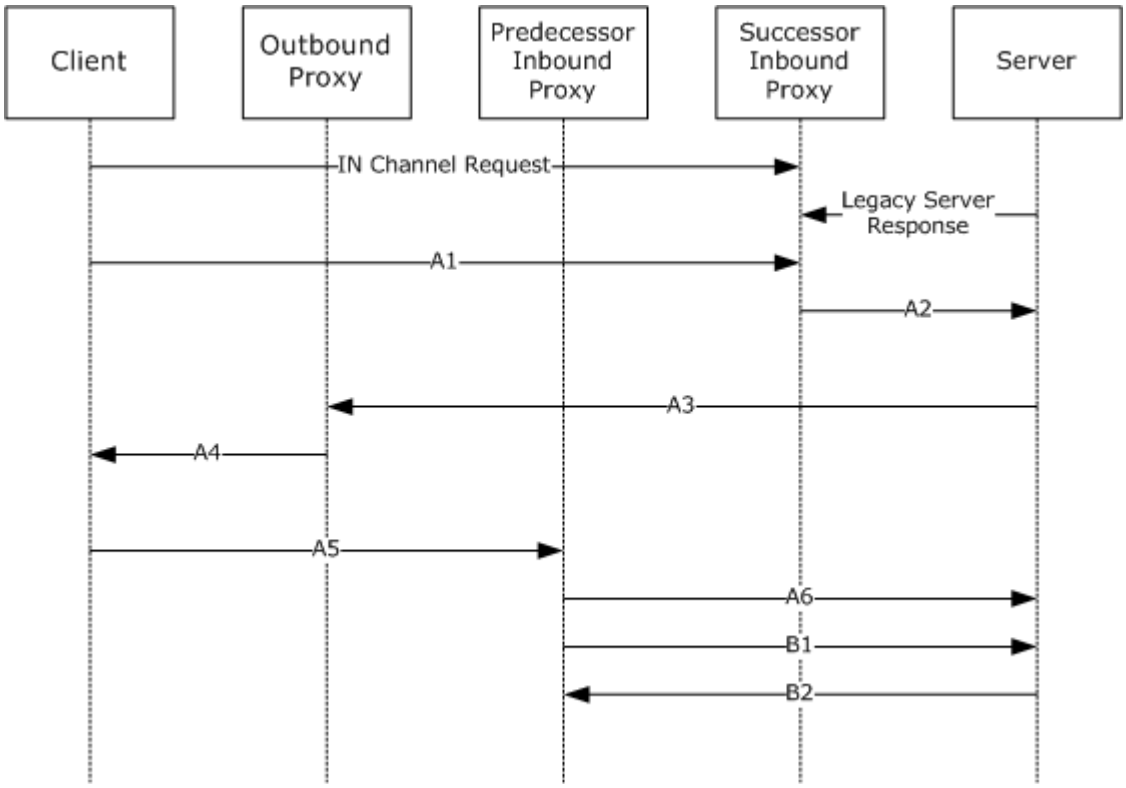


Figure 19: IN channel recycling 1 protocol sequence

The references for the PDUs used in this protocol sequence are shown in the following table.

Diagram label	PDU name and reference section
IN channel request	IN channel request (reference IN Channel Request (section 2.1.2.1.1))
Legacy Server Response	Response sent to legacy proxy servers (reference Legacy Server Response (section 2.1.2.2.1))
A1	IN_R1/A1 RTS PDU (reference IN_R1/A1 RTS PDU (section 2.2.4.10))
A2	IN_R1/A2 RTS PDU (reference IN_R1/A2 RTS PDU (section 2.2.4.11))
A3	IN_R1/A3 RTS PDU (reference IN_R1/A3 RTS PDU (section 2.2.4.12))
A4	IN_R1/A4 RTS PDU (reference IN_R1/A4 RTS PDU (section 2.2.4.13))
A5	IN_R1/A5 RTS PDU (reference IN_R1/A5 RTS PDU (section 2.2.4.14))
A6	IN_R1/A6 RTS PDU (reference IN_R1/A6 RTS PDU (section 2.2.4.15))
B1	IN_R1/B1 RTS PDU (reference IN_R1/B1 RTS PDU (section 2.2.4.16))
B2	IN_R1/B2 RTS PDU (reference IN_R1/B2 RTS PDU (section 2.2.4.17))

The processing rules for this protocol sequence are specified in sections 3.2.2 through 3.2.5.

Note In an effort to improve readability, the establishment of TCP connections are not shown in the figure.

3.2.1.4.3.4 IN Channel Recycling 2

This protocol sequence illustrates recycling of a virtual IN channel. The name of this sequence is IN_R2. This protocol sequence is very similar to protocol sequence IN_R1. They start identically, and only when processing IN_R1/A1 RTS PDU do they diverge based on dynamic decisions made by the inbound proxy as defined in section 3.2.3.5.5. This protocol sequence has a single PDU group: A.

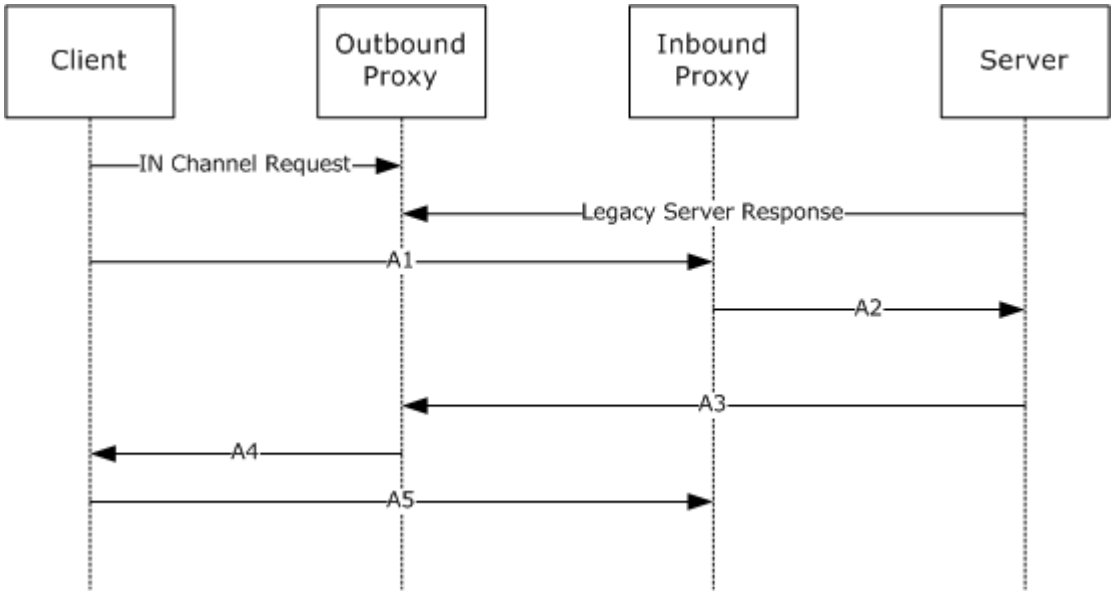


Figure 20: IN channel recycling 2 protocol sequence

The references for the PDUs used in this protocol sequence are:

Diagram label	PDU name and reference section
IN channel request	IN channel request (reference IN Channel Request (section 2.1.2.1.1))
Legacy Server Response	Response sent to legacy proxy servers (reference Legacy Server Response (section 2.1.2.2.1))
A1	IN_R2/A1 RTS PDU (reference IN_R2/A1 RTS PDU (section 2.2.4.18))
A2	IN_R2/A2 RTS PDU (reference IN_R2/A2 RTS PDU (section 2.2.4.19))
A3	IN_R2/A3 RTS PDU (reference IN_R2/A3 RTS PDU (section 2.2.4.20))
A4	IN_R2/A4 RTS PDU (reference IN_R2/A4 RTS PDU (section 2.2.4.21))
A5	IN_R2/A5 RTS PDU (reference IN_R2/A5 RTS PDU (section 2.2.4.22))

The processing rules for this protocol sequence are specified in sections 3.2.2 through 3.2.5 of this document.

Note In an effort to improve readability, the establishments of TCP connections are not shown in the figure.

3.2.1.4.3.5 OUT Channel Recycling 1

This protocol sequence illustrates recycling of a virtual OUT channel. The name of this sequence is OUT_R1. It has a single PDU group: A.

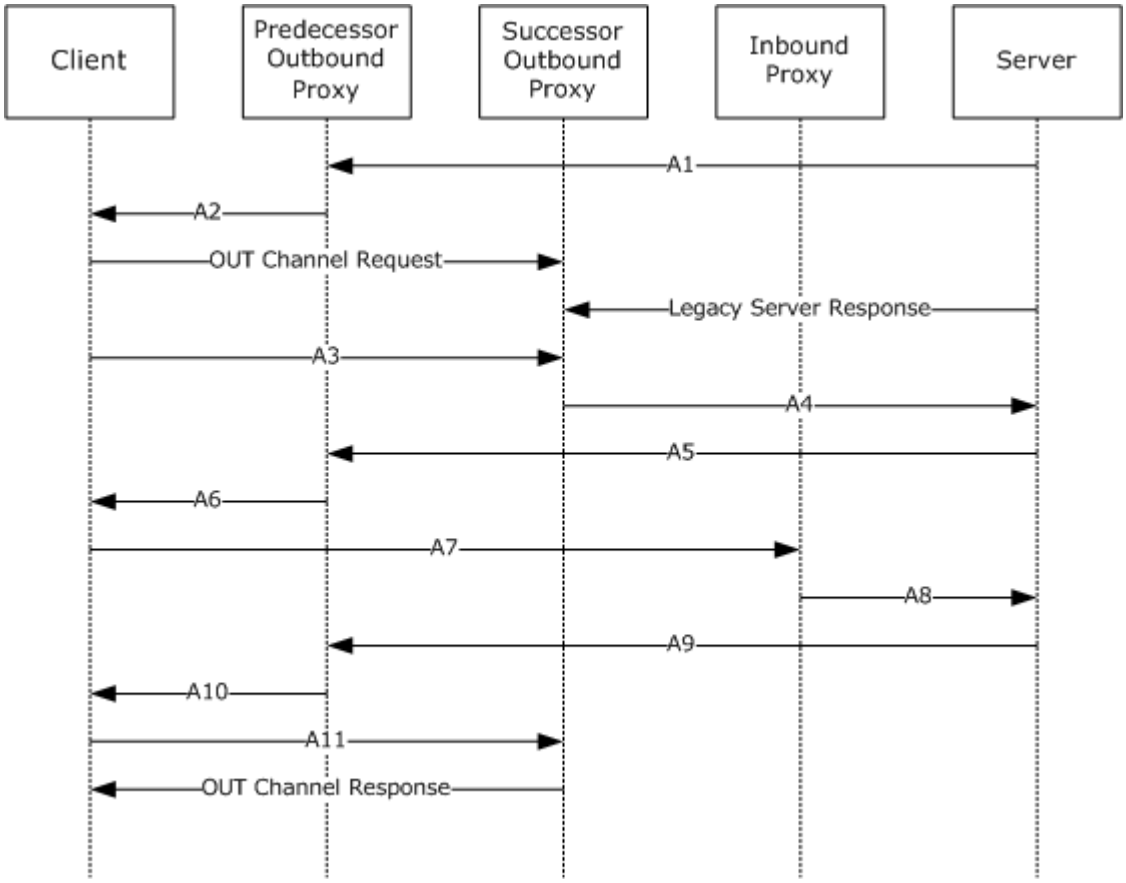


Figure 21: OUT channel recycling 1 protocol sequence

The references for the PDUs used in this protocol sequence are:

Diagram label	PDU name and reference section
OUT channel request	OUT channel request (reference OUT Channel Request (section 2.1.2.1.2))
OUT channel response	OUT channel response (reference OUT Channel Response (section 2.1.2.1.4))
Legacy Server Response	Response sent to legacy proxy servers (reference Legacy Server Response (section 2.1.2.2.1))
A1	OUT_R1/A1 RTS PDU (reference OUT_R1/A1 RTS PDU (section 2.2.4.23))
A2	OUT_R1/A2 RTS PDU (reference OUT_R1/A2 RTS PDU (section 2.2.4.24))
A3	OUT_R1/A3 RTS PDU (reference OUT_R1/A3 RTS PDU (section 2.2.4.25))
A4	OUT_R1/A4 RTS PDU (reference OUT_R1/A4 RTS PDU (section 2.2.4.26))
A5	OUT_R1/A5 RTS PDU (reference OUT_R1/A5 RTS PDU (section 2.2.4.27))
A6	OUT_R1/A6 RTS PDU (reference OUT_R1/A6 RTS PDU (section 2.2.4.28))

Diagram label	PDU name and reference section
A7	OUT_R1/A7 RTS PDU (reference OUT_R1/A7 RTS PDU (section 2.2.4.29))
A8	OUT_R1/A8 RTS PDU (reference OUT_R1/A8 RTS PDU (section 2.2.4.30))
A9	OUT_R1/A9 RTS PDU (reference OUT_R1/A9 RTS PDU (section 2.2.4.31))
A10	OUT_R1/A10 RTS PDU (reference OUT_R1/A10 RTS PDU (section 2.2.4.32))
A11	OUT_R1/A11 RTS PDU (reference OUT_R1/A11 RTS PDU (section 2.2.4.33))

The processing rules for this protocol sequence are specified in sections 3.2.2 through 3.2.5.

Note In an effort to improve readability, the establishments of TCP connections are not shown in the figure.

3.2.1.4.3.6 OUT Channel Recycling 2

This protocol sequence recycles a virtual OUT channel. The name of this sequence is OUT_R1. This protocol sequence is very similar to protocol sequence OUT_R1. The two start identically and while processing OUT_R1/A3 RTS PDU they diverge based on dynamic decisions made by the outbound proxy as specified in section 3.2.4.5.6. It has two PDU groups:

Group name	Meaning
A	PDUs that initiate and perform the virtual channel recycling
B	PDUs that complete the virtual channel recycling

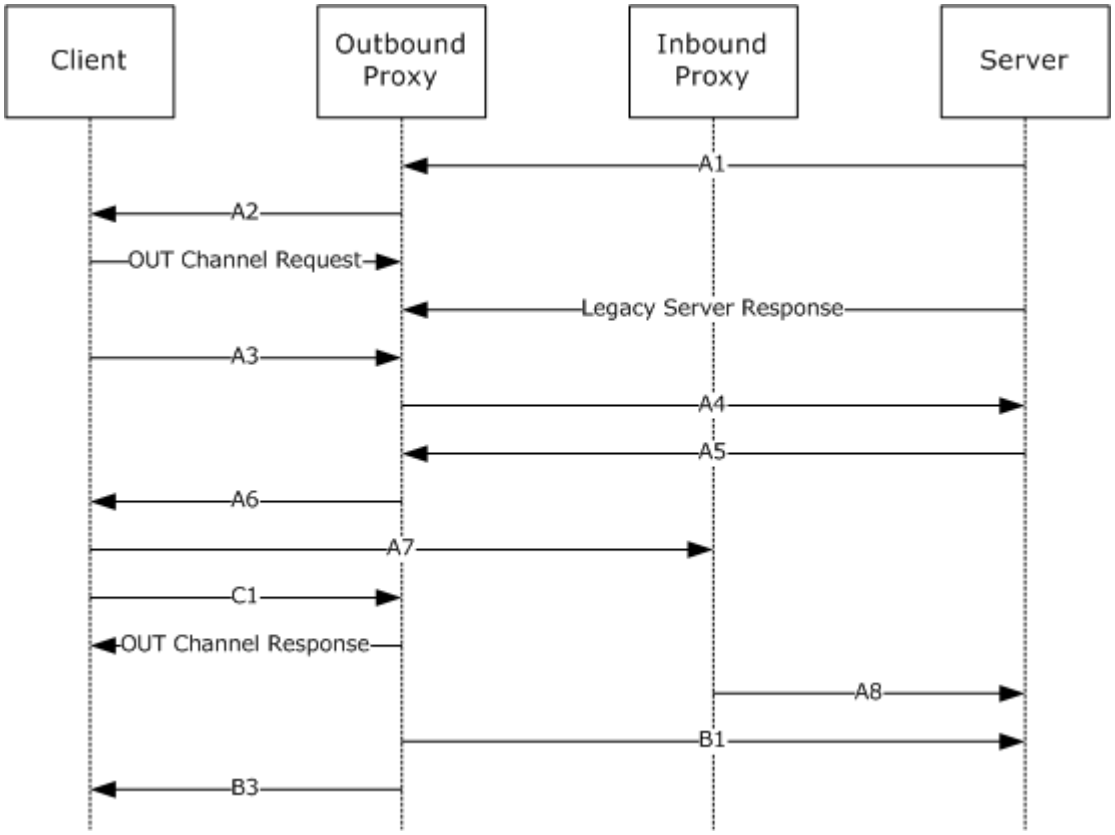


Figure 22: OUT channel recycling 2 protocol sequence

The references for the PDUs used in this protocol sequence are:

Diagram label	PDU name and reference section
OUT channel request	OUT channel request (reference OUT Channel Request (section 2.1.2.1.2))
OUT channel response	OUT channel response (reference OUT Channel Response (section 2.1.2.1.4))
Legacy Server Response	Response sent to legacy proxy servers (reference Legacy Server Response (section 2.1.2.2.1))
A1	OUT_R2/A1 RTS PDU (reference OUT_R2/A1 RTS PDU (section 2.2.4.34))
A2	OUT_R2/A2 RTS PDU (reference OUT_R2/A2 RTS PDU (section 2.2.4.35))
A3	OUT_R2/A3 RTS PDU (reference OUT_R2/A3 RTS PDU (section 2.2.4.36))
A4	OUT_R2/A4 RTS PDU (reference OUT_R2/A4 RTS PDU (section 2.2.4.37))
A5	OUT_R2/A5 RTS PDU (reference OUT_R2/A5 RTS PDU (section 2.2.4.38))
A6	OUT_R2/A6 RTS PDU (reference OUT_R2/A6 RTS PDU (section 2.2.4.39))
A7	OUT_R2/A7 RTS PDU (reference OUT_R2/A7 RTS PDU (section 2.2.4.40))

Diagram label	PDU name and reference section
A8	OUT_R2/A8 RTS PDU (reference OUT_R2/A8 RTS PDU (section 2.2.4.41))
B1	OUT_R2/B1 RTS PDU (reference OUT_R2/B1 RTS PDU (section 2.2.4.42))
B3	OUT_R2/B3 RTS PDU (reference OUT_R2/B3 RTS PDU (section 2.2.4.44))
C1	OUT_R2/C1 RTS PDU (reference OUT_R2/C1 RTS PDU (section 2.2.4.45))

The processing rules for this protocol sequence are specified in sections 3.2.2 through 3.2.5.

Note In an effort to improve readability, the establishments of TCP connections are not shown in the figure.

3.2.2 Client Details

This section defines the protocol details for the client role in the RPC over HTTP v2 Protocol Dialects.

An implementation of this protocol on the client MUST conform to the state machines given below. The first state machine is the overall client state machine for the virtual connection. This overall client state machine defines the relationship of the other state machines given here. Details about when the state machines are started and the state transitions made by these state machines are given later in this section.

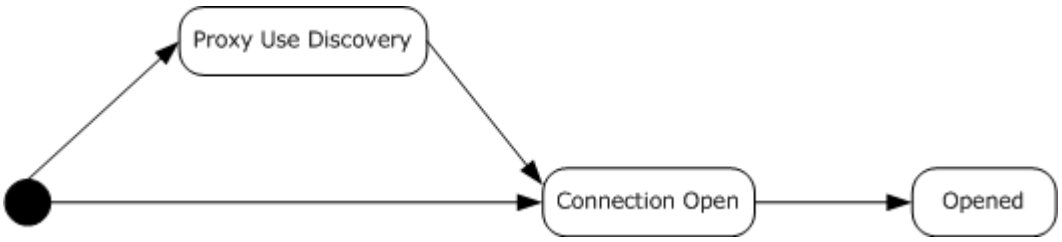


Figure 23: Overall client state machine

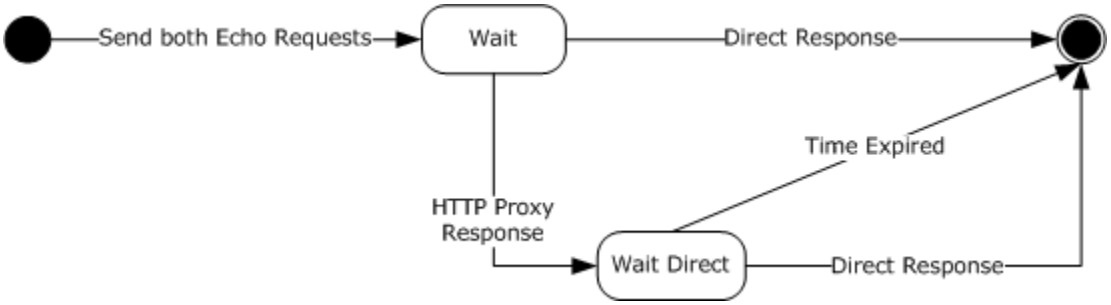


Figure 24: Proxy use determination

The proxy use determination state machine MUST be used when the client is trying to determine if it will use an HTTP proxy for communicating with the inbound proxy and outbound proxy. For more details on proxy use determination, see sections 3.2.1.4.3.1 and 3.2.2.4.1.1.

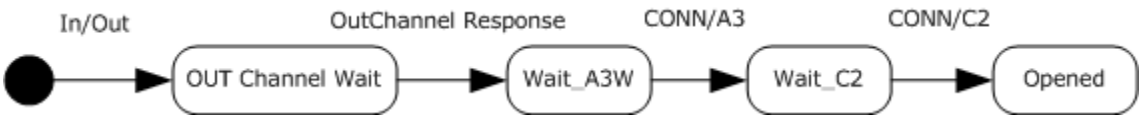


Figure 25: Virtual connection open

The virtual connection open state machine MUST be used when the client is trying to establish a virtual connection to the server. For more details on establishing a virtual connection, see sections 3.2.1.4.3.2 and 3.2.2.4.1.2.

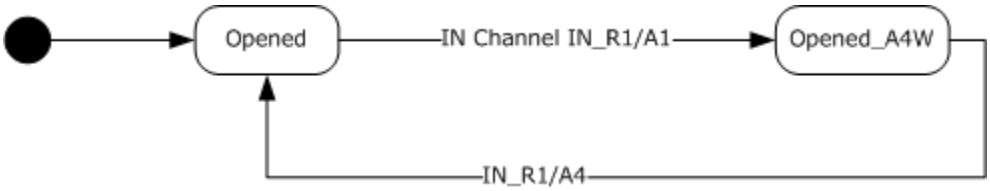


Figure 26: Client virtual IN channel state machine

The virtual IN channel state machine MUST be used when the client is trying to recycle an IN channel. It uses the protocol sequence IN_R1 as specified in section 3.2.1.4.3.3 or IN_R2 as specified in section 3.2.1.4.3.4. For more details on recycling an IN channel, see section 3.2.2.4.2.1.

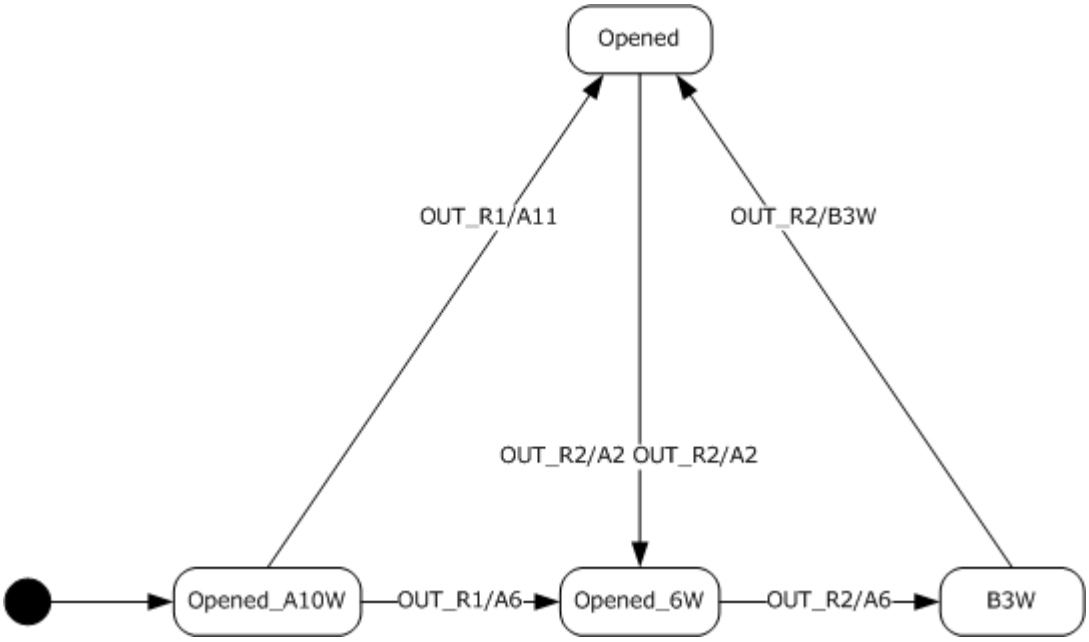


Figure 27: Client virtual OUT channel state machine

The virtual OUT channel state machine MUST be used when the client is trying to recycle an OUT channel. It uses the protocol sequence OUT_R1 as specified in section 3.2.1.4.3.5 or OUT_R2 as specified in section 3.2.1.4.3.6. For more details on recycling an OUT channel, see section 3.2.2.5.6.

3.2.2.1 Abstract Data Model

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

A client maintains several abstract protocol variables as detailed in the following sections.

3.2.2.1.1 Connection Timeout

ConnectionTimeout is a protocol variable as specified in section 3.2.1.1.3.

3.2.2.1.2 KeepAlive Interval

KeepAlive interval is a protocol variable that may be changed by higher layers. Implementations of this protocol SHOULD interpret this variable as the maximum time interval that a higher layer can wait before it establishes with certainty whether the server has dropped out of a conversation.<29>

3.2.2.1.3 Proxy Use

The proxy use variable tells the client implementation if it should use an HTTP proxy to connect to the RPC over HTTP v2 proxy. It can have two values: use an HTTP proxy (called indirect connection) or not use an HTTP proxy (called direct connection). It MUST be initialized in one of three ways:

- From local client configuration.
- By using a protocol not described in this document.
- Combination of one of the two previous bullet points and proxy use determination protocol sequence as described in this section of the protocol documentation.

3.2.2.1.4 Default IN Channel

During channel recycling, a client has two IN channels active. A default IN channel is a protocol variable that indicates which of the two channels is the default channel. Outside channel recycling, there is only one IN channel at a given point in time, and this channel is always considered the default channel. The default channel MUST be used for sending all RPC PDUs. Sending of RTS PDUs is specified in this section.

3.2.2.1.5 Channel Lifetime Sent

An implementation of this protocol MUST maintain a protocol variable that indicates the number of bytes sent by all RTS PDUs and RPC PDUs on a specific IN channel. Each time an RPC PDU or RTS PDU is sent, this protocol variable MUST be incremented by the size in bytes of the PDU that was sent.

3.2.2.2 Timers

An implementation of the RPC over HTTP v2 Protocol Dialect on the client SHOULD implement the timers defined in this section.

3.2.2.2.1 Connection Timeout Timer

This timer is a recurring timer set to an interval equal to the value of the **ConnectionTimeout** field value from CONN/A3 RTS PDU, IN_R1/A4 RTS PDU, or IN_R2/A4 RTS PDU as specified in section 2.2.4. A client implementation MAY choose a lower value for this timer.<30>

3.2.2.2.2 Keep-Alive Timer

This timer is a recurring timer set when the virtual connection is opened. The interval is controlled by the keep-alive interval protocol variable, which is set by a higher layer.

3.2.2.2.3 Proxy Use Determination Timer

A proxy use determination timer SHOULD be used for the duration of the proxy use determination protocol sequence only. It MAY have a value of 200 milliseconds (ms) or use a heuristic that adjusts this value based on network and past results of proxy use determination.<31>

3.2.2.3 Initialization

For this protocol to be initialized successfully, the higher-level RPC protocol as specified in [MS-RPCE] MUST be initialized successfully. Specifically, the initialization steps as specified in [MS-RPCE] section 3.3.2.3 MUST be completed. This protocol imposes an additional initialization step where the network options passed to RPC by higher-level protocols MUST contain a valid RPC over HTTP proxy name. Higher-level protocols also MUST indicate in an implementation-specific way whether HTTP or HTTPS will be used, and whether HTTP authentication or client certificate authentication will be used.<32>

3.2.2.4 Higher-Layer Triggered Events

This section defines the higher-layer triggered events for the RPC over HTTP v2 Protocol Dialect. These events include opening a connection, sending a PDU, closing a connection, and setting the keep-alive interval protocol variable.

3.2.2.4.1 Opening a Connection

When an implementation of a higher-level protocol calls into an implementation of this protocol to open a new connection to the server, an implementation of this protocol MUST perform the following sequence of steps. Each of the steps is broken down into more detailed steps in the two subsequent sections:

1. Establish whether the implementation needs to perform proxy use determination, and if it does, perform the proxy use determination. This step is optional as specified in section 3.2.1.4.3.1.
2. Open a virtual connection to the server as specified in section 3.2.2.4.1.2. For more information on the protocol sequence for opening a virtual connection, see section 3.2.1.4.3.2.

3.2.2.4.1.1 Determining Proxy Use

The first step of opening a connection is to determine proxy use. This step is optional and MAY be skipped by an implementation if it has information from other sources about whether an HTTP proxy is needed to connect to the RPC over HTTP v2 proxy, and which HTTP proxy it needs to use.<33>

If a client implementation knows the name of an HTTP proxy but it does not know whether this proxy needs to be used, it MUST perform the following sequence of steps to determine proxy use:

1. It will send an echo request as specified in section 2.1.2.1.5 to the RPC over HTTP proxy through the HTTP proxy it knows about.

2. It will send an echo request as specified in section 2.1.2.1.5 directly to the RPC over HTTP proxy without going through the HTTP proxy it knows about.

3. It will move to wait state and wait for events from the network.

Once proxy use has been determined, the client can proceed with the rest of the connection opening.

3.2.2.4.1.2 Connection Opening

The client sends an IN channel request as specified in section 2.1.2.1.1.

The client also sends an OUT channel request as specified in section 2.1.2.1.2.

Finally the client MUST transition to Wait_OutChannel state and wait for network events.

3.2.2.4.2 Sending a PDU

This event is valid only in the virtual connection opened state. In any other state an implementation of this protocol MUST treat this as an error and return an implementation-specific error to higher layers.

When a higher-level protocol requests that an implementation of this protocol send a PDU to the server, the implementation of this protocol MUST copy the PDU as a BLOB in the message body of the default RPC IN channel request as specified in section 2.1.2.1.7. If an implementation of this protocol encounters an error while sending the data, it MUST:

- Indicate to a higher layer in an implementation-specific way that the operation failed.<34>
- Treat the connection as closed.
- Request the HTTP protocol stack to close all IN channels and OUT channels for this virtual connection.

If the channel lifetime sent protocol variable for the default IN channel approaches the channel lifetime (as specified later in this paragraph), the implementation of this protocol MUST initiate channel recycling as defined in this section. An implementation MAY define when the number of bytes sent is approaching the channel lifetime in an implementation-specific way, but it SHOULD define it in such a way as to balance between two conflicting objectives: to open the successor IN channel early enough that it is fully opened before the predecessor channel has expired, and yet use as much of the predecessor channel as it can.<35>

For more information on the protocol sequence for recycling an IN channel, see sections 3.2.1.4.3.3 and 3.2.1.4.3.4.

3.2.2.4.2.1 IN Channel Recycling

IN channel recycling MUST NOT be started unless the IN channel is in an opened state. If the number of bytes sent on the channel approaches the channel lifetime and the IN channel is not in an opened state, implementations of this protocol SHOULD return an implementation-specific error to higher layers.<36>

An implementation of this protocol MUST start IN channel recycling by sending out an IN channel request as specified in section 2.1.2.1.1 and 3.2.2.4.1.2 immediately followed by an IN_R1/A1 RTS

PDU as specified in section 2.2.4.10 in the message body of the IN channel request. This IN channel request is the beginning of the successor channel and the existing IN channel request is the predecessor channel. The successor IN channel request is set to be the nondefault IN channel. Then the implementation MUST transition the IN channel state machine to Opened_A4W state and wait for network events. The client implementation MUST be able to execute the IN channel recycling and **OUT channel recycling** state machines in parallel.

3.2.2.4.3 Closing a Connection

When an implementation of a higher-level protocol calls an implementation of this protocol to close a connection, implementations of this protocol MUST:

- Treat the connection as closed.
- Request the HTTP protocol stack to close all IN channels and OUT channels for this virtual connection.

3.2.2.4.4 Setting the Keep-Alive Interval Protocol Variable

When this higher-level event occurs, implementations of this protocol MUST change the keep-alive interval protocol variable as requested by the higher-layer protocol. The keep-alive interval protocol variable is defined in section 3.2.2.1.2.

3.2.2.5 Message Processing Events and Sequencing Rules

All messages meeting any of the following criteria SHOULD be treated by the client as protocol errors and be processed as specified in section 3.2.2.5.11:

- Messages not specifically listed in this section.
- Messages whose syntax is specified in section 2 of this protocol as invalid.
- Events that are specified in this section as protocol errors.

3.2.2.5.1 Echo Response

This response is only expected in the states that are part of the proxy use determination state machine. All other states SHOULD treat this response as a protocol error.

If this response arrives in wait state, the following actions MUST be performed:

- If the echo response arrives to the echo request sent directly to the RPC over HTTP proxy before the echo response sent to the RPC over HTTP proxy through an HTTP proxy, then the client is finished with proxy use determination. It MUST initialize the proxy use variable to directly connect and proceed to connection opening.
- If the echo responsearrives to the echo request sent to the RPC over HTTP proxy through the HTTP proxy first, the client MUST start a proxy use determination timer as specified in section 3.2.2.2.3 and transition to Wait_Direct state and wait for further events from the network.

If the response arrives in Wait_Direct state, the following actions are performed:

- The response is, by virtue of the position in the state diagram, a response to the echo request sent directly to the RPC over HTTP proxy.

- The client is finished with proxy use determination, and MUST: cancel the proxy use determination timer, initialize the proxy use variable to direct connect, and move to connection opening.

3.2.2.5.2 OUT Channel Response

A client implementation MUST NOT accept this HTTP response in any other state than Wait_OutChannel. If received in another state, this HTTP response is a protocol error and the client MUST consider the virtual connection opening a failure and indicate this to higher layers in an implementation-specific way.<37>

If this HTTP response is received in Wait_OutChannel state, the client MUST process the fields of this response as defined below.

First, the client MUST determine whether the response indicates a success or failure. If the status-code is set to 200, the client MUST interpret this as success, and it MUST:

- Ignore the values of all other header fields.
- Transition to Wait_A3W state.
- Wait for network events.
- Skip the rest of the processing in this section.

If the status code is not set to 200, the client MUST interpret this as a failure and follow the same processing rules as specified in section 3.2.2.5.6.

3.2.2.5.3 CONN/A3 RTS PDU

A client implementation MUST NOT accept this RTS PDU in any other state than Wait_A3. If received in another state, this PDU is a protocol error and the client MUST consider the virtual connection opening a failure and indicate this to higher layers in an implementation-specific way.

If this RTS PDU is received in Wait_A3 state, the client MUST transition the state machine to Wait_C2 state and wait for network events.

3.2.2.5.4 CONN/C2 RTS PDU

A client implementation MUST NOT accept this RTS PDU in any other state than Wait_C2. If received in another state, this PDU is a protocol error and the client MUST consider the virtual connection opening a failure and indicate this to higher layers in an implementation-specific way.

If this RTS PDU is received in Wait_C2 state, the client implementation MUST:

- Transition the state machine to opened state.
- Set the connection time-out protocol variable to the value of the **ConnectionTimeout** field from the CONN/C2 RTS PDU.
- Indicate to higher-layer protocols that the virtual connection opening is a success.

From this moment on, the client implementation MUST conform to the virtual IN channel and virtual OUT channel state machines separately as specified in the beginning of section 3.2.2. Both of these state machines start in opened state.

3.2.2.5.5 IN_R1/A4 and IN_R2/A4 RTS PDUs

IN_R1/A4 and IN_R2/A4 are identically processed by implementations of this protocol. This section defines processing of IN_R1/A4, but all definitions provided herein apply to IN_R2/A4 as well.

A client implementation MUST NOT accept this RTS PDU in any state other than Opened_A4W. If received in another state, this PDU is a protocol error and the client MUST close the virtual connection and indicate this to higher layers in an implementation-specific way.<38>

If this RTS PDU is received in Opened_A4W, the client implementation MUST perform the following actions in the sequence given below:

1. Switch the successor IN channel to **Plugged Channel Mode**.
2. Set default IN channel to be the successor IN channel.
3. Set the connection time-out protocol variable to the **ConnectionTimeout** field in IN_R1/A4.
4. Wait until all RTS and RPC PDUs on the predecessor IN channel are sent.
5. Send IN_R1/A5 RTS PDU on the predecessor IN channel.
6. Unplug the successor IN channel.
7. Transition the IN virtual channel to opened state.

3.2.2.5.6 OUT_R1/A2 and OUT_R2/A2 RTS PDUs

OUT_R1/A4 and OUT_R2/A4 are identically processed by implementations of this protocol. This section defines processing for OUT_R1/A2, but all definitions provided herein apply to OUT_R2/A2 as well.

A client implementation MUST NOT accept these RTS PDUs in any other state of the virtual OUT channel than opened. If received in another state, the client MUST treat it as a protocol error as defined in section 3.2.2.5.11.

If this RTS PDU is received in opened, the client implementation MUST perform the following actions in the sequence given below:

1. Create a successor OUT channel instance and send an OUT channel request to the outbound proxy as specified in section 3.2.2.4.1.2. The successor OUT channel instance MUST be considered the successor OUT channel, and the existing MUST be considered the predecessor OUT channel. The successor OUT channel is attached as a component to the virtual OUT channel.
2. Send OUT_R1/A3 RTS PDU on the successor OUT channel.
3. Transition the virtual IN channel state machine to Opened_A6W state.

3.2.2.5.7 OUT_R1/A6 RTS PDU

A client implementation MUST NOT accept this RTS PDU in any state of the virtual OUT channel other than Opened_A6W. If received in another state, the client MUST treat it as a protocol error as defined in section 3.2.2.5.11.

If this RTS PDU is received in Opened_A6W, the client implementation MUST perform the following actions in the sequence given below:

1. Send OUT_R1/A7 RTS PDU on the IN channel.

2. Transition the virtual OUT channel state machine to Opened_A10W state.

3.2.2.5.8 OUT_R1/A10 RTS PDU

A client implementation MUST NOT accept this RTS PDU in any state of the virtual OUT channel other than Opened_A10W. If received in another state, the client MUST treat it as a protocol error as specified in section 3.2.2.5.11.

If this RTS PDU is received in Opened_A10W, the client implementation MUST perform the following actions in the sequence given below:

1. Set the successor OUT channel as the default OUT channel.

2. Send OUT_R1/A11 RTS PDU on the successor OUT channel.

3. Transition the virtual OUT channel state machine to opened state.

3.2.2.5.9 OUT_R2/A6 RTS PDU

A client implementation MUST NOT accept this RTS PDU in any state of the virtual OUT channel other than Opened_A6W. If received in another state, the client MUST treat it as a protocol error as specified in section 3.2.2.5.11.

If this RTS PDU is received in Opened_A6W, the client implementation MUST perform the following actions in the sequence given below:

1. Send OUT_R2/A7 RTS PDU on the IN channel.

2. Send OUT_R2/C1 RTS PDU on the successor OUT channel.

3. Transition the virtual OUT channel state machine to B3W state.

3.2.2.5.10 OUT_R2/B3 RTS PDU

A client implementation MUST NOT accept this RTS PDU in any state of the virtual OUT channel other than B3W. If received in another state, the client MUST treat it as a protocol error as specified in section 3.2.2.5.11.

If this RTS PDU is received in B3W, the client implementation MUST perform the following actions in the sequence given below:

1. Switch the default OUT channel to the successor OUT channel.

2. Transition the virtual OUT channel state machine to the opened state.

3.2.2.5.11 Connection Closed, Connection Error, and Protocol Error Encountered

Connection closed and connection error encountered MUST be handled and indicated to higher layers identically by implementations of this protocol. This section discusses connection close only, and implementations of this protocol MUST handle connection errors that it encounters in the same way. A connection close can come from either the inbound proxy or outbound proxy. Processing is equivalent in both cases.

This section discusses connection close from the outbound proxy, but all parts of the specification in this section apply equally to connection close received from the inbound proxy. If a connection is

closed by the outbound proxy, the client implementation MUST find the virtual connection to which the OUT channel belongs, and unless the OUT channel is in state opened and the connection close comes from a predecessor outbound proxy, the client implementation MUST:

- Free any data structures associated with it.
- Close all the channels that belong to this virtual connection.
- Stop execution on the state machine.

If the connection is closed in state opened, and the connection close comes from a predecessor outbound proxy, the client implementation MUST ignore this event.

If a connection close by the inbound proxy is preceded by an IN channel response as specified in section 2.1.2.1.3, the client MUST process its fields as specified below:

Section 2.1.2.1.3 defines the reason-phrase, which should be interpreted as follows:

```
reason_phrase = "RPC Error: " RPC_Error [ee_info]
RPC_Error = 1*HEX
ee_info = ", EEInfo: " EncodedEEInfo
```

RPC_Error: MUST be interpreted by the client implementation as a hexadecimal representation of an error code and MUST be returned to a higher-layer protocol in an implementation-specific way.<39>

Clients SHOULD ignore ee_info in the message header if the message body contains it.

EncodedEEInfo: MUST be interpreted by the client implementation as a base64 encoded [MS-EERR] BLOB and MUST be processed as specified in [MS-EERR] and made available to higher-layer protocols in an implementation-specific way.

Section 2.1.2.1.3 specifies that the message body MUST be in the format:

```
message_body = ["RPC EEInfo:" EncodedEEInfo]
```

If the message_body has EncodedEEInfo, the client SHOULD use that and ignore the EncodedEEInfo from the message header.

EncodedEEInfo: MUST be interpreted by the client implementation as a base64 encoded [MS-EERR] BLOB and MUST be processed as specified in [MS-EERR] and made available to higher-layer protocols in an implementation-specific way.<40>

Protocol error MUST be handled by the client implementation by:

- Closing all channels to the inbound and outbound proxy for the virtual connection on which the error was encountered.
- Free all data structures associated with the virtual connection.
- Stop execution on the state machine.

3.2.2.6 Timer Events

An implementation of an RPC over HTTP v2 client SHOULD implement the following timers.

3.2.2.6.1 Connection Timeout Timer Expiry

Each time this timer expires, the client implementation of this protocol MUST send a Ping RTS PDU as specified in section 2.2.4.49 unless any other RPC or RTS PDU has been sent recently. Recently MAY be interpreted in an implementation-specific way.<41>

3.2.2.6.2 Keep-Alive Timer Expiry

Each time this timer expires, the client implementation of this protocol MUST send a Ping RTS PDU as specified in section 2.2.4.49 unless any other RPC or RTS PDU has been sent recently. Recently MAY be interpreted in an implementation-specific way.<42>

3.2.2.6.3 Proxy Use Determination Timer Expiry

If this timer expires, the client MUST complete proxy use determination and MUST initialize the proxy use variable to indirect connect and move to connection opening.

3.2.2.7 Other Local Events

An implementation of this protocol is not required to handle other local events.

3.2.3 Inbound Proxy Details

This section contains details specific to an implementation of an inbound proxy. The state machine below specifies the states and the transitions between them for the inbound proxy. Which event causes which transition is specified in section 3.2.3.5.

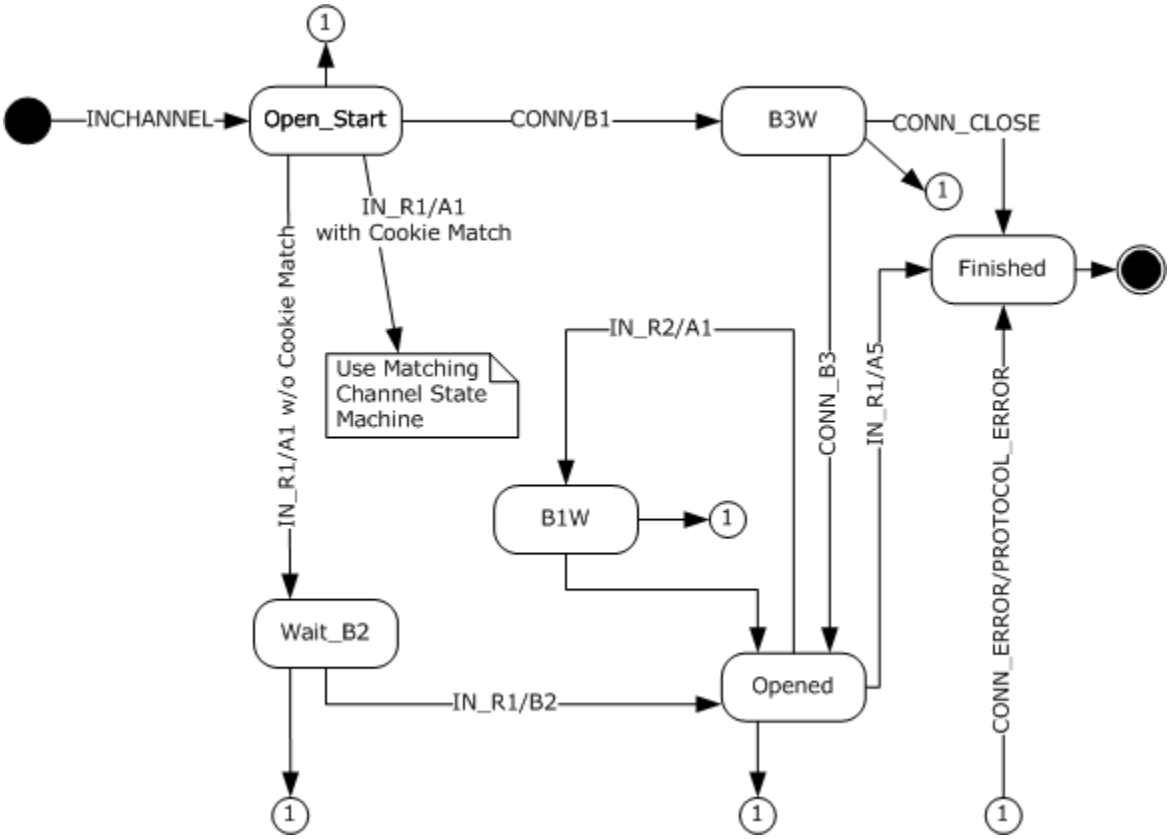


Figure 28: Inbound proxy state machine

The inbound proxy state machine is used when the inbound proxy is processing messages and PDUs coming from the network. When the state machine transitions to the "Use Matching Channel State Machine," the state machine execution for this state machine stops and the current event (IN_R1/A1 with Cookie Match) is interpreted as IN_R2/A1 event for the state machine of the matching IN channel as specified in section 3.2.3.5.5.

3.2.3.1 Abstract Data Model

This section describes a conceptual model of possible data organization that an implementation might maintain to participate in this protocol. The described organization explains 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.

An inbound proxy maintains several abstract protocol variables and data structures.

3.2.3.1.1 Connection Timeout

ConnectionTimeout is a protocol variable specified in section 3.2.1.1.3. An inbound proxy MUST maintain a local copy of it. The value of this variable is retrieved in an implementation-specific way.<43>

3.2.3.1.2 KeepAlive Interval

KeepAlive Interval is a protocol variable that SHOULD be changed in response to network events. Implementations of this protocol SHOULD interpret this variable as the maximum time interval that a client can wait before it establishes with certainty whether the server has dropped out of a conversation. The initial value is an implementation-specific value.<44>

3.2.3.1.3 Virtual Connection Cookie Table

Implementations of this protocol MUST maintain a table indexed by the virtual connection RTS cookie. On the abstract level, all IN channels that belong to a given virtual connection hold a reference count to a row in this table. When the reference count drops to zero, the row in the table MUST be deleted.

3.2.3.1.4 Resource Type UUID

Implementations of this protocol MAY maintain a protocol variable for each virtual IN channel called resource type UUID. Initially, when the IN channel is created, the value of this variable is not set. This protocol variable is not currently used but is reserved for future extensibility.<45>

3.2.3.1.5 Session UUID

Implementations of this protocol MAY maintain a protocol variable for each virtual IN channel called Session UUID. Initially, when the IN channel is created, the value of this variable is not set. This protocol variable is not currently used but is reserved for future extensibility.<46>

3.2.3.1.6 Default IN Channel

During channel recycling, an inbound proxy MAY have two IN channels active. A default IN channel is a protocol variable that indicates which of the two channels is the default channel. Outside channel recycling, there is only one IN channel at a given point in time and this channel is always considered the default channel. The default channel MUST be used for sending all RPC PDUs and all RTS PDUs not specifically defined to have different processing rules in this section.

3.2.3.2 Timers

An implementation of the RPC over HTTP v2 Protocol Dialect on the inbound proxy SHOULD implement the timers defined in this section.

3.2.3.2.1 Keep-Alive Timer

This recurring timer MUST be first set when an IN channel from the inbound proxy to the server is opened. The interval MUST be equal to the keep-alive interval protocol variable that is set through network events. This timer is set in the RPC over HTTP v2 protocol layer but is implemented by the TCP implementation on the inbound proxy and its expiry shows up as a connection error on the IN channel to the server.

3.2.3.3 Initialization

As part of initialization, implementations of this protocol MUST listen on HTTP/HTTPS URL namespace "/rpc/rpcproxy.dll" and SHOULD listen on HTTP/HTTPS URL namespace "/rpcwithcert/rpcproxy.dll".<47>

3.2.3.4 Higher-Layer Triggered Events

There are no higher-layer triggered events on the inbound proxy.

3.2.3.5 Message Processing Events and Sequencing Rules

The messages and PDUs listed below in this section correspond to events in the state diagram at the beginning of this section.

All messages not specifically listed in this section, or messages whose syntax is specified in section 2 of this protocol as invalid, SHOULD be treated by implementations of this protocol on the inbound proxy as protocol errors, as defined in section 3.2.3.5.9.

3.2.3.5.1 RPC IN Channel Request Received

When an RPC over HTTP v2 proxy receives this HTTP request it MUST assume the role of an inbound proxy and transition to Open_Start state. The processing of the HTTP header fields from the HTTP request are defined below:

Accept: Implementations of this protocol on the inbound proxy SHOULD ignore this header field.

Cache-Control: Implementations of this protocol on the inbound proxy SHOULD ignore this header field.

Connection: Implementations of this protocol on the inbound proxy SHOULD ignore this header field.

Content-Length: Implementations of this protocol on the inbound proxy SHOULD ignore this header field.

Pragma Directives:

- Implementations of this protocol on the inbound proxy SHOULD ignore the "No-cache" pragma directive if present.
- Implementations of this protocol on the inbound proxy SHOULD ignore the "Pragma:MinConnTimeout=T" directive if present.
- Implementations of this protocol on the inbound proxy SHOULD check for the presence of the "Pragma:ResourceTypeUuid=R" directive and if present, MUST set the Resource Type UUID protocol variable to the R value.
- Implementations of this protocol on the inbound proxy SHOULD check for the presence of the "Pragma:SessionId=S" directive and if present, MUST set the Session UUID protocol variable to the S value.

Protocol: Implementations of this protocol on the inbound proxy SHOULD ignore this header field.

User-Agent: Implementations of this protocol on the inbound proxy SHOULD ignore this header field.

3.2.3.5.2 RPC PDU Received

An RPC PDU MUST be received from the client only and MUST NOT be received from the server. If the RPC PDU is received on an IN channel from the server, the inbound proxy MUST close the IN channel to the server and the IN channel to the client for the virtual IN channel to which the IN channel to the server belongs.

If the PDU is received from the client as specified in section 2.1.2.1.7, an implementation of this protocol MUST forward it to the server using the default IN channel and conforming to flow control provisions as specified in section 3.2.1.3.1.2.

3.2.3.5.3 CONN/B1 RTS PDU

An inbound proxy implementation MUST NOT accept this RTS PDU in any other state than Open_Start. If received in another state, the inbound proxy MUST treat this PDU as a protocol error as defined in section 3.2.3.5.9.

If this RTS PDU is received in Open_Start state, the inbound proxy implementation MUST perform the following actions in the sequence given below:

1. Establish a TCP connection to the server using the server name and port from the IN channel request as specified in section 2.2.2.
2. Send CONN/B2 RTS PDU to the server as specified in section 2.2.4.6.
3. Set the keep-alive protocol variable to the value from the **ClientKeepalive** command of this PDU.
4. Add the virtual connection cookie to the virtual connection cookie table.
5. Switch the IN channel to the server to Plugged Channel Mode.
6. Transition the state to B3W.

3.2.3.5.4 CONN/B3 RTS PDU

An inbound proxy implementation MUST NOT accept this RTS PDU in any other state than B3W. If received in another state, this PDU is a protocol error and the inbound proxy MUST treat it as a protocol error as specified in section 3.2.3.5.9.

If this RTS PDU is received in B3W state, the inbound proxy implementation MUST perform the following actions in the sequence given below:

1. Switch the IN channel to the server to unplugged channel mode.
2. Transition the state to opened.

3.2.3.5.5 IN_R1/A1 and IN_R2/A1 RTS PDUs

These two RTS PDU have the same format and are processed identically by the inbound proxy. This section defines processing for IN_R1/A1 only, but the same processing rules apply to IN_R2/A2.

An inbound proxy implementation MUST NOT accept this RTS PDU in any other state than Open_Start. If received in another state, this PDU is a protocol error and the inbound proxy MUST treat it as a protocol error as specified in section 3.2.3.5.9.

If this RTS PDU is received in Open_Start state, the inbound proxy implementation MUST retrieve the virtual connection cookie from the IN_R1/A1 RTS PDU and search for a matching entry in the virtual connection cookie table. If found, it MUST execute the sequence of steps in section 3.2.3.5.5.1. If not found, it MUST execute the sequence of steps in section 3.2.3.5.5.2.

3.2.3.5.5.1 Virtual Connection Cookie Found

If the virtual connection cookie is found in the virtual connection cookie table, an implementation of this protocol MUST execute these steps:

1. Inbound proxy MUST conform to IN_R2 protocol sequence.
2. The virtual IN channel that belongs to the virtual connection found in the virtual connection cookie table is verified to be in opened state. If the verification fails, it is a protocol error and MUST be treated, as specified in section 3.2.3.5.9. If the verification succeeds, the IN channel instance MUST be set as a non-default IN channel and a component of the virtual IN channel found through the virtual connection cookie table. In terms of virtual IN channel state machine, this message MUST effect a transition as an IN_R2/A1 event from the opened state to the Opened_A5W state.
3. The successor IN channel from client to inbound proxy is switched to plugged channel mode.
4. Wait for further network events.

3.2.3.5.5.2 Virtual Connection Cookie Not Found

If the virtual connection cookie is not found in the virtual connection cookie table, an implementation of this protocol MUST execute these steps:

1. The inbound proxy MUST conform to IN_R1 protocol sequence.
2. Establish a TCP connection to the server using the server name and port from the IN channel request, as specified in section 2.2.2.
3. Send IN_R1/A2 RTS PDU, as specified in section 2.2.4.11 to the server.
4. Set the keep-alive protocol variable to the value from the **ClientKeepalive** command of the IN_R1/A1 RTS PDU.
5. Switch the successor IN channel to plugged channel mode.
6. Transition to state Wait_B2.

3.2.3.5.6 IN_R1/A5 RTS PDU

An inbound proxy implementation MUST NOT accept this RTS PDU in any other state than opened. If received in another state, this PDU is a protocol error and the inbound proxy MUST treat it as a protocol error as specified in section 3.2.3.5.9.

If this RTS PDU is received in opened state, the inbound proxy implementation MUST perform the following actions in the sequence given below:

1. Send IN_R1/A6 RTS PDU to the server.
2. Send all RTS PDUs queued due to flow control if it has any to the server. The sending MUST still conform to flow control processing rules as specified in section 3.2.1.3.1.2.
3. Send IN_R1/B1 RTS PDU to the server.
4. Close the connection to the client and to the server.
5. Transition to the finished state.

3.2.3.5.7 IN_R1/B2 RTS PDU

An inbound proxy implementation MUST NOT accept this RTS PDU in any other state than Wait_B2. If received in another state, this PDU is a protocol error and the inbound proxy MUST treat it as a protocol error as specified in section 3.2.3.5.9.

If this RTS PDU is received in Wait_B2 state, the inbound proxy implementation MUST perform the following actions in the sequence given below:

1. Switch the successor IN channel to unplugged channel mode.

2. Transition the state to opened.

3.2.3.5.8 IN_R2/A5 RTS PDU

An inbound proxy implementation MUST NOT accept this RTS PDU in any other state than Opened_A5W and it MUST NOT accept this RTS PDU unless it is sent on the predecessor IN channel from the client to the inbound proxy. If either of these conditions is not met, this PDU is a protocol error and the inbound proxy MUST treat it as protocol error as specified in section 3.2.3.5.9.

If this RTS PDU is received in Opened_A5W state, the inbound proxy implementation MUST perform the following actions in the sequence given below:

1. Verify that the channel cookie in this RTS PDU matches the successor IN channel cookie. If it does not match, it MUST close the successor IN channel, transition to opened state and skip the rest of the steps in this section.

2. Transition the state to opened.

3. Close the predecessor IN channel to the client.

3.2.3.5.9 Connection Closed, Connection Error, and Protocol Error Encountered

Connection closed and connection error encountered MUST be processed identically by implementations of this protocol. This section discusses connection closed only and implementations of this protocol MUST process connection errors that it encounters in the same way. A connection close can come from either the client or the server. If a connection close comes from the client, the inbound proxy MUST free any data structures associated with it. If the connection closed does not come while in a finished state, the inbound proxy MUST close all IN channels to the client and all IN channels to the server that belong to the virtual connection on which the close occurred, free all data structures associated with the virtual connection, and transition to the finished state. If the connection closed comes in the finished state, the inbound proxy MUST ignore this event.

If a connection close comes from the server, the inbound proxy MUST close all IN channels to the client and all IN channels to the server that belong to the virtual connection on which the close occurred, free all data structures associated with the virtual connection and transition to the finished state.

Protocol error MUST be handled by the inbound proxy implementation by closing all IN channels to the client and all IN channels to the server that belong to the virtual connection on which the error occurred, freeing all data structures associated with the virtual connection and transition to the finished state.

3.2.3.5.10 Processing Errors

If an implementation of this protocol on the inbound proxy encounters a processing error outside protocol errors, it SHOULD try to send an IN channel response as specified in section 2.1.2.1.3. If an implementation runs out of local resource to create a well-formed IN channel response as defined in this section, it SHOULD close the connection as if a protocol error was encountered as specified in section 3.2.3.5.9. If it is able to create a well-formed IN channel response, an implementation of this protocol:

- MUST set the Status-Code to 503.
- MUST set the RPC_Error field from the IN channel response to a hexadecimal representation of an implementation-specific error code.<48>
- SHOULD set the ee_info part of the reason-phrase from the IN channel response whenever the inbound proxy has additional error information in the format specified in [MS-EERR], as follows: EncodedEEInfo from the IN channel response SHOULD be set to a base64-encoded BLOB of the extended error information, as specified in [MS-EERR]. The base64 encoding MUST be as specified in [RFC3548]. The content of the BLOB itself is specified in [MS-EERR]. Implementations of this protocol SHOULD ensure that the total length of the reason-phrase line does not exceed 1024 bytes.
- SHOULD set the message body as specified in section 2.1.2.1.3, and the EncodedEEInfo SHOULD be set to a base64 encoded BLOB. The base64 encoding MUST be as specified in [RFC3548]. The content of the BLOB is as specified in [MS-EERR].

3.2.3.5.11 Legacy Server Response

Inbound proxies MUST ignore the legacy server response and MUST NOT treat the absence of a legacy server response as a protocol error.

3.2.3.6 Timer Events

None of the timers specified in section 3.2.3.2 expires in a way that results in events for this protocol.

3.2.3.7 Other Local Events

An implementation of this protocol is not required to handle other local events.

3.2.4 Outbound Proxy Details

This section gives details specific to an implementation of an outbound proxy. The state machine below specifies the states and the transitions between them for the outbound proxy. Which event causes which transition is specified in section 3.2.4.5.

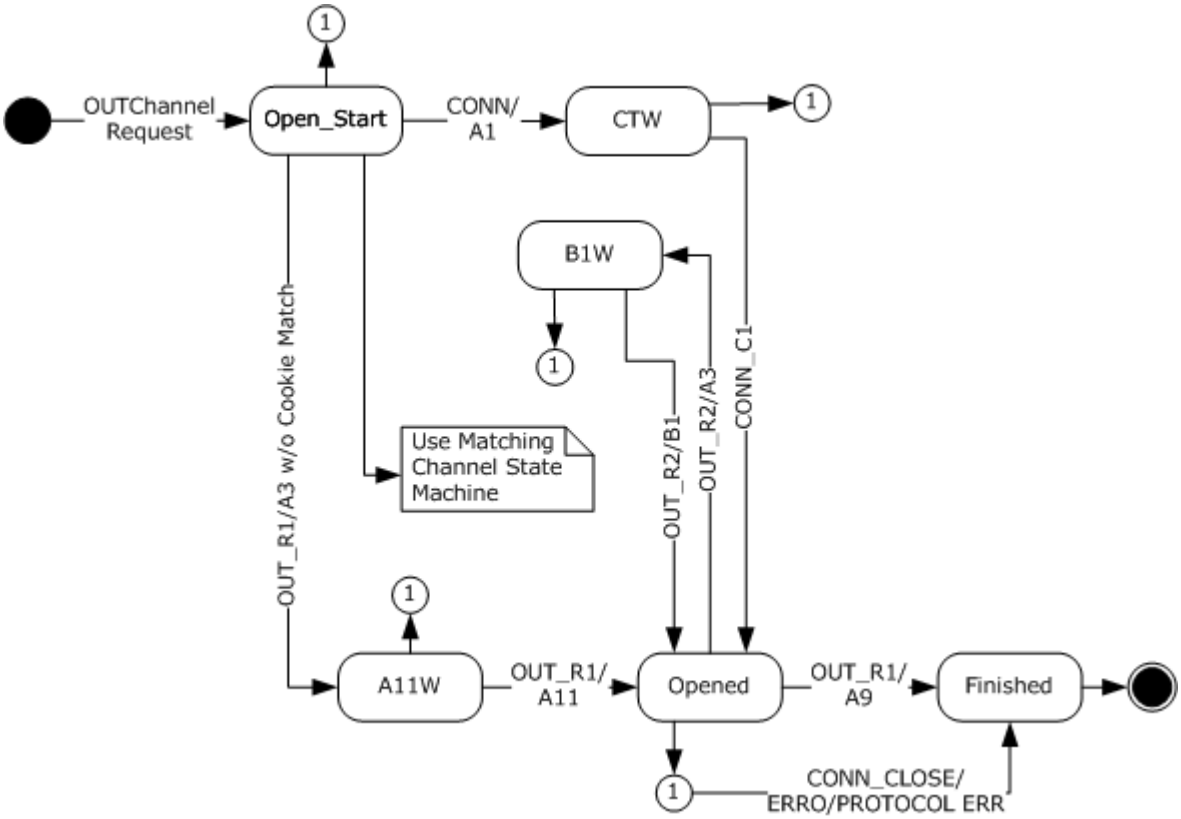


Figure 29: Outbound proxy state machine

The outbound proxy state machine is used when the outbound proxy is processing messages and PDUs coming from the network. When the state machine transitions to "Use Matching Channel State Machine," this means the state machine execution for this state machine stops and the current event (OUT_R1/A3 with Cookie Match) is interpreted as OUT_R2/A3 event for the state machine of the matching IN channel as specified in section 3.2.4.5.6.

3.2.4.1 Abstract Data Model

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

An outbound proxy maintains several abstract protocol variables and data structures.

3.2.4.1.1 Connection Timeout

ConnectionTimeout is a protocol variable specified in section 3.2.1.1.3. An outbound proxy MUST maintain a local copy of it. The value of this variable is retrieved in an implementation-specific way.<49>

3.2.4.1.2 Virtual Connection Cookie Table

Implementations of this protocol MUST maintain a table indexed by the virtual connection RTS cookie. On the abstract level, all OUT channels that belong to a given virtual connection hold a reference count to a row in this table. When the reference count drops to zero, the row in the table MUST be deleted.

3.2.4.1.3 Default OUT Channel

During channel recycling, an outbound proxy MAY have two OUT channels active. A default OUT channel is a protocol variable that indicates which of the two channels is the default channel. Outside channel recycling, there is only one OUT channel at a given point in time and this channel is always considered the default channel. The default channel MUST be used for sending all RPC PDUs and all RTS PDUs not specifically defined to have different processing rules in this section.

3.2.4.1.4 Resource Type UUID

Implementations of this protocol MAY maintain a protocol variable for each virtual OUT channel called resource type UUID. Initially, when the OUT channel is created, the value of this variable is not set. This protocol variable is not currently used but is reserved for future extensibility.<50>

3.2.4.1.5 Session UUID

Implementations of this protocol MAY maintain a protocol variable for each virtual OUT channel called session UUID. Initially, when the OUT channel is created, the value of this variable is not set. This protocol variable is not currently used but is reserved for future extensibility.<51>

3.2.4.2 Timers

An implementation of the RPC over HTTP v2 Protocol Dialect on the outbound proxy SHOULD implement the timer defined in this section.

3.2.4.2.1 Connection Timeout Timer

This recurring timer is set when an OUT channel from the outbound proxy to the client is opened. The interval is controlled by the connection time-out protocol variable as defined in section 3.2.4.1.1. The interval of this timer MUST be changed to the value of the connection time-out protocol variable each time the protocol variable is changed.

3.2.4.3 Initialization

As part of initialization, implementations of this protocol MUST listen on HTTP/HTTPS URL namespace "/rpc/rpcproxy.dll" and SHOULD listen on HTTP/HTTPS URL namespace "/rpcwithcert/rpcproxy.dll".<52>

3.2.4.4 Higher-Layer Triggered Events

There are no higher-layer triggered events on the outbound proxy.

3.2.4.5 Message Processing Events and Sequencing Rules

The messages and PDUs listed in this section correspond to events in the state diagram at the beginning of this section.

All messages not specifically listed in this section, or messages whose syntax is specified in section 2 of this protocol as invalid, SHOULD be treated by implementations of this protocol on the outbound proxy as protocol errors as specified in section 3.2.4.5.12.

3.2.4.5.1 RPC OUT Channel Request Received

When an RPC over HTTP v2 proxy receives this HTTP request, it MUST assume the role of an outbound proxy and transition to the Open_Start state. The processing of the HTTP header fields from the HTTP request are defined below:

Accept: Implementations of this protocol on the outbound proxy SHOULD ignore this header field.

Cache-Control: Implementations of this protocol on the outbound proxy SHOULD ignore this header field.

Connection: Implementations of this protocol on the outbound proxy SHOULD ignore this header field.

Content-Length: Implementations of this protocol on the outbound proxy SHOULD ignore this header field.

Pragma Directives:

- Implementations of this protocol on the outbound proxy SHOULD ignore pragma directive "No-cache".
- If the "Pragma:MinConnTimeout=T" directive is present, implementations of this protocol on the outbound proxy MUST initialize the connection time-out protocol variable to the value of T from the pragma.
- Implementations of this protocol on the outbound proxy SHOULD check for the presence of the "Pragma:ResourceTypeUuid=R" directive, and if present, MUST set the Resource Type UUID protocol variable to the R value.
- Implementations of this protocol on the inbound proxy SHOULD check for the presence of the "Pragma:SessionId=S" directive, and if present, MUST set the session UUID protocol variable to the S value.

Protocol: Implementations of this protocol on the outbound proxy SHOULD ignore this header field.

User-Agent: Implementations of this protocol on the outbound proxy SHOULD ignore this header field.

3.2.4.5.2 RPC PDU Received

An RPC PDU MUST be received from the server only and MUST NOT be received from the client. If the RPC PDU is received on an OUT channel from the client, the outbound proxy MUST close the OUT channel to the client and the OUT channel to the server for the virtual OUT channel to which the OUT channel to the client belongs.

If the PDU is received from the server on a given connection, an implementation of this protocol MUST find the default OUT channel that belongs to the same virtual connection as the connection on which the PDU from the server was received. Once the OUT channel is found, an implementation of this protocol MUST copy the PDU as a BLOB in the message body of this OUT channel request as defined in section 2.1.2.1.2 and send the PDU subject to flow control requirements as specified in section 3.2.1.4.1.

3.2.4.5.3 CONN/A1 RTS PDU

An outbound proxy implementation MUST NOT accept this RTS PDU in any state other than Open_Start. If received in another state, the inbound proxy MUST treat this PDU as a protocol error as specified in section 3.2.4.5.12.

If this RTS PDU is received in Open_Start state, the outbound proxy implementation MUST perform the following actions in the sequence given below:

1. Establish a TCP connection to the server using the server name and port from the OUT channel request as specified in section 2.2.2.
2. Send CONN/A2 RTS PDU to the server.
3. If all operations so far have been successful, send an OUT channel response on the OUT channel to the client. The fields for OUT channel response are defined below:
   - HTTP-Version: MUST be the string HTTP/1.1.
   - Status-Code: MUST be the string 200.
   - Reason-Phrase: MUST be the string Success.
   - Content-Type: Outbound proxies MUST set this header field to the string "application/rpc".
   - Content-Length: Outbound proxies MUST set this field to an implementation-specific value in the inclusive range of 128 KB to 2 GB.<53>

   In failure case, the outbound proxy MUST use the same processing rules as the inbound proxy as defined in section 3.2.3.5.10 and skip the rest of the processing in this section.
4. Send CONN/A3 RTS PDU on the OUT channel to the client.
5. Add the virtual connection cookie to the virtual connection cookie table.
6. Transition the state to C1W and wait for further network events.

3.2.4.5.4 CONN/C1 RTS PDU

An outbound proxy implementation MUST NOT accept this RTS PDU in any state other than C1W. If received in another state, the outbound proxy MUST treat this PDU as a protocol error as specified in section 3.2.4.5.12.

If this RTS PDU is received in C1W state, the outbound proxy implementation MUST perform the following actions in the sequence given below:

1. Send CONN/C2 RTS PDU on the OUT channel to the client.
2. Transition the state to opened.

3.2.4.5.5 OUT_R1/A1 or OUT_R2/A1 RTS PDUs

These two RTS PDUs have the same format and are processed identically by the outbound proxy. This section explains processing for OUT_R1/A1 only, but the same processing rules apply to OUT_R2/A2.

An outbound proxy implementation MUST NOT accept this RTS PDU in any state other than opened. If received in another state, the outbound proxy MUST treat this PDU as a protocol error as specified in section 3.2.4.5.12.

If this RTS PDU is received in opened state, the outbound proxy implementation MUST send OUT_R1/A2 or OUT_R2/A2 to the client depending on whether OUT_R1/A1 or OUT_R2/A1 is received on the default OUT channel. OUT_R1/A2 and OUT_R2/A2 have the same format.

3.2.4.5.6 OUT_R1/A3 or OUT_R2/A3 RTS PDUs

These two RTS PDUs have the same format and the outbound proxy determines which RTS PDU it got based on internal state as defined in this section.

An outbound proxy implementation MUST NOT accept any of these RTS PDUs in any state other than Open_Start. If received in another state, the outbound proxy MUST treat these PDUs as a protocol error as specified in section 3.2.4.5.12.

If this RTS PDU is received in the Open_Start state, the outbound proxy implementation MUST retrieve the virtual connection cookie from the OUT_R1/A1 RTS PDU and search for a matching cookie in the virtual connection cookie table. If found, it MUST execute the sequence of steps specified in section 3.2.4.5.6.1. If not found, it MUST execute the sequence of steps specified in section 3.2.4.5.6.2.

3.2.4.5.6.1 Virtual Connection Cookie Found

If the virtual connection cookie is found in the virtual connection cookie table, an implementation of this protocol MUST execute these steps:

1. The outbound proxy MUST conform to the OUT_R2 protocol sequence.
2. Extract the virtual OUT channel that belongs to the virtual connection found in the virtual connection cookie table. The OUT channel instance in which the OUT_R2/A3 RTS PDU arrived MUST be made a component of that virtual OUT channel, be set as the non-default channel of that virtual OUT channel, and be considered the successor channel. The existing OUT channel instance that belonged to that virtual OUT channel instance MUST be considered the predecessor OUT channel. This RTS PDU MUST be interpreted as OUT_R2/A3 and the state machine MUST transition to state B1W as if an OUT_R2/A3 message was received in the opened state of the virtual OUT channel.
3. Send OUT_R2/A4 RTS PDU to the server.
4. Switch the successor OUT channel instance to plugged channel mode.

3.2.4.5.6.2 Virtual Connection Cookie Not Found

If the virtual connection cookie is not found in the virtual connection cookie table, an implementation of this protocol MUST execute these steps:

1. The outbound proxy MUST conform to the OUT_R1 protocol sequence.
2. Establish a TCP connection to the server using the server name and port from the OUT channel request as specified in section 2.2.2.
3. Send OUT_R1/A4 RTS PDU to the server.
4. Switch the successor OUT channel instance to plugged channel mode.

5. Transition to state A11W.

3.2.4.5.7 OUT_R1/A5 RTS PDU

An outbound proxy implementation MUST NOT accept this RTS PDU in any state other than opened. If received in another state, the inbound proxy MUST treat this PDU as a protocol error as specified in section 3.2.4.5.12.

If this RTS PDU is received in opened state, the inbound proxy implementation MUST send OUT_R1/A6 RTS PDU to the client on the default OUT channel. State remains unchanged.

3.2.4.5.8 OUT_R1/A9 RTS PDU

An outbound proxy implementation MUST NOT accept this RTS PDU in any state other than opened. If received in another state, the inbound proxy MUST treat this PDU as a protocol error as specified in section 3.2.4.5.12.

If this RTS PDU is received in opened state, an implementation of the outbound proxy implementation MUST execute these steps:

- Send all RPC PDUs that might be queued due to flow control to the client on the default OUT channel, observing flow control rules as specified in section 3.2.1.3.1.2.
- Send OUT_R1/A10 RTS PDU to the client on the default OUT channel.
- Close the connection to the client and to the server.
- Transition the state to the finished state.

3.2.4.5.9 OUT_R1/A11 RTS PDU

An outbound proxy implementation MUST NOT accept this RTS PDU in any state other than A11W. If received in another state, the outbound proxy MUST treat this PDU as a protocol error as specified in section 3.2.4.5.12.

If this RTS PDU is received in A11W state, the outbound proxy implementation MUST perform the following actions in the sequence given below:

1. Switch the successor OUT channel to unplugged channel mode.
2. Transition the state to opened.

3.2.4.5.10 OUT_R2/B1 RTS PDU

An outbound proxy implementation MUST NOT accept this RTS PDU in any state other than B1W. If this condition is not met, the outbound proxy MUST treat this PDU as a protocol error as specified in section 3.2.4.5.12.

If this RTS PDU is received in B1W state, the outbound proxy implementation MUST perform the following actions in the sequence given below:

1. Send all RPC PDUs that might be queued due to flow control to the client on the default OUT channel, observing flow control rules as specified in section 3.2.1.3.1.2.
2. Send OUT_R2/B3 RTS PDU to the client on the default OUT channel.

3. Send OUT channel response header as specified in section 2.1.2.1.4 on the successor OUT channel instance.

4. Switch the default OUT channel to the successor OUT channel instance.

5. Switch the default OUT channel to unplugged channel mode.

6. Transition the state to opened.

7. Close the predecessor OUT channel to the client.

3.2.4.5.11 OUT_R2/B2 RTS PDU

An outbound proxy implementation MUST NOT accept this RTS PDU in any state other than B1W. If this condition is not met, this PDU is a protocol error and the outbound proxy MUST treat it as a protocol error as specified in section 3.2.4.5.12.

If this RTS PDU is received in B1W state, the outbound proxy implementation MUST perform the following actions in the sequence given below:

1. Transition the state to opened.

2. Close the successor OUT channel to the client.

3.2.4.5.12 Connection Close, Connection Error, and Protocol Error Encountered

Connection close and connection error encountered MUST be handled identically by implementations of this protocol. This section discusses connection close. Implementations of this protocol MUST handle connection errors that it encounters in the same way. A connection close can come from either the client or the server. If a connection close comes from the client, the outbound proxy MUST free any data structures associated with it. If the connection close does not come while in a finished state, the outbound proxy MUST close all OUT channels to the client and all OUT channels to the server, free all data structures associated with the virtual connection, and transition to finished state. If the connection close comes in the finished state, the outbound proxy MUST ignore this event.

If a connection close comes from the server, the outbound proxy MUST close all OUT channels to the client and all OUT channels to the server, free all data structures associated with the virtual connection, and transition to finished state.

Protocol error MUST be handled by the outbound proxy implementation by closing all OUT channels to the client and all OUT channels to the server, freeing all data structures associated with the virtual connection, and transitioning to finished state.

3.2.4.5.13 Legacy Server Response

Outbound proxies MUST ignore the legacy server response and MUST NOT treat the absence of a legacy server response as a protocol error.

3.2.4.6 Timer Events

Each time the connection time-out timer defined in section 3.2.4.2.1 expires, an implementation of this protocol MUST send a ping RTS PDU, as specified in section 2.2.4.49, on the default OUT channel for this virtual connection. This will prevent network agents from closing the connection used by the OUT channel response due to it being idle. An implementation of this protocol MAY choose when to notify the server that it has sent a PDU to the client and thus consume part of the

OUT channel lifetime. When it chooses to notify the server, it MUST do so using a ping traffic sent notify RTS PDU as specified in section 2.2.4.47.<54>

3.2.4.7 Other Local Events

An implementation of this protocol is not required to handle other local events.

3.2.5 Server Details

This section gives details specific to an implementation of a server. The state machine below specifies the states and the transitions between them for the server. Which event causes which transition is specified in sections 3.2.5.4 and 3.2.5.5.

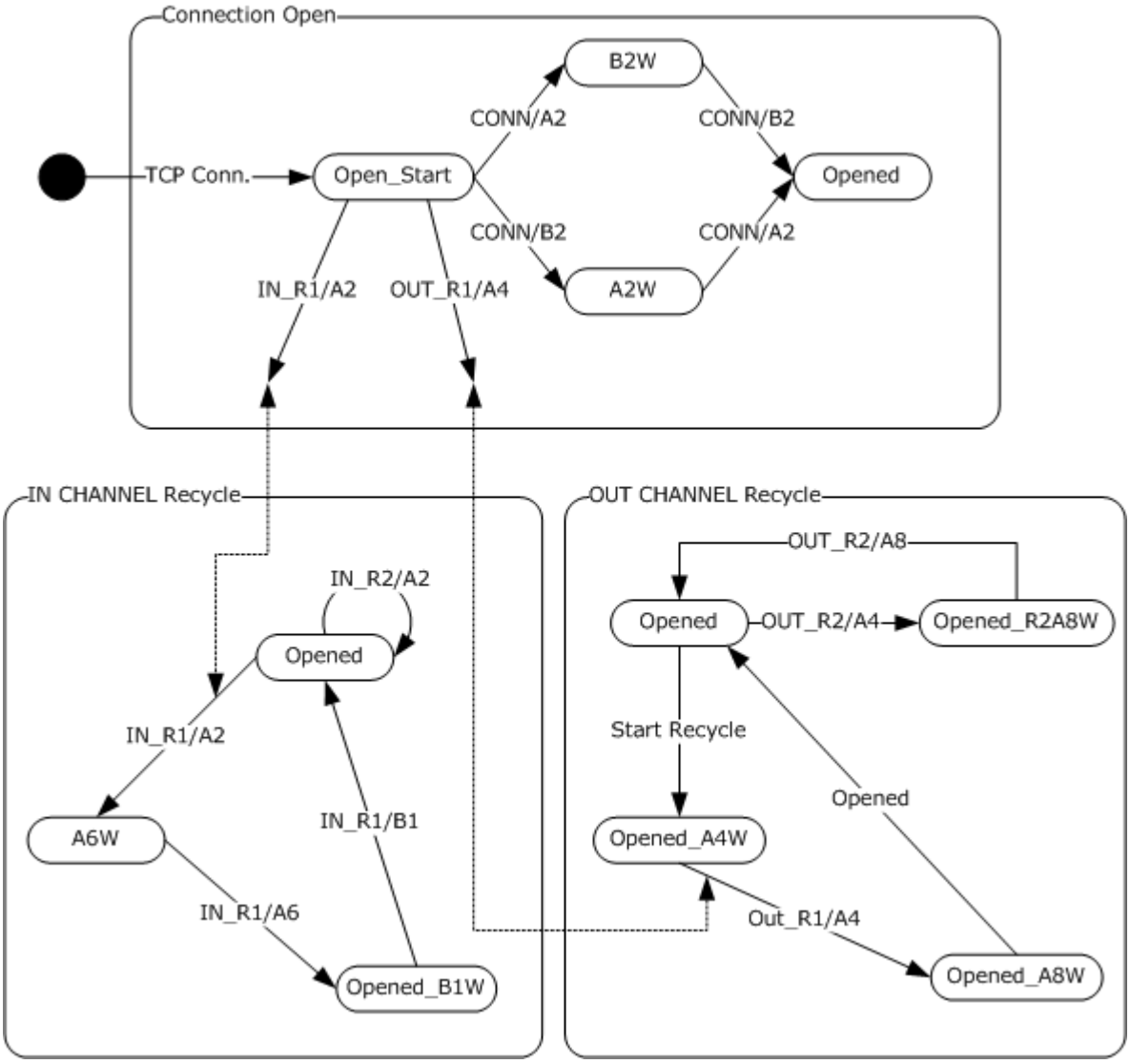


Figure 30: Server state machine

The server state machine is used when the server is processing messages and PDUs coming from the network. The following description of the state machine is provided as an aid to understanding the overall work of the state machines. This description is not a substitute for the processing specifications in section 3.2.5.5.

The connection open state machine is used during connection opening. Once a transition to the opened state of that state machine is made, the IN channel and OUT channel state machines are started from the opened state. The IN channel and OUT channel have independent state machines that run in parallel.

When a new TCP connection to the server is established, the server implementation does not yet know whether this connection will be used to establish a new virtual connection or to recycle an IN channel or an OUT channel. This is why, at this stage, the state machine is in Open_Start state. Once an RTS PDU is received, an implementation of this protocol can inspect the RTS PDU and determine which state machine it will use.

3.2.5.1 Abstract Data Model

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

A server maintains several abstract protocol variables.

3.2.5.1.1 Virtual Connection Cookie Table

Implementations of this protocol MUST maintain a table indexed by the virtual connection RTS cookie. On the abstract level, all IN channel and OUT channel that belong to a given virtual connection hold a reference count to a row in this table. When the reference count drops to zero, the row in the table is deleted.

3.2.5.1.2 Default OUT Channel

During channel recycling, a server has two OUT channels active. A default OUT channel is a protocol variable that indicates which of the two channels is the default channel. Outside channel recycling, there is only one OUT channel at a given point in time, and this channel is always considered the default channel. The default channel MUST be used for sending all RPC PDUs. Sending RTS PDUs is described in this section.

3.2.5.1.3 Temporary Cookie Variable

An implementation of this protocol MUST maintain a temporary cookie variable that it uses to store and retrieve state between RTS PDU.

3.2.5.1.4 Channel Lifetime Sent

An implementation of this protocol MUST maintain a protocol variable that indicates the number of bytes sent by all RTS PDU and RPC PDUs on a specific OUT channel. Each time an RPC or RTS PDU is sent, this protocol variable MUST be incremented by the size in bytes of the PDU that was sent.

3.2.5.2 Timers

An implementation of the RPC over HTTP v2 Protocol Dialect on the server SHOULD implement the timer defined in this section.

3.2.5.2.1 Connection Setup Timer

The connections setup timer SHOULD be set to expire in 15 minutes. It is used to detect a case where the IN channel of a virtual connection is set up, but the OUT channel of the same virtual connection is not, or vice versa.

3.2.5.3 Initialization

Implementations of this protocol MUST listen on a TCP port defined by a higher-level protocol.

3.2.5.4 Higher-Layer Triggered Events

An implementation of this Protocol Dialect on the server MUST handle sending a PDU from a higher layer.

3.2.5.4.1 Sending a PDU

When an implementation of a higher-level protocol calls to an implementation of this protocol to send a PDU to the client, the implementation of this protocol MUST send the PDU on the default OUT channel to the outbound proxy, subject to flow control requirements as specified in section 3.2.1.4.1.

If the implementation of this protocol encounters an error while sending the data, it MUST:

- Indicate to the higher layer, in an implementation-specific way, that the operation failed.<55>
- Treat the connection as closed.
- Request the TCP protocol stack to close all IN channel and OUT channels that belong to this virtual connection.

If the channel lifetime sent protocol variable for the default OUT channel approaches the channel lifetime as specified later in this paragraph, the implementation of this protocol MUST initiate channel recycling as defined in this section. An implementation MAY define when the number of bytes sent is approaching the channel lifetime in an implementation-specific way. However, it SHOULD define it in such a way as to open the successor OUT channel early enough so that it is fully opened before the predecessor channel has channel lifetime, and yet use as much of the predecessor channel as it can.<56>

For more information on the protocol sequences associated with OUT channel recycling, see sections 3.2.1.4.3.5 and 3.2.1.4.3.6 of this document.

3.2.5.4.1.1 OUT Channel Recycling

OUT channel recycling MUST NOT be started unless the OUT channel is in the opened state. If the number of bytes sent on the channel approaches the channel lifetime and the OUT channel is not in the opened state, implementations of this protocol SHOULD return an implementation-specific error to higher layers.<57>

An implementation of this protocol MUST start OUT channel recycling by sending out an OUT_R2/A1 RTS PDU as specified in section 2.2.4.34 to the outbound proxy. Then it MUST transition the OUT

channel state to Opened_A4W state and wait for network events. The server implementation MUST be able to execute the IN channel recycling and associated state machine and OUT channel recycling and associated state machines in parallel.

3.2.5.5 Message Processing Events and Sequencing Rules

Unless explicitly specified in a message or PDU section, the messages and PDUs listed in this section correspond to events in the state diagram at the beginning of section 3.2.5.

All messages not specifically listed in this section, or messages whose syntax is specified in section 2 of this protocol as invalid, SHOULD be treated by implementations of this protocol on the server as protocol errors, as defined in section 3.2.5.5.13.

3.2.5.5.1 Establishing a Connection

When a connection to the server is established, the server SHOULD send the legacy server response as specified in section 2.1.1.2.1 and transition to state Open_Start.

3.2.5.5.2 Receiving an RPC PDU

When an implementation of this protocol receives an RPC PDU, it MUST pass it on to a higher-layer protocol without modifying the contents of the RPC PDU. This happens in an implementation-specific way. If it encounters a protocol error while processing the RPC PDU, it MUST handle the error as defined in section 3.2.5.5.3.<58>

3.2.5.5.3 CONN/A2 RTS PDU

A server implementation MUST NOT accept this RTS PDU in any state other than Open_Start or A2W. If this condition is not met, the server MUST treat this PDU as a protocol error as specified in section 3.2.5.5.13.

A server implementation MUST extract the virtual connection cookie from the CONN/A2 RTS PDU and search for this cookie value in the virtual connection cookie table. If found, the virtual connection is called the existing virtual connection. In such a case, the server implementation MUST verify that the existing virtual connection is in state A2W. If it is, the server implementation MUST continue execution on the state machine of the existing virtual connection, and MUST continue processing this PDU as specified in section 3.2.5.5.3.2.

If the server implementation fails to find a virtual connection in the virtual connection cookie table with the same cookie as the virtual connection cookie from this PDU, then the server MUST continue processing as specified in section 3.2.5.5.3.1 of this document.

3.2.5.5.3.1 Virtual Connection Not Found

If the virtual connection is not found in the virtual connection cookie table as specified in the previous section, an implementation of this protocol MUST perform the following actions in the sequence given below:

1. Set up the connection setup timer defined in section 3.2.5.2.1.
2. Transition to state B2W and wait for further events.

3.2.5.5.3.2 Virtual Connection Found

If the virtual connection is found in the virtual connection cookie table as specified in the previous section, an implementation of this protocol MUST perform the following actions in the sequence given below:

1. Cancel the connection setup timer defined in section 3.2.5.2.1.
2. Send CONN/C1 RTS PDU on the OUT channel to the outbound proxy.
3. Send CONN/B2 RTS PDU on the IN channel to the inbound proxy.
4. Transition to opened state.
5. The virtual IN channels and virtual OUT channel MUST start their own state machines as specified in the beginning of section 3.2.5 of this document.

3.2.5.5.4 CONN/B2 RTS PDU

A server implementation MUST extract the virtual connection cookie from the CONN/B2 RTS PDU and search for this cookie value in the virtual connection cookie table. If found, the virtual connection is called an existing virtual connection. In such a case, the server implementation MUST verify that the existing virtual connection is in state B2W, and if it is, then it MUST continue execution on the state machine of the existing virtual connection and MUST continue processing this PDU as specified in section 3.2.5.5.4.2.

If the server implementation fails to find a virtual connection in the virtual connection cookie table with the same cookie as the virtual connection cookie from this PDU, the server MUST continue processing as specified in section 3.2.5.5.4.1.

3.2.5.5.4.1 Virtual Connection Not Found

If the virtual connection is not found in the virtual connection cookie table as specified in the previous section, an implementation of this protocol MUST perform the following actions in the sequence given below:

1. Set up the connection setup timer specified in section 3.2.5.2.1.
2. Transition to state A2W.

3.2.5.5.4.2 Virtual Connection Found

If the virtual connection is found in the virtual connection cookie table as specified in the previous section, an implementation of this protocol MUST perform the following actions in the sequence given below:

1. Cancel the connection setup timer defined in section 3.2.5.2.1.
2. Send CONN/C1 RTS PDU on the OUT channel to the outbound proxy.
3. Send CONN/B3 RTS PDU9 on the IN channel to the inbound proxy.
4. Transition to the opened state.
5. The virtual IN and OUT channels MUST start their own state machines as specified in the beginning of section 3.2.5.

3.2.5.5.5 IN_R1/A2 RTS PDU

A server implementation MUST NOT accept this RTS PDU in any state other than Open_Start. If this condition is not met, the server MUST treat this PDU as a protocol error as specified in section 3.2.5.5.13.

If this RTS PDU is received in Open_Start state, an implementation of this protocol MUST perform the following actions in the sequence given below:

1. Retrieve the virtual connection cookie from this RTS PDU and find it in the virtual connection cookie table. If the connection is not found, an implementation of this protocol MUST treat this as a protocol error, MUST handle this as specified in section 3.2.5.5.13, and MUST skip the rest of the processing in this section. If found, an implementation of this protocol MUST execute the remaining steps in this section.

2. Once the virtual connection is found, the OUT channel on which this RTS PDU arrived MUST be attached as a component to the virtual IN channel for the virtual connection, and it MUST also be set as non-default and successor IN channel. The existing IN channel is considered the predecessor channel.

3. Verify that the IN channel cookie from this RTS PDU matches the IN channel cookie on the predecessor IN channel. If they don't match, an implementation of this protocol MUST treat this as a protocol error, MUST handle this as specified in section 3.2.5.5.13, and MUST skip the rest of the processing in this section. If they match, an implementation of this protocol MUST execute the remaining steps in this section.

4. Set up the connection setup timer defined in section 3.2.5.2.1.

5. Send IN_R1/A3 RTS PDU on the default OUT channel to the outbound proxy.

6. Transition to the A6W state.

3.2.5.5.6 IN_R1/A6 RTS PDU

A server implementation MUST NOT accept this RTS PDU in any state other than A6W. If this condition is not met, the server MUST treat this PDU as a protocol error as specified in section 3.2.5.5.13.

If this RTS PDU is received in A6W state, an implementation of this protocol MUST perform the following actions in the sequence given below:

1. Cancel the connection setup timer defined in section 3.2.5.2.1.

2. Transition to the Opened_B1W state.

3.2.5.5.7 IN_R1/B1 RTS PDU

A server implementation MUST NOT accept this RTS PDU in any state other than Opened_B1W. If this condition is not met, the server MUST treat this PDU as a protocol error as specified in section 3.2.5.5.13.

If this RTS PDU is received in Opened_B1W state, an implementation of this protocol MUST perform the following actions in the sequence given below:

1. Switch the default IN channel from the predecessor IN channel to the successor IN channel.

2. Send IN_R1/B2 RTS PDU on the successor IN channel to the inbound proxy.

3. Close the IN channel connection to the predecessor inbound proxy.

4. Transition to the opened state.

3.2.5.5.8 IN_R2/A2 RTS PDU

A server implementation MUST NOT accept this RTS PDU in any state other than opened. If this condition is not met, the server MUST treat this PDU as a protocol error as specified in section 3.2.5.5.13.

If this RTS PDU is received in opened state, an implementation of this protocol MUST perform the following actions in the sequence given below:

1. Update the **channel cookie** for the channel on which this RTS PDU is received with the **channel cookie** from this RTS PDU.

2. Send IN_R2/A3 RTS PDU on the default OUT channel to the outbound proxy.

State does not change as a result of this event.

3.2.5.5.9 OUT_R1/A4 RTS PDU

A server implementation MUST NOT accept this RTS PDU in any state other than Opened_A4W. If this condition is not met, the server MUST treat this PDU as a protocol error as specified in section 3.2.5.5.13.

If this RTS PDU is received in Opened_A4W state, an implementation of this protocol MUST perform the following actions in the sequence given below:

1. Send OUT_R1/A5 RTS PDU on the predecessor OUT channel to the outbound proxy.

2. Transition state to Opened_A8W.

3.2.5.5.10 OUT_R1/A8 RTS PDU

A server implementation MUST NOT accept this RTS PDU in any state other than Opened_A8W. If this condition is not met, the server MUST treat this PDU as a protocol error as specified in section 3.2.5.5.13.

If this RTS PDU is received in Opened_A8W state, an implementation of this protocol MUST perform the following actions in the sequence given below:

1. Transition to opened state.

2. Switch the default OUT channel from the predecessor OUT channel to the successor OUT channel.

3. Send OUT_R2/A9 RTS PDU on the predecessor OUT channel to the outbound proxy.

3.2.5.5.11 OUT_R2/A4 RTS PDU

A server implementation MUST NOT accept this RTS PDU in any state other than Opened_A4W. If this condition is not met, the server MUST treat this PDU as a protocol error as specified in section 3.2.5.5.13.

If this RTS PDU is received in Opened_A4W state, an implementation of this protocol MUST perform the following actions in the sequence given below:

1. Transition to Opened_R2A8W state.

2. Send OUT_R2/A5 RTS PDU on the OUT channel to the outbound proxy.

3. Store the channel cookie from this PDU in the temporary cookie protocol variable defined in section 3.2.5.1.3.

3.2.5.5.12 OUT_R2/A8 RTS PDU

A server implementation MUST NOT accept this RTS PDU in any state other than Opened_R2A8W. If this condition is not met, the server MUST treat this PDU as a protocol error as specified in section 3.2.5.5.13.

If this RTS PDU is received in Opened_R2A8W state, an implementation of this protocol MUST compare the **channel cookie** from this RTS PDU to the one stored in the temporary cookie variable as specified in section 3.2.5.1.3. If the cookies match, implementations of this protocol MUST send OUT_R2/B1 RTS PDU on the OUT channel to the outbound proxy, transition to opened state, and update the channel cookie on the OUT channel with the one from this RTS PDU. If the cookies don't match, this protocol MUST send OUT_R2/B1 RTS PDU on the OUT channel and handle this as a protocol error as specified in section 3.2.5.5.13.

3.2.5.5.13 Connection Close, Connection Error, and Protocol Error Encountered

Connection close and connection errors encountered MUST be handled identically by implementations of this protocol. This section discusses connection close only. Implementations of this protocol MUST handle connection errors that it encounters in the same way. A connection close can come from either the inbound or outbound proxy. Processing is equivalent in both cases. This section discusses connection close from the inbound proxy, but all parts of the specification in this section apply equally to connection close received from the outbound proxy. If a connection close comes from the inbound proxy, the server implementation MUST find the virtual connection to which the IN channel belongs, and unless the IN channel is in state opened, and the connection close comes from a predecessor inbound proxy, the server implementation MUST:

- Free any data structures associated with it.
- Close all the channels that belong to this virtual connection.
- Transition to the finished state.

If the connection close comes in state opened, and the connection close comes from a predecessor inbound proxy, the server implementation MUST ignore this event.

Protocol error MUST be handled by the server implementation by closing all channels to the inbound and outbound proxy for the virtual connection on which the error was encountered, free all data structures associated with the virtual connection, and transition to the finished state.

3.2.5.5.14 Ping Traffic Sent Notify RTS PDU on Server

This PDU does not correspond to an event in the state machine. It can be received and is valid in any state. When an implementation of the server receives this PDU, it MUST add the value in the *PingTrafficSent* field of the PingTrafficSentNotify command, as specified in section 2.2.3.5.15, to the channel lifetime Sent protocol variable. If as a result of this addition, the channel lifetime protocol variable approaches the channel lifetime as specified in section 3.2.5.4.1, the implementation of this protocol MUST start channel recycling exactly as specified in section 3.2.5.4.1.

3.2.5.6 Timer Events

This section defines the processing that occurs when one of the timers in section 3.2.5.2 expire.

3.2.5.6.1 Connection Setup Timer Expiry

This timer event is treated as a connection error as specified in section 3.2.5.5.13.

3.2.5.7 Other Local Events

An implementation of this protocol is not required to handle other local events.

4 Protocol Examples

The following sections specify protocol examples: virtual connection open, and flow control and receive windows.

4.1 Virtual Connection Open Example

This example illustrates the sequence of RTS PDUs that is sent during the process of opening a virtual connection.

The process of opening a virtual connection starts by a higher-layer protocol implementation (for example, RPC Runtime) requesting an implementation of this protocol to open a connection to an RPC server.

As a first step, an implementation of this protocol determines whether it needs to use an HTTP proxy or not. For the purposes of this example, assume that it cannot determine through means outside this protocol whether it should use a specific HTTP proxy or whether it should connect to the predecessor RPC over HTTP proxy directly. In this case, the client implementation runs the proxy use determination protocol sequence. It sends an echo request message as specified in section 2.1.2.1.5 to the inbound proxy without using the HTTP proxy. It also sends an echo request message as specified in section 2.1.2.1.5 to the outbound proxy through the HTTP proxy.

Then the client transitions to the wait state in the proxy use determination state machine defined in Figure 24 and wait for an echo response message. The inbound proxy replies first with an echo responsemessage and the proxy use determination is completed. The proxy use protocol variable defined in section 3.2.2.1.3 is set to "direct connection" and the client implementation proceeds to the next step and state machine, that is, connection opening.

Connection opening is started by the client implementation sending an IN channel request and an OUT channel request to the inbound proxy and outbound proxy respectively. Then it sends CONN/A1 RTS PDU to the outbound proxy and CONN/B1 RTS PDU to the inbound proxy. Then it transitions to the "OUT Channel Wait" state in the virtual connection open in the state machine in Figure 25.

The inbound proxy receives the IN channel request and transitions to the Open_Start state. Then it receives CONN/B1 RTS PDU. It extracts the server name and port from the URL part of the IN channel request as specified in section 2.2.2 and establishes a TCP connection to that server and port. The inbound proxy sends CONN/B2 RTS PDU (section 2.2.4.6) to the server and sets the keep-alive protocol variable to the value from the ClientKeepalive command from the CONN/B1 RTS PDU. As a final processing step for this PDU, the inbound proxy adds a row in the virtual connection cookie table for the inbound proxy with the virtual connection cookie extracted from the CONN/B1 RTS PDU, switches the IN channel to the server to Plugged Channel Mode, and transitions to state B3W.

The outbound proxy receives the OUT channel request and transitions to the Open_Start state. Then it receives CONN/A1 RTS PDU. It extracts the server name and port from the URL part of the OUT channel request as specified in section 2.2.2 and establishes a TCP connection to that server and port. The outbound proxy sends CONN/A2 RTS PDU (section 2.2.4.3) to the server, sends an OUT channel response to the client, adds a row in the virtual connection cookie table for the outbound proxy with the virtual connection cookie extracted from the CONN/A1 RTS PDU, and transitions to the C1W state.

When the TCP connection from the inbound proxy to the server is established, the server transitions to the Open_Start state for that connection. Then it receives the CONN/B2 RTS PDU and searches for the virtual connection cookie from the CONN/B2 RTS PDU in its virtual connection table. It is not

found, so the connection setup timer is started and the virtual connection is transitioned to the A2W state.

When the TCP connection from the outbound proxy to the server is established, the server transitions to the Open_Start state for that connection. Then it receives the CONN/A2 RTS PDU and searches for the virtual connection cookie from the CONN/A2 RTS PDU in its virtual connection table. It is not found, so the connection setup timer is started and the virtual connection is transitioned to the A2W state.

When the TCP connection from the inbound proxy to the server is established, the server transitions to the Open_Start state for that connection. Then it receives the CONN/B2 RTS PDU and searches for the virtual connection cookie from the CONN/B2 RTS PDU in its virtual connection table. It is found, so the connection setup timer is canceled and execution continues on the state machine of the existing virtual connection, which is A2W. The server implementation sends CONN/C1 RTS PDU on its OUT channel to the outbound proxy, sends CONN/B3 RTS PDU on the IN channel to the inbound proxy, and transitions to the opened state.

When the inbound proxy receives the CONN/B3 RTS PDU, it switches the IN channel to the server to unplugged channel mode and transitions to opened state.

When the outbound proxy receives the CONN/C1 RTS PDU, it sends CONN/C2 RTS PDU on the OUT channel to the client and transitions to the opened state.

When the client receives the OUT channel response, it transitions to the Wait_A3W state. When it receives CONN/A3 RTS PDU, it transitions to the Wait_C2 state. When it receives CONN/C2 RTS PDU, it transitions to the opened state, sets the connection time-out protocol variable to the value of the **ConnectionTimeout** field from the CONN/C2 RTS PDU, and indicates to a higher-layer protocol that the connection is opened.

4.2 Flow Control and Receive Windows Example

This example demonstrates how flow control and receive windows work on the abstract level between a sender and a recipient on a channel instance A with fictitious numbers:

Action	Sender local available window	Bytes sent	Recipient local available window	Bytes received
Initial state where the receiver on channel A has successfully advertised a receive window of 1000 bytes but no RPC PDUs have been sent on channel A.	1000	0	1000	0
The sender sends 250 bytes of data to the recipient on channel A and decrements its local available receive window for channel A by the amount of data sent. The sender also increments its total bytes sent by the number of bytes sent.	750	250	1000	0
The recipient receives the 250 bytes of data on channel A, but does not release it from the receive window yet. The recipient decrements its local available receive window for channel A by the number of bytes received. The recipient also increments its total bytes received by the number of bytes received.	750	250	750	250

Action	Sender local available window	Bytes sent	Recipient local available window	Bytes received
The recipient releases 100 bytes of data from the receive window for channel A and increments its local available receive window by the number of bytes removed. The recipient sends a flow control acknowledgment back to the sender on channel A with 250 for the BytesReceived and 850 for the AvailableWindow.	750	250	850	250
Before the flow control acknowledgment is received by the sender, the sender sends another 500 bytes of data to the recipient on channel A and decrements its local available receive window for channel A by the amount of data sent. The sender also increments its total bytes sent by the number of bytes sent.	250	750	850	250
The sender receives the flow control acknowledgment packet and updates its local available receive window for channel A with the following formula: AvailableWindow = AvailableWindow_from_ack - (BytesSent - BytesReceived_from_ack) In this example, the formula expands to: 850 - (750 - 250) = 350	350	750	850	250
The recipient receives the 500 bytes of data on channel A, but does not release it from the receive window yet. The recipient decrements its local available receive window for channel A by the number of bytes received.	350	750	350	750
The recipient releases 200 bytes of data from the receive buffer for channel A and increments its local available receive window by the number of bytes removed. The recipient sends a flow control acknowledgment back to the sender on channel A with 750 for the BytesReceived and 550 for the AvailableWindow.	350	750	550	750
The sender receives the second flow control acknowledgment and updates its local available receive window for channel A with the following formula: 550 - (750 - 750) = 550	550	750	550	750
The recipient releases the remaining 550 bytes of data from the receive window for channel A and increments its local available receive window by the number of bytes removed. The recipient sends a flow control acknowledgment packet back to the server on channel A with 750 for the BytesReceived and 1000 for the AvailableWindow.	550	750	1000	750

Action	Sender local available window	Bytes sent	Recipient local available window	Bytes received
The sender receives the third flow control acknowledgment and updates its local available receive window for channel A with the following formula: 1000 - (750 - 750) = 1000	1000	750	1000	750

5 Security

The following sections specify security considerations for implementers of the Remote Procedure Call Over HTTP Protocol Extensions.

5.1 Security Considerations for Implementers

RPC over HTTP v1 has inadequate security. Implementers SHOULD consider using and implementing RPC over HTTP v2.

Implementers SHOULD consider building implementations that use or encourage the use of RPC over HTTP v2 built on top of HTTPS and enforce the use of HTTP authentication and mandate authorization of the client on the inbound and outbound proxies.

5.2 Index of Security Parameters

Security Parameter	Section
Authentication information	1.7
Server authentication	2.1.2.1
Server authentication	2.2.2

6 Appendix A: Windows Behavior

The information in this specification is applicable to the following versions of Windows:

- Windows NT
- Windows 2000
- Windows XP
- Windows Server 2003
- Windows Vista

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

<1> Section 1.3.2: RPC over HTTP v2 is supported in Windows XP SP1 and later service packs, and Windows Server 2003 and later versions of Windows.

<2> Section 1.6: All versions of Windows starting with Windows NT 4.0 support RPC over HTTP v1. Windows still supports RPC over HTTP v1 for backward compatibility reasons, but Microsoft is actively looking to remove support for RPC over HTTP v1 in future versions of Windows. Windows XP SP1and later service packs and Windows Server 2003 and later versions of Windows support RPC over HTTP v2.

<3> Section 2.1: RPC over HTTP v1 does not support IPv6 addresses, and RPC over HTTP v2 supports IPv6 addresses in Windows Vista and later releases.

<4> Section 2.1: RPC over HTTP v1 and RPC over HTTP v2 on Windows allow a higher-level protocol to specify an HTTP Proxy to be used by this protocol.

<5> Section 2.1.2.1: For HTTPS, the RPC Over HTTP Protocol uses the machine default settings for negotiating security options and does not modify them.

<6> Section 2.1.2.1: Windows Server 2003 and later versions of Windows support authentication using a client-side SSL/TLS certificate.

<7> Section 2.1.2.1: Windows implementations of this protocol request the HTTP protocol stack to use a thirty-minute time-out.

<8> Section 2.1.2.1.1: Windows clients will set this value to 1 GB by default, but this can be overridden by client configuration.

<9> Section 2.1.2.1.3: Windows implementations use Windows error codes, as specified in [MS-ERREF].

<10> Section 2.1.2.1.4: Windows outbound proxies will set this value to 1 GB by default but this can be overridden by outbound proxy configuration.

<11> Section 2.1.2.1.5: Windows clients will set this value to 4.

<12> Section 2.1.2.1.5: Windows clients will send an array of 4 octets in the message body with the successive values being: 0xF8, 0xE8, 0x18, 0x08. These values have no special significance and only serve as a signature for this message.

<13> Section 2.1.2.1.6: Windows implementations of this protocol always send **status-code** 200 and **reason-phrase** "OK".

<14> Section 2.2.2: Windows versions prior to Windows Server 2003 SP1 do not accept the second version of abs_path.

<15> Section 2.2.3.1: Windows implementations of this protocol use a UUID for all RTS cookies.

<16> Section 2.2.3.5.1: Windows uses 64-KB receive windows by default, but registry configuration can override that.

<17> Section 2.2.3.5.5: Windows clients will set this value to 1 GB by default, but this can be overridden by configuration.

<18> Section 2.2.3.5.15: Windows servers impose a limit that an outbound proxy does not send more than 8 KB of ping traffic within a window of four minutes.

<19> Section 3: Windows implementations of this protocol will return one of the errors, as specified in [MS-ERREF], to higher-level protocols. The exact error depends on the failure condition that occurred.

<20> Section 3.1: Windows implementations of this protocol pass data arriving from the Winsock APIs to the Windows implementation of the RPCE Protocol [MS-RPCE] extensions, and send data from the Windows implementations of the RPCE Protocol [MS-RPCE] extensions to the Winsock APIs.

<21> Section 3.1.1.2.1: Windows implementations of this protocol hand off data received from the Winsock APIs to the Windows implementation of the RPCE Protocol [MS-RPCE] extensions.

<22> Section 3.1.1.2.2: Windows implementations of this protocol will return an error to the Windows implementation of the RPCE Protocol [MS-RPCE] extensions, indicating that an error has occurred.

<23> Section 3.1.3.3.2: Windows implementations of this protocol pass on received data from the Winsock APIs to the Windows implementation of the [MS-RPCE] extensions.

<24> Section 3.1.3.3.3: Windows implementations of this protocol will return an error to the Windows implementation of the RPCE Protocol [MS-RPCE] extensions to indicate the occurrence of an error.

<25> Section 3.2.1.1.2.1: Windows implementations of this protocol choose by default a receive window of 64 KB. Administrators can override this size via configuration.

<26> Section 3.2.1.1.2.7: Windows implementations of this protocol maintain this variable.

<27> Section 3.2.1.3.1.1: When an RPC PDU is consumed on the receiver, if FreeWindowAdvertised is less than half of the originally advertised receive window, a new FlowControlAck is sent by the recipient to the sender.

<28> Section 3.2.1.4.3.1: Windows client can use configuration information on the machine to determine if a direct connection to the inbound and outbound RPC over HTTP v2 proxy is needed.

<29> Section 3.2.2.1.2: The higher-level Remote Procedure Call Protocol Extensions specifies usage of this in [MS-RPCE] section 3.3.2.2.1. In TCP RPC transport (ncacn_ip_tcp), Remote Procedure Call Protocol Extensions specifies that the keep-alive interval is changed. In HTTP RPC transport, this protocol variable is changed instead.

<30> Section 3.2.2.2.1: Windows clients allow a system administrator to force a lower connection time-out interval through the registry.

<31> Section 3.2.2.2.3: Windows always uses 200 milliseconds.

<32> Section 3.2.2.3: Higher-level protocols indicate whether HTTP or HTTPS will be used by specifying the RPC_C_HTTP_FLAG_USE_SSL flag in the RPC_SECURITY_QOS_V2 structure when calling the RpcBindingSetAuthInfoEx API as documented in http://msdn.microsoft.com/library/en-us/rpc/rpc/rpc_security_qos_v2.asp. HTTP authentication or client certificate authentication is specified by setting the authentication schemes in the RPC_SECURITY_QOS_V2 structure when calling the pcBindingSetAuthInfoEx API, as documented in http://msdn.microsoft.com/library/default.asp?url=/library/en-us/rpc/rpc/rpc_http_transport_credentials.asp.

<33> Section 3.2.2.4.1.1: Windows consults registry configuration to see if it should do proxy use determination, and depending on registry contents, it uses WinHttp auto-proxy discovery to find out if it needs to use an HTTP proxy.

<34> Section 3.2.2.4.2: Windows implementations of this protocol will return an error to the Windows implementation of the Remote Procedure Call Protocol Extensions as specified in [MS-RPCE] to indicate the send PDU operation failed.

<35> Section 3.2.2.4.2: Windows implementations of this protocol will start IN channel recycling on the client when there is 4 KB left of the channel lifetime.

<36> Section 3.2.2.4.2.1: The Windows implementation returns RPC_S_PROTOCOL_ERROR, as specified in [MS-ERREF], to higher-layer protocols.

<37> Section 3.2.2.5.2: The Windows implementation returns RPC_S_PROTOCOL_ERROR as specified in [MS-ERREF] to higher-layer protocols.

<38> Section 3.2.2.5.5: The Windows implementation returns RPC_S_PROTOCOL_ERROR as specified in [MS-ERREF] to higher-layer protocols.

<39> Section 3.2.2.5.11: The Windows implementation of this protocol returns the value of the RPC_Error field as an error code to the RPC method call during which the error was encountered.

<40> Section 3.2.2.5.11: Versions of Windows earlier than Windows Vista will only send EncodedEEInfo in the message header. Windows Vista and later versions of Windows will send EncodedEEInfo in both the message header and message body.

<41> Section 3.2.2.6.1: Windows implementations interpret "recently" to mean that another RPC or RTS PDU was sent on this channel more recently than one-half of the value of the connection time-out protocol variable.

<42> Section 3.2.2.6.2: Windows implementations interpret "recently" to mean that another RPC or RTS PDU was sent on this channel more recently than one-half of the value of the keep-alive protocol variable.

<43> Section 3.2.3.1.1: The Windows implementation of this protocol reads this value from the local machine configuration. The default value is 900.

<44> Section 3.2.3.1.2: Windows inbound proxies leave the system default value for the keep-alive value for the TCP stack.

<45> Section 3.2.3.1.4: Windows maintains this protocol variable as specified in this section.

<46> Section 3.2.3.1.5: Windows maintains this protocol variable as specified in this section.

<47> Section 3.2.3.3: Windows Server 2003 SP1 and later service packs and Windows versions listen on HTTP/HTTPS URL namespace "/rpcwithcert/rpcproxy.dll".

<48> Section 3.2.3.5.10: Windows implementations use Windows error codes as specified in [MS-ERREF].

<49> Section 3.2.4.1.1: The Windows implementation of this protocol reads this value from local machine configuration. In versions of Windows before Windows Vista, this value is read from the IIS Metabase. In versions of Windows starting with Windows Vista, this value is read from the IIS application host configuration file.

<50> Section 3.2.4.1.4: The Windows implementation of this protocol maintains this protocol variable.

<51> Section 3.2.4.1.5: The Windows implementation of this protocol maintains this protocol variable.

<52> Section 3.2.4.3: Windows Server 2003 SP1 and later service pack and Windows versions listen on HTTP/HTTPS URL namespace "/rpcwithcert/rpcproxy.dll".

<53> Section 3.2.4.5.3: Windows outbound proxies will set this value to 1 GB by default, but this can be overridden by the outbound proxy configuration.

<54> Section 3.2.4.6: The Windows implementation of this protocol notifies the server that it sent ping RTS PDU as follows: Each time it sends a ping RTS PDU, it increments a protocol variable by the size in bytes of the ping RTS PDU. Each time this protocol variable exceeds 1031, the protocol implementation will send a ping traffic sent notify RTS PDU with the size of this protocol variable being set in the PingTrafficSent field of the PingTrafficSentNotify command as defined in section 2.2.3.5.15, and it will reset the protocol variable to zero.

<55> Section 3.2.5.4.1: The Windows implementation of this protocol will return an error to the Windows implementation as specified in [MS-RPCE] section 1.2.1 extensions to indicate the send PDU operation failed.

<56> Section 3.2.5.4.1: Windows implementations of this protocol will start OUT channel recycling on the client when there is 8 KB left of the channel lifetime.

<57> Section 3.2.5.4.1.1: The Windows implementation returns RPC_S_PROTOCOL_ERROR as specified in [MS-ERREF] to higher-layer protocols.

<58> Section 3.2.5.5.2: The Windows implementation of this protocol will pass on PDUs it received from the Winsock APIs to the Windows implementation as specified in [MS-RPCE].

7 Index

G

H

I

K

L

M

N

S

T

U

V

W