**[MS-RDPEUDP]:**

**Remote Desktop Protocol: UDP Transport Extension**

Intellectual Property Rights Notice for Open Specifications Documentation

* **Technical Documentation.** Microsoft publishes Open Specifications documentation (“this documentation”) for protocols, file formats, data portability, computer languages, and standards support. Additionally, overview documents cover inter-protocol relationships and interactions.
* **Copyrights**. This documentation is covered by Microsoft copyrights. Regardless of any other terms that are contained in the terms of use for the Microsoft website that hosts this documentation, you can make copies of it in order to develop implementations of the technologies that are described in this documentation and can distribute portions of it in your implementations that use these technologies or in your documentation as necessary to properly document the implementation. You can also distribute in your implementation, with or without modification, any schemas, IDLs, or code samples that are included in the documentation. This permission also applies to any documents that are referenced in the Open Specifications documentation.
* **No Trade Secrets**. Microsoft does not claim any trade secret rights in this documentation.
* **Patents**. Microsoft has patents that might cover your implementations of the technologies described in the Open Specifications documentation. Neither this notice nor Microsoft's delivery of this documentation grants any licenses under those patents or any other Microsoft patents. However, a given Open Specifications document might be covered by the Microsoft [Open Specifications Promise](http://go.microsoft.com/fwlink/?LinkId=214445) or the [Microsoft Community Promise](http://go.microsoft.com/fwlink/?LinkId=214448). If you would prefer a written license, or if the technologies described in this documentation are not covered by the Open Specifications Promise or Community Promise, as applicable, patent licenses are available by contacting iplg@microsoft.com.
* **Trademarks**. The names of companies and products contained in this documentation might be covered by trademarks or similar intellectual property rights. This notice does not grant any licenses under those rights. For a list of Microsoft trademarks, visit [www.microsoft.com/trademarks](http://www.microsoft.com/trademarks).
* **Fictitious Names**. The example companies, organizations, products, domain names, email addresses, logos, people, places, and events that are depicted in this documentation are fictitious. No association with any real company, organization, product, domain name, email address, logo, person, place, or event is intended or should be inferred.

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

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

**Revision Summary**

| Date | Revision History | Revision Class | Comments |
| --- | --- | --- | --- |
| 12/16/2011 | 1.0 | New | Released new document. |
| 3/30/2012 | 1.0 | None | No changes to the meaning, language, or formatting of the technical content. |
| 7/12/2012 | 2.0 | Major | Significantly changed the technical content. |
| 10/25/2012 | 3.0 | Major | Significantly changed the technical content. |
| 1/31/2013 | 4.0 | Major | Significantly changed the technical content. |
| 8/8/2013 | 5.0 | Major | Significantly changed the technical content. |
| 11/14/2013 | 6.0 | Major | Significantly changed the technical content. |
| 2/13/2014 | 7.0 | Major | Significantly changed the technical content. |
| 5/15/2014 | 7.0 | None | No changes to the meaning, language, or formatting of the technical content. |
| 6/30/2015 | 8.0 | Major | Significantly changed the technical content. |
| 10/16/2015 | 8.0 | None | No changes to the meaning, language, or formatting of the technical content. |
| 3/2/2016 | 9.0 | Major | Significantly changed the technical content. |
| 7/14/2016 | 9.0 | None | No changes to the meaning, language, or formatting of the technical content. |

Table of Contents

[1 Introduction 5](#_Toc456187167)

[1.1 Glossary 5](#_Toc456187168)

[1.2 References 6](#_Toc456187169)

[1.2.1 Normative References 6](#_Toc456187170)

[1.2.2 Informative References 6](#_Toc456187171)

[1.3 Overview 7](#_Toc456187172)

[1.3.1 RDP-UDP Protocol 7](#_Toc456187173)

[1.3.2 Message Flows 8](#_Toc456187174)

[1.3.2.1 UDP Connection Initialization 8](#_Toc456187175)

[1.3.2.2 UDP Data Transfer 9](#_Toc456187176)

[1.4 Relationship to Other Protocols 9](#_Toc456187177)

[1.5 Prerequisites/Preconditions 10](#_Toc456187178)

[1.6 Applicability Statement 10](#_Toc456187179)

[1.7 Versioning and Capability Negotiation 10](#_Toc456187180)

[1.8 Vendor-Extensible Fields 10](#_Toc456187181)

[1.9 Standards Assignments 10](#_Toc456187182)

[2 Messages 11](#_Toc456187183)

[2.1 Transport 11](#_Toc456187184)

[2.2 Message Syntax 11](#_Toc456187185)

[2.2.1 Enumerations 11](#_Toc456187186)

[2.2.1.1 VECTOR\_ELEMENT\_STATE Enumeration 11](#_Toc456187187)

[2.2.2 Structures 11](#_Toc456187188)

[2.2.2.1 RDPUDP\_FEC\_HEADER Structure 11](#_Toc456187189)

[2.2.2.2 RDPUDP\_FEC\_PAYLOAD\_HEADER Structure 13](#_Toc456187190)

[2.2.2.3 RDPUDP\_PAYLOAD\_PREFIX Structure 13](#_Toc456187191)

[2.2.2.4 RDPUDP\_SOURCE\_PAYLOAD\_HEADER Structure 13](#_Toc456187192)

[2.2.2.5 RDPUDP\_SYNDATA\_PAYLOAD Structure 14](#_Toc456187193)

[2.2.2.6 RDPUDP\_ACK\_OF\_ACKVECTOR\_HEADER Structure 14](#_Toc456187194)

[2.2.2.7 RDPUDP\_ACK\_VECTOR\_HEADER Structure 14](#_Toc456187195)

[2.2.2.8 RDPUDP\_CORRELATION\_ID\_PAYLOAD Structure 15](#_Toc456187196)

[2.2.2.9 RDPUDP\_SYNDATAEX\_PAYLOAD Structure 16](#_Toc456187197)

[2.2.3 Vectors 16](#_Toc456187198)

[2.2.3.1 ACK Vector 16](#_Toc456187199)

[3 Protocol Details 18](#_Toc456187200)

[3.1 Common Details 18](#_Toc456187201)

[3.1.1 Abstract Data Model 18](#_Toc456187202)

[3.1.1.1 Transport Modes 18](#_Toc456187203)

[3.1.1.2 Sequence Numbers 18](#_Toc456187204)

[3.1.1.3 MTU Negotiation 19](#_Toc456187205)

[3.1.1.4 Acknowledgments 19](#_Toc456187206)

[3.1.1.4.1 Lost Datagrams 19](#_Toc456187207)

[3.1.1.5 Retransmits 20](#_Toc456187208)

[3.1.1.6 FEC Computations 20](#_Toc456187209)

[3.1.1.6.1 Finite Field Arithmetic 20](#_Toc456187210)

[3.1.1.6.1.1 Addition and Subtraction 20](#_Toc456187211)

[3.1.1.6.1.2 Multiplication and Division 21](#_Toc456187212)

[3.1.1.6.1.3 Logarithms and Exponents 22](#_Toc456187213)

[3.1.1.6.2 FEC Encoding 22](#_Toc456187214)

[3.1.1.6.3 FEC Decoding 24](#_Toc456187215)

[3.1.1.6.4 Selecting the Coefficients Matrix 25](#_Toc456187216)

[3.1.1.6.5 Structure of Source Packets used for FEC Encoding 26](#_Toc456187217)

[3.1.1.7 Flow Control 26](#_Toc456187218)

[3.1.1.8 Congestion Control 26](#_Toc456187219)

[3.1.1.9 Keepalives 27](#_Toc456187220)

[3.1.2 Timers 27](#_Toc456187221)

[3.1.3 Initialization 27](#_Toc456187222)

[3.1.4 Higher-Layer Triggered Events 28](#_Toc456187223)

[3.1.4.1 Initializing a Connection 28](#_Toc456187224)

[3.1.4.2 Sending a Datagram 28](#_Toc456187225)

[3.1.4.3 Receiving a Datagram 28](#_Toc456187226)

[3.1.4.4 Terminating a Connection 28](#_Toc456187227)

[3.1.5 Message Processing Events and Sequencing Rules 28](#_Toc456187228)

[3.1.5.1 Constructing Messages 30](#_Toc456187229)

[3.1.5.1.1 SYN Datagrams 30](#_Toc456187230)

[3.1.5.1.2 ACK Datagrams 31](#_Toc456187231)

[3.1.5.1.3 SYN and ACK Datagrams 31](#_Toc456187232)

[3.1.5.1.4 ACK and Source Packets Data 31](#_Toc456187233)

[3.1.5.1.5 ACK and FEC Packets Data 32](#_Toc456187234)

[3.1.5.2 Connection Sequence 32](#_Toc456187235)

[3.1.5.3 Data Transfer Phase 33](#_Toc456187236)

[3.1.5.3.1 Sender Receives Data 33](#_Toc456187237)

[3.1.5.3.2 Sender Sends Data 33](#_Toc456187238)

[3.1.5.3.2.1 Source Packet 33](#_Toc456187239)

[3.1.5.3.2.2 FEC Packet 34](#_Toc456187240)

[3.1.5.3.3 Receiver Receives Data 34](#_Toc456187241)

[3.1.5.3.4 User Consumes Data 34](#_Toc456187242)

[3.1.5.4 Termination 34](#_Toc456187243)

[3.1.5.4.1 Retransmit Limit 34](#_Toc456187244)

[3.1.5.4.2 Keepalive Timer Fires 34](#_Toc456187245)

[3.1.6 Timer Events 34](#_Toc456187246)

[3.1.6.1 Retransmit Timer 34](#_Toc456187247)

[3.1.6.2 Keepalive Timer on the Sender 35](#_Toc456187248)

[3.1.6.3 Delayed ACK Timer 35](#_Toc456187249)

[3.1.7 Other Local Events 35](#_Toc456187250)

[4 Protocol Examples 36](#_Toc456187251)

[4.1 UDP Connection Initialization Packets 36](#_Toc456187252)

[4.1.1 SYN Packet 36](#_Toc456187253)

[4.1.2 SYN and ACK Packet 36](#_Toc456187254)

[4.2 UDP Data Transfer Packets 37](#_Toc456187255)

[4.2.1 Source Packet 37](#_Toc456187256)

[4.2.2 FEC Packet 38](#_Toc456187257)

[4.2.2.1 Payload of an FEC Packet 39](#_Toc456187258)

[4.2.3 ACK Packet 39](#_Toc456187259)

[5 Security 41](#_Toc456187260)

[5.1 Security Considerations for Implementers 41](#_Toc456187261)

[5.1.1 Using Sequence Numbers 41](#_Toc456187262)

[5.1.2 RDP-UDP Datagram Validation 41](#_Toc456187263)

[5.1.3 Congestion Notifications 41](#_Toc456187264)

[5.2 Index of Security Parameters 41](#_Toc456187265)

[6 Appendix A: Product Behavior 42](#_Toc456187266)

[7 Change Tracking 43](#_Toc456187267)

[8 Index 44](#_Toc456187268)

# Introduction

The Remote Desktop Protocol: UDP Transport Extension specifies extensions to the transport mechanisms in the [**Remote Desktop Protocol (RDP)**](#gt_17c795a6-68bf-46bf-8ea8-467c8df1a0b3). This document specifies network connectivity between the user's machine and a remote computer system over the [**User Datagram Protocol (UDP)**](#gt_a70f5e84-6960-42f0-a160-ba0281eb548d).

Sections 1.5, 1.8, 1.9, 2, and 3 of this specification are normative. All other sections and examples in this specification are informative.

## Glossary

This document uses the following terms:

**acknowledgment (ACK)**: A signal passed between communicating processes or computers to signify successful receipt of a transmission as part of a communications protocol.

**binary large object (BLOB)**: A collection of binary data stored as a single entity in a database.

**Coded Packet**: A Source Packet or an FEC Packet.

**FEC block**: An FEC Packet that is added to the data stream after a group of Source Packets have been processed. In case one of the Source Packets in the group is lost, the redundant information that is contained in the FEC Packet can be used for recovery.

**FEC Packet**: A packet that encapsulates the payload after running an FEC logic.

**forward error correction (FEC)**: A process in which a sender uses redundancy to enable a receiver to recover from packet loss.

**Internet Protocol version 4 (IPv4)**: An Internet protocol that has 32-bit source and destination addresses. IPv4 is the predecessor of IPv6.

**Internet Protocol version 6 (IPv6)**: A revised version of the Internet Protocol (IP) designed to address growth on the Internet. Improvements include a 128-bit IP address size, expanded routing capabilities, and support for authentication (2) and privacy.

**maximum transmission unit (MTU)**: The size, in bytes, of the largest packet that a given layer of a communications protocol can pass onward.

**network address translation (NAT)**: The process of converting between IP addresses used within an intranet, or other private network, and Internet IP addresses.

**network byte order**: The order in which the bytes of a multiple-byte number are transmitted on a network, most significant byte first (in big-endian storage). This may or may not match the order in which numbers are normally stored in memory for a particular processor.

**Remote Desktop Protocol (RDP)**: A multi-channel protocol that allows a user to connect to a computer running Microsoft Terminal Services (TS). RDP enables the exchange of client and server settings and also enables negotiation of common settings to use for the duration of the connection, so that input, graphics, and other data can be exchanged and processed between client and server.

**round-trip time (RTT)**: The time that it takes a packet to be sent to a remote partner and for that partner's acknowledgment to arrive at the original sender. This is a measurement of latency between partners.

**run-length encoding (RLE)**: A form of data compression in which repeated values are represented by a count and a single instance of the value.

**Source Packet**: A packet that encapsulates data that was generated by the user.

**terminal client**: The client that initiated the remote desktop connection.

**terminal server**: A computer on which terminal services is running.

**Transmission Control Protocol (TCP)**: A protocol used with the Internet Protocol (IP) to send data in the form of message units between computers over the Internet. TCP handles keeping track of the individual units of data (called packets) that a message is divided into for efficient routing through the Internet.

**User Datagram Protocol (UDP)**: The connectionless protocol within TCP/IP that corresponds to the transport layer in the ISO/OSI reference model.

**MAY, SHOULD, MUST, SHOULD NOT, MUST NOT:** These terms (in all caps) are used as defined in [[RFC2119]](http://go.microsoft.com/fwlink/?LinkId=90317). All statements of optional behavior use either MAY, SHOULD, or SHOULD NOT.

## References

Links to a document in the Microsoft Open Specifications library point to the correct section in the most recently published version of the referenced document. However, because individual documents in the library are not updated at the same time, the section numbers in the documents may not match. You can confirm the correct section numbering by checking the [Errata](http://msdn.microsoft.com/en-us/library/dn781092.aspx).

### 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.

[MS-DTYP] Microsoft Corporation, "[Windows Data Types](file:///E%3A%5CTarget%5CWindows%5CPublished%5CBooks%5CMS-RDPEUDP%5C%5BMS-DTYP%5D.pdf#Section_cca2742956894a16b2b49325d93e4ba2)".

[MS-RDPBCGR] Microsoft Corporation, "[Remote Desktop Protocol: Basic Connectivity and Graphics Remoting](file:///E%3A%5CTarget%5CWindows%5CPublished%5CBooks%5CMS-RDPEUDP%5C%5BMS-RDPBCGR%5D.pdf#Section_5073f4ed1e9345e1b0396e30c385867c)".

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997, [http://www.rfc-editor.org/rfc/rfc2119.txt](http://go.microsoft.com/fwlink/?LinkId=90317)

### Informative References

[RFC1948] Bellovin, S., "Defending Against Sequence Number Attacks", RFC 1948, May 1996, [http://tools.ietf.org/html/rfc1948.txt](http://go.microsoft.com/fwlink/?LinkId=225732)

[RFC3782] Floyd, S., Henderson, T., and Gurtov, A., "The NewReno Modification to TCP's Fast Recovery Algorithm", RFC 3782, April 2004, [http://tools.ietf.org/html/rfc3782.txt](http://go.microsoft.com/fwlink/?LinkId=225733)

[RFC4340] Kohler, E., Handley, M., and Floyd, S., "Datagram Congestion Control Protocol (DCCP)", RFC 4340, March 2006, [http://www.ietf.org/rfc/rfc4340.txt](http://go.microsoft.com/fwlink/?LinkId=90473)

[RFC4341] Floyd, S., and Kohler, E., "Profile for Datagram Congestion Control Protocol (DCCP) Congestion Control ID 2: TCP-like Congestion Control", RFC 4341, March 2006, [http://tools.ietf.org/html/rfc4341.txt](http://go.microsoft.com/fwlink/?LinkId=225734)

[RFC5681] Allman, M., Paxson, V., and Blanton, E., "TCP Congestion Control", RFC 5681, September 2009, [http://tools.ietf.org/html/rfc5681.txt](http://go.microsoft.com/fwlink/?LinkId=225735)

[RFC793] Postel, J., Ed., "Transmission Control Protocol: DARPA Internet Program Protocol Specification", RFC 793, September 1981, [http://www.rfc-editor.org/rfc/rfc793.txt](http://go.microsoft.com/fwlink/?LinkId=150872)

## Overview

The Remote Desktop Protocol: UDP Transport Extension Protocol has been designed to improve the performance of the network connectivity compared to a corresponding RDP-TCP connection, especially on wide area networks (WANs) or wireless networks.

It has the following two primary goals:

* Gain a higher network share while reducing the variation in packet transit delays.
* Share network resources with other users.

To achieve these goals, the protocol has two modes of operation. The first mode is a reliable mode where data is transferred reliably through persistent retransmits. The second mode is an unreliable mode, where no guarantees are made about reliability and the timeliness of data is preserved by avoiding retransmits. In addition, the Remote Desktop Protocol: UDP Transport Extension Protocol includes a [**forward error correction (FEC)**](#gt_abc86a79-bd31-443b-9e73-83ef488303ff) logic that can be used to recover from random packet losses.

The protocol’s two communicating parties, the endpoints of the UDP connection, are peers and use the same protocol. The connection between the two endpoints is bidirectional – data and acknowledgments (section [3.1.1.4](#Section_ae448afed83f479792e1a8596456ebff)) can be transmitted in both directions simultaneously. Logically, each single connection can be viewed as two unidirectional connections, as shown in the following figure. Both of these unidirectional connections are symmetrical and each endpoint has both a Sender and a Receiver entity. In this specification, the initiating endpoint A is referred to as the [**terminal client**](#gt_0e260a32-2049-4eaa-bdec-bfdee19bad4b) and endpoint B is referred to as the [**terminal server**](#gt_b416f72e-cf04-4d80-bf93-f5753f3b0998).



Figure 1: The UDP bidirectional endpoints connection

### RDP-UDP Protocol

The Remote Desktop Protocol: UDP Transport Extension Protocol has two distinct phases of operation. The initial phase, UDP Connection Initialization (section [1.3.2.1](#Section_8275cf8a6b98497fb8b7a8d015888b89)), occurs when a UDP connection is initialized between the terminal client and the terminal server. Data pertaining to the connection is exchanged and the **UDP** connection is set up. Once this phase is completed successfully, the protocol enters the UDP Data Transfer (section [1.3.2.2](#Section_d6dc8925ca524045845195a0f58a870e)) phase, where [**Coded Packets**](#gt_4790b884-caac-4877-aea6-d9a2ae4a3a47) are exchanged.

The protocol can operate in one of two modes. The operational mode is determined during the UDP Connection Initialization phase. These modes are as follows:

* RDP-UDP-R or "Reliable" Mode: In this mode, the endpoint retransmits datagrams that have been lost by the underlying network fabric.
* RDP-UDP-L or "Best-Efforts" Mode: In this mode, the reliable delivery of datagrams is not guaranteed, and the endpoint does not retransmit datagrams.

The connection between the endpoints is terminated when either the terminal client or terminal server terminates the connection. No protocol-specific messages are exchanged to communicate that the endpoint is no longer present.

### Message Flows

The two endpoints, the terminal client and the terminal server, first set up a connection, and then transfer the data as shown in the following figure.



Figure 2: The UDP connection initialization and UDP data transfer message flow

The following sections describe the two phases of the communication and the detailed data transfer.

#### UDP Connection Initialization

In this phase, both endpoints are initialized with mutually agreeable parameters for the connection.

The terminal client initiates the connection by sending a SYN datagram. The terminal client also determines the mode of operation, RDP-UDP-R or RDP-UDP-L, as described in section [1.3.1](#Section_aea14a52baa14486bcd8f30505ec707d). The terminal server responds with a datagram with the SYN flag set, along with an ACK flag, to acknowledge the receipt of the SYN datagram. The terminal client acknowledges the SYN datagram by sending an ACK. The terminal client can append the Coded Packets along with the ACK datagram. This datagram indicates that a connection has been set up and data can be exchanged.

All datagrams in this phase – the SYN, SYN+ACK, and ACK – are delivered reliably by using persistent retransmits, irrespective of the mode that the transport is operating in.

#### UDP Data Transfer

In this phase, which follows the UDP Connection Initialization (section [1.3.2.1](#Section_8275cf8a6b98497fb8b7a8d015888b89)) phase, the data generated by the users of this protocol is exchanged. This phase ends when either the connection is terminated by the user, or when an endpoint determines that the remote endpoint is no longer present.

The terminal server (sender) and terminal client (receiver) exchange Coded Packets in this phase. A schematic diagram of the FEC engine is shown in the following diagram.



Figure 3: FEC engine

The Remote Desktop Protocol: UDP Transport Extension Protocol uses the FEC mechanism for recovery from packet losses. An [**FEC Packet**](#gt_846ebd3d-2bc4-40c5-ba01-00272fc7a1ae) is added to the data stream after processing a block of m [**Source Packets**](#gt_6f3820df-3806-4e53-afe8-9f9b992a48ce). Each FEC Packet carries redundant information regarding these Source Packets. This information can be used in case one of the m Source Packets is lost and needs to be recovered. A generic equation for generating an FEC Packet is listed as follows.



Figure 4: Generic equation for an FEC Packet

The FEC Packets require no acknowledgments (section [3.1.1.4](#Section_ae448afed83f479792e1a8596456ebff)), and they are not retransmitted. The sender can either set the [**FEC block**](#gt_d3f4c532-ee5f-4d3a-8ecb-f03d2f384b3a) size to any value up to 255 or to not send any FEC Packets in the stream. Likewise, the receiver, upon a receipt of an FEC Packet, can ignore the FEC Packet and not use it for any decoding operations.

Upon receiving notification of a packet loss, the sender retransmits the lost datagram. The implementation of the FEC mechanism in the RDP-UDP protocol is only used for recovery from packet losses.

## Relationship to Other Protocols

The Remote Desktop Protocol: UDP Transport Extension Protocol works on top of the User Datagram Protocol (UDP).

## Prerequisites/Preconditions

The protocol endpoints require UDP connectivity to be established. The network path between the endpoints allows the transfer of **UDP** datagrams in both directions.

The prerequisites for this protocol are identical to those for the **UDP** protocol.

## Applicability Statement

This protocol can be used in place of any [**Transmission Control Protocol (TCP)**](#gt_b08d36f6-b5c6-4ce4-8d2d-6f2ab75ea4cb) transport for the Remote Desktop Protocol (RDP) protocol. The protocol's two modes of operation are required to be considered. The RDP-UDP-R mode is used when a stream-based, reliable transport, akin to **TCP**, is required. The RDP-UDP-L mode is used when a datagram/message-based, best-efforts transport, akin to UDP, is required.

## Versioning and Capability Negotiation

The version of the Remote Desktop Protocol: UDP Transport Extension is negotiated in the SYN request and the SYN + ACK response between the two endpoints. The first endpoint optionally indicates the maximum protocol version it supports in the SYN datagram, and the second endpoint optionally indicates the maximum protocol version supported by both endpoints in the SYN + ACK datagram. The highest version supported by both endpoints is used, and if either endpoint does not indicate a protocol version, version 1 is used by both.

* Version 1: The first version of the protocol has a minimum retransmit time-out of 500 ms (section [3.1.6.1](#Section_78e889c926c34ffe8c29e9cb4ed34345)), and a minimum delayed ACK time-out of 200 ms (section [3.1.6.3](#Section_8c80ed0f0164479db94e23a08014941b)).
* Version 2: The second version improves performance on low-latency networks by reducing the minimum retransmit time-out to 300 ms (section 3.1.6.1), and the minimum delayed ACK time-out to 50 ms (section 3.1.6.3).

Implementations can support either version 1 or both version 1 and version 2 of the protocol. The negotiation of the protocol version between the two endpoints is described in section [3.1.5.1](#Section_5af070be92774196b57006e61ce81b43).

## Vendor-Extensible Fields

None.

## Standards Assignments

None.

# Messages

## Transport

The RDP protocol packets are encapsulated in the User Datagram Protocol (UDP). The **UDP** datagrams MUST be encapsulated in the [**Internet Protocol version 4 (IPv4)**](#gt_0f25c9b5-dc73-4c3e-9433-f09d1f62ea8e) or the [**Internet Protocol version 6 (IPv6)**](#gt_64c29bb6-c8b2-4281-9f3a-c1eb5d2288aa).

The default port for incoming **UDP** connection requests on the terminal server is port 3389. All of the **RDP** traffic over **UDP** is handled by this single port on the terminal server.

The terminal client MUST open a unique **UDP** socket for each instance of this transport. Each socket is bound to a different port.

## Message Syntax

All of the messages written to the network or read from the network MUST be in [**network byte order**](#gt_502de58c-ffc0-4dda-8fcb-b152b2c31fba), as described in [[RFC4340]](http://go.microsoft.com/fwlink/?LinkId=90473) section 11.

The protocol references commonly used data types as defined in [[MS-DTYP]](file:///E%3A%5CTarget%5CWindows%5CPublished%5CBooks%5CMS-RDPEUDP%5C%5BMS-DTYP%5D.pdf#Section_cca2742956894a16b2b49325d93e4ba2).

### Enumerations

#### VECTOR\_ELEMENT\_STATE Enumeration

The VECTOR\_ELEMENT\_STATE enumeration is sent along with every ACK vector (section [2.2.3.1](#Section_2d5bfb5ec5fe4a9bb9addbafae206948)) that acknowledges the receipt of a continuous array of datagrams.

| Field/Value | Description |
| --- | --- |
| DATAGRAM\_RECEIVED0 | A datagram was received. |
| DATAGRAM\_RESERVED\_11 | Not used. |
| DATAGRAM\_RESERVED\_22 | Not used. |
| DATAGRAM\_NOT\_YET\_RECEIVED3 | A datagram has not been received yet. |

### Structures

#### RDPUDP\_FEC\_HEADER Structure

The **RDPUDP\_FEC\_HEADER** structure forms the basic header for every datagram sent or received by the endpoint.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 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 |
| snSourceAck |
| uReceiveWindowSize | uFlags |

**snSourceAck (4 bytes):** A 32-bit unsigned value that specifies the highest sequence number for a Source Packet detected by the remote endpoint. This value wraps around; for more information about the sequence numbers range, see [[RFC793]](http://go.microsoft.com/fwlink/?LinkId=150872) section 3.3.

**uReceiveWindowSize (2 bytes):** A 16-bit unsigned value that specifies the size of the receiver's buffer.

**uFlags (2 bytes):** A 16-bit unsigned integer that indicates supported options, or additional headers.

The following table describes the meaning of each flag.

| Flags | Meaning |
| --- | --- |
| RDPUDP\_FLAG\_SYN0x0001 | Corresponds to the SYN flag, for initializing connection. |
| RDPUDP\_FLAG\_FIN0x0002 | Corresponds to the FIN flag. Currently unused. |
| RDPUDP\_FLAG\_ACK0x0004 | Specifies that the RDPUDP\_ACK\_VECTOR\_HEADER Structure (section [2.2.2.7](#Section_56ab5c6b92a946b7b998ca9e86fad728)) is present. |
| RDPUDP\_FLAG\_DATA0x0008 | Specifies that the RDPUDP\_SOURCE\_PAYLOAD\_HEADER Structure (section [2.2.2.4](#Section_d98c71ec945b4d1c8a03c16818ae3f20)) or the RDPUDP\_FEC\_PAYLOAD\_HEADER Structure (section [2.2.2.2](#Section_ca61eb58943b48c7a5995ccea0876441)) is present. This flag specifies that the datagram has additional data beyond the UDP ACK headers. |
| RDPUDP\_FLAG\_FEC0x0010 | Specifies that the RDPUDP\_FEC\_PAYLOAD\_HEADER Structure (section 2.2.2.2) is present.  |
| RDPUDP\_FLAG\_CN0x0020 | **Congestion Notification** flag (section [3.1.1](#Section_857bae4ea5224d00a0f0aea98bbbb961)), the receiver reports missing datagrams. |
| RDPUDP\_FLAG\_CWR0x0040 | **Congestion Window Reset** flag (section 3.1.1), the sender has reduced the congestion window, and informs the receiver to stop adding the RDPUDP\_FLAG\_CN. |
| RDPUDP\_FLAG\_SACK\_OPTION 0x0080 | Not used. |
| RDPUDP\_FLAG\_ACK\_OF\_ACKS 0x0100 | Specifies that the RDPUDP\_ACK\_OF\_ACKVECTOR\_HEADER Structure (section [2.2.2.6](#Section_8cdfce71517b4d68b545b708e8f84d5e)) is present. |
| RDPUDP\_FLAG\_SYNLOSSY 0x0200 | Specifies that the connection does not require persistent retransmits. |
| RDPUDP\_FLAG\_ACKDELAYED0x0400 | Specifies that the receiver delayed generating the ACK for the source sequence numbers received. The sender is not to use this ACK for estimating the network RTT. |
| RDPUDP\_FLAG\_CORRELATION\_ID0x0800 | Specifies that the optional RDPUDP\_CORRELATION\_ID\_PAYLOAD Structure (section [2.2.2.8](#Section_f42d4b49dc6242d387769fcf0d998586)) is present. |
| RDPUDP\_FLAG\_SYNEX0x1000 | Specifies that the optional RDPUDP\_SYNDATAEX\_PAYLOAD Structure (section [2.2.2.9](#Section_f5984e6516ef448fa49520971a99082f)) is present. |

#### RDPUDP\_FEC\_PAYLOAD\_HEADER Structure

The **RDPUDP\_FEC\_PAYLOAD\_HEADER** structure accompanies every datagram that contains an FEC payload.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 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 |
| snCoded |
| snSourceStart |
| uRange | uFecIndex | uPadding |

**snCoded (4 bytes):** A 32-bit unsigned value that contains the sequence number for a Coded Packet.

**snSourceStart (4 bytes):** A 32-bit unsigned value that specifies the first sequence number of a Source Packet that is contained in the FEC payload.

**uRange (1 byte):** An unsigned 8-bit value that, when added to **snSourceStart**, yields the range of packets that are contained in the FEC payload.

**uFecIndex (1 byte):** An 8-bit unsigned value. This value is generated by the FEC engine.

**uPadding (2 bytes):** An array of UINT8 ([[MS-DTYP]](file:///E%3A%5CTarget%5CWindows%5CPublished%5CBooks%5CMS-RDPEUDP%5C%5BMS-DTYP%5D.pdf#Section_cca2742956894a16b2b49325d93e4ba2) section 2.2.47).

#### RDPUDP\_PAYLOAD\_PREFIX Structure

The **RDPUDP\_PAYLOAD\_PREFIX** structure specifies the length of a data payload. This header is used for generating an FEC Packet or for decoding an FEC Packet. Once a datagram is decoded by using FEC, this field specifies the size of the recovered datagram.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 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 |
| cbPayloadSize |

**cbPayloadSize (2 bytes):** An unsigned 16-bit value that specifies the size of the data payload.

#### RDPUDP\_SOURCE\_PAYLOAD\_HEADER Structure

The **RDPUDP\_SOURCE\_PAYLOAD\_HEADER** structure specifies the metadata of a data payload.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 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 |
| snCoded |
| snSourceStart |

**snCoded (4 bytes):** An unsigned 32-bit value that specifies the sequence number for the current Coded Packet.

**snSourceStart (4 bytes):** An unsigned 32-bit value that specifies the sequence number for the current Source Packet.

#### RDPUDP\_SYNDATA\_PAYLOAD Structure

The **RDPUDP\_SYNDATA\_PAYLOAD** structure specifies the parameters that are used to initialize the UDP connection.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 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 |
| snInitialSequenceNumber |
| uUpStreamMtu | uDownStreamMtu |

**snInitialSequenceNumber (4 bytes):** A 32-bit unsigned value that specifies the starting value for sequence numbers for Source Packets and Coded Packets.

**uUpStreamMtu (2 bytes):** A 16-bit unsigned value that specifies the maximum size for a datagram that can be generated by the endpoint. This value MUST be greater than or equal to 1132 and less than or equal to 1232.

**uDownStreamMtu (2 bytes):** A 16-bit unsigned value that specifies the maximum size of the [**maximum transmission unit (MTU)**](#gt_03aae42f-32fd-47ab-b413-d5ec92d29d45) that the endpoint can accept. This value MUST be greater than or equal to 1132 and less than or equal to 1232.

#### RDPUDP\_ACK\_OF\_ACKVECTOR\_HEADER Structure

The **RDPUDP\_ACK\_OF\_ACKVECTOR\_HEADER** structure resets the start position of an ACK vector (section [2.2.3.1](#Section_2d5bfb5ec5fe4a9bb9addbafae206948)).

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 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 |
| snAckOfAcksSeqNum |

**snAckOfAcksSeqNum (4 bytes):** This value specifies the new sequence number from which the ACK vector starts encoding the state of the receiver queue. The receiver generates the ACK Vector for sequence numbers greater than the **snAckOfAcksSeqNum**. The minimum ACK Vector sequence number is to be greater of the **snAckOfAcksSeqNum** and the lowest sequence number the receiver expects (current window).

The sender sets the **AckOfAck** sequence number with the greatest cumulative ACK it has received and processed. The sender SHOULD send **AckOfAck** every 20 packets.

#### RDPUDP\_ACK\_VECTOR\_HEADER Structure

The **RDPUDP\_ACK\_VECTOR\_HEADER** structure contains the [ACK vector (section 2.2.3.1)](#Section_2d5bfb5ec5fe4a9bb9addbafae206948) that specifies the states of the datagram in the receiver’s queue. This vector is a variable-size array. The states are encoded by using [**run-length encoding (RLE)**](#gt_df9ca00e-9abb-4642-b365-2cb0d91d533d) and are stored in this array.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 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 |
| uAckVectorSize | AckVectorElement (variable) |
| ... |
| ... |
| Padding (variable) |
| ... |
| ... |

**uAckVectorSize (2 bytes):** A 16-bit unsigned value that contains the size of the **AckVectorElement** array. The maximum size of the ACK Vector is 2048 bytes.

**AckVectorElement (variable):** An array of ACK Vector elements. Each element is composed of a state, and the number of contiguous datagrams that share the same state.

**Padding (variable):** A variable-sized array, of length zero or more, such that this structure ends on a DWORD ([[MS-DTYP]](file:///E%3A%5CTarget%5CWindows%5CPublished%5CBooks%5CMS-RDPEUDP%5C%5BMS-DTYP%5D.pdf#Section_cca2742956894a16b2b49325d93e4ba2) section 2.2.9) boundary.

#### RDPUDP\_CORRELATION\_ID\_PAYLOAD Structure

The **RDPUDP\_CORRELATION\_ID\_PAYLOAD** structure allows a terminal client to specify the correlation identifier for the connection, which can appear in some of the terminal server's event logs. Otherwise, the terminal server can generate a random identifier.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 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 |
| uCorrelationId (16 bytes) |
| ... |
| ... |
| uReserved (16 bytes) |
| ... |
| ... |

**uCorrelationId (16 bytes):** DTYP.GUID. An array of 16 8-bit, unsigned integers that specifies a unique identifier to associate with the connection. The value MUST be transmitted in big-endian byte order. The most-significant byte SHOULD NOT have a value of 0x00 or 0xF4. The value 0x0D SHOULD NOT be used in any of the bytes. The value of this field SHOULD be the same as the value provided in the RDP\_NEG\_CORRELATION\_INFO structure ([[MS-RDPBCGR]](file:///E%3A%5CTarget%5CWindows%5CPublished%5CBooks%5CMS-RDPEUDP%5C%5BMS-RDPBCGR%5D.pdf#Section_5073f4ed1e9345e1b0396e30c385867c) section 2.2.1.1.2).

**uReserved (16 bytes):** 16 8-bit values, all set to 0x00.

#### RDPUDP\_SYNDATAEX\_PAYLOAD Structure

The RDPUDP\_SYNDATAEX\_PAYLOAD structure specifies extended parameters that are used to configure the UDP connection.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 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 |
| uSynExFlags | uUdpVer |

**uSynExFlags (2 bytes)**: A 16-bit unsigned integer that indicates supported options. The following table describes the meaning of each flag.

| Flags | Meaning |
| --- | --- |
| RDPUDP\_VERSION\_INFO\_VALID0x0001 | The **uUdpVer** field indicates a supported version of the RDP-UDP protocol. |

**uUdpVer (2 bytes)**: A 16-bit unsigned value. When the RDPUDP\_VERSION\_INFO\_VALID flag is present, this specifies a supported version of the UDP Transport Extension, used to negotiate with the other endpoint.

| Flags | Meaning |
| --- | --- |
| RDPUDP\_PROTOCOL\_VERSION\_10x0001 | The minimum retransmit time-out is 500 ms (section [3.1.6.1](#Section_78e889c926c34ffe8c29e9cb4ed34345)), and the minimum delayed ACK time-out is 200 ms (section [3.1.6.3](#Section_8c80ed0f0164479db94e23a08014941b)).[<1>](#Appendix_A_1" \o "Product behavior note 1) |
| RDPUDP\_PROTOCOL\_VERSION\_20x0002 | The minimum retransmit time-out is 300 ms (section 3.1.6.1), and the minimum delayed ACK time-out is 50 ms (section 3.1.6.3).[<2>](#Appendix_A_2" \o "Product behavior note 2) |

### Vectors

#### ACK Vector

The ACK vector captures the state of the queue of Source Packets at the receiver endpoint.

Each position in the queue can have two values that indicate whether a Source Packet is present in the queue, or not. The run-length encoding (RLE) compression is used for encoding the states of Source Packets in the array.

An ACK Vector comprises a number of elements, as specified by the **uAckVectorSize** field in the RDPUDP\_ACK\_VECTOR\_HEADER structure (section [2.2.2.7](#Section_56ab5c6b92a946b7b998ca9e86fad728)). Each element is 8 bits long.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 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 |
| uAckVectorSize | S | S | L | L | L | L | L | L | AckVec Element[2] |

The two most significant bits of each element compose the VECTOR\_ELEMENT\_STATE enumeration (section [2.2.1.1](#Section_7e7e58dc2b7d441dbc02e9ca5ba66323)). The next 6 bits are the length of a continuous sequence of datagrams that share the same state.

The ACK vectors form a [**binary large object (BLOB)**](#gt_ad861812-8cb0-497a-80bb-13c95aa4e425), and are padded so that they are aligned to WORD ([[MS-DTYP]](file:///E%3A%5CTarget%5CWindows%5CPublished%5CBooks%5CMS-RDPEUDP%5C%5BMS-DTYP%5D.pdf#Section_cca2742956894a16b2b49325d93e4ba2) section 2.2.61) boundaries.

This is similar to the description of ACK vectors in the Datagram Congestion Control Protocol (DCCP), as described in [[RFC4341]](http://go.microsoft.com/fwlink/?LinkId=225734).

# Protocol Details

## Common Details

### 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 an 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.

**Initial Sequence Number:** Each endpoint advertises the first sequence number that will be used when sending the datagrams. The Coded sequence number (section [3.1.1.2](#Section_5fd219caa9194305a4c112b4f84cb484)) and the Source sequence number (section 3.1.1.2) for the first datagram sent will be equal to this value.

**Congestion Control:** Each endpoint MUST notify the remote endpoint of congestion events. Congestion events are characterized by lost or missing datagrams.

**Congestion Notification:** The **RDPUDP\_FLAG\_CN** flag (section [2.2.2.1](#Section_ceaae261e53840f08678254e31621054)) indicates that the remote endpoint has detected congestion events.

**Congestion Window Reset:** The **RDPUDP\_FLAG\_CWR** flag (section 2.2.2.1) indicates that the endpoint has reacted to the congestion notification message, and that the remote endpoint MUST stop sending **Congestion Notifications**.

#### Transport Modes

When the connection is initialized in the RDP-UDP-R mode, as described in section [1.3.1](#Section_aea14a52baa14486bcd8f30505ec707d), persistent retransmits ensure that all datagrams written to the sender will be read respectively at the receiver.

When the connection is initialized in the RDP-UDP-L mode with the **RDPUDP\_FLAG\_SYNLOSSY** flag (section [2.2.2.1](#Section_ceaae261e53840f08678254e31621054)), the sender does not retransmit any datagrams. In this mode, not all datagrams generated by the user on the sender side are received by the user on the receiver side. However, the ordering of datagrams MUST be preserved and datagrams MUST be read at the receiver in the same order in which they were written by the sender.

In RDP-UDP-L, the receiver SHOULD maintain a timer for out-of-order packets. This timer SHOULD be enabled when the first out-of-order packet is received and disabled when all missing datagrams have been received. When this timer fires, the receiver SHOULD stop the timer and process datagrams it has received. The receiver SHOULD process any out-of-order packet that is in the right edge of the receiver window. This ensures new packets are not dropped.

The order of the datagrams is determined according to their sequence numbers, as specified in section [3.1.1.2](#Section_5fd219caa9194305a4c112b4f84cb484).

#### Sequence Numbers

All Coded Packets and Source Packets have a sequence number that identifies their sending order. The sequence numbers for the Coded Packets and the Source Packets are independent of each other.

The **Initial Sequence Number** abstract data model (ADM) element for both Coded Packets and Source Packets is initialized as follows:

**Initial Sequence Number** = **snInitialSequenceNumber** in the RPDUDP\_SYNDATA\_PAYLOAD Structure (section [2.2.2.5](#Section_3cbfe21d143d4fc88463813b42264c58)).

This initial value is a true random number. This field is similar to the initial sequence number (ISN) field used in the TCP transport protocol; for more information about the ISN field, see [[RFC1948]](http://go.microsoft.com/fwlink/?LinkId=225732).

The Coded Packet sequence number is referred to as the Coded sequence number. The Coded sequence number uniquely identifies each datagram sent by the sender. The Coded sequence number value is increased by one for each Coded Packet that was sent. Retransmitted Source Packets can have different Coded sequence numbers.

The Source Packet sequence number is referred to as the Source sequence number. Each Source Packet encapsulates a data payload. The Source sequence number uniquely identifies this data payload. The Source sequence number value is increased by one for each data payload that was sent.

The sequence numbers wrap around due to space limitations. Implementations MUST handle this wrap-around scenario. For more information about the sequence numbers range, see [[RFC793]](http://go.microsoft.com/fwlink/?LinkId=150872) section 3.3.

#### MTU Negotiation

The largest data payload that can be transferred over this protocol is negotiated during the 3-way UDP handshake process, called MTU negotiation. The size of the Internet Protocol (IP) or MAC layer headers and other underlying network headers is not a part of this negotiation.

The RDP-client advertises the largest payload it can send (**uUpStreamMtu**) and the largest payload it can receive (**uDownStreamMtu)** as a part of the SYN datagram, as specified in section [2.2.2.5](#Section_3cbfe21d143d4fc88463813b42264c58). The minimum of these values and the data payload sizes the server can send or receive determines the negotiated MTU, as shown in the following equation.

Negotiated **uUpStreamMtu** = minimum (Advertised **uUpStreamMtu**, Received **uDownStreamMtu**, 1232) + Maximum size of the RDPUDP\_ACK\_OF\_ACKVECTOR\_HEADER Structure (section [2.2.2.6](#Section_8cdfce71517b4d68b545b708e8f84d5e))

Negotiated **uDownStreamMtu** = minimum (Advertised **uDownstreamMtu**, Received **uUpStreamMtu**, 1232) + Maximum size of the RDPUDP\_ACK\_OF\_ACKVECTOR\_HEADER Structure (section 2.2.2.6)

The server sends these values to the client as a part of the SYN+ACK packet (section [3.1.5.1.3](#Section_96eaa81aff4240a2884c96b3834db6c8)); this is the final negotiated MTU size. The client MUST NOT send a data payload larger than the value specified in **uUpStreamMtu**, and the server MUST NOT send data larger than **uDownStreamMtu**. Values that do not fall within this range are unacceptable. If such oversized payloads are detected, either endpoint MUST ignore such UDP datagrams. This could possibly lead to a connection termination, initiated by any layer in the RDP stack, because some part of the data was lost.

The range of **uUpStreamMtu** and **uDownStreamMtu** is in the closed interval [1132, 1232]. The advertised MTU MUST NOT be smaller than 1132 or larger than 1232.

#### Acknowledgments

An [**acknowledgment (ACK)**](#gt_6aa258ea-917f-461a-9c54-1b1a66791965) is sent from the receiver to the sender, informing the sender about the receipt of a Source Packet. An acknowledgment MUST be generated for every Source Packet received. However, because acknowledgments are cumulative, the number of Source Packets for which a receiver generates an acknowledgment is implementation-specific.[<3>](#Appendix_A_3" \o "Product behavior note 3) Only Source Packets MUST be acknowledged by the receiver; FEC Packets MUST NOT be acknowledged by the receiver.

Each acknowledgment contains an ACK Vector (section [2.2.3.1](#Section_2d5bfb5ec5fe4a9bb9addbafae206948)).

##### Lost Datagrams

Lost datagrams notification is a part of the **Congestion Control** ADM element implementation. It is used to control the rate of the data that is transferred between the endpoints as described in section [5.1.3](#Section_9810e7e522774acea3df42ff1b805893).

The receiver marks a datagram as lost only when it receives three other datagrams after its original transmission, with sequence numbers greater than the original datagram. Similarly, the sender marks a packet as lost only when it receives an acknowledgment (section [3.1.1.4](#Section_ae448afed83f479792e1a8596456ebff)) for any three packets that have a sequence number greater than the lost packet.

#### Retransmits

The Remote Desktop Protocol: UDP Transport Extension does not specify a retransmit mechanism. An implementation can choose any retransmit method; for example, the Fast Retransmit method, as described in [[RFC5681]](http://go.microsoft.com/fwlink/?LinkId=225735).

When the sender detects that the receiver did not receive a specific Source Packet (section [3.1.1.4.1](#Section_b7c69b866ef241fd946eab2bf5c05c04)), the sender retransmits that Source Packet. Only Source Packets MUST be retransmitted.

#### FEC Computations

This section explains the operations involved in generating an FEC Packet. An FEC Packet is generated by a linear combination of a number of Source Packets, as described in section [1.3.2.2](#Section_d6dc8925ca524045845195a0f58a870e), over a Galois Field, as specified in [Bewersdorff]. A brief introduction on finite field arithmetic is given in section [3.1.1.6.1](#Section_2c86bd06db6743e38c23f6658ad90110). The coefficients of the equation are described in section [3.1.1.6.4](#Section_2b42363d9eed4159ab240811d7bf7d56). The actual FEC encoding and decoding are described in section [3.1.1.6.2](#Section_80d87a0b10c542c2af5eb978e5f76f34) and section [3.1.1.6.3](#Section_4cfcbf75c0a44127b516200ec9f3c99b), respectively.

##### Finite Field Arithmetic

A finite field is a finite set of numbers. All arithmetic operations performed on this field will yield a result that belongs to the same finite field. For example, a finite field of size 256 with numbers from 0 to 255 is defined. All the arithmetic operations (addition, subtraction, multiplication, and division) on this field will yield a result in the range of 0 to 255, thus belonging to the original finite field itself. Conventional arithmetic differs from finite field arithmetic as it operates on an infinite set of real numbers. For more details on finite fields, see [Lidl].

All binary numbers belonging to a finite field (also known as a Galois field, GF(pn)), where p is a prime number and n is a positive integer, can be represented in a polynomial form and in a finite field with binary numbers (for example in GF(256)=GF(28)), where a is the coefficient of this equation with a value equal to zero or 1.



Figure 5: Galois field and binary representation example

###### Addition and Subtraction

Adding or subtracting two polynomials is done by grouping coefficients of the same order, similar to regular algebra. However, since this operation is performed in GF(28), the result is brought into the finite field by performing a modulo 2 operation on each of the coefficients in the polynomial representation.

The addition operation over the finite field is logically equivalent to a XOR operation. Thus, adding or subtracting two polynomials means XORing them together, as described in the following figure.



Figure 6: Addition and subtraction example

In a finite field of GF(2n), such as GF(256), addition and subtraction are equivalent operations.

Pseudo-code example:

1. BYTE Add(const BYTE x, const BYTE y)
2. {
3. return (x ^ y);
4. }
5. BYTE Sub(const BYTE x, const BYTE y)
6. {
7. return (x ^ y);
8. }

###### Multiplication and Division

Multiplication in the finite field can be performed in one of the following two ways:

* Using logarithms
* Multiplying the two polynomials and reducing the result with an irreducible polynomial to bring it back in the finite field

It is simpler to perform multiplications and divisions using logarithms, as it involves a table lookup for the log function, followed by an addition of the polynomials, followed by an exponent function.



Figure 7: Multiplication equation

Division is performed similarly using logarithms and exponentiation.



Figure 8: Division equation

Since the discrete logarithm of an element in the finite field is a regular integer, the addition in the exponent is a regular addition modulo 2n.

Pseudo-code example:

1. BYTE Div(const int x, const int y)
2. {
3. if (y==0) return 0;
4. if (x==0) return 0;
5. return (BYTE)(m\_ffExp2Poly[m\_ffPoly2Exp[x] - m\_ffPoly2Exp[y] + (MAX\_FIELD\_SIZE-1)]);
6. }
7. BYTE Mul(const int x, const int y)
8. {
9. if (((x-1) | (y-1)) < 0)
10. return (0);
11. return (BYTE)(m\_ffExp2Poly[m\_ffPoly2Exp[x] + m\_ffPoly2Exp[y]]);
12. }

Where m\_ffExp2Poly and m\_ffPoly2Exp are exponent and log tables respectively.

###### Logarithms and Exponents

Exponents can be calculated by repeatedly multiplying the same number, and then using a modulo operation to ensure that the result stays in the finite field.

Pseudo-code example:

1. reduction = 0x1d;
2. m\_ffExp2Poly[0] = 0x01;
3. for (i = 1; i < m\_fieldSize - 1; i++)
4. {
5. temp = m\_ffExp2Poly[i - 1] << 1;
6. if (temp & m\_fieldSize)
7. {
8. m\_ffExp2Poly[i] = (temp & ~m\_fieldSize) ^ reduction;
9. }
10. else
11. {
12. m\_ffExp2Poly[i] = (byte)temp;
13. }
14. }

Where m\_fieldSize is 256 for GF(28)

Logarithms are the inverse of exponents, and can be easily calculated by reversing the previous operation as shown in the following pseudo-code example:

1. m\_ffPoly2Exp[0] = 2 \* m\_fieldSize; // no exponential representation, doesn't exist
2. for (i = 0; i < m\_fieldSize - 1; i++)
3. {
4. m\_ffPoly2Exp[m\_ffExp2Poly[i]] = (byte)i;
5. }

Logarithms and exponents can be obtained by using the methods described previously to generate logarithms and exponent lookup tables.

##### FEC Encoding

As described in section [1.3.2.2](#Section_d6dc8925ca524045845195a0f58a870e), an FEC Packet is added to the data stream after processing a block of Source Packets. The size of the FEC Packet is equal to the size of the largest Source Packet in the group. In the following representation, each Source Packet Sn contains at most k bytes. All the Source Packets with a size smaller than k are padded with bytes containing zero.



Figure 9: Source Packet and FEC Packet representation

The FEC Packet is generated with the following equation.



Figure 10: FEC encoding

The product of these two matrices will give us a row matrix, which is the FEC Packet of size 1 \* k. The method in which the coefficients are generated is explained in the following pseudo-code example and in the following sections.

Pseudo-code example:

1. //
2. // Generate the log and exponent tables.
3. //
4. PrepareExpLogArrays();
5. //
6. // Generate a set of packets. Fill them with random data for this example.
7. //
8. Packet S1, S2, S3, S4, S5, F15;
9. S1.GeneratePacketData(10);
10. S2.GeneratePacketData(20);
11. S3.GeneratePacketData(15);
12. S4.GeneratePacketData(15);
13. S5.GeneratePacketData(20);
14. //
15. // Print the packets out for verification.
16. //
17. S1.PrintPacketData();
18. S2.PrintPacketData();
19. S3.PrintPacketData();
20. S4.PrintPacketData();
21. S5.PrintPacketData();
22. //
23. // The coefficient arrays and the fecIndex generated from FEC calculations
24. //
25. BYTE fecIndex = 0;
26. BYTE CoEfficientArray[5] = {0, 0, 0, 0, 0};
27. GenerateCoeffArray(CoEfficientArray, 5, 1, 5, &fecIndex);
28. printf("CoEff Array [%d %d %d %d %d]\n", CoEfficientArray[0],
29. CoEfficientArray[1],
30. CoEfficientArray[2],
31. CoEfficientArray[3],
32. CoEfficientArray[4]);
33. //
34. // Generating a matrix of source packets
35. //
36. BYTE\* FECGeneratorArray[5] = {S1.m\_pbPacket,
37. S2.m\_pbPacket,
38. S3.m\_pbPacket,
39. S4.m\_pbPacket,
40. S5.m\_pbPacket
41. };
42. //
43. // Generate the FEC packet.
44. //
45. MatrixMultiply(F15.m\_pbPacket, CoEfficientArray, 5, FECGeneratorArray, 5, 22);
46. //
47. // Print the FEC packet for verification.
48. //
49. F15.PrintFECData(22);
50. .....
51. void MatrixMultiply(BYTE \*fecArr, BYTE\* CoEffArray, int cbCoEffArrayCount, BYTE\*\* FECGeneratorArray, int cbRowCount, int cbColumnCount)
52. {
53. for (int i = 0; i < cbColumnCount; i++)
54. {
55. fecArr[i] = 0;
56. for (int j = 0; j < cbCoEffArrayCount; j++)
57. {
58. fecArr[i] = Mul(CoEffArray[j],FECGeneratorArray[j][i]) ^ fecArr[i];
59. }
60. }
61. }

##### FEC Decoding

An FEC decoding operation is the reverse of the FEC encoding (section [3.1.1.6.2](#Section_80d87a0b10c542c2af5eb978e5f76f34)) operation. The FEC decoding operation solves the linear equation that is used to recover the lost Source Packets. Each FEC Packet can be used to recover only one Source Packet in the range covered by that FEC Packet.

To decode, or recover a missing datagram using FEC, the following matrix is constructed where packet F1 is the FEC block for Source Packets S1 – Sn.

For simplicity, assume n=5. If packet S4 is missing, it can be recovered by using the following matrix operation.



Figure 11: Matrix operation for FEC decoding

Here, matrix S’ contains an unknown term (S4) that needs to be computed. This can be done by converting Cd to an identity matrix using the Gauss-Jordan elimination. For more details on the Gauss-Jordan elimination, see [Press].

Not all matrices have an inverse, and in some cases, Cd’ doesn’t exist. For such operations, the FEC Packet cannot be used to recover from that particular Source Packet. Thus, not all FEC operations are reversible, and not being able to decode a FEC Packet is not fatal. The missing Source Packet is always retransmitted in RDP-UDP-R mode (section [3.1.1.7](#Section_a102f4abe4114f9a93d76d07b3b98e1b)), and can be ignored for RDP-UDP-L mode (section 3.1.1.7).

Pseudo-code example:

1. // Regenerate Coefficient array from fecIndex.
2. //
3. RegenerateCoeffArrayFromFecIndex(CoEfficientArray, 5, fecIndex, 1, 5);
4. //
5. // Compute the missing packet (S3) by inverting the matrix.
6. // This is the algebraic equivalent
7. // of a matrix inverse.
8. //
9. for (int i = 0; i < 22; i++)
10. {
11. printf("%d ", Div(Mul(CoEfficientArray[0], S1.m\_pbPacket[i]) ^
12. Mul(CoEfficientArray[1], S2.m\_pbPacket[i]) ^
13. Mul(CoEfficientArray[3], S4.m\_pbPacket[i]) ^
14. Mul(CoEfficientArray[4], S5.m\_pbPacket[i]) ^
15. F15.m\_pbPacket[i],CoEfficientArray[2]));
16. }
17. printf("\n");

##### Selecting the Coefficients Matrix

If the Source sequence numbers (section [3.1.1.2](#Section_5fd219caa9194305a4c112b4f84cb484)) for packets S1, S2, S3 … Sn are s1, s2, s3 … sn, the coefficient matrix is calculated as follows.



Figure 12: Matrix coefficient calculation

The division uses finite field division as described in section [3.1.1.6.1.2](#Section_a587ef707ad84515a4b3c2db19004ab4). Note that since all the packets in an FEC Packet are sequential, s2=s1+1, s3=s1+2, …, sn=s1+(n-1).

Only the last byte of the Source sequence number is used in calculating the coefficient. The **fecIndex** field described in the following pseudo-code example is equivalent to the **uFecIndex** field, as specified in section [2.2.2.2](#Section_ca61eb58943b48c7a5995ccea0876441). The value of the **fecIndex** field is updated using the following code prior to every call for encoding an FEC Packet:

1. if ((sn&0xf) >= (s1 &0xf) && ((fecIndex >= (s1 &0xf)) && (fecIndex <= (sn&0xf))) ||
2. (sn&0xf) < (s1 &0xf) && ((fecIndex >= (s1 &0xf)) || (fecIndex <= (sn&0xf))))
3. fecIndex = (sn+1) & 0xf;

Pseudo-code example:

1. void GenerateCoeffArray(BYTE \*pbCoEfficientArray,
2. int cLength,
3. USHORT ucOrigStart,
4. USHORT ucOrigEnd,
5. \_\_out BYTE \*pucFecIndex)
6. {
7. if ((ucOrigEnd >= ucOrigStart) &&
8. ((\*pucFecIndex >= ucOrigStart) && (\*pucFecIndex <= ucOrigEnd)))
9. \*pucFecIndex = (BYTE)(ucOrigEnd+1);
10. if ((ucOrigEnd < ucOrigStart) &&
11. ((\*pucFecIndex >= ucOrigStart) || (\*pucFecIndex <= ucOrigEnd)))
12. \*pucFecIndex = (BYTE)(ucOrigEnd+1);
13. for (int i=0; i < cLength; i++, ucOrigStart++)
14. {
15. BYTE e = Div(1, (\*pucFecIndex)^ucOrigStart);
16. pbCoEfficientArray[i] = (BYTE)m\_ffPoly2Exp[e];
17. }
18. }
19. void RegenerateCoeffArrayFromFecIndex(BYTE \*pbCoefficientArray,
20. int cLength,
21. BYTE fecIndex,
22. USHORT ucOrigStart,
23. USHORT ucOrigEnd)
24. {
25. for (int i=0; i < cLength; i++, ucOrigStart++)
26. {
27. BYTE e = Div(1, fecIndex^ucOrigStart);
28. pbCoefficientArray[i] = (BYTE)m\_ffPoly2Exp[e];
29. }
30. }

##### Structure of Source Packets used for FEC Encoding

Only for the FEC Encoding operations, Source Packets are prepended with a 2 byte RDPUDP\_PAYLOAD\_PREFIX (section [2.2.2.3](#Section_3eddc65b0e314ae8a6dc2024082ba427)) header. This header is used only for the FEC encoding and decoding operations, and is not transmitted to the terminal client. This field contains the size of each Source Packet, specified in the network byte order. When a datagram is recovered using FEC, the first 2 bytes constitute of this header, and specify the size of the recovered datagram to the decoder.

#### Flow Control

The Flow Control feature is similar to the TCP transport protocol Flow Control, as specified in [[RFC793]](http://go.microsoft.com/fwlink/?LinkId=150872).

The main objective of Flow Control is to prevent a fast sender from sending too many datagrams to a slow receiver and congesting it. The receiver advertises the number of datagrams it can accommodate at any given time. The sender MUST NOT send more datagrams than the advertised number of datagrams. The receiver SHOULD discard all datagrams that fall outside the advertised window.

The Flow Control algorithm allows the sender to transmit packets in the following range:

(**CumAcked** + 1) to (**CumAcked** + **uReceiveWindowSize**)

**CumAcked:**  An internal state variable of the sender.

* For an RDP-UDP-R sender (section [1.3.1](#Section_aea14a52baa14486bcd8f30505ec707d)), this is the highest sequence number where all datagrams with a smaller sequence number have already been received by the receiver.
* For an RDP-UDP-L sender (section 1.3.1), this is the highest sequence number where all datagrams with a smaller sequence number have been either received or marked as lost by the receiver.

**uReceiveWindowSize:**  The receiver advertised window defined in the **RDPUDP\_FEC\_HEADER** structure, as specified in section [2.2.2.1](#Section_ceaae261e53840f08678254e31621054).

#### Congestion Control

The **Congestion Control** abstract data model (ADM) element is used to limit the rate at which the sender sends Source Packets. Controlling the network throughput enables sharing the network resources with other users and avoiding network congestion. The sender MUST implement some form of **Congestion Control** logic. Any NewReno variant implementation can be an acceptable option. For more information about NewReno variants, see [[RFC3782]](http://go.microsoft.com/fwlink/?LinkId=225733).

When the sender receives the **RDPUDP\_FLAG\_CN** flag (section [2.2.2.1](#Section_ceaae261e53840f08678254e31621054)), which notifies of a datagram loss, the sender MUST immediately react and reduce its network throughput. The next Source Packet sent by the sender MUST have an **RDPUDP\_FLAG\_CWR** flag (section 2.2.2.1) to indicate that the sender has reacted to the **Congestion Notification** ADM element. The sender will remember the source packet that carries the **RDPUDP\_FLAG\_CWR**. The receiver will stop setting the **RDPUDP\_FLAG\_CN** on acknowledgment once it receives the **RDPUDP\_FLAG\_CWR**. On the other side, the sender will then ignore the set **RDPUDP\_FLAG\_CN** flags on subsequent acknowledgments from any receiver that has an snSourceAck ADM in the acknowledgment that is less than the previously remembered sequence number.

Additionally, the sender SHOULD set the **RDPUDP\_FLAG\_CWR** flag whenever a retransmit occurs due to the Retransmit Timer (section [3.1.6.1](#Section_78e889c926c34ffe8c29e9cb4ed34345)) firing to indicate that a datagram loss was detected, even if the **RDPUDP\_FLAG\_CN** flag was not set by the receiver. If the receiver is not setting the **RDPUDP\_FLAG\_CN** flag, no action is needed on receipt of the **RDPUDP\_FLAG\_CWR** flag.

The sender reacts to losses that take place every [**round-trip time (RTT)**](#gt_ae37224f-415b-4e14-9cf3-62f666d8a976) only. There could be multiple losses in an RTT, and the sender MUST NOT react to those events. This behavior is similar to the NewReno variants behavior, as described in [RFC3782].

#### Keepalives

As the underlying transport is based on UDP and is connectionless, each pair of endpoints MUST constantly send data to make sure that the other endpoint is present and is responding to network events. If there is no data to send, each endpoint MUST periodically acknowledge the last received datagram. Otherwise, the [**network address translation (NAT)**](#gt_7ee5c1a4-6768-4256-817c-6686382e0f39) en route between the peers can block the UDP connection.

If the sender does not receive any datagram from the receiver after 65 seconds, it is determined that the remote endpoint has entered the Closed state (section [3.1.5](#Section_575660d7a69848de92dbd4d4c9fcc783)), and that the connection has been terminated.

Because the delivery of acknowledgments (section [3.1.1.4](#Section_ae448afed83f479792e1a8596456ebff)) is not guaranteed, the receiver SHOULD send one or more keepalive datagrams in implementation-specific[<4>](#Appendix_A_4" \o "Product behavior note 4) time intervals smaller or equal to 65 seconds. If the sender does not receive at least one keep-alive datagram every 65 seconds, it terminates the connection.

### Timers

The following timers are used by the Remote Desktop Protocol: UDP Transport Extension and MUST be implemented:

**Retransmit**: This timer is used for indicating that no acknowledgment (section [3.1.1.4](#Section_ae448afed83f479792e1a8596456ebff)) has been received for a datagram that was transmitted earlier.

**Keepalive at the sender**: This timer is used for maintaining an active connection between the endpoints.

**Delayed ACK**: This timer is used for indicating the receipt of a Source Packet that was not acknowledged yet and has no acknowledgment scheduled for it.

### Initialization

Before the protocol operation can commence, UDP network connectivity has to be established between the endpoints: the terminal client and the terminal server.

The terminal server MUST open a **UDP** socket, and bind it to the default RDP port 3389, as specified in section [2.1](#Section_efafc7a931af43d8a8304b80044797d8). The terminal server listens on this socket for incoming connections.

The terminal client MUST open a **UDP** socket to the terminal server. The terminal client MUST connect to the port that the terminal server is listening on. If there are multiple connections, each connection MUST have a unique port number on the terminal client.

### Higher-Layer Triggered Events

#### Initializing a Connection

The user of this protocol MUST initialize a UDP connection between the endpoints as described in section [1.3.2.1](#Section_8275cf8a6b98497fb8b7a8d015888b89).

#### Sending a Datagram

The user of this protocol can send data from one endpoint to another using this protocol. The protocol MUST send the data across only if the two endpoints are in the Established state.

#### Receiving a Datagram

The user of this protocol MUST be notified on receipt of a datagram when one endpoint receives data sent by the remote endpoint. The endpoints MUST be in the Established state.

#### Terminating a Connection

The user of this protocol can terminate a connection at any point in time. Datagrams SHOULD NOT be sent by the transport after the user has terminated the connection. All of the datagrams received after the connection termination MUST be ignored.

### Message Processing Events and Sequencing Rules

The states of the protocol, divided into the terminal server states and the terminal client states, are illustrated in the following figure.



Figure 13: State diagram for the terminal server and terminal client states

The states are described as follows:

Closed state: Both the terminal server sender and the terminal client receiver can be in the Closed state. The endpoint in a Closed state MUST NOT respond to any networking events, and MUST NOT generate or process any datagrams. The endpoint enters the Closed state when the Retransmit timer or the Keepalive timer is fired, as specified in section [3.1.5.4](#Section_e63ba09d2a554e4783513f73723ca2db).

Listen state: Only the terminal server sender can enter this state. The terminal server listens on the port for incoming UDP connections, as specified in section [3.1.3](#Section_85cc555bd749476da2cfde1e95bc0c42).

SYN\_SENT: Only the terminal client receiver can enter this state, after sending a SYN packet and thus initiating the connection.

SYN\_RECEIVED: Only the terminal server sender can enter this state, after receiving a SYN packet from the terminal client receiver.

Established: This state indicates that a connection has been established, and datagrams are exchanged between the two endpoints.

Duplicate messages are ignored and discarded by either endpoint. The exchanged messages are specified in the following sections.

#### Constructing Messages

##### SYN Datagrams

The following steps specify the creation of a SYN datagram:

1. An RDPUDP\_FEC\_HEADER structure (section [2.2.2.1](#Section_ceaae261e53840f08678254e31621054)) MUST be appended to the UDP datagram.
	* The **snSourceAck** variable MUST be set to -1.
	* The **uReceiveWindowSize** variable MUST be set to the size of the receive buffer. The receive buffer is the number of packets the receiver specified it can buffer.
	* The **uFlags** variable MUST be set as follows:
		+ The **RDPUDP\_FLAG\_SYN** flag MUST be set.
		+ The **RDPUDP\_FLAG\_SYNLOSSY** flag MUST be set only when neither endpoint requires retransmission of lost datagrams.
		+ The **RDPUDP\_FLAG\_CORRELATION\_ID** flag MUST be set only when the RDPUDP\_CORRELATION\_ID\_PAYLOAD structure (section [2.2.2.8](#Section_f42d4b49dc6242d387769fcf0d998586)) is included.
		+ The **RDPUDP\_FLAG\_SYNEX** flag MUST be set only when the RDPUDP\_SYNDATAEX\_PAYLOAD structure (section [2.2.2.9](#Section_f5984e6516ef448fa49520971a99082f)) is included.
2. The RDPUDP\_SYNDATA\_PAYLOAD structure (section [2.2.2.5](#Section_3cbfe21d143d4fc88463813b42264c58)) MUST be appended to the UDP datagram.
	* The **snInitialSequenceNumber** variable MUST be set to a 32-bit number generated by using a truly random function.
	* The **uUpStreamMtu** field MUST be set to a value in the range of 1132 to 1232.
	* The **uDownStreamMtu** field MTU MUST be set to a value in the range of 1132 to 1232.
3. The RDPUDP\_CORRELATION\_ID\_PAYLOAD structure (section 2.2.2.8) MUST be appended to the UDP datagram if the RDPUDP\_FLAG\_CORRELATION\_ID flag is set in **uFlags**.
	* The **uCorrelationId** variable MUST be filled with 8-bit numbers generated by using a truly random function, except that: The value MUST be transmitted in big-endian byte order. The most-significant byte is not to have a value of 0x00 or 0xF4. None of the bytes are to have the value 0x0D. This value is to be the same as provided in the RDP\_NEG\_CORRELATION\_INFO structure ([[MS-RDPBCGR]](file:///E%3A%5CTarget%5CWindows%5CPublished%5CBooks%5CMS-RDPEUDP%5C%5BMS-RDPBCGR%5D.pdf#Section_5073f4ed1e9345e1b0396e30c385867c) section 2.2.1.1.2).
	* The **uReserved** variable MUST be filled with 16 8-bit numbers, all with value 0x00.
4. The RDPUDP\_SYNEX\_PAYLOAD structure (section 2.2.2.9) MUST be appended to the UDP datagram if the RDPUDP\_FLAG\_SYNEX flag is set in **uFlags**. Not appending this structure implies that RDPUDP\_PROTOCOL\_VERSION\_1 is the highest protocol version supported. This structure SHOULD NOT be appended if this datagram is in response to a SYN from the other endpoint where the RDP\_FLAG\_SYNEX flag was not specified. The **uSynExFlags** field MUST be set as follows:
	* The RDPUDP\_VERSION\_INFO\_VALID flag MUST be set only if the structure contains a valid RDP-UDP protocol version.
	* If the RDPUDP\_VERSION\_INFO\_VALID flag is present, the **uUdpVer** field MUST be set to the highest RDP-UDP protocol version supported by the endpoint, or if the other endpoint has already sent a SYN, the highest version supported by both endpoints.
5. This datagram MUST be zero-padded to increase the size of this datagram to 1232 bytes.

##### ACK Datagrams

The following steps specify the creation of an ACK datagram:

1. An RDPUDP\_FEC\_HEADER structure (section [2.2.2.1](#Section_ceaae261e53840f08678254e31621054)) MUST be appended to the UDP datagram.
	* The **snSourceAck** variable MUST be set to the largest sequence number the receiver has seen so far. Sequence numbers will wrap over after overflow, and the receiver MUST handle this case.
	* The **uReceiveWindowSize** variable MUST be set to the size of the receive buffer. The receive buffer is the number of packets the receiver specified it can buffer.
	* The **uFlags** flag MUST be set as follows:
		+ The **RDPUDP\_FLAG\_ACK** flag MUST be set.
		+ The **RDPUDP\_FLAG\_CN** flag SHOULD be set only if the receiver has detected a lost datagram and has not received a datagram with the **RDPUDP\_FLAG\_CWR** flag corresponding to that **RDPUDP\_FLAG\_CN** flag.
		+ The **RDPUDP\_FLAG\_ACK\_OF\_ACKS** flag SHOULD be set only if the sender sends an ACK for the section ACK Vector (section [2.2.3.1](#Section_2d5bfb5ec5fe4a9bb9addbafae206948)).
2. An RDPUDP\_ACK\_VECTOR\_HEADER structure (section [2.2.2.7](#Section_56ab5c6b92a946b7b998ca9e86fad728)) header MUST be appended as follows:
	* The **uAckVectorSize** variable MUST be set to the number of elements in the array.
	* An array of elements, that captures the receiver’s queue by using run-length encoding (RLE), as specified in section [3.1.1.4.1](#Section_b7c69b866ef241fd946eab2bf5c05c04).
3. An RDPUDP\_ACK\_OF\_ACKVECTOR\_HEADER structure (section [2.2.2.6](#Section_8cdfce71517b4d68b545b708e8f84d5e)) SHOULD be appended by the sender if both of the following occur:
	* The **RDPUDP\_FLAG\_ACK\_OF\_ACKS** flag is set.
	* The **snAckOfAcksSeqNum** variable was set as the new start position of the ACK Vector.

##### SYN and ACK Datagrams

A SYN datagram is generated, as specified in section [3.1.5.1.1](#Section_066f9acffd574f95ab3f334e748bab10), with the following fields set as follows:

* The **snSourceAck** field in the RDPUDP\_FEC\_HEADER structure (section [2.2.2.1](#Section_ceaae261e53840f08678254e31621054)) MUST be set to the **snInitialSequenceNumber** value received in the SYN packet (section 3.1.5.1.1).
* The **RDPUDP\_FLAG\_ACK** flag MUST be set in the RDPUDP\_FEC\_HEADER structure (section 2.2.2.1).
* The **uUpStreamMtu** and **uDownStreamMtu** in the RDPUDP\_SYNDATA\_PAYLOAD structure (section [2.2.2.5](#Section_3cbfe21d143d4fc88463813b42264c58)) MUST be set as specified in the algorithm described in section [3.1.1.3](#Section_ffbc66efe79f49f7b20876d4a68339e8). The values of these fields MUST be in the range of 1132 to 1232 bytes.
* The RDPUDP\_SYNEX\_PAYLOAD structure (section [2.2.2.9](#Section_f5984e6516ef448fa49520971a99082f)) SHOULD only be present if it is also present in the received SYN packet. The **uUdpVer** field MUST be set to the highest RDP-UDP protocol version supported by both endpoints. The highest version supported by both endpoints, which is RDPUDP\_PROTOCOL\_VERSION\_1 if either this packet or the SYN packet does not specify a version, is the version that MUST be used by both endpoints.

##### ACK and Source Packets Data

The following steps specify the creation of an ACK and Source Packet datagram:

1. An ACK datagram is generated, as specified in section [3.1.5.1.2](#Section_facb0b3163c644f4aeec03b5163aedae).
	* The **RDPUDP\_FLAG\_DATA** flag MUST be set.
	* The **RDPUDP\_FLAG\_CWR** flag SHOULD be set for the first **RDPUDP\_FLAG\_CN** flag seen in an RTT.
2. An RDPUDP\_SOURCE\_PAYLOAD structure (section [2.2.2.4](#Section_d98c71ec945b4d1c8a03c16818ae3f20)) header MUST be appended.
	* The **snCoded** variable value MUST be set to the previously transmitted datagram’s **snCoded** value plus 1. If this is the first datagram, this value is the advertised **Initial Sequence Number** ADM element plus 1.
	* The **snSourceStart** variable MUST be set. It is incremented for each chunk of data written to the transport. The initial value is the advertised **Initial Sequence Number** ADM element plus 1**.**
3. The data payload protocol data MUST be appended.

##### ACK and FEC Packets Data

The following steps specify the creation of an ACK and FEC Packet datagram.

1. An ACK datagram is generated, as specified in section [3.1.5.1.2](#Section_facb0b3163c644f4aeec03b5163aedae).
	* The **RDPUDP\_FLAG\_DATA** flag MUST be set.
	* The **RDPUDP\_FLAG\_FEC** flag MUST be set.
2. An RDPUDP\_FEC\_PAYLOAD\_HEADER structure (section [2.2.2.2](#Section_ca61eb58943b48c7a5995ccea0876441)) MUST be appended.
	* The **snCoded** variable's value MUST be set to the previously transmitted datagram's **snCoded** value plus 1. If this is the first datagram, this value is the advertised **Initial Sequence Number** ADM element.
	* The **snSourceStart** variable MUST be set to the Source sequence number of the first datagram included in this FEC operation.
	* The **uRange** variable MUST be set to the number of datagrams included in this FEC operation.
	* The **uPadding** variable MUST be set to zero and ignored by the receiver.
3. The FEC payload data MUST be appended.

#### Connection Sequence

The protocol's connection sequence is illustrated in the figure in section [3.1.5](#Section_575660d7a69848de92dbd4d4c9fcc783). The following list describes the states that the terminal server and terminal client enter:

1. Listen/- : The terminal server enters the Listen state:
	1. The terminal server binds to a UDP socket, and is ready to accept incoming connections.
2. Connect/SYN-:
	1. The terminal client establishes a **UDP** socket connection with the terminal server.
	2. The terminal client constructs and sends a SYN datagram, as specified in section [3.1.5.1.1](#Section_066f9acffd574f95ab3f334e748bab10).
3. SYN/SYN+ACK:
	1. The terminal server receives the SYN datagram.
	2. The terminal server constructs and sends a SYN+ACK datagram, as specified in section [3.1.5.1.3](#Section_96eaa81aff4240a2884c96b3834db6c8).
4. SYN+ACK/ACK(+DATA):
	1. The terminal client receives a SYN+ACK datagram. If the terminal client does not receive a response for a SYN datagram that was retransmitted at least three and no more than five times, the endpoint will enter the Closed state.[<5>](#Appendix_A_5" \o "Product behavior note 5)
	2. The terminal client generates an ACK for the SYN+ACK datagram.
	3. The terminal client can append Source Packets to the ACK datagram.
5. ACK/-:
	1. The server receives an ACK for the SYN+ACK datagram sent. If the terminal server does not receive a response for a SYN + ACK datagram that was retransmitted at least three and no more than five times, the endpoint will enter the Closed state.[<6>](#Appendix_A_6" \o "Product behavior note 6)
	2. The server enters the Established state.

#### Data Transfer Phase

##### Sender Receives Data

Each Source Packet is identified by a unique Source sequence number, as specified in section [3.1.1.2](#Section_5fd219caa9194305a4c112b4f84cb484). The sender assigns a Source sequence number to each datagram. This number is increased by one for each datagram. The initial value is the **Initial Sequence Number** advertised by the Sender.

The size of the data a user can write to the sender is limited to the negotiated MTU for the RDP-UDP transport, obtained through the MTU negotiation process, as specified in section [3.1.1.3](#Section_ffbc66efe79f49f7b20876d4a68339e8).

An RDP-UDP-R sender (section [1.3.1](#Section_aea14a52baa14486bcd8f30505ec707d)) is similar to the TCP protocol, and operates like a stream-based transport. Data of any arbitrary size can be handed to the RDP-UDP-R sender. The sender fragments this block of data into MTU-sized chunks before transmitting it.

An RDP-UDP-L sender (section 1.3.1) is similar to the UDP protocol, and operates like a pure datagram-based transport. Each block of data the RDP-UDP-L sender can send is no more than the MTU size negotiated in section 3.1.1.3. Blocks of data larger than the negotiated MTU are not transferred by this protocol.

##### Sender Sends Data

Each Coded Packet is identified by a Coded sequence number, as specified in section [3.1.1.2](#Section_5fd219caa9194305a4c112b4f84cb484). The sender MUST implement a form of **Congestion Control**, and generate applicable messages, as specified in section [3.1.1.8](#Section_dee0ec22a3d54b22909b6bcd2e6587e8).

###### Source Packet

A Source Packet is generated as specified in section [3.1.5.1.4](#Section_427c5d296b084cdbbbdfa1ed09e76e2d). A Source Packet is sent only if one of the following occurs:

* A datagram has been marked as a lost datagram (section [3.1.1.4.1](#Section_b7c69b866ef241fd946eab2bf5c05c04)), and it has not been retransmitted.
* There is space in the receiver-advertised window for this datagram and the **Congestion Control** logic permits transmission of a datagram.

###### FEC Packet

An FEC Packet is generated, as specified in section [3.1.5.1.5](#Section_7eaacd177012468faa006e629cb88df8). An FEC Packet is generated when the sender has sent one or more data packets and the receiver has not acknowledged one or more of these data packets.

##### Receiver Receives Data

The receiver MUST accept all of the datagrams with Source sequence numbers (section [3.1.1.2](#Section_5fd219caa9194305a4c112b4f84cb484)) that fall within the range of the receiver-advertised window. All other datagrams MUST be ignored and discarded. If the datagram has already been received, the received datagram is a duplicate, and MUST be ignored. Acknowledgments (section [3.1.1.4](#Section_ae448afed83f479792e1a8596456ebff)) are generated for datagrams that were not discarded by the receiver.

The receiver MUST generate an acknowledgment for received Source Packets, as specified in section [3.1.5.1.2](#Section_facb0b3163c644f4aeec03b5163aedae). The receiver MUST generate **Congestion Notification** messages, as specified in section [3.1.1.8](#Section_dee0ec22a3d54b22909b6bcd2e6587e8).

##### User Consumes Data

The receiver-advertised window MUST increase by 1 for every datagram read by the user from the receiver.

#### Termination

##### Retransmit Limit

If a datagram has been retransmitted at least three and no more than five times without a response, the sender terminates the connection. The endpoint is terminated and enters the Closed state.[<7>](#Appendix_A_7" \o "Product behavior note 7)

##### Keepalive Timer Fires

If the sender does not receive any ACK from the receiver after 65 seconds, the connection is terminated and the endpoint enters the Closed state.

### Timer Events

#### Retransmit Timer

This timer fires if no acknowledgment (section [3.1.1.4](#Section_ae448afed83f479792e1a8596456ebff)) has been received for a datagram that was transmitted earlier. This timer MUST fire at the minimum retransmit time-out or twice the RTT, whichever is longer, after the datagram is first transmitted. The minimum retransmit time-out depends on the negotiated protocol version (section 3.1.5.1) as follows:

* RDPUDP\_PROTOCOL\_VERSION\_1: the minimum retransmit time-out is 500 ms.
* RDPUDP\_PROTOCOL\_VERSION\_2: the minimum retransmit time-out is 300 ms.

When a datagram is scheduled for retransmission, a Source Packet is generated, as specified in section [3.1.5.1.4](#Section_427c5d296b084cdbbbdfa1ed09e76e2d). The timer MUST continue to fire with a time-out of at least the same length for multiple retransmissions of the same datagram.[<8>](#Appendix_A_8" \o "Product behavior note 8) If the same datagram has already been retransmitted at least three and no more than five times, the endpoints move to the Closed state, and the connection is terminated.

#### Keepalive Timer on the Sender

This timer fires when the sender has not received any datagram from the receiver within 65 seconds, as specified in section [3.1.1.9](#Section_16c136b27d9641f9b52b4a6a5a881457). This indicates that the receiver is no longer present or has disconnected. The upper layers are notified of this event, the endpoints move to the Closed state, and the connection is terminated.

#### Delayed ACK Timer

This timer fires on the receiver at the delayed ACK time-out after the receipt of a Source Packet if no acknowledgment (section [3.1.1.4](#Section_ae448afed83f479792e1a8596456ebff)) has been scheduled for that Source Packet. The delayed ACK time-out depends on the negotiated protocol version (section [3.1.5.1](#Section_5af070be92774196b57006e61ce81b43)) as follows:

* RDPUDP\_PROTOCOL\_VERSION\_1: the delayed ACK time-out is 200 ms.
* RDPUDP\_PROTOCOL\_VERSION\_2: the delayed ACK time-out is 50 ms or half the RTT, whichever is longer, up to a maximum of 200 ms.

Once the timer is fired, an acknowledgment for that Source Packet MUST be generated and sent. The receiver MUST set the RDPUDP\_FLAG\_ACKDELAYED flag in the **uFlags** field of the RDPUDP\_FEC\_HEADER structure.

This timer is needed only when the receiver generates one cumulative acknowledgment for a number of Source Packets, as specified in section 3.1.1.4. In this case, this timer indicates that there is at least one Source Packet at the receiver for which an acknowledgment has not been generated and sent.

### Other Local Events

None.

# Protocol Examples

## UDP Connection Initialization Packets

The following sections describe examples for packets that are created during the UDP Connection Initialization (section [1.3.2.1](#Section_8275cf8a6b98497fb8b7a8d015888b89)) phase.

For readability, the network captures headers have been divided with the "/" delimiter and additional information is provided in the field and value tables.

### SYN Packet

This packet is used in the reliable, best-effort mode, as described in section [1.3.1](#Section_aea14a52baa14486bcd8f30505ec707d). The following is an example of a network capture of a SYN packet as described in section [3.1.5.1.1](#Section_066f9acffd574f95ab3f334e748bab10).

1. ff ff ff ff 04 00 0A 01 00 00 00 42 04 D0 04 D0 00 00 00
2. D2 35 AC 43 89 41 42 DA B1 0E DD 68 87 F7 F9 FB

The following table describes the fields and values for each header structure.

| Field | Value |
| --- | --- |
| RDPUDP\_FEC\_HEADER | ff ff ff ff 04 00 0A 01 |
| snSourceAck | 0xff ff ff ff |
| uReceiveWindowSize | 0x04 00 = 1024 (decimal) |
| uFlags | 0x0A 01 = RDPUDP\_FLAG\_CORRELATION\_ID | RDPUDP\_FLAG\_SYNLOSSY | RDPUDP\_FLAG\_SYN |
| RDPUDP\_SYNDATA\_PAYLOAD | 00 00 00 42 04 D0 04 D0 |
| snInitialSequenceNumber | 0x00 00 00 42 |
| uUpStreamMtu | 0x04 D0 = 1232 (decimal) |
| uDownStreamMtu | 0x04 D0 = 1232 (decimal) |
| RDPUDP\_CORRELATION\_ID\_PAYLOAD | 0xD2 35 AC 43 89 41 42 DA B1 0E DD 68 87 F7 F9 FB0x00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |
| uCorrelationId | 0xD2 35 AC 43 89 41 42 DA B1 0E DD 68 87 F7 F9 FB |
| uReserved | 0x00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |
|  | 00 00 00 (zero padded to 1232 bytes) |

### SYN and ACK Packet

The following is an example of a network capture of a SYN and ACK packet as described in section [3.1.5.1.3](#Section_96eaa81aff4240a2884c96b3834db6c8).

1. 00 00 00 42 04 00 00 05 00 00 00 42 04 D0 04 D0 00 00 00

The following table describes the fields and values for each header structure.

| Field | Value |
| --- | --- |
| RDPUDP\_FEC\_HEADER | 00 00 00 42 04 00 02 01 |
| snSourceAck | 0x00 00 00 42 |
| uReceiveWindowSize | 0x04 00 = 1024 (decimal) |
| uFlags | 0x 00 05 = RDPUDP\_FLAG\_SYN | RDPUDP\_FLAG\_ACK |
| RDPUDP\_SYNDATA\_PAYLOAD | 00 00 00 42 04 D0 04 D0 |
| snInitialSequenceNumber | 0x00 00 00 42 |
| uUpStreamMtu | 0x04 D0 = 1232 (decimal) |
| uDownStreamMtu | 0x04 D0 = 1232 (decimal) |
|  | 00 00 00 (zero padded to 1232 bytes) |

## UDP Data Transfer Packets

The following sections describe examples for packets that are created during the section UDP Data Transfer (section [1.3.2.2](#Section_d6dc8925ca524045845195a0f58a870e)) phase.

For readability, the network captures headers have been divided with the "/" delimiter and additional information is provided in the field and value tables.

### Source Packet

The following is an example of a network capture of a Source Packet, as described in section [3.1.5.3.2.1](#Section_d2fccbe736c94c8db8f4d0532e9699d6).

1. d6 cf 0a b8 04 00 00 0c 00 01 04 00 ec 47 1a e4 ec 47 1a e4 17 03 03 00 40 bb…

The following table describes the fields and values for each header structure.

| Field | Value |
| --- | --- |
| RDPUDP\_FEC\_HEADER | d6 cf 0a b8 04 00 00 0c |
| snSourceAck | 0xd6 cf 0a b8 = -691074376 (decimal) |
| uReceiveWindowSize | 0x0400 = 1024 (decimal) |
| uFlags | 0x000c = RDPUDP\_FLAG\_DATA | RDPUDP\_FLAG\_ACK  |
| Ack Vector | 04 00 |
| Size | 0x00 01 = 1 |
| Element 1 | 0x04 |
| State | 0x0 (2 bits) DATAGRAM\_RECEIVED |
| State | 0x04 length of the vector, 4 datagrams received |
| RDPUDP\_SOURCE\_PAYLOAD\_HEADER | ec 47 1a e4 ec 47 1a e4 |
| snCoded | 0xec 47 1a e4 = -330884380 |
| snSourceStart | 0xec 47 1a e4 = -330884380 |
| Payload data | 17 03 03 00 40 bb … |

### FEC Packet

The following is an example of a network capture of an FEC Packet, as described in section [3.1.5.3.2.2](#Section_e5c6e44b0c9041ba870b41804756d9bc).

1. d6 cf 0a cb 04 00 00 1c 00 01 04 00 ec 47 1a fd ec 47 1a fd 10 01 00 00 40 25 04 f1 …

The following table describes the fields and values for each header structure.

| Field | Value |
| --- | --- |
| RDPUDP\_FEC\_HEADER | d6 cf 0a b8 04 00 00 0c |
| snSourceAck | 0xd6 cf 0a b8 = -691074376 (decimal) |
| uReceiveWindowSize | 0x0400 = 1024 (decimal) |
| uFlags | 0x001c= 0x0010 | 0x0008 | 0x0004= RDPUDP\_FLAG\_FEC | RDPUDP\_FLAG\_DATA | RDPUDP\_FLAG\_ACK  |
| Ack Vector | 04 00 |
| Size | 0x00 01 = 1 |
| Element 1 | 0x04 |
| State | 0x0 (2 bits) DATAGRAM\_RECEIVED |
| State | 0x04 length of the vector, 4 datagrams received |
| RDPUDP\_FEC\_PAYLOAD\_HEADER | ec 47 1a fd ec 47 1a fd 10 01 00 00 |
| snCoded | 0xec 47 1a e4 = -330884380 |
| snSourceStart | 0xec 47 1a e4 = -330884380 |
| uRange | 0x10 = 16 |
| uFecIndex | 0x01 = 1 |
| uPadding | 0x0000 |
| Payload data | 40 25 04 f1 … |

#### Payload of an FEC Packet

The following is an example of an FEC Packet network payload.

| Sequence number | Size | Value |
| --- | --- | --- |
| RDP Payload S1 | 10 | 155 110 240 230 64 115 74 226 112 181 |
| RDP Payload S2 | 20 | 72 219 238 65 213 222 36 36 219 1 93 208 17 236 52 194 21 152 76 98 |
| RDP Payload S3 | 15 | 186 87 66 43 163 21 224 11 17 221 148 13 249 159 32 |
| RDP Payload S4 | 15 | 53 90 48 146 171 205 146 119 29 94 118 76 94 154 255 |
| RDP Payload S5 | 20 | 53 83 233 201 242 15 30 42 14 61 77 183 89 190 220 10 153 148 221 195 |
| FEC Payload |  | 0 66 208 168 239 37 29 238 180 193 24 58 66 252 233 126 172 211 135 31 206 27 |

The following are FEC encoding internals; these packets are not transferred on the wire:

* CoEff Array [0 254 230 253 205]
* RDPUDP\_FEC\_PAYLOAD\_HEADER:: uFecIndex = 0
* RDPUDP\_FEC\_PAYLOAD\_HEADER:: snSourceStart = 1
* RDPUDP\_FEC\_PAYLOAD\_HEADER:: uRange = 5
* If RDP Payload S3 is lost, it will be recovered as

0 15 186 87 66 43 163 21 224 11 17 221 148 13 249 159 32 0 0 0 0 0

The first 2 bytes (0, 15) form the RDPUDP\_PAYLOAD\_PREFIX header (section [2.2.2.3](#Section_3eddc65b0e314ae8a6dc2024082ba427)), which gives the length of packet S3.

### ACK Packet

The following is an example of a network capture of an ACK Packet, with the option ACK of ACKS, as described in section [3.1.5.1.2](#Section_facb0b3163c644f4aeec03b5163aedae).

1. d6 cf 0a b8 04 00 01 0c 00 01 04 00 d6 cf 0a b8 ec 47 1a e4 ec 47 1a e4 17 03 03 00

The following table describes the fields and values for each header structure.

| Field | Value |
| --- | --- |
| RDPUDP\_FEC\_HEADER | d6 cf 0a b8 04 00 01 0c |
| snSourceAck | 0xd6 cf 0a b8 = -691074376 (decimal) |
| uReceiveWindowSize | 0x0400 = 1024 (decimal) |
| uFlags | 0x010c= 0x0100 | 0x0008 | 0x0004= RDPUDP\_FLAG\_ACK\_OF\_ACKS | RDPUDP\_FLAG\_DATA | RDPUDP\_FLAG\_ACK |
| Ack Vector | 04 00 |
| Size | 0x00 01 = 1 |
| Element 1 | 0x04 |
| State | 0x0 (2 bits) DATAGRAM\_RECEIVED |
| State | 0x04 length of the vector, 4 datagrams received |
| Ack of Acks | d6 cf 0a b8 |
| RDPUDP\_SOURCE\_PAYLOAD\_HEADER | ec 47 1a e4 ec 47 1a e4 |
| snCoded | 0xec 47 1a e4 = -330884380 |
| snSourceStart | 0xec 47 1a e4 = -330884380 |
| Payload data | 17 03 03 00 … |

# Security

## Security Considerations for Implementers

The Remote Desktop Protocol: UDP Transport Extension Protocol shares a number of security considerations with the TCP protocol. The following sections describe these security considerations.

### Using Sequence Numbers

The two communicating endpoints exchange the range of sequence numbers they will be generating and/or are willing to accept through the **Initial Sequence Number** and acknowledgments (section [3.1.1.4](#Section_ae448afed83f479792e1a8596456ebff)). All of the datagrams that arrive at the receiver with sequence numbers that fall outside the advertised window are considered malicious, and are not processed.

Similarly, the sender maintains a range of sequence numbers that are valid and can be acknowledged. All of the acknowledgments with sequence numbers that fall outside this range are ignored. These datagrams can be a consequence of packet reordering or packet duplication in the network and do not result in a connection termination.

### RDP-UDP Datagram Validation

All headers require validation. The size of the headers and data payload in the datagram tally with the size of the UDP datagram and is within the ranges specified by the sender.

When decoding ACK vectors (section [2.2.3.1](#Section_2d5bfb5ec5fe4a9bb9addbafae206948)), some state changes are considered illegal. For example, a datagram that has been marked as received cannot arrive with the state unknown in the subsequent datagrams. Such acknowledgments can be ignored, as they can either be delayed or invalid.

### Congestion Notifications

The receiver generates congestion notifications for lost datagrams. The sender reduces the rate at which data is written to the wire. Failure to do so increases congestion on the network, and drives the network towards congestion collapse, which impacts all users.

## Index of Security Parameters

None.

# Appendix A: Product Behavior

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

* Windows 8 operating system
* Windows Server 2012 operating system
* Windows 8.1 operating system
* Windows Server 2012 R2 operating system
* Windows 10 operating system
* Windows Server 2016 operating system

Exceptions, if any, are noted below. If a service pack or Quick Fix Engineering (QFE) number appears with the product version, behavior changed in that service pack or QFE. The new behavior also applies to subsequent service packs of the product unless otherwise specified. If a product edition appears with the product version, behavior is different in that product edition.

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

[<1> Section 2.2.2.9](#Appendix_A_Target_1): Windows 8 and Windows Server 2012 support version 1 of the RDP-UDP protocol.

[<2> Section 2.2.2.9](#Appendix_A_Target_2): Windows 8.1 and Windows Server 2012 R2 support version 1 and version 2 of the RDP-UDP protocol, and are the first product versions to send the RDPUDP\_SYNDATAEX\_PAYLOAD structure.

[<3> Section 3.1.1.4](#Appendix_A_Target_3): The Remote Desktop Protocol: UDP Transport Extension generates one ACK for every two Source Packets received from the sender.

[<4> Section 3.1.1.9](#Appendix_A_Target_4): The Remote Desktop Protocol: UDP Transport Extension generates four keep-alive datagrams every 65 seconds when the transport is quiescent.

[<5> Section 3.1.5.2](#Appendix_A_Target_5): The Remote Desktop Protocol: UDP Transport Extension retransmits SYN and SYN+ACK packets three times, with a time-out of 800 ms, before terminating the connection.

[<6> Section 3.1.5.2](#Appendix_A_Target_6): The Remote Desktop Protocol: UDP Transport Extension retransmits SYN and SYN+ACK packets three times, with a time-out of 800 ms, before terminating the connection.

[<7> Section 3.1.5.4.1](#Appendix_A_Target_7): The Remote Desktop Protocol: UDP Transport Extension retransmits datagrams five times before terminating the connection.

[<8> Section 3.1.6.1](#Appendix_A_Target_8): The Remote Desktop Protocol: UDP Transport Extension doubles the retransmit time-out each time the same packet is retransmitted, up to 120 seconds.

# Change Tracking

No table of changes is available. The document is either new or has had no changes since its last release.

# Index

A

Abstract data model

 [client](#section_857bae4ea5224d00a0f0aea98bbbb961) 18

 [server](#section_857bae4ea5224d00a0f0aea98bbbb961) 18

[Applicability](#section_6b958fe385ac4cbdb68fc44c5ed51040) 10

C

[Capability negotiation](#section_965fbceb72c84524934acf182c554dd5) 10

[Change tracking](#section_21b050dda6064c22b34e846ed57ce68a) 43

Client

 [abstract data model](#section_857bae4ea5224d00a0f0aea98bbbb961) 18

 higher-layer triggered events

 [initializing connection](#section_ff305355dd3c40d68c2d973a33e032bd) 28

 [receiving datagram](#section_7037ca0b5285467e9ec1758ffcc1d449) 28

 [sending datagram](#section_0d25f1c682774018b076603a3c02e1cc) 28

 [terminating connection](#section_f4e45f3667834826a128c018db310238) 28

 [initialization](#section_85cc555bd749476da2cfde1e95bc0c42) 27

 [message processing](#section_575660d7a69848de92dbd4d4c9fcc783) 28

 [sequencing rules](#section_575660d7a69848de92dbd4d4c9fcc783) 28

 timer events

 [Delayed ACK timer](#section_8c80ed0f0164479db94e23a08014941b) 35

 [Keepalive timer on sender](#section_997909d6d4564e5b8c69d8c9667934e3) 35

 [Retransmit timer](#section_78e889c926c34ffe8c29e9cb4ed34345) 34

 [timers](#section_a39e93e6851d4ec6b200782b34d4e77f) 27

D

Data model - abstract

 [client](#section_857bae4ea5224d00a0f0aea98bbbb961) 18

 [server](#section_857bae4ea5224d00a0f0aea98bbbb961) 18

F

[Fields - vendor-extensible](#section_e7360d44bb4c4c8b8eee02043e9c5e2d) 10

G

[Glossary](#section_5a0a391a4ff143c6b7d9fb08075492f4) 5

H

Higher-layer triggered events

 client

 [initializing connection](#section_ff305355dd3c40d68c2d973a33e032bd) 28

 [receiving datagram](#section_7037ca0b5285467e9ec1758ffcc1d449) 28

 [sending datagram](#section_0d25f1c682774018b076603a3c02e1cc) 28

 [terminating connection](#section_f4e45f3667834826a128c018db310238) 28

 server

 [initializing connection](#section_ff305355dd3c40d68c2d973a33e032bd) 28

 [receiving datagram](#section_7037ca0b5285467e9ec1758ffcc1d449) 28

 [sending datagram](#section_0d25f1c682774018b076603a3c02e1cc) 28

 [terminating connection](#section_f4e45f3667834826a128c018db310238) 28

I

[Implementer - security considerations](#section_3163f9edb0094bb78a30d6cf3f36f6e3) 41

[Index of security parameters](#section_1552a0be95f04b5f8674e6f90e180303) 41

[Informative references](#section_521c4128ea5342fcbb809906dc75d45b) 6

Initialization

 [client](#section_85cc555bd749476da2cfde1e95bc0c42) 27

 [server](#section_85cc555bd749476da2cfde1e95bc0c42) 27

[Introduction](#section_1ed440f46e8f4c79a9a30aea32a21daf) 5

M

Message processing

 [client](#section_575660d7a69848de92dbd4d4c9fcc783) 28

 [server](#section_575660d7a69848de92dbd4d4c9fcc783) 28

Messages

 [syntax](#section_fa40443d6ea645a8b441fa020bbdfea1) 11

 [transport](#section_efafc7a931af43d8a8304b80044797d8) 11

N

[Normative references](#section_1a87147bf068457990b3578d8c9b06b6) 6

O

[Overview (synopsis)](#section_fe211d9792dd47e68fa3b23f2c1a5af9) 7

P

[Parameters - security index](#section_1552a0be95f04b5f8674e6f90e180303) 41

[Preconditions](#section_fb1a250c74e54f2b9fc07cc3a2369a81) 10

[Prerequisites](#section_fb1a250c74e54f2b9fc07cc3a2369a81) 10

[Product behavior](#section_cb93d5bf25d14780b58b54c5579902d3) 42

R

[References](#section_254b7b10099c4b56b764465904392d81) 6

 [informative](#section_521c4128ea5342fcbb809906dc75d45b) 6

 [normative](#section_1a87147bf068457990b3578d8c9b06b6) 6

[Relationship to other protocols](#section_cfac35e57ada4c6b997ff77b1bbec91d) 9

S

Security

 [implementer considerations](#section_3163f9edb0094bb78a30d6cf3f36f6e3) 41

 [parameter index](#section_1552a0be95f04b5f8674e6f90e180303) 41

Sequencing rules

 [client](#section_575660d7a69848de92dbd4d4c9fcc783) 28

 [server](#section_575660d7a69848de92dbd4d4c9fcc783) 28

Server

 [abstract data model](#section_857bae4ea5224d00a0f0aea98bbbb961) 18

 higher-layer triggered events

 [initializing connection](#section_ff305355dd3c40d68c2d973a33e032bd) 28

 [receiving datagram](#section_7037ca0b5285467e9ec1758ffcc1d449) 28

 [sending datagram](#section_0d25f1c682774018b076603a3c02e1cc) 28

 [terminating connection](#section_f4e45f3667834826a128c018db310238) 28

 [initialization](#section_85cc555bd749476da2cfde1e95bc0c42) 27

 [message processing](#section_575660d7a69848de92dbd4d4c9fcc783) 28

 [sequencing rules](#section_575660d7a69848de92dbd4d4c9fcc783) 28

 timer events

 [Delayed ACK timer](#section_8c80ed0f0164479db94e23a08014941b) 35

 [Keepalive timer on sender](#section_997909d6d4564e5b8c69d8c9667934e3) 35

 [Retransmit timer](#section_78e889c926c34ffe8c29e9cb4ed34345) 34

 [timers](#section_a39e93e6851d4ec6b200782b34d4e77f) 27

[Standards assignments](#section_63f33e16d12e41fb9f2cc9bbf9fe8c36) 10

[Syntax](#section_fa40443d6ea645a8b441fa020bbdfea1) 11

T

Timer events

 client

 [Delayed ACK timer](#section_8c80ed0f0164479db94e23a08014941b) 35

 [Keepalive timer on sender](#section_997909d6d4564e5b8c69d8c9667934e3) 35

 [Retransmit timer](#section_78e889c926c34ffe8c29e9cb4ed34345) 34

 server

 [Delayed ACK timer](#section_8c80ed0f0164479db94e23a08014941b) 35

 [Keepalive timer on sender](#section_997909d6d4564e5b8c69d8c9667934e3) 35

 [Retransmit timer](#section_78e889c926c34ffe8c29e9cb4ed34345) 34

Timers

 [client](#section_a39e93e6851d4ec6b200782b34d4e77f) 27

 [server](#section_a39e93e6851d4ec6b200782b34d4e77f) 27

[Tracking changes](#section_21b050dda6064c22b34e846ed57ce68a) 43

[Transport](#section_efafc7a931af43d8a8304b80044797d8) 11

Triggered events

 client

 [initializing connection](#section_ff305355dd3c40d68c2d973a33e032bd) 28

 [receiving datagram](#section_7037ca0b5285467e9ec1758ffcc1d449) 28

 [sending datagram](#section_0d25f1c682774018b076603a3c02e1cc) 28

 [terminating connection](#section_f4e45f3667834826a128c018db310238) 28

 server

 [initializing connection](#section_ff305355dd3c40d68c2d973a33e032bd) 28

 [receiving datagram](#section_7037ca0b5285467e9ec1758ffcc1d449) 28

 [sending datagram](#section_0d25f1c682774018b076603a3c02e1cc) 28

 [terminating connection](#section_f4e45f3667834826a128c018db310238) 28

V

[Vendor-extensible fields](#section_e7360d44bb4c4c8b8eee02043e9c5e2d) 10

[Versioning](#section_965fbceb72c84524934acf182c554dd5) 10