Network Working Group
Internet-Draft
Intended status: Informational
Expires: January 11, 2014

M. Thornburgh
Adobe
July 10, 2013

Adobe's Secure Real-Time Media Flow Protocol
draft-thornburgh-adobe-rtmfp-10

Abstract

This memo describes Adobe's Secure Real-Time Media Flow Protocol (RTMFP), an endpoint-to-endpoint communication protocol designed to securely transport parallel flows of real-time video, audio, and data messages, as well as bulk data, over IP networks. RTMFP has features making it effective for peer-to-peer (P2P) as well as client-server communications, even when Network Address Translators (NATs) are used.

Status of this Memo

Copyright Notice

Table of Contents

1. Introduction

Adobe's Secure Real-Time Media Flow Protocol (RTMFP) is intended for use as a general purpose endpoint-to-endpoint data transport service in IP networks. It has features that make it well suited to the transport of real-time media (such as low-delay video, audio, and data) as well as bulk data, and for client-server as well as peer-to-peer (P2P) communication. These features include independent parallel message flows which may have different delivery priorities, variable message reliability (from TCP-like full reliability to UDP-like best effort), multi-point congestion control, and built-in security. Session multiplexing and facilities to support UDP hole-punching simplify Network Address Translator (NAT) traversal in peer-to-peer systems.

RTMFP is implemented in Flash Player, Adobe Integrated Runtime (AIR), and Adobe Media Server (AMS, formerly Flash Media Server or FMS), all from Adobe Systems Incorporated, and is used as the foundation transport protocol for real-time video, audio, and data communication, both client-server and P2P, in those products. At the time of writing, the Adobe Flash Player runtime is installed on more than one billion end-user desktop computers.

RTMFP was developed by Adobe Systems Incorporated, and is not the product of an IETF activity.

This memo describes the syntax and operation of the Secure Real-Time Media Flow Protocol.

This memo describes a general security framework that, when combined with an application-specific Cryptography Profile, can be used to establish a confidential and authenticated session between endpoints. The application-specific Cryptography Profile, not defined herein, would detail the specific cryptographic algorithms, data formats, and semantics to be used within this framework. Interoperation between applications of RTMFP requires common or compatible Cryptography Profiles.

Note to implementers: at the time of writing, the Cryptography Profile used by the above mentioned Adobe products is not publicly described by Adobe. Implementers should investigate the availability of documentation of that Cryptography Profile prior to implementing RTMFP for the purpose of interoperation with the above mentioned Adobe products.

1.1. Design Highlights of RTMFP

Between any pair of communicating endpoints is a single, bidirectional, secured, congestion controlled session. Unidirectional flows convey messages from one end to the other within the session. An endpoint can have concurrent sessions with multiple other far endpoints.

Design highlights of RTMFP include:

- The security framework is an inherent part of the basic protocol. The application designer chooses the cryptographic formats and algorithms to suit the needs of the application, and may update them as the state of the security arts progresses.

- Cryptographic Endpoint Discriminators can resist port scanning.

- All header, control, and framing information, except for network addressing information and a session identifier, is encrypted according to the Cryptography Profile.

- There is a single session and associated congestion control state between a pair of endpoints.

- Each session may have zero or more unidirectional message-oriented flows in each direction. All of a session's sending flows share the session's congestion control state.

- Return Flow Association (Section 2.3.11.1.2) generalizes bidirectional communication to arbitrarily complex trees of flows.

- Messages in flows can be arbitrarily large and are fragmented for transmission.

- Messages of any size may be sent with full, partial, or no reliability (sender's choice). Messages may be delivered to the receiving user in original queuing order or network arrival order (receiver's choice).

- Flows are named with arbitrary, user-defined metadata (Section 2.3.11.1.1) rather than port or stream numbers.

- The sequence numbers of each flow are independent of all other flows, and are not permanently bound to a session-wide transmission ordering. This allows real-time priority decisions to be made at transmission or retransmission time.

- Each flow has its own receive window, and therefore independent flow control.

- Round-trips are expensive, and are minimized or eliminated when possible.

- After a session is established, flows begin by sending the flow's messages with no additional handshake (and associated round-trips).

- Transmitting bytes on the network is much more expensive than moving bytes in a CPU or memory. Wasted bytes are minimized or eliminated when possible and practical, and variable length encodings are used, even at the expense of breaking 32-bit alignment and making the text diagrams in this specification look awkward.

- P2P lookup and peer introduction (including UDP hole punching for NAT and firewall traversal) is supported directly by the session startup handshake.

- Session identifiers allow an endpoint to multiplex many sessions over a single local transport address while allowing sessions to survive changes in transport address (as may happen in mobile or wireless deployments).

The syntax of the protocol is detailed in Section 2. The operation of the protocol is detailed in Section 3.

1.2. Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC2119].

2. Syntax

Definitions of types and structures in this specification use traditional text diagrams paired with procedural descriptions using a C-like syntax. The C-like procedural descriptions SHALL be construed as definitive.

Structures are packed to take only as many bytes as explicitly indicated. There is no 32-bit alignment constraint, and fields are not padded for alignment unless explicitly indicated or described. Text diagrams may include a bit ruler across the top; this is a

convenience for counting bits in individual fields and does not necessarily imply field alignment on a multiple of the ruler width.

Unless specified otherwise, reserved fields SHOULD be set to 0 by a sender and MUST be ignored by a receiver.

The procedural syntax of this specification defines correct and error-free encoded inputs to a parser. The procedural syntax does not describe a fully featured parser, including error detection and handling. Implementations MUST include means to identify error circumstances, including truncations causing elementary or composed types to not fit inside containing structures, fields or elements. Unless specified otherwise, an error circumstance SHALL abort the parsing and processing of an element and its enclosing elements, up to the containing packet.

2.1. Common Elements

This section lists types and structures that are used throughout this specification.

2.1.1. Elementary Types and Constructs

This section lists the elementary types and constructs out of which all of the following sections' definitions are built.

```
uint8_t var;
```

An unsigned integer 8 bits (one byte) in length and byte aligned.

```
uint16_t var;
```

An unsigned integer 16 bits in length, in network byte order ("big endian") and byte aligned.

```
uint32_t var;
```

An unsigned integer 32 bits in length, in network byte order and byte aligned.

```
uint128_t var;
```

An unsigned integer 128 bits in length, in network byte order and byte aligned.

```
uintn_t var :bitsize;
```

An unsigned integer of any other size, potentially not byte aligned.

Its size in bits is specified explicitly by bitsize.

bool_t var :1;

A boolean flag having the value true (1 or set) or false (0 or clear) and being one bit in length.

type var[num];

A packed array of type with length num*sizeof(type)*8 bits.

struct name_t { ... } name :bitsize;

A packed structure. Its size in bits is specified by bitsize.

remainder();

The number of bytes from the current offset to the end of the enclosing structure.

type var[remainder()];

A packed array of type, its size extending to the end of the enclosing structure.

Note that a bitsize of "variable" indicates that the size of the structure is determined by the sizes of its interior components. A bitsize of "n*8" indicates that the size of the structure is a whole number of bytes and is byte aligned.

2.1.2. Variable Length Unsigned Integer (VLU)

A VLU encodes any finite non-negative integer into one or more bytes. For each encoded byte, if the high bit is set, the next byte is also part of the VLU. If the high bit is clear, this is the final byte of the VLU. The remaining bits encode the number, seven bits at a time, from most significant to least significant.

```
    0 1 2 3 4 5 6 7                 0 1 2 3 4 5 6 7
   +~+~+~+~+~+~+~+~+               +-+-+-+-+-+-+-+-+
   |1|    digit    |...............|0|    digit    |
   +~+~+~+~+~+~+~+~+               +-+-+-+-+-+-+-+-+
   ^                               ^
   +--------- zero or more --------+

   struct vlu_t
   {
       value = 0;
       do {
           bool_t  more  :1;
           uintn_t digit :7;
           value = (value * 128) + digit;
       } while(more);
   } :variable*8;


                             +-------------/-+
                             |             \ |
                             +-------------/-+
```

Figure 1: VLU depiction in following diagrams

Unless stated otherwise in this specification, implementations SHOULD handle VLUs encoding unsigned integers at least 64 bits in length (that is, encoding a maximum value of at least 2^64 - 1).

2.1.3. Option

An Option is a Length-Type-Value triplet. Length and Type are encoded in VLU format. Length is the number of bytes of payload following the Length field. The payload comprises the Type and Value fields. Type identifies the kind of option this is. The syntax of the Value field is determined by the type of option.

An option can have a length of zero, in which case it has no type and no value and is empty. An empty option is called a "Marker".

```
   +-------------/-+~~~~~~~~~~~~~/~+~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~+
   |   length    \ |    type     \ |            value              |
   +-------------/-+~~~~~~~~~~~~~/~+~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~/
                   ^                                               ^
                   +-------- length bytes long (may be 0) ---------+

   struct option_t
   {
       vlu_t length :variable*8; // "L"
       if(length > 0)
       {
           struct {
               vlu_t   type :variable*8;   // "T"
               uint8_t value[remainder()]; // "V"
           } payload :length*8;
       }
   } :variable*8;


                             +---/---/-------+
                             | L \ T \   V   |
                             +---/---/-------+
```

Figure 2: Option depiction in following diagrams

2.1.4. Option List

An Option List is a sequence of zero or more non-empty Options terminated by a Marker.

```
   +~~~/~~~/~~~~~~~+               +~~~/~~~/~~~~~~~+-------------/-+
   | L \ T \   V   |...............| L \ T \   V   |       0     \ |
   +~~~/~~~/~~~~~~~+               +~~~/~~~/~~~~~~~+-------------/-+
   ^                                               ^     Marker
   +------- zero or more non-empty Options --------+ (empty Option)

   struct optionList_t
   {
       do
       {
           option_t option :variable*8;
       } while(option.length > 0);
   } :variable*8;
```

2.1.5. Internet Socket Address (Address)

When communicating an Internet Socket Address (a combination of a 32-bit IPv4 [RFC0791] or 128-bit IPv6 [RFC2460] address and a 16-bit port number) to another RTMFP, this encoding is used. This encoding additionally allows an address to be tagged with an origin type, which an RTMFP MAY use to modify the use or disposition of the address.

```
                                                     1
 0 1 2 3 4 5 6 7                  0 1 2 3 4 5 6 7|8 9 0 1 2 3 4 5
+-+-+-+-+-+-+-+-+-----/.../-----+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|I|         | O |    internet   |                               |
|P|0 0 0 0 0| R |    address    |              port             |
|6|   rsv   | I |32 or 128 bits |                               |
+-+-+-+-+-+-+-+-+-----/.../-----+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

```
struct address_t
{
    bool_t  inet6    :1;     // "IP6"
    uintn_t reserved :5 = 0; // "rsv"
    uintn_t origin   :2;     // "ORI"
    if(inet6)
        uint128_t ipAddress;
    else
        uint32_t ipAddress;
    uint16_t port;
} :variable*8;
```

inet6 : If set, the Internet address is a 128-bit IPv6 address. If clear, the Internet address is a 32-bit IPv4 address.

origin : The origin tag of this address. Possible values are:

0 : Unknown, unspecified, or "other"

1 : Address was reported by the origin as a local, directly-attached interface address

2 : Address was observed to be the source address from which a packet was received (a "reflexive transport address" in the terminology of [RFC5389])

3 : Address is a relay, proxy, or introducer (a Redirector and/or Forwarder)

ipAddress : The Internet address, in network byte order.

port : The 16 bit port number, in network byte order.

2.2. Network Layer

2.2.1. Encapsulation

RTMFP Multiplex packets are usually carried in UDP [RFC0768] datagrams so that they may transit commonly deployed NATs and firewalls, and so that RTMFP may be implemented on commonly deployed operating systems without special privileges or permissions.

RTMFP Multiplex packets MAY be carried by any suitable datagram transport or encapsulation where endpoints are addressed by an Internet socket address (that is, an IPv4 or IPv6 address and a 16-bit port number).

The choice of port numbers is not mandated by this specification. Higher protocol layers or the application define the port numbers used.

2.2.2. Multiplex

```
    0 1 2 3 4 5 6 7|0 1 2 3 4 5 6 7|0 1 2 3 4 5 6 7|0 1 2 3 4 5 6 7
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |                 scrambled session ID (SSID)                   |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |             e             first32[0]                          |
   |- - - - - -  n  - - - - - - - - - - - - - - - - - - - - - - - -|
   |             c             first32[1]                          |
   +- - - - - -  r  - - - - - - - - - - - - - - - - - - - - - - - -+
   |             y                                                 |
   |             pted packet                                       |
   +---------------------------------------------------------------/

   struct multiplex_t
   {
       uint32_t scrambledSessionID; // "SSID"
       union {
           uint32_t first32[2]; // see note
           uint8_t  encryptedPacket[remainder()];
       } :(encapsulation.length - 4)*8;

       // if encryptedPacket is less than 8 bytes long, treat it
       // as if it were end-padded with 0s for the following:
       sessionID = scrambledSessionID XOR first32[0] XOR first32[1];
   } :encapsulation.length*8;
```

The 32-bit Scrambled Session ID is the 32-bit Session ID modified by performing a bitwise exclusive-or with the bitwise exclusive-or of the first two 32-bit words of the encrypted packet.

The Session ID is a 32-bit value that the receiver has requested to be used by the sender when sending packets to this receiver (Section 2.3.7, Section 2.3.8). The Session ID identifies the Session to which this packet belongs and the decryption key to be used to decrypt the encrypted packet.

Note: Session ID 0 (prior to scrambling) denotes the startup pseudo-session and implies the Default Session Key.

Note: If the encrypted packet is less than 8 bytes long, then for the scrambling operation, perform the exclusive-or as though the encrypted packet were end-padded with enough 0-bytes to bring its length to 8.

2.2.3. Encryption

RTMFP packets are encrypted according to a Cryptography Profile. This specification doesn't define a Cryptography Profile or mandate a particular choice of cryptography. The application defines the cryptographic syntax and algorithms.

Packet encryption is RECOMMENDED to be a block cipher operating in Cipher Block Chaining [CBC] or similar mode. Encrypted packets MUST be decipherable without inter-packet dependency, since packets may be lost, duplicated, or reordered in the network.

The packet encryption layer is responsible for data integrity and authenticity of packets, for example by means of a checksum or cryptographic message authentication code. To mitigate replay attacks, data integrity SHOULD comprise duplicate packet detection, for example by means of a session-wide packet sequence number. The packet encryption layer SHALL discard a received packet that does not pass integrity or authenticity tests.

Note that the structures described below are of plain, unencrypted packets. Encrypted packets MUST be decrypted according to the Session Key associated with the Multiplex Session ID before being interpreted according to this specification.

The cryptography profile defines a well-known Default Session Key that is used at session startup, during which per-session key(s) are negotiated by the two endpoints. A Session ID of zero denotes use of the Default Session Key. The Default Session Key is also used with non-zero Session IDs during the latter phases of session startup

(Section 2.3.6, Section 2.3.8). See Security Considerations (Section 5) for more about the Default Session Key.

2.2.4. Packet

An (unencrypted, plain) RTMFP Packet consists of a variable sized common header, zero or more chunks, and padding. Padding can be inserted by the encryption layer of the sender to meet cipher block size constraints, and is ignored by the receiver. A sender's encryption layer MAY pad the end of a packet with bytes with value 0xff such that the resulting packet is a natural and appropriate size for the cipher. Alternatively, the Cryptography Profile MAY define its own framing and padding scheme, if needed, such that decrypted packets are compatible with the syntax defined in this section.

```
    0 1 2 3 4 5 6 7|0 1 2 3 4 5 6 7|0 1 2 3 4 5 6 7|0 1 2 3 4 5 6 7
   +-+-+-+-+-+-+-+-+
   |T|T| r |T|T| M |
   |C|C| s |S|S| O |
   | |R| v | |E| D |
   +-+-+-+-+-+-+-+-+
   +~+~+~+~+~+~+~+~+~+~+~+~+~+~+~+~+~+~+~+~+~+~+~+~+~+~+~+~+~+~+~+~+
   |        if(TS) timestamp       |     if(TSE) timestampEcho     |
   +~+~+~+~+~+~+~+~+~+~+~+~+~+~+~+~+~+~+~+~+~+~+~+~+~+~+~+~+~+~+~+~+
   +~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~+
   |                             Chunk                             |
   +~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~/
                                   :
                                   :
   +~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~+
   |                             Chunk                             |
   +~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~/
   +~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~+
   |                            padding                            |
   +~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~/
```

```
   struct packet_t
   {
       bool_t  timeCritical         :1; // "TC"
       bool_t  timeCriticalReverse  :1; // "TCR"
       uintn_t reserved             :2; // "rsv"
       bool_t  timestampPresent     :1; // "TS"
       bool_t  timestampEchoPresent :1; // "TSE"
       uintn_t mode                 :2; // "MOD"
       if(0 != mode)
       {
           if(timestampPresent)
               uint16_t timestamp;
           if(timestampEchoPresent)
               uint16_t timestampEcho;
           while(remainder() > 2)
           {
               uint8_t  chunkType;
               uint16_t chunkLength;
               if(remainder() < chunkLength)
                   break;
               uint8_t  chunkPayload[chunkLength];
           } // chunks
           uint8_t padding[remainder()];
       }
   } :plainPacket.length*8;
```

timeCritical : Time Critical Forward Notification. If set, indicates that this packet contains real-time user data.

timeCriticalReverse : Time Critical Reverse Notification. If set, indicates that the sender is currently receiving packets on other sessions that have the timeCritical flag set.

timestampPresent : If set, indicates that the timestamp field is present. If clear, there is no timestamp field.

timestampEchoPresent : If set, indicates that the timestamp echo field is present. If clear, there is no timestamp echo field.

mode : The mode of this packet. See below for additional discussion of packet modes. Possible values are:

0 : Forbidden value

1 : Initiator Mark

2 : Responder Mark

3 : Startup

timestamp : If the timestampPresent flag is set, this field is present and contains the low 16 bits of the sender's 250 Hz clock (4 milliseconds per tick) at transmit time. The sender's clock MAY have its origin at any time in the past.

timestampEcho : If the timestampEchoPresent flag is set, this field is present and contains the sender's estimate of what the timestamp field of a packet received from the other end would be at the time this packet was transmitted, using the method described in Section 3.5.2.2.

chunks : Zero or more chunks follow the header. It is RECOMMENDED that a packet contain at least one chunk.

padding : Zero or more bytes of padding follow the chunks. The following conditions indicate padding:

- Fewer than three bytes (the size of a chunk header) remain in the packet.

- The chunkLength field of what would be the current chunk header indicates that the hypothetical chunk payload wouldn't fit in the remaining bytes of the packet.

Packet mode 0 is not allowed. Packets marked with this mode are invalid and MUST be discarded.

The original initiator of a session MUST mark all non-startup packets it sends in that session with packet mode 1 "Initiator Mark". It SHOULD ignore any packet received in that session with packet mode 1.

The original responder of a session MUST mark all non-startup packets it sends in that session with packet mode 2 "Responder Mark". It SHOULD ignore any packet received in that session with packet mode 2.

Packet mode 3 is for session startup. Session startup chunks are only allowed in packets with this mode.

Chunks that are not for session startup are only allowed in packets with modes 1 or 2.

2.3. Chunks

```
 0 1 2 3 4 5 6 7|0 1 2 3 4 5 6 7|0 1 2 3 4 5 6 7|0 1 2 3 4 5 6 7
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|   chunkType   |          chunkLength          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
+~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~+
|        chunkPayload (chunkLength bytes, may be zero)          |
+~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~/
```

```
struct chunk_t
{
    uint8_t  chunkType;
    uint16_t chunkLength;
    uint8_t  chunkPayload[chunkLength];
} :variable*8;
```

chunkType : The chunk type code.

chunkLength : The size, in bytes, of the chunk payload.

chunkPayload : The type-specific payload of this chunk, chunkLength bytes in length (may be empty).

Defined chunk types are enumerated here in the order they might be encountered in the course of a typical session. The following chunk type codes are defined:

0x7f :	Packet Fragment (Section 2.3.1)
0x30 :	Initiator Hello (Section 2.3.2)
0x0f :	Forwarded Initiator Hello (Section 2.3.3)
0x70 :	Responder Hello (Section 2.3.4)
0x71 :	Responder Redirect (Section 2.3.5)
0x79 :	RHello Cookie Change (Section 2.3.6)
0x38 :	Initiator Initial Keying (Section 2.3.7)
0x78 :	Responder Initial Keying (Section 2.3.8)
0x01 :	Ping (Section 2.3.9)
0x41 :	Ping Reply (Section 2.3.10)
0x10 :	User Data (Section 2.3.11)
0x11 :	Next User Data (Section 2.3.12)
0x50 :	Data Acknowledgement Bitmap (Section 2.3.13)
0x51 :	Data Acknowledgement Ranges (Section 2.3.14)
0x18 :	Buffer Probe (Section 2.3.15)
0x5e :	Flow Exception Report (Section 2.3.16)
0x0c :	Session Close Request (Section 2.3.17)
0x4c :	Session Close Acknowledgement (Section 2.3.18)
0x00 :	Ignore/Padding
0xff :	Ignore/Padding

A receiver MUST ignore a chunk having an unrecognized chunk type code. A receiver MUST ignore a chunk appearing in a packet having a mode inappropriate to that chunk type.

Unless specified otherwise, if a chunk has a syntax or processing error (for example, the chunk's payload field is not long enough to contain the specified syntax elements), the chunk SHALL be ignored as though it was not present in the packet, and parsing and processing

SHALL commence with the next chunk in the packet, if any.

2.3.1. Packet Fragment Chunk

This chunk is used to divide a plain RTMFP packet (Section 2.2.4) that is unavoidably larger than the path MTU (such as session startup packets containing Responder Hello (Section 2.3.4) or Initiator Initial Keying (Section 2.3.7) chunks with large certificates) into segments that do not exceed the path MTU, and to allow the segments to be sent through the network at a moderated rate to avoid jamming interfaces, links, or paths.

```
 0 1 2 3 4 5 6 7|0 1 2 3 4 5 6 7|0 1 2 3 4 5 6 7|0 1 2 3 4 5 6 7
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|      0x7f     |          chunkLength          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
+-+-+-+-+-+-+-+-+-------------/-+-------------/-+
|M|  reserved   |   packetID  \ | fragmentNum \ |
+-+-+-+-+-+-+-+-+-------------/-+-------------/-+
+---------------------------------------------------------------+
|                          packetFragment                       |
+---------------------------------------------------------------/
```

```
struct fragmentChunkPayload_t
{
    bool_t  moreFragments :1; // M
    uintn_t reserved      :7;
    vlu_t   packetID      :variable*8;
    vlu_t   fragmentNum   :variable*8;
    uint8_t packetFragment[remainder()];
} :chunkLength*8;
```

moreFragments : If set, the indicated packet comprises additional fragments. If clear, this fragment is the final fragment of the packet.

reserved : Reserved for future use.

packetID : VLU, the identifier of this segmented packet. All fragments of the same packet have the same packetID.

fragmentNum : VLU, the index of this fragment of the indicated packet. The first fragment of the packet MUST be index 0. Fragments are numbered consecutively.

packetFragment : The bytes of the indicated segment of the indicated original plain RTMFP packet. A packetFragment MUST NOT be empty.

The use of this mechanism is detailed in Section 3.4.

2.3.2. Initiator Hello Chunk (IHello)

This chunk is sent by the initiator of a new session to begin the startup handshake. This chunk is only allowed in a packet with Session ID 0, encrypted with the default session key, and having packet mode 3 (Startup).

```
 0 1 2 3 4 5 6 7|0 1 2 3 4 5 6 7|0 1 2 3 4 5 6 7|0 1 2 3 4 5 6 7
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|      0x30     |          chunkLength          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
+-------------/-+-----------------------------------------------+
|  epdLength  \ |    endpointDiscriminator (epdLength bytes)    |
+-------------/-+-----------------------------------------------/
+---------------------------------------------------------------+
|                              tag                              |
+---------------------------------------------------------------/
```

```
struct ihelloChunkPayload_t
{
    vlu_t   epdLength :variable*8;
    uint8_t endpointDiscriminator[epdLength];
    uint8_t tag[remainder()];
} :chunkLength*8;
```

epdLength : VLU, the length of the following endpointDiscriminator field in bytes.

endpointDiscriminator : The Endpoint Discriminator for the identity with which the initiator wants to communicate.

tag : Initiator-provided data to be returned in a Responder Hello's tagEcho field. The tag/tagEcho is used to match Responder Hellos to the initiator's session startup state independent of the responder's address.

The use of IHello is detailed in Section 3.5.1.

2.3.3. Forwarded Initiator Hello Chunk (FIHello)

This chunk is sent on behalf of an initiator by a Forwarder. It is only allowed in packets of an established session having packet mode 1 or 2. A receiver MAY treat this chunk as though it was an

Initiator Hello received directly from replyAddress. Alternatively, if the receiver is selected by the Endpoint Discriminator, it MAY respond to replyAddress with an Implied Redirect (Section 2.3.5).

```
    0 1 2 3 4 5 6 7|0 1 2 3 4 5 6 7|0 1 2 3 4 5 6 7|0 1 2 3 4 5 6 7
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |      0x0f     |          chunkLength          |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   +-------------/-+-----------------------------------------------+
   |  epdLength  \ |    endpointDiscriminator (epdLength bytes)    |
   +-------------/-+-----------------------------------------------/
   +---------------------------------------------------------------+
   |                          replyAddress                         |
   +---------------------------------------------------------------/
   +---------------------------------------------------------------+
   |                              tag                              |
   +---------------------------------------------------------------/
```

```
   struct fihelloChunkPayload_t
   {
       vlu_t     epdLength :variable*8;
       uint8_t   endpointDiscriminator[epdLength];
       address_t replyAddress :variable*8;
       uint8_t   tag[remainder()];
   } :chunkLength*8;
```

epdLength : VLU, the length of the following endpointDiscriminator field in bytes.

endpointDiscriminator : The Endpoint Discriminator for the identity with which the original initiator wants to communicate, copied from the original Initiator Hello.

replyAddress : Address format (Section 2.1.5), the address that the forwarding node derived from the received Initiator Hello, to which the receiver should respond.

tag : Copied from the original Initiator Hello.

The use of FIHello is detailed in Section 3.5.1.5.

2.3.4. Responder Hello Chunk (RHello)

This chunk is sent by a responder in response to an Initiator Hello or Forwarded Initiator Hello if the Endpoint Discriminator indicates the responder's identity. This chunk is only allowed in a packet with Session ID 0, encrypted with the default session key, and having packet mode 3 (Startup).

```
    0 1 2 3 4 5 6 7|0 1 2 3 4 5 6 7|0 1 2 3 4 5 6 7|0 1 2 3 4 5 6 7
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |      0x70     |          chunkLength          |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   +-------------/-+-----------------------------------------------+
   |  tagLength  \ |            tagEcho (tagLength bytes)          |
   +-------------/-+-----------------------------------------------/
   +-------------/-+-----------------------------------------------+
   | cookieLength\ |           cookie (cookieLength bytes)         |
   +-------------/-+-----------------------------------------------/
   +---------------------------------------------------------------+
   |                     responderCertificate                      |
   +---------------------------------------------------------------/

   struct rhelloChunkPayload_t
   {
       vlu_t   tagLength :variable*8;
       uint8_t tagEcho[tagLength];
       vlu_t   cookieLength :variable*8;
       uint8_t cookie[cookieLength];
       uint8_t responderCertificate[remainder()];
   } :chunkLength*8;
```

tagLength : VLU, the length of the following tagEcho field in bytes.

tagEcho : The tag from the Initiator Hello, unaltered.

cookieLength : VLU, the length of the following cookie field in bytes.

cookie : Responder-created state data to authenticate a future Initiator Initial Keying message (in order to prevent denial of service attacks).

responderCertificate : The responder's cryptographic credentials.

Note: this specification doesn't mandate a specific choice of certificate format. The Cryptography Profile determines the syntax, algorithms, and interpretation of the responderCertificate.

The use of RHello is detailed in Section 3.5.1.

2.3.5. Responder Redirect Chunk (Redirect)

This chunk is sent in response to an Initiator Hello or Forwarded Initiator Hello to indicate that the requested endpoint can be reached at one or more of the indicated address(es). A receiver can add none, some, or all of the indicated address(es) to the set of

addresses to which it is sending Initiator Hello messages for the opening session associated with tagEcho. This chunk is only allowed in a packet with Session ID 0, encrypted with the default session key, and having packet mode 3 (Startup).

```
 0 1 2 3 4 5 6 7|0 1 2 3 4 5 6 7|0 1 2 3 4 5 6 7|0 1 2 3 4 5 6 7
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|      0x71     |          chunkLength          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
+-------------/-+-----------------------------------------------+
|  tagLength  \ |            tagEcho (tagLength bytes)          |
+-------------/-+-----------------------------------------------/
+~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~+
|                     redirectDestination 1                     |
+~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~/
                                :
                                :
+~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~+
|                     redirectDestination N                     |
+~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~/
```

```
struct responderRedirectChunkPayload_t
{
    vlu_t   tagLength :variable*8;
    uint8_t tagEcho[tagLength];
    addressCount = 0;
    while(remainder() > 0)
    {
        address_t redirectDestination :variable*8;
        addressCount++;
    }
    if(0 == addressCount)
        redirectDestination = packetSourceAddress();
} :chunkLength*8;
```

tagLength : VLU, the length of the following tagEcho field in bytes.

tagEcho : The tag from the Initiator Hello, unaltered.

redirectDestination : (Zero or more) Address format (Section 2.1.5), addresses to add to the opening set for the indicated session.

If this chunk lists zero redirectDestination addresses, then this is an Implied Redirect, and the indicated address is the address from which the packet containing this chunk was received.

The use of Redirect is detailed in Section 3.5.1.1.1, Section 3.5.1.1.2, and Section 3.5.1.4.

2.3.6. RHello Cookie Change Chunk

This chunk SHOULD be sent by a responder to an initiator in response to an Initiator Initial Keying if that chunk's cookie appears to have been created by the responder but the cookie is incorrect (for example, it includes a hash of the initiator's address, but the initiator's address is different than the one which elicited the Responder Hello containing the original cookie).

This chunk is only allowed in a packet encrypted with the default session key and having packet mode 3, and with the Session ID indicated in the initiatorSessionID field of the Initiator Initial Keying to which this is a response.

```
    0 1 2 3 4 5 6 7|0 1 2 3 4 5 6 7|0 1 2 3 4 5 6 7|0 1 2 3 4 5 6 7
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |      0x79     |          chunkLength          |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   +-------------/-+------------------------------------------------+
   | oldCookieLen\ |        oldCookie (oldCookieLen bytes)          |
   +-------------/-+------------------------------------------------/
   +----------------------------------------------------------------+
   |                           newCookie                            |
   +----------------------------------------------------------------/
```

```
   struct rhelloCookieChangeChunkPayload_t
   {
       vlu_t   oldCookieLen :variable*8;
       uint8_t oldCookie[oldCookieLen];
       uint8_t newCookie[remainder()];
   } :chunkLength*8;
```

oldCookieLen : VLU, the length of the following oldCookie field in bytes.

oldCookie : The cookie that was sent in a previous Responder Hello and Initiator Initial Keying.

newCookie : The new cookie that the responder would like sent (and signed) in a replacement Initiator Initial Keying. The old and new cookies need not have the same lengths.

On receipt of this chunk, the initiator SHOULD compute, sign, and send a new Initiator Initial Keying having newCookie in place of oldCookie. The use of this chunk is detailed in Section 3.5.1.2.

2.3.7. Initiator Initial Keying Chunk (IIKeying)

This chunk is sent by an initiator to establish a session with a responder. The initiator MUST have obtained a valid cookie to use with the responder, typically by receiving a Responder Hello from it. This chunk is only allowed in a packet with Session ID 0, encrypted with the default session key, and having packet mode 3 (Startup).

```
    0 1 2 3 4 5 6 7|0 1 2 3 4 5 6 7|0 1 2 3 4 5 6 7|0 1 2 3 4 5 6 7
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |      0x38     |          chunkLength          |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |                       initiatorSessionID                      |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   +-------------/-+-----------------------------------------------+
   | cookieLength\ |                  cookieEcho                   |
   +-------------/-+-----------------------------------------------/
   +-------------/-+-----------------------------------------------+
   |  certLength \ |             initiatorCertificate              |
   +-------------/-+-----------------------------------------------/
   +-------------/-+-----------------------------------------------+
   |  skicLength \ |          sessionKeyInitiatorComponent         |
   +-------------/-+-----------------------------------------------/
   +---------------------------------------------------------------+
   |                           signature                           |
   +---------------------------------------------------------------/
```

```
   struct iikeyingChunkPayload_t
   {
       struct
       {
           uint32_t initiatorSessionID;
           vlu_t    cookieLength :variable*8;
           uint8_t  cookieEcho[cookieLength];
           vlu_t    certLength :variable*8;
           uint8_t  initiatorCertificate[certLength];
           vlu_t    skicLength :variable*8;
           uint8_t  sessionKeyInitiatorComponent[skicLength];
       } initiatorSignedParameters :variable*8;
       uint8_t signature[remainder()];
   } :chunkLength*8;
```

initiatorSessionID : The Session ID to be used by the responder when sending packets to the Initiator.

cookieLength : VLU, the length of the following cookieEcho field in bytes.

cookieEcho : The cookie from the Responder Hello, unaltered.

certLength : VLU, the length of the following initiatorCertificate field in bytes.

initiatorCertificate : The initiator's identity credentials.

skicLength : VLU, the length of the following sessionKeyInitiatorComponent field in bytes.

sessionKeyInitiatorComponent : The initiator's portion of the session key negotiation according to the Cryptography Profile.

initiatorSignedParameters : The payload portion of this chunk up to the signature field.

signature : The initiator's digital signature of the initiatorSignedParameters according to the Cryptography Profile.

Note: this specification doesn't mandate a specific choice of cryptography. The Cryptography Profile determines the syntax, algorithms, and interpretation of the initiatorCertificate, responderCertificate, sessionKeyInitiatorComponent, sessionKeyResponderComponent, and signature, and how the sessionKeyInitiatorComponent and sessionKeyResponderComponent are combined to derive the session keys.

The use of IIKeying is detailed in Section 3.5.1.

2.3.8. Responder Initial Keying Chunk (RIKeying)

This chunk is sent by a responder in response to an Initiator Initial Keying as the final phase of session startup. This chunk is only allowed in a packet encrypted with the default session key, having packet mode 3 (Startup), and sent to the initiator with the Session ID specified by the initiatorSessionID field from the Initiator Initial Keying.

```
    0 1 2 3 4 5 6 7|0 1 2 3 4 5 6 7|0 1 2 3 4 5 6 7|0 1 2 3 4 5 6 7
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |      0x78     |          chunkLength          |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |                       responderSessionID                      |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   +-------------/-+-----------------------------------------------+
   |  skrcLength \ |         sessionKeyResponderComponent          |
   +-------------/-+-----------------------------------------------/
   +---------------------------------------------------------------+
   |                           signature                           |
   +---------------------------------------------------------------/

   struct rikeyingChunkPayload_t
   {
       struct
       {
           uint32_t responderSessionID;
           vlu_t    skrcLength :variable*8;
           uint8_t  sessionKeyResponderComponent[skrcLength];
       } responderSignedParametersPortion :variable*8;
       uint8_t  signature[remainder()];
   } :chunkLength*8;

   struct
   {
       responderSignedParametersPortion;
       sessionKeyInitiatorComponent;
   } responderSignedParameters;
```

responderSessionID : The Session ID to be used by the Initiator when sending packets to the Responder.

skrcLength : VLU, the length of the following sessionKeyResponderComponent field in bytes.

sessionKeyResponderComponent : The responder's portion of the session key negotiation according to the Cryptography Profile.

responderSignedParametersPortion : The payload portion of this chunk up to the signature field.

signature : The responder's digital signature of the responderSignedParameters (see below) according to the Cryptography Profile.

responderSignedParameters : The concatenation of the responderSignedParametersPortion (the payload portion of this chunk up to the signature field) and the sessionKeyInitiatorComponent from the Initiator Initial Keying to which this chunk is a response.

Note: this specification doesn't mandate a specific choice of cryptography. The Cryptography Profile determines the syntax, algorithms, and interpretation of the initiatorCertificate, responderCertificate, sessionKeyInitiatorComponent, sessionKeyResponderComponent, and signature, and how the sessionKeyInitiatorComponent and sessionKeyResponderComponent are combined to derive the session keys.

Once the responder has computed the sessionKeyResponderComponent, it has all of the information and state necessary for an established session with the initiator. Once the responder has sent this chunk to the initiator, the session is established and ready to carry flows of user data.

Once the initiator receives, verifies, and processes this chunk, it has all of the information and state necessary for an established session with the responder. The session is established and ready to carry flows of user data.

The use of RIKeying is detailed in Section 3.5.1.

2.3.9. Ping Chunk

This chunk is sent in order to elicit a Ping Reply from the receiver. It is only allowed in a packet belonging to an established session and having packet mode 1 or 2.

```
    0 1 2 3 4 5 6 7|0 1 2 3 4 5 6 7|0 1 2 3 4 5 6 7|0 1 2 3 4 5 6 7
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |      0x01     |          chunkLength          |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   +~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~+
   |                             message                           |
   +~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~/

   struct pingChunkPayload_t
   {
       uint8_t message[chunkLength];
   } :chunkLength*8;
```

message : The (potentially empty) message that is expected to be returned by the other end of the session in a Ping Reply.

The receiver of this chunk SHOULD reply as immediately as is practical with a Ping Reply.

Ping and the expected Ping Reply are typically used for session keepalive, endpoint address change verification, and path MTU discovery. See Section 3.5.4 for details.

2.3.10. Ping Reply Chunk

This chunk is sent in response to a Ping chunk. It is only allowed in a packet belonging to an established session and having packet mode 1 or 2.

```
 0 1 2 3 4 5 6 7|0 1 2 3 4 5 6 7|0 1 2 3 4 5 6 7|0 1 2 3 4 5 6 7
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|      0x41     |          chunkLength          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
+~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~+
|                           messageEcho                         |
+~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~/
```

```
struct pingReplyChunkPayload_t
{
    uint8_t messageEcho[chunkLength];
} :chunkLength*8;
```

messageEcho : The message from the Ping to which this is a response, unaltered.

2.3.11. User Data Chunk

This chunk is the basic unit of transmission for the user messages of a flow. A user message comprises one or more fragments. Each fragment is carried in its own chunk and has a unique sequence number in its flow. It is only allowed in a packet belonging to an established session and having packet mode 1 or 2.

```
    0 1 2 3 4 5 6 7|0 1 2 3 4 5 6 7|0 1 2 3 4 5 6 7|0 1 2 3 4 5 6 7
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |      0x10     |          chunkLength          |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   +-+-+-+-+-+-+-+-+
   |O|r| F | r |A|F|
   |P|s| R | s |B|I|
   |T|v| A | v |N|N|
   +-+-+-+-+-+-+-+-+
   +-------------/-+-------------/-+-------------/-+
   |   flowID    \ |     seq#    \ |  fsnOffset  \ |
   +-------------/-+-------------/-+-------------/-+
   +~~~/~~~/~~~~~~~+               +~~~/~~~/~~~~~~~+-------------/-+
   | L \ T \   V   |... options ...| L \ T \   V   |       0     \ |
   \~~~/~~~/~~~~~~~+   [if(OPT)]   +~~~/~~~/~~~~~~~+-------------/-/
   +~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~+
   |                            userData                           |
   +~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~/
```

```
   struct userDataChunkPayload_t
   {
       bool_t  optionsPresent :1;  // "OPT"
       uintn_t reserved1 :1;       // "rsv"
       uintn_t fragmentControl :2; // "FRA
           // 0=whole, 1=begin, 2=end, 3=middle
       uintn_t reserved2 :2;       // "rsv"
       bool_t  abandon :1;         // "ABN"
       bool_t  final :1;           // "FIN"
       vlu_t   flowID :variable*8;
       vlu_t   sequenceNumber :variable*8; // "seq#"
       vlu_t   fsnOffset :variable*8;
       forwardSequenceNumber = sequenceNumber - fsnOffset;
       if(optionsPresent)
           optionList_t options :variable*8;
       uint8_t userData[remainder()];
   } :chunkLength*8;
```

optionsPresent : If set, indicates the presence of an option list before the user data. If clear, there is no option list in this chunk.

fragmentControl : Indicates how this fragment is assembled, potentially with others, into a complete user message. Possible values:

0 : This fragment is a complete message.

1 : This fragment is the first of a multi-fragment message.

2 : This fragment is the last of a multi-fragment message.

3 : This fragment is in the middle of a multi-fragment message.

A single-fragment user message has a fragment control of "0-whole". When a message has more than one fragment, the first fragment has a fragment control of "1-begin", then zero or more "3-middle" fragments, and finally a "2-end" fragment. The sequence numbers of a multi-fragment message MUST be contiguous.

abandon : If set, this sequence number has been abandoned by the sender. The userData, if any, MUST be ignored.

final : If set, this is the last sequence number of the flow.

flowID : VLU, the flow identifier.

sequenceNumber : VLU, the sequence number of this fragment. Fragments are assigned contiguous increasing sequence numbers in a flow. The first sequence number of a flow SHOULD be 1. The first sequence number of a flow MUST be greater than zero. Sequence numbers are unbounded and do not wrap.

fsnOffset : VLU, the difference between the Sequence Number and the Forward Sequence Number. This field MUST NOT be zero if the abandon flag is not set. This field MUST NOT be greater than sequenceNumber.

forwardSequenceNumber : The flow sender will not send (or resend) any fragment with a sequence number less than or equal to the forward sequence number.

options : If the optionsPresent flag is set, a list of zero or more Options terminated by a Marker is present. See Section 2.3.11.1 for defined options.

userData : The actual user data for this fragment.

The use of User Data is detailed in Section 3.6.2.

2.3.11.1. Options for User Data

This section lists options that may appear in User Data option lists. A conforming implementation MUST support the options in this section.

A flow receiver MUST reject a flow containing a flow option that is not understood if the option type is less than 8192 (0x2000). A flow receiver MUST ignore any flow option that is not understood if the option type is 8192 or greater.

The following option type codes are defined for User Data:

0x00 : Users's Per-Flow Metadata (Section 2.3.11.1.1)

0x0a : Return Flow Association (Section 2.3.11.1.2)

2.3.11.1.1. User's Per-Flow Metadata

This option conveys the user's per-flow metadata for the flow to which it's attached.

```
+-------------/-+-------------/-+~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~+
|   length    \ |     0x00    \ |         userMetadata          |
+-------------/-+-------------/-+~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~/

struct userMetadataOptionValue_t
{
    uint8_t userMetadata[remainder()];
} :remainder()*8;
```

The user associates application-defined metadata with each flow. The metadata does not change over the life of the flow. Every flow MUST have metadata. A flow sender MUST send this option with the first User Data chunk for this flow in each packet until an acknowledgement for this flow is received. A flow sender SHOULD NOT send this option more than once for each flow in any one packet. A flow sender SHOULD NOT send this option for a flow once the flow has been acknowledged.

This specification doesn't mandate the encoding, syntax, or interpretation of the user's per-flow metadata; this is determined by the application.

The userMetadata SHOULD NOT exceed 512 bytes. The userMetadata MAY be 0 bytes in length.

2.3.11.1.2. Return Flow Association

A new flow can be considered to be in return (or response) to a flow sent by the other endpoint. This option encodes the receive flow identifier to which this new sending flow is a response.

```
+-------------/-+-------------/-+-------------/-+
|   length    \ |     0x0a    \ |    flowID   \ |
+-------------/-+-------------/-+-------------/-+
```

```
struct returnFlowAssociationOptionValue_t
{
    vlu_t flowID :variable*8;
} :variable*8;
```

Consider endpoints A and B. Endpoint A begins a flow with identifier 5 to endpoint B. A is the flow sender for A's flowID=5 and B is the flow receiver for A's flowID=5. B begins a return flow with identifier 7 to A in response to A's flowID=5. B is the flow sender for B's flowID=7 and A is the flow receiver for B's flowID=7. B sends this option with flowID set to 5 to indicate that B's flowID=7 is in response to and associated with A's flowID=5.

If there is a return association, the flow sender MUST send this option with the first User Data chunk for this flow in each packet until an acknowledgement for this flow is received. A flow sender SHOULD NOT send this option more than once for each flow in any one packet. A flow sender SHOULD NOT send this option for a flow once the flow has been acknowledged.

A flow MUST NOT indicate more than one return association.

A flow MUST indicate its return association, if any, upon its first transmission of a User Data chunk. A return association can't be added to a sending flow after it begins.

A flow receiver MUST reject a new receiving flow having a return flow association that does not indicate an F_OPEN sending flow.

2.3.12. Next User Data Chunk

This chunk is equivalent to the User Data Chunk for purposes of sending the user messages of a flow. When used, it MUST follow a User Data or another Next User Data chunk in the same packet.

```
    0 1 2 3 4 5 6 7|0 1 2 3 4 5 6 7|0 1 2 3 4 5 6 7|0 1 2 3 4 5 6 7
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |      0x11     |          chunkLength          |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   +-+-+-+-+-+-+-+-+
   |O|r| F | r |A|F|
   |P|s| R | s |B|I|
   |T|v| A | v |N|N|
   +-+-+-+-+-+-+-+-+
   +~~~/~~~/~~~~~~~+               +~~~/~~~/~~~~~~~+-------------/-+
   | L \ T \   V   |... options ...| L \ T \   V   |       0     \ |
   \~~~/~~~/~~~~~~~+   [if(OPT)]   +~~~/~~~/~~~~~~~+-------------/-/
   +~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~+
   |                            userData                           |
   +~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~/

   struct nextUserDataChunkPayload_t
   {
       bool_t  optionsPresent :1;  // "OPT"
       uintn_t reserved1 :1;       // "rsv"
       uintn_t fragmentControl :2; // "FRA
           // 0=whole, 1=begin, 2=end, 3=middle
       uintn_t reserved2 :2;       // "rsv"
       bool_t  abandon :1;         // "ABN"
       bool_t  final :1;           // "FIN"
       if(optionsPresent)
           optionList_t options :variable*8;
       uint8_t userData[remainder()];
   } :chunkLength*8;
```

This chunk is considered to be for the same flowID as the most recently preceding User Data or Next User Data chunk in the same packet, having the same Forward Sequence Number, and having the next sequence number. The optionsPresent, fragmentControl, abandon, and final flags, and the options (if present) have the same interpretation as for the User Data chunk.

```
   ...
   ----------+------------------------------------
   10 00 07  | User Data chunk, length=7
   00        | OPT=0, FRA=0 "whole", ABN=0, FIN=0
   02 05 03  | flowID=2, seq#=5, fsn=(5-3)=2
   00 01 02  | data 3 bytes: 00, 01, 02
   ----------+------------------------------------
   11 00 04  | Next User Data chunk,length=4
   00        | OPT=0, FRA=0 "whole", ABN=0, FIN=0
             | flowID=2, seq#=6, fsn=2
   03 04 05  | data 3 bytes: 03, 04, 05
   ----------+------------------------------------
   11 00 04  | Next User Data chunk, length=4
   00        | OPT=0, FRA=0 "whole", ABN=0, FIN=0
             | flowID=2, seq#=7, fsn=2
   06 07 08  | data 3 bytes: 06, 07, 08
   ----------+------------------------------------
```

Figure 3: Sequential messages in one packet using Next User Data

The use of Next User Data is detailed in Section 3.6.2.3.2.

2.3.13. Data Acknowledgement Bitmap Chunk (Bitmap Ack)

This chunk is sent by the flow receiver to indicate to the flow sender the User Data fragment sequence numbers that have been received for one flow. It is only allowed in a packet belonging to an established session and having packet mode 1 or 2.

The flow receiver can choose to acknowledge User Data with this chunk or with a Range Ack. It SHOULD choose whichever format has the most compact encoding of the sequence numbers received.

```
    0 1 2 3 4 5 6 7|0 1 2 3 4 5 6 7|0 1 2 3 4 5 6 7|0 1 2 3 4 5 6 7
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |      0x50     |          chunkLength          |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   +-------------/-+-------------/-+-------------/-+
   |   flowID    \ |   bufAvail  \ |    cumAck   \ |
   +-------------/-+-------------/-+-------------/-+
   +~+~+~+~+~+~+~+~+~+~+~+~+~+~+~+~+~+~+~+~+~+~+~+~+
   |C|C|C|C|C|C|C|C|C|C|C|C|C|C|C|C|C|C|C|C|C|C|C|C|
   |+|+|+|+|+|+|+|+|+|+|+|+|+|+|+|+|+|+|+|+|+|+|+|+|
   |9|8|7|6|5|4|3|2|1|1|1|1|1|1|1|1|2|2|2|2|2|2|1|1| ....
   | | | | | | | | |7|6|5|4|3|2|1|0|5|4|3|2|1|0|9|8|
   +~+~+~+~+~+~+~+~+~+~+~+~+~+~+~+~+~+~+~+~+~+~+~+~+
```

```
   struct dataAckBitmapChunkPayload_t
   {
       vlu_t flowID :variable*8;
       vlu_t bufferBlocksAvailable :variable*8; // "bufAvail"
       vlu_t cumulativeAck :variable*8; // "cumAck"
       bufferBytesAvailable = bufferBlocksAvailable * 1024;
       acknowledge(0 through cumulativeAck);
       ackCursor = cumulativeAck + 1;
       while(remainder() > 0)
       {
           for(bitPosition = 8; bitPosition > 0; bitPosition--)
           {
               bool_t bit :1;
               if(bit)
                   acknowledge(ackCursor + bitPosition);
           }
           ackCursor += 8;
       }
   } :chunkLength*8;
```

flowID : VLU, the flow identifier.

bufferBlocksAvailable : VLU, the number of 1024-byte blocks of User Data that the receiver is currently able to accept. Section 3.6.3.5 describes how to calculate this value.

cumulativeAck : VLU, the acknowledgement of every fragment sequence number in this flow that is less than or equal to this value. This MUST NOT be less than the highest forward sequence number received in this flow.

bit field : A sequence of zero or more bytes representing a bit field of received fragment sequence numbers after the cumulative acknowledgement, least significant bit first. A set bit indicates receipt of a sequence number. A clear bit indicates that sequence number was not received. The least significant bit of the first byte is the second sequence number following the cumulative acknowledgement, the next bit is the third sequence number following, and so on.

```
     50 00 05  | Bitmap Ack, length=5 bytes
     05 7f 10  | flowID=5, bufAvail=127*1024 bytes, cumAck=0..16
     79 06     | 01111001 00000110 = 18, 21, 22, 23, 24, 27, 28
```

Example bitmap ack indicating acknowledgement of fragment sequence numbers 0 through 16, 18, 21 through 24, 27 and 28.

Figure 4

2.3.14. Data Acknowledgement Ranges Chunk (Range Ack)

This chunk is sent by the flow receiver to indicate to the flow sender the User Data fragment sequence numbers that have been received for one flow. It is only allowed in a packet belonging to an established session and having packet mode 1 or 2.

The flow receiver can choose to acknowledge User Data with this chunk or with a Bitmap Ack. It SHOULD choose whichever format has the most compact encoding of the sequence numbers received.

```
  0 1 2 3 4 5 6 7|0 1 2 3 4 5 6 7|0 1 2 3 4 5 6 7|0 1 2 3 4 5 6 7
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 |      0x51     |          chunkLength          |
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 +-------------/-+-------------/-+-------------/-+
 |   flowID    \ |   bufAvail  \ |    cumAck   \ |
 +-------------/-+-------------/-+-------------/-+
 +~~~~~~~~~~~~~/~+~~~~~~~~~~~~~/~+
 |   #holes-1  \ |   #recv-1   \ |
 +~~~~~~~~~~~~~/~+~~~~~~~~~~~~~/~+
                 :
                 :
 +~~~~~~~~~~~~~/~+~~~~~~~~~~~~~/~+
 |   #holes-1  \ |   #recv-1   \ |
 +~~~~~~~~~~~~~/~+~~~~~~~~~~~~~/~+

 struct dataAckRangesChunkPayload_t
 {
     vlu_t flowID :variable*8;
     vlu_t bufferBlocksAvailable :variable*8; // "bufAvail"
     vlu_t cumulativeAck :variable*8; // "cumAck"
     bufferBytesAvailable = bufferBlocksAvailable * 1024;
     acknowledge(0 through cumulativeAck);
     ackCursor = cumulativeAck;
     while(remainder() > 0)
     {
         vlu_t holesMinusOne :variable*8; // "#holes-1"
         vlu_t receivedMinusOne :variable*8; // "#recv-1"

         ackCursor++;
         rangeFrom = ackCursor + holesMinusOne + 1;
         rangeTo = rangeFrom + receivedMinusOne;
         acknowledge(rangeFrom through rangeTo);

         ackCursor = rangeTo;
     }
 } :chunkLength*8;
```

flowID : VLU, the flow identifier.

bufferBlocksAvailable : VLU, the number of 1024-byte blocks of User Data that the receiver is currently able to accept. Section 3.6.3.5 describes how to calculate this value.

cumulativeAck : VLU, the acknowledgement of every fragment sequence number in this flow that is less than or equal to this value. This MUST NOT be less than the highest forward sequence number received in this flow.

holesMinusOne / receivedMinusOne : Zero or more acknowledgement ranges, run-length encoded. Runs are encoded as zero or more pairs of VLUs indicating the number (minus one) of missing sequence numbers followed by the number (minus one) of received sequence numbers, starting at the cumulative acknowledgement. NOTE: If a parser syntax error is encountered here (that is, if the chunk is truncated such that not enough bytes remain to completely encode both VLUs of the acknowledgement range), then treat and process this chunk as though it was properly formed up to the last completely encoded range.

```
51 00 07  | Range Ack, length=7
05 7f 10  | flowID=5, bufAvail=127*1024 bytes, cumAck=0..16
00 00     | holes=1, received=1 -- missing 17, received 18
01 03     | holes=2, received=4 -- missing 19..20, received 21..24
```

Example range ack indicating acknowledgement of fragment sequence numbers 0 through 16, 18, 21, 22, 23, 24.

Figure 5

```
51 00 07  | Range Ack, length=9
05 7f 10  | flowID=5, bufAvail=127*1024 bytes, cumAck=0..16
00 00     | holes=1, received=1 -- missing 17, received 18
01 83     | holes=2, received=VLU parse error, ignore this range
```

Example range ack indicating acknowledgement of fragment sequence numbers 0 through 16 and 18, with a truncated last range. Note the truncation and parse error does not abort the entire chunk in this case.

Figure 6

2.3.15. Buffer Probe Chunk

This chunk is sent by the flow sender in order to request the current available receive buffer (in the form of a Data Acknowledgement) for a flow. It is only allowed in a packet belonging to an established session and having packet mode 1 or 2.

```
 0 1 2 3 4 5 6 7|0 1 2 3 4 5 6 7|0 1 2 3 4 5 6 7|0 1 2 3 4 5 6 7
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|      0x18     |          chunkLength          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
+-------------/-+
|   flowID    \ |
+-------------/-+

struct bufferProbeChunkPayload_t
{
    vlu_t flowID :variable*8;
} :chunkLength*8;
```

flowID : VLU, the flow identifier.

The receiver of this chunk SHOULD reply as immediately as is practical with a Data Acknowledgement.

2.3.16. Flow Exception Report Chunk

This chunk is sent by the flow receiver to indicate that it is not (or is no longer) interested in the flow and would like the flow sender to close the flow. This chunk SHOULD precede every Data Acknowledgement chunk for the same flow in this condition.

This chunk is only allowed in a packet belonging to an established session and having packet mode 1 or 2.

```
 0 1 2 3 4 5 6 7|0 1 2 3 4 5 6 7|0 1 2 3 4 5 6 7|0 1 2 3 4 5 6 7
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|      0x5e     |          chunkLength          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
+-------------/-+-------------/-+
|   flowID    \ |  exception  \ |
+-------------/-+-------------/-+

struct flowExceptionReportChunkPayload_t
{
    vlu_t flowID :variable*8;
    vlu_t exception :variable*8;
} :chunkLength*8;
```

flowID : VLU, the flow identifier.

exception : VLU, the application-defined exception code being reported.

A receiving RTMFP might reject a flow automatically, for example if

it is missing metadata, or if an invalid return association is specified. In circumstances where an RTMFP rejects a flow automatically, the exception code MUST be 0. The application can specify any exception code, including 0, when rejecting a flow. All non-zero exception codes are reserved for the application.

2.3.17. Session Close Request Chunk (Close)

This chunk is sent to cleanly terminate a session. It is only allowed in a packet belonging to an established or closing session and having packet mode 1 or 2.

```
 0 1 2 3 4 5 6 7|0 1 2 3 4 5 6 7|0 1 2 3 4 5 6 7|0 1 2 3 4 5 6 7
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|      0x0c     |               0               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

This chunk has no payload.

The use of Close is detailed in Section 3.5.5.

2.3.18. Session Close Acknowledgement Chunk (Close Ack)

This chunk is sent in response to a Session Close Request to indicate the sender has terminated the session. It is only allowed in a packet belonging to an established or closing session and having packet mode 1 or 2.

```
 0 1 2 3 4 5 6 7|0 1 2 3 4 5 6 7|0 1 2 3 4 5 6 7|0 1 2 3 4 5 6 7
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|      0x4c     |               0               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

This chunk has no payload.

The use of Close Ack is detailed in Section 3.5.5.

3. Operation

3.1. Overview

```
             +--------+                             +--------+
             | Peer A |        S E S S I O N        | Peer B |
             |        /=============================\        |
             |       ||            Flows            ||       |
             |       ||---------------------------->||       |
             |       ||---------------------------->||       |
             |       ||<----------------------------||       |
             |       ||<----------------------------||       |
             |       ||<----------------------------||       |
             |        \=============================/        |
             |        |                             |        |
             |        |                             +--------+
             |        |
             |        |                             +--------+
             |        |        S E S S I O N        | Peer C |
             |        /=============================\        |
             |       ||            Flows            ||       |
             |       ||---------------------------->||       |
             |       ||<----------------------------||       |
             |       ||<----------------------------||       |
             |        \=============================/        |
             |        |                             |        |
             +--------+                             +--------+
```

Figure 7: Sessions between pairs of communicating endpoints

Between any pair of communicating endpoints is a single, bidirectional, secured, congestion controlled session. Unidirectional flows convey messages from one end to the other within the session.

An endpoint initiates a session to a far end when communication is desired. An initiator begins with one or more candidate destination socket addresses, and may learn and try more candidate addresses during startup handshaking. Eventually a first suitable response is received, and that endpoint is selected. Startup proceeds to the selected endpoint. In the case of session startup glare, one endpoint is the prevailing initiator and the other assumes the role of responder. Encryption keys and session identifiers are negotiated between the endpoints, and the session is established.

Each endpoint may begin sending message flows to the other end. For each flow, the far end may accept it and deliver its messages to the user, or it may reject the flow and transmit an exception to the sender. The flow receiver may close and reject a flow at a later time, after first accepting it. The flow receiver acknowledges all

data sent to it regardless of whether the flow was accepted. Acknowledgements drive a congestion control mechanism.

An endpoint may have concurrent sessions with other far endpoints. The multiple sessions are distinguished by a session identifier rather than by socket address. This allows an endpoint's address to change mid-session without having to tear down and re-establish a session. The existing cryptographic state for a session can be used to verify a change of address while protecting against session hijacking or denial-of-service.

A sender may indicate to a receiver that some user messages are of a time-critical or real-time nature. A receiver may indicate to senders on concurrent sessions that it is receiving time-critical messages from another endpoint. The other senders SHOULD modify their congestion control parameters to yield capacity to the session carrying time-critical messages.

A sender may close a flow. The flow is completed when the receiver has no outstanding gaps before the final fragment of the flow. The sender and receiver reserve a completed flow's identifier for a time to allow in flight messages to drain from the network.

Eventually, neither end will have any flows open to the other. The session will be idle and quiescent. Either end may reliably close the session to recover its resources.

In certain circumstances, an endpoint may be ceasing operation and not have time to wait for acknowledgement of a reliable session close. In this case the halting endpoint may send an abrupt session close to advise the far end that it is halting immediately.

3.2. Endpoint Identity

Each RTMFP endpoint has an identity. The identity is encoded in a certificate. This specification doesn't mandate any particular certificate format, cryptographic algorithms, or cryptographic properties for certificates.

An endpoint is named by an Endpoint Discriminator. This specification doesn't mandate any particular format for Endpoint Discriminators.

An Endpoint Discriminator MAY select more than one identity, and MAY match more than one distinct certificate.

Multiple distinct Endpoint Discriminators MAY match one certificate.

It is RECOMMENDED that multiple endpoints not have the same identity. Entities with the same identity are indistinguishable during session startup, which could be undesirable in some applications.

An endpoint MAY have more than one address.

The Cryptography Profile implements the following functions for identities, certificates, and Endpoint Discriminators, whose operation MUST be deterministic:

- Test whether a given certificate is authentic. Authenticity can comprise verifying an issuer signature chain in a public key infrastructure.

- Test whether a given Endpoint Discriminator selects a given certificate.

- Test whether a given Endpoint Discriminator selects the local endpoint.

- Generate a Canonical Endpoint Discriminator for a given certificate. Canonical Endpoint Discriminators for distinct identities SHOULD be distinct. If two distinct identities have the same Canonical Endpoint Discriminator, an initiator might abort a new opening session to the second identity (Section 3.5.1.1.1), which might not be desired.

- Given a certificate, a message, and a digital signature over the message, test whether the signature is valid and generated by the owner of the certificate.

- Generate a digital signature for a given message corresponding to the near identity.

- Given the near identity and a far certificate, determine which one shall prevail as Initiator and which shall assume the Responder role in the case of startup glare. The far end MUST arrive at the same conclusion. A comparison function can comprise performing a lexicographic ordering of the binary certificates, and declaring the far identity the prevailing endpoint if the far certificate is ordered before the near certificate, and otherwise declaring the near identity to be the prevailing endpoint.

- Given a first certificate and a second certificate, test whether a new incoming session from the second shall override an existing session with the first. It is RECOMMENDED that the test comprise testing whether the certificates are bitwise identical.

All other semantics for certificates and Endpoint Discriminators are determined by the Cryptography Profile and the application.

3.3. Packet Multiplex

An RTMFP typically has one or more interfaces through which it communicates with other RTMFP endpoints. RTMFP can communicate with multiple distinct other RTMFP endpoints through each local interface. Session multiplexing over a shared interface can facilitate peer-to-peer communications through a NAT, by enabling third party endpoints such as Forwarders (Section 3.5.1.5) and Redirectors (Section 3.5.1.4) to observe the translated public address and inform peers of the translation.

An interface is typically a UDP socket (Section 2.2.1), but MAY be any suitable datagram transport service where endpoints can be addressed by IPv4 or IPv6 socket addresses.

RTMFP uses a session ID to multiplex and demultiplex communications with distinct endpoints (Section 2.2.2), in addition to the endpoint socket address. This allows an RTMFP to detect a far-end address change (as might happen for example in mobile and wireless scenarios) and for communication sessions to survive address changes. This also allows an RTMFP to act as a Forwarder or Redirector for an endpoint with which it has an active session, by distinguishing startup packets from those of the active session.

On receiving a packet, an RTMFP decodes the session ID to look up the corresponding session information context and decryption key. Session ID 0 is reserved for session startup and MUST NOT be used for an active session. A packet for session ID 0 uses the Default Session Key as defined by the Cryptography Profile.

3.4. Packet Fragmentation

When an RTMFP packet (Section 2.2.4) is unavoidably larger than the path MTU (such as a startup packet containing an RHello (Section 2.3.4) or IIKeying (Section 2.3.7) chunk with a large certificate), it can be fragmented into segments that do not exceed the path MTU using the Packet Fragment chunk (Section 2.3.1).

The packet fragmentation mechanism SHOULD be used only to segment unavoidably large packets. Accordingly, this mechanism SHOULD be employed only during session startup with session ID 0. This mechanism MUST NOT be used instead of the natural fragmentation mechanism of the User Data (Section 2.3.11) and Next User Data (Section 2.3.12) chunks for dividing the messages of the user's data flows into segments that do not exceed the path MTU.

A fragmented plain RTMFP packet is reassembled by concatenating the packetFragment fields of the fragments for the packet in contiguous ascending order, starting from index 0 through and including the final fragment.

When reassembling packets for Session ID 0, a receiver SHOULD identify the packets by both the socket address from which the packet containing the fragment was received as well as the indicated packetID.

A receiver SHOULD allow up to 60 seconds to completely receive a fragmented packet for which progress is being made. A packet is progressing if at least one new fragment for it was received in the last second.

A receiver MUST discard a Packet Fragment chunk having an empty packetFragment field.

The mode of each packet containing Packet Fragments for the same fragmented packet MUST match the mode of the fragmented packet. A receiver MUST discard any new Packet Fragment chunk received in a packet with a mode different from the mode of the packet containing the first received fragment. A receiver MUST discard any reassembled packet with a mode different than the packets containing its fragments.

In order to avoid jamming the network, the sender MUST rate limit packet transmission. In the absence of specific path capacity information (for instance, during session startup), a sender SHOULD NOT send more than 4380 bytes nor more than four packets per distinct endpoint every 200ms.

To avoid resource exhaustion, a receiver SHOULD limit the number of concurrent packet reassembly buffers and the size of each buffer. Limits can depend on the expected size of reassembled packets, the rate at which fragmented packets are expected to be received and degree of interleaving, and the expected function of the receiver. Limits can depend on the available resources of the receiver. There can be different limits for packets with Session ID 0 and packets for established sessions. For example, a busy server might need to allow for several hundred concurrent packet reassembly buffers to accommodate hundreds of connection requests per second with potentially interleaved fragments, but a client device with constrained resources could allow just a few reassembly buffers. In the absence of specific information regarding the expected size of reassembled packets, a receiver should set the limit for each packet reassembly buffer to 65536 bytes.

3.5. Sessions

A session is the protocol relationship between a pair of communicating endpoints, comprising the shared and endpoint-specific information context necessary to carry out the communication. The session context at each end includes at least:

- TS_RX: the last timestamp received from the far end;
- TS_RX_TIME: the time at which TS_RX was first observed to be different than its previous value;
- TS_ECHO_TX: the last timestamp echo sent to the far end;
- MRTO: the measured retransmission timeout;
- ERTO: the effective retransmission timeout;
- Cryptographic keys for encrypting and decrypting packets, and for verifying the validity of packets, according to the Cryptography Profile;
- Cryptographic near and far nonces, according to the Cryptography Profile, the near nonce being the far end's far nonce, and vice versa;
- The certificate of the far end;
- The receive session identifier, used by the far end when sending packets to this end;
- The send session identifier to use when sending packets to the far end;
- DESTADDR: the socket address to which to send packets to the far end;
- The set of all sending flow contexts (Section 3.6.2);
- The set of all receiving flow contexts (Section 3.6.3);
- The transmission budget, which controls the rate at which data is sent into the network (for example, a congestion window);
- S_OUTSTANDING_BYTES: the total amount of user message data outstanding, or in flight, in the network; that is the sum of the F_OUTSTANDING_BYTES of each sending flow in the session;

- RX_DATA_PACKETS: a count of the number of received packets containing at least one User Data chunk since the last acknowledgement was sent, initially 0;

- ACK_NOW: a boolean flag indicating whether an acknowledgement should be sent immediately, initially false;

- DELACK_ALARM: an alarm to trigger an acknowledgement after a delay, initially unset;

- The state, at any time being one of the following values: the opening states S_IHELLO_SENT and S_KEYING_SENT; the open state S_OPEN; the closing states S_NEARCLOSE and S_FARCLOSE_LINGER; and the closed states S_CLOSED and S_OPEN_FAILED; and

- The role of this end of the session, which is either Initiator or Responder.

Note: this diagram is only a summary of state transitions and their causing events, and is not a complete operational specification.

```
         rcv IIKeying Glare
         far prevails +-------------+   ultimate open timeout
       +--------------|S_IHELLO_SENT|-------------+
       |              +-------------+             |
       |                     |rcv RHello          |
       |                     |                    v
       |                     v             +-------------+
       |<-----------(duplicate session?)   |S_OPEN_FAILED|
       |         yes         |no           +-------------+
       |                     |                    ^
       | rcv IIKeying Glare  v                    |
       | far prevails +-------------+             |
       |<-------------|S_KEYING_SENT|-------------+
       |              +-------------+   ultimate open timeout
       |                     |rcv RIKeying
       |                     |
       |       rcv           v
       |   +-+ IIKeying  +--------+ rcv Close Request
       |   |X|---------->| S_OPEN |--------------------+
       |   +-+           +--------+                    |
       |                   |    |ABRUPT CLOSE          |
       |      ORDERLY CLOSE|    |or rcv Close Ack      |
       |                   |    |or rcv IIKeying       |
       |                   |    |   session override   |
       |                   |    +-------+              |
       |                   v            |              v
       |             +-----------+      |     +-----------------+
       |             |S_NEARCLOSE|      |     |S_FARCLOSE_LINGER|
       |             +-----------+      |     +-----------------+
       |      rcv Close Ack|            |              |rcv Close Ack
       |      or 90 seconds|            v              |or 19 seconds
       |                   |       +--------+          |
       |                   +------->|S_CLOSED|<---------+
       +-------------------------->|        |
                                   +--------+
```

Figure 8: Session state diagram

3.5.1. Startup

3.5.1.1. Normal Handshake

RTMFP sessions are established with a 4-way handshake in two round trips. The Initiator begins by sending an IHello to one or more candidate addresses for the desired destination endpoint. A

Responder statelessly sends an RHello in response. The first correct RHello received at the Initiator is selected; all others are ignored. The Initiator computes its half of the session keying and sends an IIKeying. The Responder receives the IIKeying, and if it is acceptable, computes its half of the session keying, at which point it can also compute the shared session keying and session nonces. The Responder creates a new S_OPEN session with the Initiator, and sends an RIKeying. The Initiator receives the RIKeying, and if it is acceptable, it computes the shared session keying and session nonces. The Initiator's session is now S_OPEN.

```
       .     Initiator                                Responder     .
                     | IHello                         |
                     |(EPD,Tag)                       |
       S_IHELLO_SENT |(SID=0)                         |
                     |------------------------------->|
                     |                                |
                     |                         RHello |
                     |              (Tag,Cookie,RCert)|
                     |                         (SID=0)|
                     |<-------------------------------|
       S_KEYING_SENT |                                |
                     | IIKeying                       |
                     |(ISID,Cookie,ICert,SKIC,ISig)   |
                     |(SID=0)                         |
                     |------------------------------->|
                     |                                |
                     |                       RIKeying |
                     |                (RSID,SKRC,RSig)|
                     |          (SID=ISID,Key=Default)| S_OPEN
                     |<-------------------------------|
              S_OPEN |                                |
                     |          S E S S I O N         |
                     |<-------------------(SID=ISID)--|
                     |--(SID=RSID)------------------->|
```

Figure 9: Normal handshake

In the following sections the handshake is detailed from the perspectives of the Initiator and Responder.

3.5.1.1.1. Initiator

The Initiator determines that a session is needed for an Endpoint Discriminator. The Initiator creates state for a new opening session and begins with a candidate endpoint address set containing at least one address. The new session is placed in the S_IHELLO_SENT state.

If the session does not move to the S_OPEN state before an ultimate open timeout, the session has failed and moves to the S_OPEN_FAILED state. The RECOMMENDED ultimate open timeout is 95 seconds.

The Initiator chooses a new, unique Tag not used by any currently opening session. It is RECOMMENDED that the Tag be cryptographically pseudorandom and be at least 8 bytes in length, so that it is hard to guess. The Initiator constructs an IHello chunk (Section 2.3.2) with the Endpoint Discriminator and the Tag.

While the Initiator is in the S_IHELLO_SENT state, it sends the IHello to each candidate endpoint address in the set, on a backoff schedule. The backoff SHOULD NOT be less than multiplicative with not less than 1.5 seconds added to the interval between each attempt. The backoff SHOULD be scheduled separately for each candidate address, since new candidates can be added over time.

If the Initiator receives a Redirect chunk (Section 2.3.5) with a Tag Echo matching this session, AND this session is in the S_IHELLO_SENT state, then for each redirect destination indicated in the Redirect: if the candidate endpoint address set contains fewer than REDIRECT_THRESHOLD addresses, add the indicated redirect destination to the candidate endpoint address set. REDIRECT_THRESHOLD SHOULD NOT be more than 24.

If the Initiator receives an RHello chunk (Section 2.3.4) with a Tag Echo matching this session, AND this session is in the S_IHELLO_SENT state, AND the Responder certificate matches the desired Endpoint Discriminator, AND the certificate is authentic according to the Cryptography Profile, then:

1. If the Canonical Endpoint Discriminator for the responder certificate matches the Canonical Endpoint Discriminator of another existing session in the S_KEYING_SENT or S_OPEN states, AND the certificate of the other opening session matches the desired Endpoint Discriminator, then: this session is a duplicate and SHOULD be aborted in favor of the other existing session; otherwise

2. Move to the S_KEYING_SENT state. Set DESTADDR, the far end address for the session, to the address from which this RHello was received. The Initiator chooses a new, unique receive session ID, not used by any other session, for the Responder to use when sending packets to the Initiator. It computes a Session Key Initiator Component appropriate to the responder's certificate according to the Cryptography Profile. Using this data and the cookie from the RHello, the Initiator constructs and signs an IIKeying chunk (Section 2.3.7).

While the Initiator is in the S_KEYING_SENT state, it sends the IIKeying to DESTADDR on a backoff schedule. The backoff SHOULD NOT be less than multiplicative with not less than 1.5 seconds added to the interval between each attempt.

If the Initiator receives an RIKeying chunk (Section 2.3.8) in a packet with this session's receive session identifier, AND this session is in the S_KEYING_SENT state, AND the signature in the chunk is authentic according to the far end's certificate (from the RHello), AND the Session Key Responder Component successfully combines with the Session Key Initiator Component and the near and far certificates to form the shared session keys and nonces according to the Cryptography Profile, then the session has opened successfully. The session moves to the S_OPEN state. The send session identifier is set from the RIKeying. Packet encryption, decryption, and verification now use the newly computed shared session keys, and the session nonces are available for application-layer cryptographic challenges.

3.5.1.1.2. Responder

On receipt of an IHello chunk (Section 2.3.2) with an Endpoint Discriminator that selects its identity, an endpoint SHOULD construct an RHello chunk (Section 2.3.4) and send it to the address from which the IHello was received. To avoid a potential resource exhaustion denial-of-service, the endpoint SHOULD NOT create any persistent state associated with the IHello. The endpoint MUST generate the cookie for the RHello in such a way that it can be recognized as authentic and valid when echoed in an IIKeying. The endpoint SHOULD use the address from which the IHello was received as part of the cookie generation formula. Cookies SHOULD be valid only for a limited time; that lifetime SHOULD NOT be less than 95 seconds (the recommended ultimate session open timeout).

On receipt of an FIHello chunk (Section 2.3.3) from a Forwarder (Section 3.5.1.5) where the Endpoint Discriminator selects its identity, an endpoint SHOULD do one of the following:

1. Compute, construct and send an RHello as though the FIHello was an IHello received from the indicated reply address; or

2. Construct and send an Implied Redirect (Section 2.3.5) to the FIHello's reply address; or

3. Ignore this FIHello.

On receipt of an IIKeying chunk (Section 2.3.7), if the cookie is not authentic or if it has expired, ignore this IIKeying; otherwise,

On receipt of an IIKeying chunk, if the cookie appears authentic but does not match the address from which the IIKeying's packet was received, perform the special processing at Cookie Change (Section 3.5.1.2); otherwise,

On receipt of an IIKeying with an authentic and valid cookie, if the certificate is authentic according to the Cryptography Profile, AND the signature in the chunk is authentic according to the far end's certificate and the Cryptography Profile, AND the Session Key Initiator Component is acceptable, then:

1. If the address from which this IIKeying was received corresponds to an opening session in the S_IHELLO_SENT or S_KEYING_SENT state, perform the special processing at Glare (Section 3.5.1.3); otherwise,

2. If the address from which this IIKeying was received corresponds to a session in the S_OPEN state, then:

    1. If the receiver was the Responder for the S_OPEN session and the session identifier, certificate, and Session Key Initiator Component are identical to those of the S_OPEN session, this IIKeying is a retransmission, so resend the S_OPEN session's RIKeying using the Default Session Key as specified below; otherwise,

    2. If the certificate from this IIKeying does not override the certificate of the S_OPEN session: ignore this IIKeying; otherwise,

    3. The certificate from this IIKeying overrides the certificate of the S_OPEN session; this is a new opening session from the same identity and the existing S_OPEN session is stale. Move the existing S_OPEN session to S_CLOSED and abort all of its flows (signaling exceptions to the user), then continue processing this IIKeying.

    Otherwise,

3. Compute a Session Key Responder Component and choose a new, unique receive session ID not used by any other session for the Initiator to use when sending packets to the Responder. Using this data, construct and, with the Session Key Initiator Component, sign an RIKeying chunk (Section 2.3.8). Using the Session Key Initiator and Responder Components and the near and far certificates, the Responder combines and computes the shared session keys and nonces according to the Cryptography Profile. The Responder creates a new session in the S_OPEN state, with the

far endpoint address DESTADDR taken from the source address of the packet containing the IIKeying and the send session identifier taken from the IIKeying. The Responder sends the RIKeying to the Initiator using the Default Session Key and the requested send session identifier. Packet encryption, decryption, and verification of all future packets for this session use the newly computed keys, and the session nonces are available for application-layer cryptographic challenges.

3.5.1.2. Cookie Change

In some circumstances, the Responder may generate an RHello cookie for an Initiator's address that isn't the address the Initiator would use when sending packets directly to the Responder. This can happen, for example, when the Initiator has multiple local addresses, and uses one to reach a Forwarder (Section 3.5.1.5) but another to reach the Responder.

Consider the following example:

```
    Initiator                    Forwarder                     Responder
    | IHello                         |                                 |
    |(Src=Ix)                        |                                 |
    |------------------------------->|                                 |
    |                                | FIHello                         |
    |                                |(RA=Ix)                          |
    |                                |-------------------------------->|
    |                                                                  |
    |                                                           RHello |
    |                                                       (Cookie:Ix)|
    |<-----------------------------------------------------------------|
    |                                                                  |
    | IIKeying                                                         |
    |(Cookie:Ix,Src=Iy)                                                |
    |----------------------------------------------------------------->|
    |                                                                  |
    |                                             RHello Cookie Change |
    |                                             (Cookie:Ix,Cookie:Iy)|
    |<-----------------------------------------------------------------|
    |                                                                  |
    | IIKeying                                                         |
    |(Cookie:Iy)                                                       |
    |----------------------------------------------------------------->|
    |                                                                  |
    |                                                         RIKeying |
    |<-----------------------------------------------------------------|
    |                                                                  |
    |<======================== S E S S I O N =========================>|
```

Figure 10: Handshake with Cookie Change

Initiator has two network interfaces, a first preferred interface with address Ix = 192.0.2.100:50000, and a second with address Iy = 198.51.100.101:50001. Responder has one interface with address Ry = 198.51.100.200:51000, on the same network as Initiator's second interface. Initiator uses its first interface to reach a Forwarder. Forwarder observes Initiator's address of Ix and sends a Forwarded IHello (Section 2.3.3) to Responder. Responder treats this as if it was an IHello from Ix, calculates a corresponding cookie, and sends an RHello to Ix. Initiator receives this RHello from Ry and selects that address as the destination for the session. It then sends an IIKeying, copying the cookie from the RHello. However, since the source of the RHello is Ry, on a network to which the Initiator is directly connected, Initiator uses its second interface Iy to send the IIKeying. Responder, on receiving the IIKeying, will compare the cookie to the expected value based on the source address of the packet, and since the IIKeying source doesn't match the IHello source used to generate the cookie, Responder will reject the IIKeying.

If Responder determines that it generated the cookie in the IIKeying but the cookie doesn't match the sender's address (for example, if the cookie is in two parts, with a first part generated independently of the Initiator's address, and a second part dependent on the address), Responder SHOULD generate a new cookie based on the address from which the IIKeying was received, and send an RHello Cookie Change chunk (Section 2.3.6) to the source of the IIKeying, using the session ID from the IIKeying and the Default Session Key.

If Initiator receives an RHello Cookie Change chunk for a session in the S_KEYING_SENT state, AND the old cookie matches the one originally sent to the Responder, then: Initiator adopts the new cookie, constructs and signs a new IIKeying chunk, and sends the new IIKeying to the Responder. Initiator SHOULD NOT change the cookie for a session more than once.

3.5.1.3. Glare

Glare occurs when two endpoints attempt to initiate sessions to each other concurrently. Glare is detected by receipt of a valid and authentic IIKeying from an endpoint address which is a destination for an opening session. Only one session is allowed between a pair of endpoints.

Glare is resolved by comparing the certificate in the received IIKeying with the near end's certificate. The Cryptography Profile defines a certificate comparison function to determine the prevailing endpoint when there is glare.

If the near end prevails, discard and ignore the received IIKeying. The far end will abort its opening session on receipt of IIKeying from the near end.

Otherwise, the far end prevails:

1. If the certificate in the IIKeying overrides the certificate associated with the near opening session according to the Cryptography Profile, then: abort and destroy the near opening session. Then,

2. Continue with normal Responder IIKeying processing (Section 3.5.1.1.2).

3.5.1.4. Redirector

```
       +-----------+           +------------+          +-----------+
       | Initiator |---------->| Redirector |          | Responder |
       |           |<----------|            |          |           |
       |           |           +------------+          |           |
       |           |<=================================>|           |
       +-----------+                                   +-----------+
```

Figure 11: Redirector

A Redirector acts like a name server for Endpoint Discriminators. An Initiator MAY use a Redirector to discover additional candidate endpoint addresses for a desired endpoint.

On receipt of an IHello chunk with an Endpoint Discriminator that does not select the Redirector's identity, the Redirector constructs and sends a Responder Redirect chunk (Section 2.3.5) back to the Initiator containing one or more additional candidate addresses for the indicated endpoint.

```
    Initiator                   Redirector                     Responder
    | IHello                         |                                 |
    |------------------------------->|                                 |
    |                                |                                 |
    |                       Redirect |                                 |
    |<-------------------------------|                                 |
    |                                                                  |
    | IHello                                                           |
    |----------------------------------------------------------------->|
    |                                                                  |
    |                                                           RHello |
    |<-----------------------------------------------------------------|
    |                                                                  |
    | IIKeying                                                         |
    |----------------------------------------------------------------->|
    |                                                                  |
    |                                                         RIKeying |
    |<-----------------------------------------------------------------|
    |                                                                  |
    |<======================== S E S S I O N =========================>|
```

Figure 12: Handshake using a Redirector

Deployment Design Note: Redirectors SHOULD NOT initiate new sessions to endpoints that might use the Redirector's address as a candidate for another endpoint, since the far end might interpret the Redirector's IIKeying as glare for the far end's initiation to the

other endpoint.

3.5.1.5. Forwarder

```
     +-----------+     +-----------+     +---+     +-----------+
     | Initiator |---->| Forwarder |<===>| N |<===>| Responder |
     |           |     +-----------+     | A |     |           |
     |           |<=====================>| T |<===>|           |
     +-----------+                       +---+     +-----------+
```

Figure 13: Forwarder

A Responder might be behind a NAT or firewall that doesn't allow inbound packets to reach the endpoint until it first sends an outbound packet for a particular far endpoint address.

A Forwarder's endpoint address MAY be a candidate address for another endpoint. A Responder MAY use a Forwarder to receive FIHello chunks sent on behalf of an Initiator.

On receipt of an IHello chunk with an Endpoint Discriminator that does not select the Forwarder's identity, if the Forwarder has an S_OPEN session with an endpoint whose certificate matches the desired Endpoint Discriminator, the forwarder constructs and sends an FIHello chunk (Section 2.3.3) to the selected endpoint over the S_OPEN session, using the Tag and Endpoint Discriminator from the IHello chunk and the source address of the packet containing the IHello for the corresponding fields of the FIHello.

On receipt of an FIHello chunk, a Responder might send an RHello or Implied Redirect to the original source of the IHello (Section 3.5.1.1.2), potentially allowing future packets to flow directly between the Initiator and Responder through the NAT or firewall.

```
    Initiator                    Forwarder           NAT       Responder
    | IHello                         |                |                |
    |------------------------------->|                |                |
    |                                | FIHello        |                |
    |                                |--------------->|--------------->|
    |                                                 |                |
    |                                                 |         RHello |
    |                                                 :<---------------|
    |<------------------------------------------------:                |
    |                                                 :                |
    | IIKeying                                        :                |
    |-------------------------------------------------:--------------->|
    |                                                 :                |
    |                                                 :       RIKeying |
    |                                                 :<---------------|
    |<------------------------------------------------:                |
    |                                                 :                |
    |<======================== S E S S I O N ========>:<==============>|
```

Figure 14: Forwarder handshake where Responder sends an RHello

```
    Initiator                    Forwarder            NAT       Responder
    | IHello                         |                 |                |
    |------------------------------->|                 |                |
    |                                | FIHello         |                |
    |                                |---------------->|--------------->|
    |                                                  |                |
    |                                                  |       Redirect |
    |                                                  | (Implied,RD={})|
    |                                                  :<---------------|
    |<-------------------------------------------------:                |
    |                                                  :                |
    | IHello                                           :                |
    |------------------------------------------------->:--------------->|
    |                                                  :                |
    |                                                  :         RHello |
    |                                                  :<---------------|
    |<-------------------------------------------------:                |
    |                                                  :                |
    | IIKeying                                         :                |
    |------------------------------------------------->:--------------->|
    |                                                  :                |
    |                                                  :       RIKeying |
    |                                                  :<---------------|
    |<-------------------------------------------------:                |
    |                                                  :                |
    |<======================== S E S S I O N ========>:<==============>|
```

Figure 15: Forwarder handshake where Responder sends an Implied Redirect

3.5.1.6. Redirector and Forwarder with NAT

```
            +---+       +---+       +---+      +---+      +---+
            | I |       | N |       | I |      | N |      | R |
            | n |------>| A |------>| n |      | A |      | e |
            | i |       | T |       | t |<====>| T |<====>| s |
            | t |<------|   |<------| r |      |   |      | p |
            | i |       |   |       | o |      |   |      | o |
            | a |       |   |       +---+      |   |      | n |
            | t |       |   |                  |   |      | d |
            | o |<=====>|   |<================>|   |<====>| e |
            | r |       |   |                  |   |      | r |
            +---+       +---+                  +---+      +---+
```

Figure 16: Introduction service for Initiator and Responder behind NATs

An Initiator and Responder might each be behind distinct NATs or

firewalls that don't allow inbound packets to reach the respective endpoints until each first sends an outbound packet for a particular far endpoint address.

An introduction service comprising Redirector and Forwarder functions may facilitate direct communication between endpoints each behind a NAT.

Responder is registered with the introduction service via an S_OPEN session to it. The service observes and records Responder's public NAT address as the DESTADDR of the S_OPEN session. The service MAY record other addresses for Responder, for example addresses Responder self-reports as being directly attached.

Initiator begins with an address of the introduction service as an initial candidate. The Redirector portion of the service sends a Responder Redirect to Initiator containing at least Responder's public NAT address as previously recorded. The Forwarder portion of the service sends a Forwarded IHello to Responder containing Initiator's public NAT address as observed as the source of the IHello.

Responder sends an RHello to Initiator's public NAT address in response to the FIHello. This will allow inbound packets to Responder through its NAT from Initiator's public NAT address.

Initiator sends an IHello to Responder's public NAT address in response to the Responder Redirect. This will allow inbound packets to Initiator through its NAT from Responder's public NAT address.

With transit paths created in both NATs, normal session startup can proceed.

```
    Initiator     NAT-I    Redirector+Forwarder     NAT-R      Responder
    |               |                |                |                |
    | IHello        |                |                |                |
    |(Dst=Intro)    |                |                |                |
    |-------------->|                |                |                |
    |               |--------------->|                |                |
    |               |                | FIHello        |                |
    |               |                |(RA=NAT-I-Pub)  |                |
    |               |                |--------------->|--------------->|
    |               |        Redirect |                |                |
    |               | (RD={NAT-R-Pub,|                |                |
    |               |           ...})|                |                |
    |<--------------|<---------------|                |                |
    |               |                                 |         RHello |
    |               |                                 | (Dst=NAT-I-Pub)|
    |               |                                 :<---------------|
    |               | (*)  <--------------------------:                |
    | IHello        |                                 :                |
    |(Dst=NAT-R-Pub)|                                 :                |
    |-------------->:                                 :                |
    |               :-------------------------------->:--------------->|
    |               :                                 :                |
    |               :                                 :         RHello |
    |               :                                 :<---------------|
    |<--------------:<--------------------------------:                |
    |               :                                 :                |
    | IIKeying      :                                 :                |
    |-------------->:                                 :                |
    |               :-------------------------------->:--------------->|
    |               :                                 :                |
    |               :                                 :       RIKeying |
    |               :                                 :<---------------|
    |<--------------:<--------------------------------:                |
    |               :                                 :                |
    |<=============>:<======== S E S S I O N ========>:<==============>|
```

Figure 17: Handshake with Redirector and Forwarder

At the point in Figure 17 marked (*), Responder's RHello from the FIHello might arrive at Initiator's NAT before or after Initiator's IHello is sent outbound to Responder's public NAT address. If it arrives before, it may be dropped by the NAT. If it arrives after, it will transit the NAT and trigger keying without waiting for another round trip time. The timing of this race depends, among other factors, on the relative distances of Initiator and Responder to each other and the introduction service.

3.5.1.7. Load Distribution and Fault Tolerance

```
            +---+    IHello/RHello    +-------------+
            | I |<------------------->| Responder 1 |
            | n |                     +-------------+
            | i |  SESSION  +-------------+
            | t |<=========>| Responder 2 |
            | i |           +-------------+
            | a |   IHello...                 +----------------+
            | t |-------------------------> X | Dead Responder |
            | o |                             +----------------+
            | r |  IHello/RHello   +-------------+
            |   |<---------------->| Responder N |
            +---+                  +-------------+
```

Figure 18: Parallel Open to Multiple Endpoints

Section 3.2 allows more than one endpoint to be selected by one Endpoint Discriminator. This will typically be the case for a set of servers, any of which could accommodate a connecting client.

Section 3.5.1.1.1 allows an Initiator to use multiple candidate endpoint addresses when starting a session, and specifies that the sender of the first acceptable RHello chunk to be received is selected to complete the session, with later responses ignored. An Initiator can start with the multiple candidate endpoint addresses, or it may learn them during startup from one or more Redirectors (Section 3.5.1.4).

Parallel open to multiple endpoints for the same Endpoint Discriminator combined with selection by earliest RHello can be used for load distribution and fault tolerance. The cost at each endpoint that is not selected is limited to receiving and processing an IHello, and generating and sending an RHello.

In one circumstance, multiple servers of similar processing and networking capacity may be located in near proximity to each other, such as in a data center. In this circumstance, a less heavily loaded server can respond to an IHello more quickly than more heavily loaded servers, and will tend to be selected by a client.

In another circumstance, multiple servers may be located in different physical locations, such as different data centers. In this circumstance, a server that is located nearer (in terms of network distance) to the client can respond earlier than more distant servers, and will tend to be selected by the client.

Multiple servers, in proximity or distant from one another, can form

a redundant pool of servers. A client can perform a parallel open to the multiple servers. In normal operation, the multiple servers will all respond, and the client will select one of them as described above. If one of the multiple servers fails, other servers in the pool can still respond to the client, allowing the client to successfully complete to an S_OPEN session with one of them.

3.5.2. Congestion Control

An RTMFP MUST implement congestion control and avoidance algorithms that are "TCP compatible", in accordance with Internet best current practice [RFC2914]. The algorithms SHOULD NOT be more aggressive in sending data than those described in TCP Congestion Control [RFC5681], and MUST NOT be more aggressive in sending data than the "slow start algorithm" described in RFC 5681 Section 3.1.

An endpoint maintains a transmission budget in the session information context of each S_OPEN session (Section 3.5), controlling the rate at which the endpoint sends data into the network.

For window-based congestion control and avoidance algorithms, the transmission budget is the congestion window, which is the amount of user data that is allowed to be outstanding, or in flight, in the network. Transmission is allowed when S_OUTSTANDING_BYTES (Section 3.5) is less than the congestion window (Section 3.6.2.3). See Appendix A for an experimental window-based congestion control algorithm for real-time and bulk data.

An endpoint avoids sending large bursts of data or packets into the network (Section 3.5.2.3).

A sending endpoint increases and decreases its transmission budget in response to acknowledgements (Section 3.6.2.4) and loss according to the congestion control and avoidance algorithms. Loss is detected by negative acknowledgement (Section 3.6.2.5) and timeout (Section 3.6.2.6).

Timeout is determined by the Effective Retransmission Timeout ERTO (Section 3.5.2.2). ERTO is measured using the Timestamp and Timestamp Echo packet header fields (Section 2.2.4).

A receiving endpoint acknowledges all received data (Section 3.6.3.4) to enable the sender to measure receipt of data, or lack thereof.

A receiving endpoint may be receiving time critical (or real-time) data from a first sender while receiving data from other senders. The receiving endpoint can signal its other senders (Section 2.2.4) to cause them to decrease the aggressiveness of their congestion

control and avoidance algorithms, in order to yield network capacity to the time critical data (Section 3.5.2.1).

3.5.2.1. Time Critical Reverse Notification

A sender can increase its transmission budget at a rate compatible with (but not exceeding) the "slow start algorithm" specified in RFC 5681 (with which the transmission rate is doubled every round trip when beginning or restarting transmission, until loss is detected). However, a sender MUST behave as though the slow start threshold SSTHRESH is clamped to 0 (disabling the slow start algorithm's exponential increase behavior) on a session where a Time Critical Reverse Notification (Section 2.2.4) indication has been received from the far end within the last 800 milliseconds, unless the sender is itself currently sending time critical data to the far end.

During each round trip, a sender SHOULD NOT increase the transmission budget by more than 0.5% or by 384 bytes per round trip (whichever is greater) on a session where a Time Critical Reverse Notification indication has been received from the far end within the last 800 milliseconds, unless the sender is itself currently sending time critical data to the far end.

3.5.2.2. Retransmission Timeout

RTMFP uses the Effective Retransmission Timeout ERTO to detect when a user data fragment has been lost in the network. The ERTO is typically calculated in a manner similar to that specified in Requirements for Internet Hosts - Communication Layers [RFC1122], and is a function of round trip time measurements and persistent timeout behavior.

The ERTO SHOULD be at least 250 milliseconds and SHOULD allow for the receiver to delay sending an acknowledgement for up to 200 milliseconds (Section 3.6.3.4.4). The ERTO MUST NOT be less than the round trip time.

To facilitate round trip time measurement, an endpoint MUST implement the Timestamp Echo facility:

- o On a session entering the S_OPEN state, initialize TS_RX_TIME to negative infinity, and TS_RX and TS_ECHO_TX to have no value.

- o On receipt of a packet in an S_OPEN session with the timestampPresent (Section 2.2.4) flag set, if the timestamp field in the packet is different than TS_RX: set TS_RX to the value of the timestamp field in the packet, and set TS_RX_TIME to the current time.

- When sending a packet to the far end in an S_OPEN session:

  1. Calculate TS_RX_ELAPSED = current time - TS_RX_TIME. If TS_RX_ELAPSED is more than 128 seconds, then set TS_RX and TS_ECHO_TX to have no value and do not include a timestamp echo; otherwise

  2. Calculate TS_RX_ELAPSED_TICKS to be the number of whole 4 millisecond periods in TS_RX_ELAPSED; then

  3. Calculate TS_ECHO = (TS_RX + TS_RX_ELAPSED_TICKS) MODULO 65536; then

  4. If TS_ECHO is not equal to TS_ECHO_TX, then: set TS_ECHO_TX to TS_ECHO, set the timestampEchoPresent flag, and set the timestampEcho field to TS_ECHO_TX.

The remainder of this section describes an OPTIONAL method for calculating the Effective Retransmission Timeout ERTO. Real-time applications and P2P mesh applications often require knowing the round trip time and RTT variance. This section additionally describes a method for measuring the round trip time and RTT variance, and calculating a smoothed round trip time.

Let the session information context contain additional variables:

- TS_TX: the last timestamp sent to the far end, initialized to have no value;

- TS_ECHO_RX: the last timestamp echo received from the far end, initialized to have no value;

- SRTT: the smoothed round-trip time, initialized to have no value;

- RTTVAR: the round-trip time variance, initialized to 0;

Initialize MRTO to 250 milliseconds.

Initialize ERTO to 3 seconds.

On sending a packet to the far end of an S_OPEN session, if the current send timestamp is not equal to TS_TX, then: set TS_TX to the current send timestamp, set the timestampPresent flag in the packet header, and set the timestamp field to TS_TX.

On receipt of a packet from the far end of an S_OPEN session, if the timestampEchoPresent flag is set in the packet header AND the timestampEcho field is not equal to TS_ECHO_RX, then:

1. Set TS_ECHO_RX to timestampEcho;

2. Calculate RTT_TICKS = (current send timestamp - timestampEcho) MODULO 65536;

3. If RTT_TICKS is greater than 32767, the measurement is invalid, so discard this measurement; otherwise

4. Calculate RTT = RTT_TICKS * 4 milliseconds;

5. If SRTT has a value, then calculate new values of RTTVAR and SRTT:

    1. RTT_DELTA = | SRTT - RTT |;

    2. RTTVAR = ((3 * RTTVAR) + RTT_DELTA) / 4;

    3. SRTT = ((7 * SRTT) + RTT) / 8;

6. If SRTT has no value, then set SRTT = RTT and RTTVAR = RTT / 2;

7. Set MRTO = SRTT + 4 * RTTVAR + 200 milliseconds;

8. Set ERTO to the greater of MRTO or 250 milliseconds.

A retransmission timeout occurs when the most recently transmitted user data fragment has remained outstanding in the network for ERTO. When this timeout occurs, increase ERTO on an exponential backoff with an ultimate backoff cap of 10 seconds:

1. Calculate ERTO_BACKOFF = ERTO * 1.4142;

2. Calculate ERTO_CAPPED to be the lesser of ERTO_BACKOFF and 10 seconds;

3. Set ERTO to the greater of ERTO_CAPPED and MRTO.

3.5.2.3. Burst Avoidance

An application's sending patterns may cause the transmission budget to grow to a large value but, at times, for there to be a comparatively small amount of data outstanding in the network. In this circumstance, especially with a window-based congestion avoidance algorithm, if the application then has a large amount of new data to send (for example, a new bulk data transfer), it could send data into the network all at once to fill the window. This kind of transmission burst can jam interfaces, links, and buffers, and is undesirable.

Accordingly, in any session, an endpoint SHOULD NOT send more than six packets containing user data between receiving acknowledgements or retransmission timeouts.

The following describes an OPTIONAL method to avoid bursting large numbers of packets into the network.

Let the session information context contain an additional variable DATA_PACKET_COUNT, initialized to 0.

Transmission of a user data fragment on this session is not allowed if DATA_PACKET_COUNT is greater than or equal to 6, regardless of any other allowance of the congestion control algorithm.

On transmission of a packet containing at least one User Data chunk (Section 2.3.11), set DATA_PACKET_COUNT = DATA_PACKET_COUNT + 1.

On receipt of an acknowledgement chunk (Section 2.3.13, Section 2.3.14), set DATA_PACKET_COUNT to 0.

On a retransmission timeout, set DATA_PACKET_COUNT to 0.

3.5.3. Address Mobility

Sessions are demultiplexed with a 32 bit session ID, rather than by endpoint address. This allows an endpoint's address to change during an S_OPEN session. This can happen, for example, when switching from a wireless to a wired network, or when moving from one wireless base station to another, or when a NAT restarts.

If the near end receives a valid packet for an S_OPEN session from a source address that doesn't match DESTADDR, the far end might have changed addresses. The near end SHOULD verify that the far end is definitively at the new address before changing DESTADDR. A suggested verification method is described in Section 3.5.4.2.

3.5.4. Ping

If an endpoint receives a Ping chunk (Section 2.3.9) in a session in the S_OPEN state, it SHOULD construct and send a Ping Reply chunk (Section 2.3.10) in response if possible, copying the message unaltered. A Ping Reply response SHOULD be sent as quickly as possible following receipt of a Ping. The semantics of a Ping's message is reserved for the sender; a receiver SHOULD NOT interpret the Ping's message.

Endpoints can use the mechanism of the Ping chunk and the expected Ping Reply for any purpose. This specification doesn't mandate any

specific constraints on the format or semantics of a Ping message. A Ping Reply MUST be sent only as a response to a Ping.

Receipt of a Ping Reply implies live bidirectional connectivity. This specification doesn't mandate any other semantics for a Ping Reply.

3.5.4.1. Keepalive

An endpoint can use Ping to test for live bidirectional connectivity, to test that the far end of a session is still S_OPEN, to keep NAT translations alive, and to keep firewall holes open.

An endpoint can use Ping to hasten detection by the far end of a near end address change.

An endpoint may declare a session to be defunct and dead after a persistent failure by the far end to return Ping Replies in response to Pings.

If used for these purposes, a Keepalive Ping SHOULD have an empty message.

A Keepalive Ping SHOULD NOT be sent more often than once per ERTO. If a corresponding Ping Reply is not received within ERTO of sending the Ping, ERTO SHOULD be increased according to Congestion Control (Section 3.5.2).

3.5.4.2. Address Mobility

This section describes an OPTIONAL but suggested method for processing and verifying a far end address change.

Let the session context contain additional variables MOB_TX_TS, MOB_RX_TS, and MOB_SECRET. MOB_TX_TS and MOB_RX_TS have initial values of negative infinity. MOB_SECRET should be a cryptographically pseudorandom value not less than 128 bits in length and known only to this end.

On receipt of a packet for an S_OPEN session, after processing all chunks in the packet: if the session is still S_OPEN, AND the source address of the packet does not match DESTADDR, AND MOB_TX_TS is at least one second in the past, then:

1. Set MOB_TX_TS to the current time;

2. Construct a Ping message comprising: a marking to indicate (to this end when returned in a Ping Reply) that it is a mobility

check (for example, the first byte being ASCII 'M' for "Mobility"), a timestamp set to MOB_TX_TS, and a cryptographic hash over the preceding as well as the address from which the packet was received and MOB_SECRET; and

3. Send this Ping to the address from which the packet was received, instead of DESTADDR.

On receipt of a Ping Reply in an S_OPEN session, if the Ping Reply's message satisfies all of these conditions:

- it has this end's expected marking to indicate it is a mobility check, and

- the timestamp in the message is not more than 120 seconds in the past, and

- the timestamp in the message is greater than MOB_RX_TS, and

- the cryptographic hash matches the expected value according to the contents of the message plus the source address of the packet containing this Ping Reply and MOB_SECRET,

then:

1. Set MOB_RX_TS to the timestamp in the message; and

2. Set DESTADDR to the source address of the packet containing this Ping Reply.

3.5.4.3. Path MTU Discovery

Packetization Layer Path MTU Discovery [RFC4821] describes a method for measuring the path MTU between communicating endpoints.

An RTMFP SHOULD perform path MTU discovery.

The method of RFC 4821 can be adapted for use in RTMFP by sending a probe packet comprising one of the Padding chunk types (type 0x00 or 0xff) and a Ping. The Ping chunk SHOULD come after the Padding chunk, to guard against a false positive response in case the probe packet is truncated.

3.5.5. Close

An endpoint may close a session at any time. Typically an endpoint will close a session when there have been no open flows in either direction for a time. In another circumstance, an endpoint may be

ceasing operation and will close all of its sessions even if they have open flows.

To close an S_OPEN session in a reliable and orderly fashion, an endpoint moves the session to the S_NEARCLOSE state.

On a session transitioning from S_OPEN to S_NEARCLOSE and every 5 seconds thereafter while still in the S_NEARCLOSE state, send a Session Close Request chunk (Section 2.3.17).

A session that has been in the S_NEARCLOSE state for at least 90 seconds (allowing time to retransmit the Session Close Request multiple times) SHOULD move to the S_CLOSED state.

On a session transitioning from S_OPEN to the S_NEARCLOSE, S_FARCLOSE_LINGER or S_CLOSED state: immediately abort and terminate all open or closing flows. Flows only exist in S_OPEN sessions.

To close an S_OPEN session abruptly, send a Session Close Acknowledgement chunk (Section 2.3.18), then move to the S_CLOSED state.

On receipt of a Session Close Request chunk for a session in the S_OPEN, S_NEARCLOSE, or S_FARCLOSE_LINGER states: send a Session Close Acknowledgement chunk; then, if the session is in the S_OPEN state: move to the S_FARCLOSE_LINGER state.

A session that has been in the S_FARCLOSE_LINGER state for at least 19 seconds (allowing time to answer 3 retransmissions of a Session Close Request) SHOULD move to the S_CLOSED state.

On receipt of a Session Close Acknowledgement chunk for a session in the S_OPEN, S_NEARCLOSE, or S_FARCLOSE_LINGER states: move to the S_CLOSED state.

3.6. Flows

A flow is a unidirectional communication channel in a session for transporting a correlated series of user messages from a sender to a receiver. Each end of a session may have zero or more sending flows to the other end. Each sending flow at one end has a corresponding receiving flow at the other end.

3.6.1. Overview

3.6.1.1. Identity

Flows are multiplexed in a session by a flow identifier. Each end of a session chooses its sending flow identifiers independently of the other end. The choice of similar flow identifiers by both ends does not imply an association. A sender MAY choose any identifier for any flow; therefore, a flow receiver MUST NOT ascribe any semantic meaning, role, or name to a flow based only on its identifier. There are no "well known" or reserved flow identifiers.

Bidirectional flow association is indicated at flow startup with the Return Flow Association option (Section 2.3.11.1.2). An endpoint can indicate that a new sending flow is in return (or response) to a receiving flow from the other end. A sending flow MUST NOT indicate more than one return association. A receiving flow can be specified as the return association for any number of sending flows. The return flow association, if any, is fixed for the lifetime of the sending flow. Note: Closure of one flow in an association does not automatically close other flows in the association except as specified in Section 3.6.3.1.

Flows are named with arbitrary user metadata. This specification doesn't mandate any particular encoding, syntax, or semantics for the user metadata except for the encoded size (Section 2.3.11.1.1); the user metadata is entirely reserved for the application. The user metadata is fixed for the lifetime of the flow.

3.6.1.2. Messages and Sequencing

Flows provide message-oriented framing. Large messages are fragmented for transport in the network. Receivers reassemble fragmented messages and only present complete messages to the user.

A sender queues messages on a sending flow one after another. A receiver can recover the original queuing order of the messages, even when they are reordered in transit by the network or as a result of loss and retransmission, by means of the messages' fragment sequence numbers. Flows are the basic units of message sequencing; each flow is sequenced independently of all other flows; inter-flow message arrival and delivery sequencing is not guaranteed.

Independent flow sequencing allows a sender to prioritize the transmission or retransmission of the messages of one flow over those of other flows in a session, allocating capacity from the transmission budget according to priority. RTMFP is designed for flows to be the basic unit of prioritization. In any flow, fragment sequence numbers are unique and monotonically increasing; that is, the fragment sequence numbers for any message MUST be greater than

the fragment sequence numbers of all messages previously queued in that flow. Receipt of fragments within a flow out of sequence number order creates discontiguous gaps at the receiver, causing it to send an acknowledgement for every packet and for the size of the encoded acknowledgements to grow. Therefore, for any flow, the sender SHOULD send lower sequence numbers first.

A sender can abandon a queued message at any time, even if some fragments of that message have been received by the other end. A receiver MUST be able to detect a gap in the flow when a message is abandoned; therefore, each message SHOULD take at least one sequence number from the sequence space even if no fragments for that message are ever sent. The sender will transmit the fragments of all messages not abandoned, and retransmit any lost fragments of all messages not abandoned, until all the fragments of all messages not abandoned are acknowledged by the receiver. A sender indicates a Forward Sequence Number (FSN) to instruct the receiver that sequence numbers less than or equal to the FSN will not be transmitted or retransmitted. This allows the receiver to move forward over gaps and continue sequenced delivery of completely received messages to the user. Any incomplete messages missing fragments with sequence numbers less than or equal to the FSN were abandoned by the sender and will never be completed. A gap indication MUST be communicated to the receiving user.

3.6.1.3. Lifetime

A sender begins a flow by sending user message fragments to the other end, and including the user metadata and, if any, the return flow association. The sender continues to include the user metadata and return flow association until the flow is acknowledged by the far end, at which point the sender knows that the receiver has received the user metadata and, if any, the return flow association. After that point, the flow identifier alone is sufficient.

Flow receivers SHOULD acknowledge all sequence numbers received for any flow, whether the flow is accepted or rejected. Flow receivers MUST NOT acknowledge sequence numbers higher than the FSN that were not received. Acknowledgements drive the congestion control and avoidance algorithms and therefore must be accurate.

An endpoint can reject a receiving flow at any time in the flow's lifetime. To reject the flow, the receiving endpoint sends a Flow Exception chunk (Section 2.3.16) immediately preceding every acknowledgement chunk for the rejected receiving flow.

An endpoint may eventually conclude and close a sending flow. The last sequence number of the flow is marked with the Final flag. The

sending flow is complete when all sequence numbers of the flow, including the final sequence number, have been cumulatively acknowledged by the receiver. The receiving flow is complete when every sequence number from the FSN to the final sequence number has been received. The sending flow and corresponding receiving flow at the respective ends hold the flow identifier of a completed flow in reserve for a time to allow delayed or duplicated fragments and acknowledgements to drain from the network without erroneously initiating a new receiving flow or erroneously acknowledging a new sending flow.

If a flow sender receives a Flow Exception indication from the other end, the flow sender SHOULD close the flow and abandon all of the undelivered queued messages. The flow sender SHOULD indicate an exception to the user.

3.6.2. Sender

Each sending flow comprises the flow-specific information context necessary to transfer that flow's messages to the other end. Each sending flow context includes at least:

- F_FLOW_ID: this flow's identifier;

- STARTUP_OPTIONS: the set of options to send to the receiver until this flow is acknowledged, including the User's Per-Flow Metadata and, if set, the Return Flow Association;

- SEND_QUEUE: the unacknowledged message fragments queued in this flow, initially empty; each message fragment entry comprising:

  * SEQUENCE_NUMBER: the sequence number of this fragment;

  * DATA: this fragment's user data;

  * FRA: the fragment control value for this message fragment, having one of the values enumerated for that purpose in User Data (Section 2.3.11);

  * ABANDONED: a boolean flag indicating whether this fragment has been abandoned;

  * SENT_ABANDONED: a boolean flag indicating whether this fragment was abandoned when sent;

  * EVER_SENT: a boolean flag indicating whether this fragment has been sent at least once, initially false;

  - NAK_COUNT: a count of the number of negative acknowledgements detected for this fragment, initially 0;

  - IN_FLIGHT: a boolean flag indicating whether this fragment is currently outstanding, or in flight, in the network, initially false;

  - TRANSMIT_SIZE: the size, in bytes, of the encoded User Data chunk (including the chunk header) for this fragment when it was transmitted into the network.

- F_OUTSTANDING_BYTES: the sum of the TRANSMIT_SIZE of each entry in SEND_QUEUE where entry.IN_FLIGHT is true;

- RX_BUFFER_SIZE: the most recent available buffer advertisement from the other end (Section 2.3.13, Section 2.3.14), initially 65536 bytes;

- NEXT_SN: the next sequence number to assign to a message fragment, initially 1;

- F_FINAL_SN: the sequence number assigned to the final message fragment of the flow, initially having no value;

- EXCEPTION: a boolean flag indicating whether an exception has been reported by the receiver, initially false;

- The state, at any time being one of the following values: the open state F_OPEN; the closing states F_CLOSING and F_COMPLETE_LINGER; and the closed state F_CLOSED.

Note: this diagram is only a summary of state transitions and their causing events, and is not a complete operational specification.

```
                                +--------+
                                | F_OPEN |
                                +--------+
                                     |CLOSE or
                                     |rcv Flow Exception
                                     |
                                     v
                                +---------+
                                |F_CLOSING|
                                +---------+
                                     |rcv Data Ack
                                     |  0..F_FINAL_SN
                                     v
                            +-----------------+
                            |F_COMPLETE_LINGER|
                            +-----------------+
                                     | 130 seconds
                                     v
                                 +--------+
                                 |F_CLOSED|
                                 +--------+
```

Figure 19: Sending flow state diagram

3.6.2.1. Startup

The application opens a new sending flow to the other end in an S_OPEN session. The implementation chooses a new flow ID that is not assigned to any other sending flow in that session in the F_OPEN, F_CLOSING, or F_COMPLETE_LINGER states. The flow starts in the F_OPEN state. The STARTUP_OPTIONS for the new flow is set with the User's Per-Flow Metadata (Section 2.3.11.1.1). If this flow is in return (or response) to a receiving flow from the other end, that flow's ID is encoded in a Return Flow Association (Section 2.3.11.1.2) option and added to STARTUP_OPTIONS. A new sending flow SHOULD NOT be opened in response to a receiving flow from the other end that is not in the RF_OPEN state when the sending flow is opened.

At this point the flow exists in the sender, but not the receiver. The flow begins when user data fragments are transmitted to the receiver. A sender can begin a flow in the absence of immediate user data by sending a Forward Sequence Number Update (Section 3.6.2.7.1), by queuing and transmitting a user data fragment that is already abandoned.

3.6.2.2. Queuing Data

The application queues messages in an F_OPEN sending flow for transmission to the far end. The implementation divides each message into one or more fragments for transmission in User Data chunks (Section 2.3.11). Each fragment MUST be small enough so that, if assembled into a Packet (Section 2.2.4) with a maximum-size common header, User Data chunk header, and, if not empty, this flow's STARTUP_OPTIONS, the Packet will not exceed the Path MTU (Section 3.5.4.3).

For each fragment, create a fragment entry and set fragmentEntry.SEQUENCE_NUMBER to flow.NEXT_SN, and increment flow.NEXT_SN by one. Set fragmentEntry.FRA according to the encoding in User Data chunks:

0 : This fragment is a complete message.

1 : This fragment is the first of a multi-fragment message.

2 : This fragment is the last of a multi-fragment message.

3 : This fragment is in the middle of a multi-fragment message.

Append fragmentEntry to flow.SEND_QUEUE.

3.6.2.3. Sending Data

A sending flow is ready to transmit if the SEND_QUEUE contains at least one entry that is eligible to send and if either RX_BUFFER_SIZE is greater than F_OUTSTANDING_BYTES or EXCEPTION is set to true.

A SEND_QUEUE entry is eligible to send if it is not IN_FLIGHT AND at least one of the following conditions holds:

o The entry is not ABANDONED; or

o The entry is the first one in the SEND_QUEUE; or

o The entry's SEQUENCE_NUMBER is equal to flow.F_FINAL_SN.

If the session's transmission budget allows, a flow that is ready to transmit is selected for transmission according to the implementation's prioritization scheme. The manner of flow prioritization is not mandated by this specification.

Trim abandoned messages from the front of the queue and find the Forward Sequence Number FSN:

1. While the SEND_QUEUE contains at least two entries, AND the first entry is not IN_FLIGHT, AND the first entry is ABANDONED: remove and discard the first entry from SEND_QUEUE;

2. If the first entry in the SEND_QUEUE is not abandoned: set FSN to entry.SEQUENCE_NUMBER - 1; otherwise

3. If the first entry in the SEND_QUEUE is IN_FLIGHT AND entry.SENT_ABANDONED is false: set FSN to entry.SEQUENCE_NUMBER - 1; otherwise

4. The first entry in the SEND_QUEUE is abandoned and is either not IN_FLIGHT or was already abandoned when sent: set FSN to entry.SEQUENCE_NUMBER.

The FSN MUST NOT be greater than any sequence number currently outstanding. The FSN MUST NOT be equal to any sequence number currently outstanding that was not abandoned when sent.

Assemble user data chunks for this flow into a packet to send to the receiver. While enough space remains in the packet and the flow is ready to transmit:

1. Starting at the head of the SEND_QUEUE, find the first eligible fragment entry;

2. Encode entry into a User Data chunk (Section 2.3.11) or, if possible (Section 3.6.2.3.2), a Next User Data chunk (Section 2.3.12);

3. If present, set chunk.flowID to flow.F_FLOW_ID;

4. If present, set chunk.sequenceNumber to entry.SEQUENCE_NUMBER;

5. If present, set chunk.fsnOffset to entry.SEQUENCE_NUMBER - FSN;

6. Set chunk.fragmentControl to entry.FRA;

7. Set chunk.abandon to entry.ABANDONED;

8. If entry.SEQUENCE_NUMBER equals flow.F_FINAL_SN: set chunk.final to true; else set chunk.final to false;

9. If any options are being sent with this chunk: set chunk.optionsPresent to true, assemble the options into the chunk, and assemble a Marker to terminate the option list;

10. If entry.ABANDONED is true, set chunk.userData to empty; otherwise set chunk.userData to entry.DATA;

11. If adding the assembled chunk to the packet would cause the packet to exceed the path MTU: do not assemble this chunk into the packet, enough space no longer remains in the packet, stop. Otherwise, continue:

12. Set entry.IN_FLIGHT to true;

13. Set entry.EVER_SENT to true;

14. Set entry.NAK_COUNT to 0;

15. Set entry.SENT_ABANDONED to entry.ABANDONED;

16. Set entry.TRANSMIT_SIZE to the size of the assembled chunk, including the chunk header;

17. Assemble this chunk into the packet; and

18. If this flow or entry is considered Time Critical (real-time), set the timeCritical flag in the packet header (Section 2.2.4).

Complete any other appropriate packet processing, and transmit the packet to the far end.

3.6.2.3.1. Startup Options

If STARTUP_OPTIONS is not empty, then when assembling the FIRST User Data chunk for this flow into a packet, add the encoded STARTUP_OPTIONS to that chunk's option list.

3.6.2.3.2. Send Next Data

The Next User Data chunk (Section 2.3.12) is a compact encoding for a user message fragment when multiple contiguous fragments are assembled into one packet. Using this chunk where possible can conserve space in a packet, potentially reducing transmission overhead or allowing additional information to be sent in a packet.

If, after assembling a user message fragment of a flow into a packet (Section 3.6.2.3), the next eligible fragment to be selected for assembly into that packet belongs to the same flow AND its sequence number is one greater than that of the fragment just assembled, it is RECOMMENDED that an implementation encode a Next User Data chunk instead of a User Data chunk.

The FIRST fragment of a flow assembled into a packet MUST be encoded as a User Data chunk.

3.6.2.4. Processing Acknowledgements

A Data Acknowledgement Bitmap chunk (Section 2.3.13) or a Data Acknowledgement Ranges chunk (Section 2.3.14) encodes the acknowledgement of receipt of one or more sequence numbers of a flow, as well as the receiver's current receive window advertisement.

On receipt of an acknowledgement chunk for a sending flow:

1. Set PRE_ACK_OUTSTANDING_BYTES to flow.F_OUTSTANDING_BYTES;

2. Set flow.STARTUP_OPTIONS to empty;

3. Set flow.RX_BUFFER_SIZE to chunk.bufferBytesAvailable;

4. For each sequence number encoded in the acknowledgement: if there is an entry in flow.SEND_QUEUE with that sequence number and its IN_FLIGHT is true, then: remove entry from flow.SEND_QUEUE; and

5. Notify the congestion control and avoidance algorithms that PRE_ACK_OUTSTANDING_BYTES - flow.F_OUTSTANDING_BYTES were acknowledged. Note that Negative Acknowledgements (Section 3.6.2.5) affect "TCP friendly" congestion control.

3.6.2.5. Negative Acknowledgement and Loss

A negative acknowledgement is inferred for an outstanding fragment if an acknowledgement is received for any other fragments sent after it in the same session.

An implementation SHOULD consider a fragment to be lost once that fragment receives three negative acknowledgements. A lost fragment is no longer outstanding in the network.

The following describes an OPTIONAL method for detecting negative acknowledgements.

Let the session track the order in which fragments are transmitted across all its sending flows by way of a monotonically increasing Transmission Sequence Number (TSN) recorded with each fragment queue entry each time that fragment is transmitted.

Let the session information context contain additional variables:

- NEXT_TSN: the next TSN to record with a fragment's queue entry when it is transmitted, initially 1;

- MAX_TSN_ACK: the highest acknowledged TSN, initially 0.

Let each fragment queue entry contain an additional variable TSN, initially 0, to track its transmission order.

On transmission of a message fragment into the network, set its entry.TSN to session.NEXT_TSN, and increment session.NEXT_TSN.

On acknowledgement of an outstanding fragment, if its entry.TSN is greater than session.MAX_TSN_ACK, set session.MAX_TSN_ACK to entry.TSN.

After processing all acknowledgements in a packet containing at least one acknowledgement, then for each sending flow in that session, for each entry in that flow's SEND_QUEUE, if entry.IN_FLIGHT is true and entry.TSN is less than session.MAX_TSN_ACK: increment entry.NAK_COUNT and notify the congestion control and avoidance algorithms that a negative acknowledgement was detected in this packet.

For each sending flow in that session, for each entry in that flow's SEND_QUEUE, if entry.IN_FLIGHT is true and entry.NAK_COUNT is at least 3, that fragment was lost in the network and is no longer considered to be in flight. Set entry.IN_FLIGHT to false. Notify the congestion control and avoidance algorithms of the loss.

3.6.2.6. Timeout

A fragment is considered lost and no longer in flight in the network if it has remained outstanding for at least ERTO.

The following describes an OPTIONAL method to manage transmission timeouts. This method REQUIRES that either Burst Avoidance (Section 3.5.2.3) is implemented, or that the implementation's congestion control and avoidance algorithms will eventually stop sending new fragments into the network if acknowledgements are persistently not received.

Let the session information context contain an alarm TIMEOUT_ALARM, initially unset.

On sending a packet containing at least one User Data chunk, set or reset TIMEOUT_ALARM to fire in ERTO.

On receiving a packet containing at least one acknowledgement, reset TIMEOUT_ALARM (if already set) to fire in ERTO.

When TIMEOUT_ALARM fires:

1. Set WAS_LOSS = false;

2. For each sending flow in the session, and for each entry in that flow's SEND_QUEUE:

   1. If entry.IN_FLIGHT is true: set WAS_LOSS = true; and

   2. Set entry.IN_FLIGHT to false.

3. If WAS_LOSS is true: perform ERTO backoff (Section 3.5.2.2); and

4. Notify the congestion control and avoidance algorithms of the timeout and, if WAS_LOSS is true, that there was loss.

3.6.2.7. Abandoning Data

The application can abandon queued messages at any time and for any reason. Example reasons include (but are not limited to): one or more fragments of a message have remained in the SEND_QUEUE for longer than a specified message lifetime; a fragment has been retransmitted more than a specified retransmission limit; a prior message on which this message depends (such as a key frame in a prediction chain) was abandoned and not delivered.

To abandon a message fragment, set its SEND_QUEUE entry's ABANDON flag to true. When abandoning a message fragment, abandon all fragments of the message to which it belongs.

An abandoned fragment MUST NOT be un-abandoned.

3.6.2.7.1. Forward Sequence Number Update

Abandoned data may leave gaps in the sequence number space of a flow. Gaps may cause the receiver to hold completely received messages for ordered delivery to allow for retransmission of the missing fragments. User Data chunks (Section 2.3.11) encode a Forward Sequence Number (FSN) to instruct the receiver that fragments with sequence numbers less than or equal to the FSN will not be transmitted or retransmitted.

When the receiver has gaps in the received sequence number space and no non-abandoned message fragments remain in the SEND_QUEUE, the sender SHOULD transmit a Forward Sequence Number Update (FSN Update) comprising a User Data chunk marked abandoned, whose sequence number is the FSN and whose fsnOffset is 0. An FSN Update allows the receiver to skip gaps that will not be repaired and deliver received

messages to the user. An FSN Update may be thought of as a transmission or retransmission of abandoned sequence numbers without actually sending the data.

The method described in Sending Data (Section 3.6.2.3) generates FSN Updates when appropriate.

3.6.2.8. Examples

```
    Sender
      |                    :
    1 |<---  Ack  ID=2, seq:0-16
    2 |--->  Data ID=2, seq#=25, fsnOff=9 (fsn=16)
    3 |--->  Data ID=2, seq#=26, fsnOff=10 (fsn=16)
    4 |<---  Ack  ID=2, seq:0-18
    5 |--->  Data ID=2, seq#=27, fsnOff=9 (fsn=18)
    6 |--->  Data ID=2, seq#=28, fsnOff=10 (fsn=18)
      |                    :

   There are 9 sequence numbers in flight with delayed acknowledgements.
```

Figure 20: Normal flow with no loss

```
    Sender
      |                   :
    1 |<---  Ack  ID=3, seq:0-30
    2 |--->  Data ID=3, seq#=45, fsnOff=15 (fsn=30)
    3 |<---  Ack  ID=3, seq:0-30, 32 (nack 31:1)
    4 |--->  Data ID=3, seq#=46, fsnOff=16 (fsn=30)
    5 |<---  Ack  ID=3, seq:0-30, 32, 34 (nack 31:2, 33:1)
    6 |<---  Ack  ID=3, seq:0-30, 32, 34-35 (nack 31:3=lost, 33:2)
    7 |--->  Data ID=3, seq#=47, fsnOff=15 (fsn=32, abandon 31)
    8 |<---  Ack  ID=3, seq:0-30, 32, 34-36 (nack 33:3=lost)
    9 |--->  Data ID=3, seq#=33, fsnOff=1 (fsn=32, retransmit 33)
   10 |<---  Ack  ID=3, seq:0-30, 32, 34-37
   11 |--->  Data ID=3, seq#=48, fsnOff=16 (fsn=32)
      |                   :
      |      (continues through seq#=59)
      |                   :
   12 |--->  Data ID=3, seq#=60, fsnOff=28(fsn=32)
   13 |<---  Ack  ID=3, seq:0-30, 34-46
   14 |--->  Data ID=3, seq#=61, fsnOff=29 (fsn=32)
   15 |<---  Ack  ID=3, seq:0-32, 34-47
   16 |--->  Data ID=3, seq#=62, fsnOff=30 (fsn=32)
   17 |<---  Ack  ID=3, seq:0-47
   18 |--->  Data ID=3, seq#=63, fsnOff=16 (fsn=47)
   19 |<---  Ack  ID=3, seq:0-49
   20 |--->  Data ID=3, seq#=64, fsnOff=15 (fsn=49)
      |                   :
   21 |<---  Ack  ID=3, seq:0-59
   22 |<---  Ack  ID=3, seq:0-59, 61 (nack 60:1)
   23 |<---  Ack  ID=3, seq:0-59, 61-62 (nack 60:2)
   24 |<---  Ack  ID=3, seq:0-59, 61-63 (nack 60:3=lost)
   25 |--->  Data ID=3, ABN=1, seq#=60, fsnOff=0 (fsn=60, abandon 60)
   26 |<---  Ack  ID=3, seq:0-59, 61-64
      |                   :
   27 |<---  Ack  ID=3, seq:0-64
```

Flow with sequence numbers 31, 33, and 60 lost in transit, and a pause at 64. 33 is retransmitted, 31 and 60 are abandoned. Note line 25 is a Forward Sequence Number Update (Section 3.6.2.7.1).

Figure 21: Flow with loss

3.6.2.9. Flow Control

The flow receiver advertises the amount of new data it's willing to accept from the flow sender with the bufferBytesAvailable derived field of an acknowledgement (Section 2.3.13, Section 2.3.14).

The flow sender MUST NOT send new data into the network if

flow.F_OUTSTANDING_BYTES is greater than or equal to the most recently received buffer advertisement, unless flow.EXCEPTION is true (Section 3.6.2.3).

3.6.2.9.1. Buffer Probe

The flow sender is suspended if the most recently received buffer advertisement is zero and the flow hasn't been rejected by the receiver; that is, while RX_BUFFER_SIZE is zero AND EXCEPTION is false. To guard against potentially lost acknowledgements that might reopen the receive window, a suspended flow sender SHOULD send a packet comprising a Buffer Probe chunk (Section 2.3.15) for this flow from time to time.

If the receive window advertisement transitions from non-zero to zero, the flow sender MAY send a Buffer Probe immediately and SHOULD send a probe within one second.

The initial period between Buffer Probes SHOULD be at least one second or ERTO, whichever is greater. The period between probes SHOULD increase over time, but the period between probes SHOULD NOT be more than one minute or ERTO, whichever is greater.

The flow sender SHOULD stop sending Buffer Probes if it is no longer suspended.

3.6.2.10. Exception

The flow receiver can reject the flow at any time and for any reason. The flow receiver sends a Flow Exception Report (Section 2.3.16) when it has rejected a flow.

On receiving a Flow Exception Report for a sending flow:

1. If the flow is F_OPEN, close the flow (Section 3.6.2.11) and notify the user that the far end reported an exception with the encoded exception code;

2. Set the EXCEPTION flag to true; and

3. For each entry in SEND_QUEUE: set entry.ABANDONED = true.

3.6.2.11. Close

A sending flow is closed by the user or as a result of an exception. To close an F_OPEN flow:

1. Move to the F_CLOSING state;

2. If the SEND_QUEUE is not empty, AND the tail entry of the SEND_QUEUE has a sequence number of NEXT_SN - 1, AND the tail entry.EVER_SENT is false: set F_FINAL_SN to entry.SEQUENCE_NUMBER; else

3. The SEND_QUEUE is empty, OR the tail entry does not have a sequence number of NEXT_SN - 1, OR the tail entry.EVER_SENT is true: enqueue a new SEND_QUEUE entry with entry.SEQUENCE_NUMBER = flow.NEXT_SN, entry.FRA = 0, and entry.ABANDONED = true, and set flow.F_FINAL_SN to entry.SEQUENCE_NUMBER.

An F_CLOSING sending flow is complete when its SEND_QUEUE transitions to empty, indicating that all sequence numbers including the FINAL_SN have been acknowledged by the other end.

When an F_CLOSING sending flow becomes complete, move to the F_COMPLETE_LINGER state.

A sending flow MUST remain in the F_COMPLETE_LINGER state for at least 130 seconds. After at least 130 seconds, move to the F_CLOSED state. The sending flow is now closed, its resources can be reclaimed, and its F_FLOW_ID MAY be used for a new sending flow.

3.6.3. Receiver

Each receiving flow comprises the flow-specific information context necessary to receive that flow's messages from the sending end and deliver completed messages to the user. Each receiving flow context includes at least:

- RF_FLOW_ID: this flow's identifier;

- SEQUENCE_SET: the set of all fragment sequence numbers seen in this receiving flow, whether received or abandoned, initially empty;

- RF_FINAL_SN: the final fragment sequence number of the flow, initially having no value;

- RECV_BUFFER: the message fragments waiting to be delivered to the user, sorted by sequence number in ascending order, initially empty; each message fragment entry comprising:

  * SEQUENCE_NUMBER: the sequence number of this fragment;

  - DATA: this fragment's user data; and

  - FRA: the fragment control value for this message fragment, having one of the values enumerated for that purpose in User Data (Section 2.3.11).

- BUFFERED_SIZE: the sum of the lengths of each fragment in RECV_BUFFER plus any additional storage overhead for the fragments incurred by the implementation, in bytes;

- BUFFER_CAPACITY: the desired maximum size for the receive buffer, in bytes;

- PREV_RWND: the most recent receive window advertisement sent in an acknowledgement, in 1024-byte blocks, initially having no value;

- SHOULD_ACK: whether or not an acknowledgement should be sent for this flow, initially false;

- EXCEPTION_CODE: the exception code to report to the sender when the flow has been rejected, initially 0;

- The state, at any time being one of the following values: the open state RF_OPEN; the closing states RF_REJECTED and RF_COMPLETE_LINGER; and the closed state RF_CLOSED.

```
    Note: this diagram is only a summary of state transitions and their
     causing events, and is not a complete operational specification.

                                      +-+
                                      |X|
                                      +-+
                                       |rcv User Data for
                                       |  no existing flow
                                       v
                                  +---------+
                                  | RF_OPEN |
                                  +---------+
             rcv all sequence numbers|   |user reject,
                     0..RF_FINAL_SN  |   |rcv bad option,
                                     |   |no metadata at open,
                                     |   |association specified
                                     |   |  but not F_OPEN at open
                                 +---+   |
                                 |       v
                                 |  +-----------+
                                 |  |RF_REJECTED|
                                 |  +-----------+
                                 |       |rcv all sequence numbers
                                 |       |  0..RF_FINAL_SN
                                 v       v
                            +------------------+
                            |RF_COMPLETE_LINGER|
                            +------------------+
                                     | 120 seconds
                                     v
                                +---------+
                                |RF_CLOSED|
                                +---------+
```

Figure 22: Receiving flow state diagram

3.6.3.1. Startup

A new receiving flow starts on receipt of a User Data chunk (Section 2.3.11) encoding a flow ID not belonging to any other receiving flow in the same session in the RF_OPEN, RF_REJECTED, or RF_COMPLETE_LINGER states.

On receipt of such a User Data chunk:

1.   Set temporary variables METADATA, ASSOCIATED_FLOWID, and ASSOCIATION to each have no value;

2. Create a new receiving flow context in this session, setting its RF_FLOW_ID to the flow ID encoded in the opening User Data chunk, and set to the RF_OPEN state;

3. If the opening User Data chunk encodes a User's Per-Flow Metadata option (Section 2.3.11.1.1), set METADATA to option.userMetadata;

4. If the opening User Data chunk encodes a Return Flow Association option (Section 2.3.11.1.2), set ASSOCIATED_FLOWID to option.flowID;

5. If METADATA has no value, the receiver MUST reject the flow (Section 3.6.3.7), moving it to the RF_REJECTED state;

6. If ASSOCIATED_FLOWID has a value, then if there is no sending flow in the same session with a flow ID of ASSOCIATED_FLOWID, the receiver MUST reject the flow, moving it to the RF_REJECTED state; otherwise set ASSOCIATION to the indicated sending flow;

7. If ASSOCIATION indicates a sending flow AND that sending flow's state is not F_OPEN, the receiver MUST reject this receiving flow, moving it to the RF_REJECTED state;

8. If the opening User Data chunk encodes any unrecognized option with a type code less than 8192 (Section 2.3.11.1), the receiver MUST reject the flow, moving it to the RF_REJECTED state;

9. If this new receiving flow is still RF_OPEN, then: notify the user that a new receiving flow has opened, including the METADATA and, if present, the ASSOCIATION, and set flow.BUFFER_CAPACITY according to the user;

10. Perform the normal data processing (Section 3.6.3.2) for the opening User Data chunk; and

11. Set this session's ACK_NOW to true.

3.6.3.2. Receiving Data

A User Data chunk (Section 2.3.11) or a Next User Data chunk (Section 2.3.12) encodes one fragment of a user data message of a flow, as well as the flow's Forward Sequence Number and potentially optional parameters (Section 2.3.11.1).

On receipt of a User Data or Next User Data chunk:

1. If chunk.flowID doesn't indicate an existing receiving flow in the same session in the RF_OPEN, RF_REJECTED, or RF_COMPLETE_LINGER state, perform the steps at Startup (Section 3.6.3.1) to start a new receiving flow;

2. Retrieve the receiving flow context for the flow indicated by chunk.flowID;

3. Set flow.SHOULD_ACK to true;

4. If the flow is RF_OPEN AND the chunk encodes any unrecognized option with a type code less than 8192 (Section 2.3.11.1), the flow MUST be rejected: notify the user of an exception, and reject the flow (Section 3.6.3.7), moving it to the RF_REJECTED state;

5. If the flow is not in the RF_OPEN state: set session.ACK_NOW to true;

6. If flow.PREV_RWND has a value and that value is less than 2 blocks, set session.ACK_NOW to true;

7. If chunk.abandon is true: set session.ACK_NOW to true;

8. If flow.SEQUENCE_SET has any gaps (that is, if it doesn't contain every sequence number from 0 through and including the highest sequence number in the set), set session.ACK_NOW to true;

9. If flow.SEQUENCE_SET contains chunk.sequenceNumber, then this chunk is a duplicate: set session.ACK_NOW to true;

10. If flow.SEQUENCE_SET doesn't contain chunk.sequenceNumber, AND chunk.final is true, AND flow.RF_FINAL_SN has no value, then: set flow.RF_FINAL_SN to chunk.sequenceNumber, and set session.ACK_NOW to true;

11. If the flow is in the RF_OPEN state, AND flow.SEQUENCE_SET doesn't contain chunk.sequenceNumber, AND chunk.abandon is false, then: create a new RECV_BUFFER entry for this chunk's data and set entry.SEQUENCE_NUMBER to chunk.sequenceNumber, entry.DATA to chunk.userData, and entry.FRA to chunk.fragmentControl, and insert this new entry into flow.RECV_BUFFER;

12. Add to flow.SEQUENCE_SET the range of sequence numbers from 0 through and including the chunk.forwardSequenceNumber derived field;

13. Add chunk.sequenceNumber to flow.SEQUENCE_SET;

14. If flow.SEQUENCE_SET now has any gaps, set session.ACK_NOW to true;

15. If session.ACK_NOW is false and session.DELACK_ALARM is not set: set session.DELACK_ALARM to fire in 200 milliseconds; and

16. Attempt delivery of completed messages in this flow's RECV_BUFFER to the user (Section 3.6.3.3).

After processing all chunks in a packet containing at least one User Data chunk, increment session.RX_DATA_PACKETS by one. If session.RX_DATA_PACKETS is at least two, set session.ACK_NOW to true.

A receiving flow that is not in the RF_CLOSED state is ready to send an acknowledgement if its SHOULD_ACK flag is set. Acknowledgements for receiving flows that are ready are sent either opportunistically by piggybacking on a packet that's already sending user data or an acknowledgement (Section 3.6.3.4.6), or when the session's ACK_NOW flag is set (Section 3.6.3.4.5).

3.6.3.3. Buffering and Delivering Data

A receiving flow's information context contains a RECV_BUFFER for reordering, reassembling, and holding the user data messages of the flow. Only complete messages are delivered to the user; an implementation MUST NOT deliver partially received messages except by special arrangement with the user.

Let the Cumulative Acknowledgement Sequence Number CSN be the highest number in the contiguous range of numbers in SEQUENCE_SET starting with 0. For example, if SEQUENCE_SET contains {0, 1, 2, 3, 5, 6}, the contiguous range starting with 0 is 0..3, so the CSN is 3.

A message is complete if all of its fragments are present in the RECV_BUFFER. The fragments of one message have contiguous sequence numbers. A message can either be a single fragment, whose fragment control value is 0-whole, or can be two or more fragments where the first's fragment control value is 1-begin, followed by zero or more fragments with control value 3-middle, and terminated by a last fragment with control value 2-end.

An incomplete message segment is a contiguous sequence of one or more fragments that do not form a complete message; that is, a 1-begin followed by zero or more 3-middle fragments but with no 2-end, or zero or more 3-middle fragments followed by a 2-end but with no 1-begin, or one or more 3-middle fragments with neither a 1-begin nor

a 2-end.

Incomplete message segments can either be in progress or abandoned. An incomplete segment is abandoned in the following cases:

- The sequence number of the segment's first fragment is less than or equal to the CSN AND that fragment's control value is not 1-begin; or

- The sequence number of the segment's last fragment is less than the CSN.

Abandoned message segments will never be completed, so they SHOULD be removed from the RECV_BUFFER to make room in the advertised receive window and the receiver's memory for messages that can be completed.

The user can suspend delivery of a flow's messages. A suspended receiving flow holds completed messages in its RECV_BUFFER until the user resumes delivery. A suspended flow can cause the receive window advertisement to go to zero even when the BUFFER_CAPACITY is non-zero; this is described in detail in Flow Control (Section 3.6.3.5).

When the receiving flow is not suspended, the original queuing order of the messages is recovered by delivering, in ascending sequence number order, complete messages in the RECV_BUFFER whose sequence numbers are less than or equal to the CSN.

The following describes a method for discarding abandoned message segments and delivering complete messages in original queueing order when the receiving flow is not suspended.

While the first fragment entry in the RECV_BUFFER has a sequence number less than or equal to CSN and delivery is still possible:

1. If entry.FRA is 0-whole: deliver entry.DATA to the user, and remove this entry from RECV_BUFFER; otherwise,

2. If entry.FRA is 2-end or 3-middle: this entry belongs to an abandoned segment, so remove and discard this entry from RECV_BUFFER; otherwise,

3. Entry.FRA is 1-begin. Let LAST_ENTRY be the last RECV_BUFFER entry that is part of this message segment (LAST_ENTRY can be entry if the segment has only one fragment so far). Then:

    1. If LAST_ENTRY.FRA is 2-end: this segment is a complete message, so concatenate the DATA fields of each fragment entry of this segment in ascending sequence number order and

deliver the complete message to the user, then remove the entries for this complete message from RECV_BUFFER; otherwise,

2. If LAST_ENTRY.SEQUENCE_NUMBER is less than CSN: this segment is incomplete and abandoned, so remove and discard the entries for this segment from RECV_BUFFER; otherwise,

3. LAST_ENTRY.SEQUENCE_NUMBER is equal to CSN and LAST_ENTRY.FRA is not 2-end: this segment is incomplete but still in progress. Ordered delivery is no longer possible until at least one more fragment is received. Stop.

If flow.RF_FINAL_SN has a value and is equal to CSN, AND RECV_BUFFER is empty: all complete messages have been delivered to the user, so notify the user that the flow is complete.

3.6.3.4. Acknowledging Data

A flow receiver SHOULD acknowledge all user data fragment sequence numbers seen in that flow. Acknowledgements drive the sender's congestion control and avoidance algorithms, clear data from the sender's buffers, and in some sender implementations clock new data into the network, and therefore must be accurate and timely.

3.6.3.4.1. Timing

For similar reasons as discussed in RFC 1122 Section 4.2.3.2 [RFC1122], it is advantageous to delay sending acknowledgements for a short time so that multiple data fragments can be acknowledged in a single transmission. However, it is also advantageous for a sender to receive timely notification about the receiver's disposition of the flow, particularly in unusual or exceptional circumstances, so that the circumstances can be addressed if possible.

Therefore, a flow receiver SHOULD send an acknowledgement for a flow as soon as is practical in any of the following circumstances:

- On receipt of a User Data chunk that starts a new flow;

- On receipt of a User Data or Next User Data chunk if the flow is not in the RF_OPEN state;

- On receipt of a User Data chunk where, before processing the chunk, the SEQUENCE_SET of the indicated flow does not contain every sequence number between 0 and the highest sequence number in the set (that is, if there was a sequence number gap before processing the chunk);

- On receipt of a User Data chunk where, after processing the chunk, the flow's SEQUENCE_SET does not contain every sequence number between 0 and the highest sequence number in the set (that is, if this chunk causes a sequence number gap);

- On receipt of a Buffer Probe for the flow;

- On receipt of a User Data chunk if the last acknowledgement sent for the flow indicated fewer than two bufferBlocksAvailable;

- On receipt of a User Data or Next User Data chunk for the flow if, after processing the chunk, the flow's BUFFER_CAPACITY is not at least 1024 bytes greater than BUFFERED_SIZE;

- On receipt of a User Data or Next User Data chunk for any sequence number that was already seen (that is, on receipt of a duplicate);

- On the first receipt of the final sequence number of the flow;

- On receipt of two packets in the session containing user data for any flows since an acknowledgement was last sent; the new acknowledgements being for the flows having any User Data chunks in the received packets (that is, for every second packet containing user data);

- After receipt of a User Data chunk for the flow, if an acknowledgement for any other flow is being sent (that is, consolidate acknowledgements);

- After receipt of a User Data chunk for the flow, if any user data for a sending flow is being sent in a packet and if there is space available in the same packet (that is, attempt to piggyback an acknowledgement with user data if possible);

- No longer than 200 milliseconds after receipt of a User Data chunk for the flow.

3.6.3.4.2. Size and Truncation

Including an encoded acknowledgement in a packet might cause the packet to exceed the path MTU. In that case:

- If the packet is being sent primarily to send an acknowledgement AND this is the first acknowledgement in the packet, truncate the acknowledgement so that the packet does not exceed the path MTU; otherwise

- The acknowledgement is being piggybacked in a packet with user data or with an acknowledgement for another flow: do not include this acknowledgement in the packet, and send it later.

3.6.3.4.3. Constructing

The Data Acknowledgement Bitmap chunk (Section 2.3.13) and Data Acknowledgement Ranges chunk (Section 2.3.14) encode a receiving flow's SEQUENCE_SET and its receive window advertisement. The two chunks are semantically equivalent; implementations SHOULD send whichever provides the most compact encoding of the SEQUENCE_SET.

When assembling an acknowledgement for a receiving flow:

1. If the flow's state is RF_REJECTED, first assemble a Flow Exception Report chunk (Section 2.3.16) for flow.flowID;

2. Choose the acknowledgement chunk type that most compactly encodes flow.SEQUENCE_SET;

3. Use the method described in Flow Control (Section 3.6.3.5) to determine the value for the acknowledgement chunk's bufferBlocksAvailable field;

3.6.3.4.4. Delayed Acknowledgement

As discussed in Acknowledging Data (Section 3.6.3.4.1), a flow receiver can delay sending an acknowledgement for up to 200 milliseconds after receiving user data. The method described in Receiving Data (Section 3.6.3.2) sets the session's DELACK_ALARM.

When DELACK_ALARM fires: set ACK_NOW to true.

3.6.3.4.5. Obligatory Acknowledgement

One or more acknowledgements should be sent as soon as is practical when the session's ACK_NOW flag is set. While the ACK_NOW flag is set:

1. Choose a receiving flow that is ready to send an acknowledgement;

2. If there is no such flow: there is no work to do, set ACK_NOW to false, set RX_DATA_PACKETS to 0, clear the DELACK_ALARM, and stop; otherwise

3. Start a new packet;

4. Assemble an acknowledgement for the flow and include it in the packet, truncating it if necessary so that the packet doesn't exceed the path MTU;

5. Set flow.SHOULD_ACK to false;

6. Set flow.PREV_RWND to the bufferBlocksAvailable field of the included acknowledgement chunk;

7. Attempt to piggyback acknowledgements for any other flows that are ready to send an acknowledgement into the packet, as described below; and

8. Send the packet.

3.6.3.4.6. Opportunistic Acknowledgement

When sending a packet with user data or an acknowledgement, any other receiving flows that are ready to send an acknowledgement should include their acknowledgements in the packet if possible.

To piggyback acknowledgements in a packet that is already being sent, where the packet contains user data or an acknowledgement: While there is at least one receiving flow that is ready to send an acknowledgement:

1. Assemble an acknowledgement for the flow;

2. If the acknowledgement cannot be included in the packet without exceeding the path MTU: the packet is full, stop; otherwise

3. Include the acknowledgement in the packet;

4. Set flow.SHOULD_ACK to false;

5. Set flow.PREV_RWND to the bufferBlocksAvailable field of the included acknowledgement chunk; and

6. If there are no longer any receiving flows in the session that are ready to send an acknowledgement: set session.ACK_NOW to false, set session.RX_DATA_PACKETS to 0, and clear session.DELACK_ALARM.

3.6.3.4.7. Example

```
   Receiver
    1 |<---  Data ID=3, seq#=29, fsnOff=11 (fsn=18)
    2 |<---  Data ID=3, seq#=30, fsnOff=12 (fsn=18)
    3 |--->  Ack  ID=3, seq:0-30
    4 |<---  Data ID=3, seq#=32, fsnOff=12 (fsn=20)
    5 |--->  Ack  ID=3, seq:0-30, 32
    6 |<---  Data ID=3, seq#=34, fsnOff=12 (fsn=22)
    7 |--->  Ack  ID=3, seq:0-30, 32, 34
      |                   :
    8 |<---  Data ID=3, seq#=46, fsnOff=16 (fsn=30)
    9 |--->  Ack  ID=3, seq:0-30, 32, 34-46
   10 |<---  Data ID=3, seq#=47, fsnOff=15 (fsn=32)
   11 |--->  Ack  ID=3, seq:0-32, 34-47
   12 |<---  Data ID=3, seq#=33, fsnOff=1 (fsn=32)
   13 |--->  Ack  ID=3, seq#=0-47
   14 |<---  Data ID=3, seq#=48, fsnOff=16 (fsn=32)
   15 |<---  Data ID=3, seq#=49, fsnOff=17 (fsn=32)
   16 |--->  Ack  ID=3, seq#=0-49
      |                   :

   Flow with sequence numbers 31 and 33 lost in transit, 31 abandoned
   and 33 retransmitted.
```

Figure 23

3.6.3.5. Flow Control

The flow receiver maintains a buffer for reassembling and reordering messages for delivery to the user (Section 3.6.3.3). The implementation and the user may wish to limit the amount of resources (including buffer memory) that a flow is allowed to use.

RTMFP provides a means for each receiving flow to govern the amount of data sent by the sender, by way of the bufferBytesAvailable derived field of acknowledgement chunks (Section 2.3.13, Section 2.3.14). This derived field indicates the amount of data that the sender is allowed to have outstanding in the network, until instructed otherwise. This amount is also called the receive window.

The flow receiver can suspend the sender by advertising a closed (zero length) receive window.

The user can suspend delivery of messages from the receiving flow (Section 3.6.3.3). This can cause the receive buffer to fill.

In order for progress to be made on completing a fragmented message or repairing a gap for sequenced delivery in a flow, the flow receiver MUST advertise at least one buffer block in an

acknowledgement if it is not suspended, even if the amount of data in the buffer exceeds the buffer capacity, unless the buffer capacity is 0. Otherwise, deadlock can occur, as the receive buffer will stay full and won't drain because of a gap or incomplete message, and the gap or incomplete message can't be repaired or completed because the sender is suspended.

The receive window is advertised in units of 1024-byte blocks. For example, advertisements for 1 byte, 1023 bytes, and 1024 bytes each require one block. An advertisement for 1025 bytes requires two blocks.

The following describes the RECOMMENDED method of calculating the bufferBlocksAvailable field of an acknowledgement chunk for a receiving flow:

1. If BUFFERED_SIZE is greater than or equal to BUFFER_CAPACITY: set ADVERTISE_BYTES to 0;

2. If BUFFERED_SIZE is less than BUFFER_CAPACITY: set ADVERTISE_BYTES to BUFFER_CAPACITY - BUFFERED_SIZE;

3. Set ADVERTISE_BLOCKS to CEIL(ADVERTISE_BYTES / 1024);

4. If ADVERTISE_BLOCKS is 0, AND BUFFER_CAPACITY is greater than 0, AND delivery to the user is not suspended: set ADVERTISE_BLOCKS to 1; and

5. Set the acknowledgement's bufferBlocksAvailable field to ADVERTISE_BLOCKS.

3.6.3.6. Receiving a Buffer Probe

A Buffer Probe chunk (Section 2.3.15) is sent by the flow sender (Section 3.6.2.9.1) to request the current receive window advertisement (in the form of an acknowledgement) from the flow receiver.

On receipt of a Buffer Probe chunk:

1. If chunk.flowID doesn't belong to a receiving flow in the same session in the RF_OPEN, RF_REJECTED, or RF_COMPLETE_LINGER state: ignore this Buffer Probe; otherwise,

2. Retrieve the receiving flow context for the flow indicated by chunk.flowID; then

3. Set flow.SHOULD_ACK to true; and

4. Set session.ACK_NOW to true.

3.6.3.7. Rejecting a Flow

A receiver can reject an RF_OPEN flow at any time and for any reason. To reject a receiving flow in the RF_OPEN state:

1. Move to the RF_REJECTED state;

2. Discard all entries in flow.RECV_BUFFER, as they are no longer relevant;

3. If the user rejected the flow, set flow.EXCEPTION_CODE to the exception code indicated by the user; otherwise the flow was rejected automatically by the implementation, so the exception code is 0;

4. Set flow.SHOULD_ACK to true; and

5. Set session.ACK_NOW to true.

The receiver indicates that it has rejected a flow by sending a Flow Exception Report chunk (Section 2.3.16) with every acknowledgement (Section 3.6.3.4.3) for a flow in the RF_REJECTED state.

3.6.3.8. Close

A receiving flow is complete when every sequence number from 0 through and including the final sequence number has been received; that is, when flow.RF_FINAL_SN has a value and flow.SEQUENCE_SET contains every sequence number from 0 through flow.RF_FINAL_SN, inclusive.

When an RF_OPEN or RF_REJECTED receiving flow becomes complete, move to the RF_COMPLETE_LINGER state, set flow.SHOULD_ACK to true, and set session.ACK_NOW to true.

A receiving flow SHOULD remain in the RF_COMPLETE_LINGER state for 120 seconds. After 120 seconds, move to the RF_CLOSED state. The receiving flow is now closed, and its resources can be reclaimed once all complete messages in flow.RECV_BUFFER have been delivered to the user (Section 3.6.3.3). The same flow ID might be used for a new flow by the sender after this point.

Discussion: The flow sender detects that the flow is complete on receiving an acknowledgement of all fragment sequence numbers of the

flow. This can't happen until after the receiver has detected that the flow is complete and acknowledged all of the sequence numbers. The receiver's RF_COMPLETE_LINGER period is two minutes (one Maximum Segment Lifetime (MSL)), which allows any in flight packets to drain from the network without being misidentified, and gives the sender an opportunity to retransmit any sequence numbers if the completing acknowledgement is lost. The sender's F_COMPLETE_LINGER period is at least two minutes plus 10 seconds, and doesn't begin until the completing acknowledgement is received; therefore, the same flow identifier won't be re-used by the flow sender for a new sending flow for at least 10 seconds after the flow receiver has closed the receiving flow context. This ensures correct operation independent of network delay and even when the sender's clock runs up to 8 percent faster than the receiver's.

4. IANA Considerations

This memo specifies chunk type code values (Section 2.3) and User Data option type code values (Section 2.3.11.1). These type code values are assigned and maintained by Adobe. Therefore, this memo has no IANA actions.

5. Security Considerations

This memo specifies a general framework that can be used to establish a confidential and authenticated session between endpoints. A Cryptography Profile, not specified herein, defines the cryptographic algorithms, data formats, and semantics as used within this framework. Designing a Cryptography Profile to ensure that communications are protected to the degree required by the application-specific threat model is outside the scope of this specification.

A block cipher in CBC mode is RECOMMENDED for packet encryption (Section 2.2.3). An attacker can predict the values of some fields from one plain RTMFP packet to the next, or predict that some fields may be the same from one packet to the next. This SHOULD be considered in choosing and implementing a packet encryption cipher and mode.

The well-known Default Session Key of a Cryptography Profile serves multiple purposes, including: to scramble session startup packets to protect interior fields from undesirable modification by middleboxes such as NATs; to increase the effort required for casual passive observation of startup packets; to allow for different applications of RTMFP using different Default Session Keys to (intentionally or

not) share network transport addresses without interference. The Default Session Key, being well-known, MUST NOT be construed to contribute to the security of session startup; session startup is essentially in the clear.

Section 3.5.4.2 describes an OPTIONAL method for processing a change of network address of a communicating peer. Securely processing address mobility using that or any substantially similar method REQUIRES at least that the Packet Encryption function of the Cryptography Profile (Section 2.2.3) employs a cryptographic verification mechanism comprising secret information known only to the two endpoints. Without this constraint, that or any substantially similar method becomes "session hijacking support".

Flows and packet fragmentation imply semantics that could cause unbounded resource utilization in receivers, causing a denial of service. Implementations SHOULD guard against unbounded or excessive resource use, and abort sessions that appear abusive.

A rogue but popular Redirector (Section 3.5.1.4) could direct session Initiators to flood a victim address or network with Initiator Hello packets, potentially causing a denial of service.

An attacker that can passively observe an IHello and that possesses a certificate matching the Endpoint Discriminator (without having to know the private key, if any, associated with it) can deny the Initiator access to the desired Responder by sending an RHello before the desired Responder does, since only the first received RHello is selected by the Initiator. The attacker needn't forge the desired Responder's source address, since the RHello is selected based on the tag echo and not the packet's source address. This can simplify the attack in some network or host configurations.

An attacker that can passively observe and record the packets of an established session can use traffic analysis techniques to infer the start and completion of flows without decrypting the packets. The User Data fragments of flows have unique sequence numbers, so flows are immune to replay while they are open. However, once a flow has completed and the linger period has concluded, the attacker could replay the recorded packets, opening a new flow in the receiver and duplicating the flow's data, which might have undesirable effects in the receiver's application. The attacker could also infer that a new flow has begun reusing the recorded flow's identifier, and replay the final sequence number or any of the other fragments in the flow, potentially denying or interfering with legitimate traffic to the receiver. Therefore, the data integrity aspect of packet encryption SHOULD comprise anti-replay measures.

6. Acknowledgements

Special thanks go to Matthew Kaufman for his contributions to the creation and design of RTMFP.

Thanks to Jari Arkko, Ben Campbell, Wesley Eddy, Stephen Farrell, Philipp Hancke, Bela Lubkin, Hilarie Orman, Richard Scheffenegger, and Martin Stiemerling for their detailed reviews of this memo.

7. References

7.1. Normative References

[RFC0768] Postel, J., "User Datagram Protocol", STD 6, RFC 768, August 1980.

[RFC0791] Postel, J., "Internet Protocol", STD 5, RFC 791, September 1981.

[RFC1122] Braden, R., "Requirements for Internet Hosts - Communication Layers", STD 3, RFC 1122, October 1989.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

[RFC2460] Deering, S. and R. Hinden, "Internet Protocol, Version 6 (IPv6) Specification", RFC 2460, December 1998.

[RFC2914] Floyd, S., "Congestion Control Principles", BCP 41, RFC 2914, September 2000.

[RFC4821] Mathis, M. and J. Heffner, "Packetization Layer Path MTU Discovery", RFC 4821, March 2007.

[RFC5681] Allman, M., Paxson, V., and E. Blanton, "TCP Congestion Control", RFC 5681, September 2009.

[CBC] Dworkin, M., "Recommendation for Block Cipher Modes of Operation", NIST Special Publication 800-38A, December 2001, <http://csrc.nist.gov/publications/nistpubs/800-38a/sp800-38a.pdf>.

7.2. Informative References

[RFC5389] Rosenberg, J., Mahy, R., Matthews, P., and D. Wing, "Session Traversal Utilities for NAT (STUN)", RFC 5389, October 2008.

[ScalableTCP]
Kelly, T., "Scalable TCP: Improving Performance in Highspeed Wide Area Networks", December 2002, <http://datatag.web.cern.ch/datatag/papers/pfldnet2003-ctk.pdf>.

Appendix A. Example Congestion Control Algorithm

Section 3.5.2 mandates that an RTMFP use TCP-compatible congestion control, but allows flexibility in exact implementation within certain limits. This section describes an experimental window-based congestion control algorithm that is appropriate for real-time and bulk data transport in RTMFP. The algorithm includes slow-start and congestion avoidance phases including modified increase and decrease parameters. These parameters are further adjusted according to whether real-time data is being sent and whether time-critical reverse notifications are received.

A.1. Discussion

RFC 5681 defines the standard window-based congestion control algorithms for TCP. These algorithms are appropriate for delay-insensitive bulk data transport, but have undesirable behaviors for delay- and loss-sensitive applications. Among the undesirable behaviors are the cutting of the congestion window in half during a loss event, and the rapidity of the slow start algorithm's exponential growth. Cutting the congestion window in half requires a large channel headroom to support a real-time application, and can cause a large amount of jitter from sender-side buffering. Doubling the congestion window during the slow-start phase can lead to the congestion window temporarily growing to twice the size it should be, causing a period of excessive loss in the path.

We found that a number of deployed TCP implementations use the method of equation 3 from RFC 5681 Section 3.1, which, when combined with the recommended behavior of acknowledging every other packet, causes the congestion window to grow at approximately half the rate as the recommended method specifies. In order to compete fairly with these deployed TCPs, we choose 768 bytes per round trip as the increment during the normal congestion avoidance phase, which is approximately half of the typical maximum segment size of 1500 bytes while also being easily subdivided.

The sender may be sending real-time data to the far end. When sending real-time data, a smoother response to congestion is desired while still competing with reasonable fairness to other flows in the Internet. In order to scale the sending rate quickly, the slow start algorithm is desired, but slow start's normal rate of increase can

cause excessive loss in the last round trip. Accordingly, slow start's exponential increase rate is adjusted to double every approximately 3 round trips instead of every round trip. The multiplicative decrease cuts the congestion window by one eighth on loss to maintain a smoother sending rate. The additive increase is done at half the normal rate (incrementing at 384 bytes per round trip), both to compensate for the less aggressive loss response and to probe the path capacity more gently.

The far end may report that it is receiving real-time data from other peers, or the sender may be sending real-time data to other far ends. In these circumstances (if not sending real-time data to this far end) it is desirable to respond differently than the standard TCP algorithms specify, both to yield capacity to the real-time flows and to avoid excessive losses while probing the path capacity. Slow start's exponential increase is disabled and the additive increase is done at half the normal rate (incrementing at 384 bytes per round trip). Multiplicative decrease is left at cutting by half to yield to other flows.

Since real-time messages may be small, and sent regularly, it is advantageous to spread congestion window increases out across the round-trip time instead of doing them all at once. We divide the round-trip into 16 segments with an additive increase of a useful size (48 bytes) per segment.

Scalable TCP [ScalableTCP] describes experimental methods of modifying the additive increase and multiplicative decrease of the congestion window in large delay-bandwidth scenarios. The congestion window is increased by 1% each round trip and decreased by one eighth on loss in the congestion avoidance phase in certain circumstances (specifically, when a 1% increase is larger than the normal additive-increase amount). Those methods are adapted here. The scalable increase amount is 48 bytes for every 4800 bytes acknowledged, to spread the increase out over the round-trip. The congestion window is decreased by one eighth on loss when it is at least 67200 bytes per round trip, which is seven eighths of 76800 (the point at which 1% is greater than 768 bytes per round trip). When sending real-time data to the far end, the scalable increase is 1% or 384 bytes per round trip, whichever is greater. Otherwise, when notified that the far end is receiving real-time data from other peers, the scaled increase is adjusted to 0.5% or 384 bytes per round trip, whichever is greater.

A.2. Algorithm

Let SMSS denote the Sender Maximum Segment Size [RFC5681], for example 1460 bytes. Let CWND_INIT denote the Initial Congestion

Window (IW) according to RFC 5681 Section 3.1, for example 4380 bytes. Let CWND_TIMEDOUT denote the congestion window after a timeout indicating lost data, being 1*SMSS (for example 1460 bytes).

Let the session information context contain additional variables:

- CWND: the Congestion Window, initialized to CWND_INIT;

- SSTHRESH: the Slow Start Threshold, initialized to positive infinity;

- ACKED_BYTES_ACCUMULATOR: a count of acknowledged bytes, initialized to 0;

- ACKED_BYTES_THIS_PACKET: a count of acknowledged bytes observed in the current packet;

- PRE_ACK_OUTSTANDING: the number of bytes outstanding in the network before processing any acknowledgements in the current packet;

- ANY_LOSS: an indication to whether any loss has been detected in the current packet;

- ANY_NAKS: an indication to whether any negative acknowledgements have been detected in the current packet;

- ANY_ACKS: an indication to whether any acknowledgement chunks have been received in the current packet;

Let FASTGROW_ALLOWED indicate whether the congestion window is allowed to grow at the normal rate versus a slower rate, being FALSE if a Time Critical Reverse Notification has been received on this session within the last 800 milliseconds (Section 2.2.4, Section 3.5.2.1) or if a Time Critical Forward Notification has been sent on ANY session in the last 800 milliseconds, and otherwise being TRUE.

Let TC_SENT indicate whether a Time Critical Forward Notification has been sent on this session within the last 800 milliseconds.

Implement the method of Section 3.6.2.6 to manage transmission timeouts, including setting the TIMEOUT_ALARM.

On being notified that the TIMEOUT_ALARM has fired, perform the function in Figure 24:

```
on TimeoutNotification(WAS_LOSS):
    set SSTHRESH to MAX(SSTHRESH, CWND * 3/4).
    set ACKED_BYTES_ACCUMULATOR to 0.
    if WAS_LOSS is TRUE:
        set CWND to CWND_TIMEDOUT.
    else:
        set CWND to CWND_INIT.
```

Figure 24: Pseudocode for handling a timeout notification

Before processing each received packet in this session:

1. Set ANY_LOSS to FALSE;

2. Set ANY_NAKS to FALSE;

3. Set ACKED_BYTES_THIS_PACKET to 0; and

4. Set PRE_ACK_OUTSTANDING to S_OUTSTANDING_BYTES.

On notification of loss (Section 3.6.2.5): set ANY_LOSS to TRUE.

On notification of negative acknowledgement (Section 3.6.2.5): set ANY_NAKS to TRUE.

On notification of acknowledgement of data (Section 3.6.2.4): set ANY_ACKS to TRUE, and add the count of acknowledged bytes to ACKED_BYTES_THIS_PACKET.

After processing all chunks in each received packet for this session, perform the function in Figure 25:

```
if ANY_LOSS is TRUE:
    if (TC_SENT is TRUE) OR (PRE_ACK_OUTSTANDING > 67200 AND \
    FASTGROW_ALLOWED is TRUE):
        set SSTHRESH to MAX(PRE_ACK_OUTSTANDING * 7/8, CWND_INIT).
    else:
        set SSHTRESH to MAX(PRE_ACK_OUTSTANDING * 1/2, CWND_INIT).
    set CWND to SSTHRESH.
    set ACKED_BYTES_ACCUMULATOR to 0.
else if (ANY_ACKS is TRUE) AND (ANY_NAKS is FALSE) AND \
(PRE_ACK_OUTSTANDING >= CWND):
    set var INCREASE to 0.
    var AITHRESH.
    if FASTGROW_ALLOWED is TRUE:
        if CWND < SSTHRESH:
            set INCREASE to ACKED_BYTES_THIS_PACKET.
        else:
            add ACKED_BYTES_THIS_PACKET to ACKED_BYTES_ACCUMULATOR.
            set AITHRESH to MIN(MAX(CWND / 16, 64), 4800).
            while ACKED_BYTES_ACCUMULATOR >= AITHRESH:
                subtract AITHRESH from ACKED_BYTES_ACCUMULATOR.
                add 48 to INCREASE.
    else FASTGROW_ALLOWED is FALSE:
        if CWND < SSTHRESH AND TC_SENT is TRUE:
            set INCREASE to CEIL(ACKED_BYTES_THIS_PACKET / 4).
        else:
            var AITHRESH_CAP.
            if TC_SENT is TRUE:
                set AITHRESH_CAP to 2400.
            else:
                set AITHRESH_CAP to 4800.
            add ACKED_BYTES_THIS_PACKET to ACKED_BYTES_ACCUMULATOR.
            set AITHRESH to MIN(MAX(CWND / 16, 64), AITHRESH_CAP).
            while ACKED_BYTES_ACCUMULATOR >= AITHRESH:
                subtract AITHRESH from ACKED_BYTES_ACCUMULATOR.
                add 24 to INCREASE.
    set CWND to MAX(CWND + MIN(INCREASE, SMSS), CWND_INIT).
```

Figure 25: Pseudocode for congestion window adjustment after processing a packet

Author's Address

Michael C. Thornburgh
Adobe Systems Incorporated
345 Park Avenue
San Jose, CA 95110-2704
US

Phone: +1 408 536 6000
Email: mthornbu@adobe.com
URI: http://www.adobe.com/