ScreenView Protocol

Josh Brown

December 21, 2021

Abstract

ScreenView is suite of cryptographical and application level networking protocols culminating to create a zero configuration end to end encrypted remote screen viewing and controlling software. ScreenView aims to replace TeamViewer, RDP, and VNC for many use cases while being more performant and more secure. ScreenView requires little setup an is just as easy or easier than other solutions. ScreenView defines four different layers of protocols, each encapsulating all the layers below it. Cryptography for communication between peers and the server is based upon TLS 1.3 and Wireguard. End-to-end cryptography used for ALL communication between peers is based upon TLS-SRP. ScreenView end-to-end cryptography prevents man-in-the-middle attacks even if the intermediary server is compromised, unlike TeamViewer. Screen data is sent over UDP to achieve superior performance than TCP based solutions such as VNC. All UDP packets must be authenticated with keys established over TCP before a response is made by the server preventing amplification attacks. A congestion control mechanism is used to handle low bandwidth and poor networking conditions. Finally, ScreenView supports advanced use cases including file transfer, multiple displays, sharing specific windows, shared whiteboards, and clipboard transfer.

Contents

1 Definitions

The key words ”MUST”, ”MUST NOT”, ”REQUIRED”, ”SHALL”, ”SHALL NOT”, ”SHOULD”, ”SHOULD NOT”, ”RECOMMENDED”, ”MAY”, and ”OPTIONAL” in this document are to be interpreted as described in RFC 2119.

2 Introduction

This section describes the high level overview of ScreenView. Terms used in this section are defined in other sections.

2.1 Application Layers

The table belows is an abstract diagram of the different layers of ScreenView. Each layer encapsulates all the below layers.

```
Transport Layer
| Server Encryption Layer
|| Server Communication Layer
||| E2EE (Peer ↔ Peer) Encryption Layer
|||| Host ↔ Client Communication Layer
```

The Transport Layer is the OSI Transport Layer level protocol for networking (TCP or UDP). The Server Encryption Layer provides security between the Server and a Peer. The Server Communication Layer facilities communication between Peers and the Client. The E2EE (Peer ↔ Peer) Encryption Layer provides end-to-end encryption between Peers and other Peers. The Host ↔ Client Communication Layer facilities communication between Peers.

2.2 Overview

To begin, a Peer connects to the server over TCP and commences the Server Encryption Layer protocol for TCP defined. Next the Server Communication Layer is used. All Server Communication Layer messages are encrypted, authenticated, and encapsulated in messages defined in the Server Encryption Layer. Once a session is established between two peers, the E2EE (Peer ↔ Peer) Encryption Layer commences. All E2EE (Peer ↔ Peer) Encryption messages are encapsulated in Server Communication Layer messages. Finally, the Host ↔ Client Communication Layer commences as defined in 6. All Host ↔ Client Communication Layer messages are encrypted, authenticated, and encapsulated in E2EE (Peer ↔ Peer) Encryption Layer messages.

2.3 Transport Messages Total Header Sizes

3 Server Encryption Layer

The Server Encryption Layer provides security for communication between Peers and the Server. TCP and UDP have different security methods.

3.1 TCP

The TCP Server Encryption Layer is heavily based on a simplification of TLS 1.3 as defined in.

3.1.1 PeerHello

Peer → Server

Bytes	**Name**	**Value**
1	type	1
16	public-key	

$$(E_{peer}^{pub},\ E_{peer}^{priv}) := \text{DH-Generate}()$$
$$\text{public-key} := E_{peer}^{pub}$$

3.1.2 ServerHello

Server → Peer

Bytes	**Name**	**Value**
1	type	2
3	certificates-length	
certificates-length	certificate_list	
16	public-key	
variable	certificate-verify	

certificate_list is defined in RFC8446 Section-4.4.2.

$$(E_{serv}^{pub},\ E_{serv}^{priv}) := \text{DH-Generate}()$$
$$\text{public-key} := E_{serv}^{pub}$$

certificate-verify is defined in RFC8445 Section-4.4.3 with the following modification. The content that is signed is:

$$\text{content} := \text{“SreenViewServerVerify”} \mid\mid 0 \mid\mid E_{serv}^{pub}$$

The Client MUST validate all signatures in accordance with the TLS spec.

3.1.3 Transport Data Key Derivation

$$C_{peer} = \text{DH}(E_{serv}^{pub},\ E_{peer}^{priv})$$
$$C_{serv} = \text{DH}(E_{peer}^{pub},\ E_{serv}^{priv})$$
$$(T_{peer}^{send} = T_{serv}^{recv},\ T_{peer}^{recv} = T_{serv}^{send}) := \text{KDF}_2(C_{peer} = C_{serv},\ \epsilon)$$
$$N_{peer}^{send} = N_{serv}^{recv} = N_{peer}^{recv} = N_{serv}^{send} := 0$$

3.1.4 Subsequent Messages: Transport Data Messages

Peer $\leftrightarrow$ Server

Bytes	**Name**	**Value**
1	type	3
2	data-length	
data-length	data	

$$\text{data} := \text{AEAD}(T_m^{send}, N_m^{send}, P, \epsilon)$$
$$N_m^{send} := N_m^{send} + 1$$

N_m is an 64 bit counter that MUST NOT wrap. After a transport message is sent, if N_m equals $(2^{64} - 1)$ the TCP connection MUST be dropped. Subsequent TCP messages MUST NOT be sent.

Where P is the payload to be transported

3.2 UDP

UDP encryption and authentication rely on the *session-id*, *peer-id* and *peer-key* values established in a session (described in 5.4). The Server MUST NOT process or reply to any messages that don't pass authentication. This prevents an amplification attack.

3.2.1 Transport Data Key Derivation

$$G := \text{session-id}$$
$$H := \text{peer-id}$$
$$J := \text{peer-key}$$
$$(V_{peer}^{send} = V_{serv}^{recv}, V_{peer}^{recv} = V_{serv}^{send}) := \text{KDF}_2(\text{HASH}(G \,||\, H \,||\, J), \epsilon)$$
$$M_{peer}^{send} = M_{serv}^{recv} = M_{peer}^{recv} = M_{serv}^{send} := 0$$

3.2.2 Transport Data Messages

Peer $\rightarrow$Server

Bytes	**Name**	**Value**
1	type	4
16	*peer-id*	
8	counter	
UDP length − 8 bytes	data	

Server $\rightarrow$Peer

Bytes	**Name**	**Value**
1	type	5
8	counter	
UDP length – 8 bytes	data	

$$\text{data} := \text{AEAD}(V_m^{send}, M_m^{send}, P, \epsilon)$$
$$\text{counter} := M_m^{send}$$
$$M_m^{send} := M_m^{send} + 1$$

M_m is an 64 bit counter that MUST NOT wrap. After a transport message is sent, if M_m equals $(2^{64}-1)$ the TCP connection MUST be dropped. Subsequent UDP messages MUST NOT be sent.

Where P is the payload to be transported

4 ScreenView Server Communication (SVSC) Protocol

The SVSC protocol is the Server Communication Layer protocol used for Peers to interact with the relay server, Server. Peers can lease an ID as well as begin a session with another Peer. Once a session is established, Peers can forward messages to another Peer. Unless otherwise noted, all messages MUST occur over TCP.

With the exception of the *Handshake* messages. All SVSC messages' first byte contain a number to indicate the message type.

4.1 Definitions

The following definitions are used globally throughout the document:

- Peer - denotes a client in classical server/client environment
- Server - The intermediary server used for routing and proxying data between

4.2 Handshake

4.2.1 ProtocolVersion

Handshaking begins by the Server sending the Peer a *ProtocolVersion* message. This lets the server know the version supported by the Host.

The *ProtocolVersion* message consists of 12 bytes interpreted as a string of ASCII characters in the format "SVSC xxx.yyy" where xxx and yyy are the major and minor version numbers, padded with zeros.

Server → Peer

Bytes	**Name**	**Value**
11	version	"`SVSC 001.000`"

The Peer replies back either `0` to indicate the version is not acceptable and that the handshake has failed or `1` if the version is acceptable to the Peer and the handshake as succeeded. If 0 is sent, all communication MUST cease and the TCP connection MUST be terminated.

Peer → Server

Bytes	**Name**	**Value**
1	ok	0 or 1

4.3 Leasing

A lease is a temporary assignment of an ID to a Peer. The ID format and generation is discussed in 4.2.5. A maximum of 1 ID can be leased per TCP connection. ID generation MUST be rate limited to prevent ID exhaustion. Rate limiting rules are out of scope for this protocol, however some suggestions are listed in 4.2.6.

4.3.1 LeaseRequest

A *LeaseRequest* message requests a lease of an ID.

Peer → Server

Bytes	**Name**	**Value**
1	type	1
1	has-cookie	0 or 1
Below only if *has-cookie* is 1		
24	cookie	

If a Peer would like to request an ID it had previously been issued after expiration, it may include the cookie value it received in the *LeaseResponse*.

4.3.2 LeaseResponse

A *LeaseResponse* message is a response to a *LeaseRequest*.
If *has-cookie* is 1, a Server MAY consider the *cookie* value in *LeaseRequest* or completely ignore it.

Server → Peer

Bytes	**Name**	**Value**
1	type	2
1	accepted	0 or 1
Below only if *accepted* is 1		
4	id	
24	cookie	
8	expiration	

expiration is a 64 bit Unix timestamp representing the expiry of lease. Disconnection of a Peer (e.g, the TCP connection is dropped) does not end a lease.

cookie a 128 bit value. The generation of this value is discussed in 4.2 .7.

Consideration of the *cookie* value MUST have no effect on the the value of *accepted*. That is, if the request is for a specific ID (implied by the presence of a cookie value and a *has-cookie* value equal to 1 in the *LeaseRequest*) and the ID requested is not available, the Server SHOULD respond with a different available ID and an *accepted* value of 1 (assuming an ID is available).

accepted SHOULD only be 0 if no IDs are left or for rate limiting reasons.

4.3.3 LeaseExtensionRequest

A *LeaseExtensionRequest* message is used to extend a lease. Before a lease has expired, the Peer can request a lease extension. The server can accept or deny this request.

Peer $\rightarrow$ Server

Bytes	**Name**	**Value**
1	type	3
24	cookie	

4.3.4 LeaseExtensionResponse

A *LeaseExtensionResponse* message is a response to a *LeaseExtensionRequest*.

Server $\rightarrow$ Peer

Bytes	**Name**	**Value**
1	type	4
1	extended	0 or 1
Below only if *extended* is 1		
8	new-expiration	

new-expiration is a 64 bit Unix timestamp representing the expiry of lease.

4.3.5 ID Generation

An ID is a 26 to 33 bit decimal number. This comes out to about up to 8 to 10 decimal digits, respectively. The Server may scale the keyspace depending on current usage. For optimal user experience while maintaining efficiency, the Server MUST only use keyspaces between 26 bits and 33 bits for ID generation. ID generation must also be uniformly random. All active IDs must be stored on the server. ID generation MAY occur using the below algorithm:

Let S represents a set of all active IDs, B be a number of bits between 26 and 33, and $generate(x)$ be a functions that returns a x uniformly random bits.

Algorithm 1 ID generation

```
$id$
**repeat**
    $id \leftarrow generate(B)$
**until** $id \notin S$
$S \leftarrow S \cup \{id\}$
**return** $id$
```

4.3.6 Rate Limits

To prevent ID exhaustion, rate limits SHOULD be in place. TCP is used for *LeaseRequests* so IP addresses can not be spoofed. However, using proxy services such as Tor, simple IP based rate limits are likely not entirely sufficient. Servers MAY want to block all known proxy IP addresses. Additionally, a Server MAY only want to allow one active ID per Peer.

4.3.7 Cookie Value

A *cookie* value is a 128 bit value used for authentication in *LeaseExtensionRequest* and *LeaseRequest* messages. Specific generation of a *cookie* is out of scope, however care must be taken to ensure it is not predictable or exploitable. This value MAY be simply a random 24 byte key, HMAC-SHA1(*id*, *key*) || *id*, or something else entirely.

4.4 Sessions

A session is a connection between two Peers. At least one Peer must have an ID. A Peer can have a maximum of one session at any time. Immediately after receiving a *EstablishSessionResponse* message with a *status* of 0 or a *EstablishSessionNotification* message a Peer MUST establish UDP connection by sending a *Keepalive* message as defined in 5.5. Failure to do so MAY result in dropped *SessionData** packets.

4.4.1 EstablishSessionRequest

An *EstablishSessionRequest* message is a Peer request to establish a connection to a Peer.

Peer → Server

Bytes	Name	Value	Description
1	type	5	
4	lease-id		The ID of the Peer to establish this connection with

4.4.2 EstablishSessionResponse

An *EstablishSessionResponse* message is a response to *EstablishSessionRequest*.

Server → Peer

Bytes	Name	Value	Description
1	type	6	
4	lease-id		the ID of the Peer attempted to connect to
1	status	0-5	described below
Below only if *status* is 0			
16	session-id		described below
16	peer-id		described below
16	peer-key		described below

status can have the following values:

Value	Description
0	session establishment was successful
1	ID not found
2	Peer is offline
3	Peer is busy
4	You are busy
5	Other error

A Peer may be considered offline if, for example, an unexpired ID has been assigned to them and then the TCP connection is dropped. Keepalive messages and timeouts are discussed in .

session-id is a 128 bit random value used for session identification

peer-id is a 128 bit random value used to authentication a peer for a given session.

4.4.3 EstablishSessionNotification

A *EstablishSessionNotification* notifies a Peer that a session has been established with them.

Server → Host

Bytes	Name	Value	Description
1	type	7	
16	session-id		described in 4.3.2
16	peer-id		described in 4.3.2
16	peer-key		described in 4.3.2

4.4.4 SessionEnd

A *SessionEnd* message is used to terminate a session. Once a Server receives a *SessionEnd* message,

Peer → Server

Bytes	Name	Value	Description
1	type	8	

4.4.5 SessionEndNotification

A *SessionEndNotification* notifies a Peer that a session has ended. If a Client sends a *SessionEnd* message, the Server MUST send a *SessionEndNotification* message to a Host.

Server → Peer

Bytes	Name	Value	Description
1	type	9	

4.4.6 SessionDataSend - TCP/UDP

A *SessionDataSend* is a message from a Peer intended to be forwarded to the Peer on the other side of the session. If a connection is not available (e.g. UDP was dropped or never established) for *SessionDataReceive* message to be sent to the other Peer, the *SessionDataSend* message is silently dropped.

Peer → Server

Bytes	Name	Value	Description
1	type	10	
3	data-length		length of the content in bytes
data-length	data		data to be forwarded

4.4.7 SessionDataReceive - TCP/UDP

A *SessionDataReceive* is a message being forwarded to a Peer from the Peer on the other side of the session. The Server SHOULD forward the message along the same transport as it was received.

Server → Peer

Bytes	Name	Value	Description
1	type	11	
3	data-length		length of the content in bytes
data-length	data		data to be forwarded

4.5 Keepalive - TCP/UDP

For each transport (TCP and UDP), if no message has been sent in *Keepalive-Timeout* a Server sends a keepalive message over the respective transport. The Peer MUST respond with another Keepalive message.

For TCP, if a *KeepaliveTimeout* response is not received by the Server in double *KeepaliveTimeout* seconds the TCP connection is considered dropped.

For UDP, if a *KeepaliveTimeout* response is not received by the Server in half *KeepaliveTimeout* seconds another *Keepalive* message is sent. If a response is not received in an additional half *KeepaliveTimeout* seconds, the UDP connection is considered dropped.

Server $\leftrightarrow$ Peer

Bytes	**Name**	**Value**
1	type	0

5 Weak Pre Shared Key, Key Authentication (WPSKKA) Protocol

The WPSKKA protocol is the E2EE Encryption layer protocol used to communicate between peers. All WPSKKA messages' first byte contain a number to indicate the message type.

WPSKKA relies on SRP as defined in https://datatracker.ietf.org/doc/html/rfc5054 to establish a shared key used to authenticate DH keys. The Host will serve as the SRP server, the client will serve as the SRP client.

Host $\rightarrow$ Client

Bytes	**Name**	**Value**
1	type	1
16	username	
16	salt	
256	srp-B	
16	public-key	

```
SRP-HOST-INIT()
$(D_{host}^{pub}, D_{host}^{priv})$ := DH-Generate()
$I := RAND(128)$
$s := RAND(128)$
$B :=$ SRP-B()
$username := I$
$salt := s$
$srp - B := B$
public-key := $C_{serv}^{pub}$
```

Client $\rightarrow$ Host

Bytes	**Name**	**Value**
1	type	2
256	srp-A	
16	public-key	
32	mac	

$$\begin{aligned}
&\text{SRP-CLIENT-INIT()}\\
&(D^{pub}_{client},\ D^{priv}_{client}) := \text{DH-Generate()}\\
&A := \text{SRP-A()}\\
&L_{client} = L_{host} := \text{SRP-PREMASTER()}\\
&\text{public-key} := \text{HMAC}(\text{KDF}_1(D^{pub}_{client}, L_{client}))
\end{aligned}$$

Host $\rightarrow$ Client

Bytes	**Name**	**Value**
1	type	3
32	mac	

$$\text{mac} := \text{HMAC}(\text{KDF}_1(C^{pub}_{host}, L_{host}))$$

5.1 Transport Data Key Derivation

$$\begin{aligned}
&D_{host} = \text{DH}(D^{pub}_{client},\ D^{priv}_{host})\\
&D_{client} = \text{DH}(D^{pub}_{host},\ D^{priv}_{client})\\
&(U^{send}_{peer} = U^{recv}_{serv},\ U^{recv}_{peer} = U^{send}_{serv}) := \text{KDF}_2(D_{host} = D_{client},\ \epsilon)\\
&O^{send}_{host} = O^{recv}_{client} = O^{recv}_{host} = O^{send}_{client} := 0
\end{aligned}$$

5.1.1 Subsequent Messages: Transport Data Messages

Host $\leftrightarrow$ Client

Bytes	**Name**	**Value**
1	type	4
8	counter	
2	data-length	
data-length	data	

$$\begin{aligned}
&\text{data} := \text{AEAD}(U^{send}_m, O^{send}_m, P, \epsilon)\\
&\text{counter} := O^{send}_m\\
&O^{send}_m := O^{send}_m + 1
\end{aligned}$$

O_m is an 64 bit counter that MUST NOT wrap. After a transport message is sent, if O_m equals $(2^{64} - 1)$ the TCP connection MUST be dropped. Subsequent messages MUST NOT be sent.

Where P is the payload to be transported

6 Remote Visual Display (RVD) Protocol

The RVD protocol is used to communicate mouse input, keyboard input, frame data, and clipboard data between the Host and the Client.

All messages MUST occur over the transport listed.

With the exception of the *Handshake* messages, all RVD messages' first byte contain a number to indicate the message type.

A *sequence-number* is an incrementing 32-bit counter each UDP message sent, separate for Host and Client. All messages sent over UDP MUST begin with a 4 bytes *sequence-number*. *sequence-number* is initialized to 0 and increments once for every UDP message sent by the respective Peer. Therefore each RVD UDP message looks like:

Bytes	Name
4	sequence-number
variable	RVD message

6.1 Combining Messages

In some situations, such as when a large screen change occurs or when the screen is first sent to the client, large amounts of *FrameData*'s may need to be sent in quick succession. Often individual *FrameData*'s will be much smaller than the MTU. Similar to TCP's Nagle algorithm, multiple UDP messages can be combined into a single UDP packet as long as the total size is remains less than the MTU. Each message directly follows the previous message with the first message directly following the sequence number. Only one *sequence-number* is used:

Bytes	Name
4	sequence-number
variable	RVD message 1
variable	RVD message 2
variable	etc.

6.2 Definitions

- Host - A peer with an ID that wants to share their screen to the Client

- Client - A peer that wants to view and maybe control the Host's screen
- Display - A rectangular visual region that is shared by a Host to a Client. May or may not be
- *Controllable*.
- Controllable - A *Display* that accepts keyboard and mouse input.

6.3 Handshake

6.3.1 ProtocolVersion - TCP

Handshaking begins by the Host sending the client a *ProtocolVersion* message. This lets the Client know the verison supported by the Host.

The *ProtocolVersion* message consists of 11 bytes interpreted as a string of ASCII characters in the format "RVD xxx.yyy" where xxx and yyy are the major and minor version numbers, padded with zeros.

Host → Client

Bytes	**Name**	**Value**
11	version	"`RVD 001.000`"

The Client replies back either `0` to indicate the version is not acceptable and that the handshake has failed or `1` if the version is acceptable to the Client and the handshake as succeeded. If 0 is sent, all communication MUST cease and an error SHOULD be displayed to user. A *SessionEnd* message should be sent by the Client.

Client → Host

Bytes	**Name**	**Value**
1	ok	0 or 1

6.3.2 Initialization

Once the handshake has succeeded the Host responds with a *DisplayChange* message.

6.4 Control messages

Control messages are messages that instruct client about changes regarding the Host.

6.4.1 DisplayChange - TCP

A *DisplayChange* message informs the client about the available *Display*s. RVD supports up to 255 displays.

Host → Client

Bytes	Name	Value
1	type	1
1	clipboard-readable	0 or 1
1	number-of-displays	1 - 255
variable	displays-information	*DisplayInformation*

Each *Display* has an associated *DisplayInformation*. *displays-information* contains *number-of-displays DisplayInformation*'s. A *DisplayInformation* is defined below:

Bytes	Name	Description
1	display-id	
2	width	number of pixels of the width of this display
2	height	number of pixels of the width of this display
2	cell-width	number of pixels of the height of a cell in the grid
2	cell-height	number of pixels of the height of a cell in the grid
1	access	defined below
1	name-length	length of the *display-name* in bytes
name-length	display-name	the display name (UTF-8)

Restrictions:

- *cell-width* MUST be less than *width*. *cell-height* must be less than *height*.

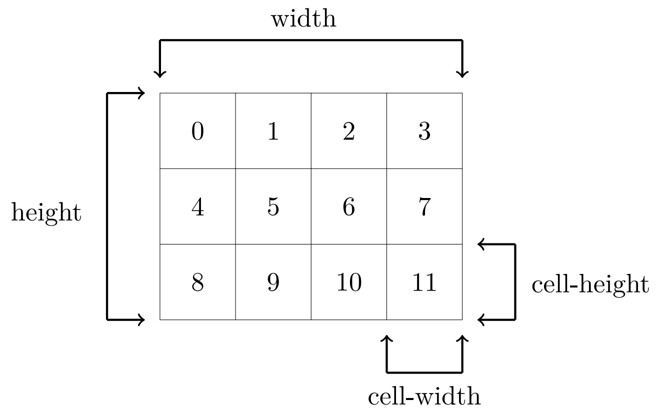
- The *access* byte defines what type of access is available for the display. The bits of the *access* byte are described below in little endian.

Bit	Name
0	Flush
1	*Controllable*
2 3 4 5 6 7	Reserved for future use

If the *Controllable* bit is 1 and the *clipboard-readable* byte is set to 1, then the clipboard is writable. The *Controllable* bit SHOULD be consistent throughout all displays.

The *Flush* bit indicates whether this display has changed, specifically if this *display-id* refers to a different *Display* than the same *display-id* did in the previous *DisplayChange* message. In initialization, this MUST always be 1 (as there is no previous *DisplayChange*). If the display hasn't changed (0) then the frame data may be maintained. If *Flush* is 0, *width*, *height* MUST remain the same as the previous *DisplayChange* specified for the *display-id*.

Display cell numbering is right to left, up to down, 0 indexed as seen below. If cell-width % width > 0 the right column's width will be width % cell-width. If height % cell-height > 0 the bottom row's height will be height % cell-height.



6.4.2 DisplayChangeReceived - TCP

The *DisplayChangeReceived* message is sent in reply after receiving a *DisplayChange* message. It indicates to the Host they may start sending *FrameData* referencing the new *DisplayInformation* in the most recent *DisplayChange*.

Client → Host

Bytes	**Name**	**Value**
1	type	2

6.4.3 MouseLocation - TCP/UDP

The *MouseLocation* message send information about where the mouse is currently on the screen. The Host sends this information periodically throughout the session. The Host SHOULD send a *MouseLocation* update when mouse input is received from the Host's system or in reply when it receives a *MouseInput*.

Host → Client

Bytes	Name	Value	Description
3	type	3	
1	display-id	0-255	
2	x-location		x coordinate of the mouse
2	y-location		y coordinate of the mouse

6.5 Input

Input messages (including *MouseLocation*) may be sent over TCP or UDP. TCP is preferred in most situations. However, in situations where speed is prioritized over the guarantees TCP provides (such as gaming), UDP can be used.

6.5.1 MouseInput - TCP/UDP

Client → Host

Bytes	Name	Value	Description
1	type	4	
1	display-id	0-255	
2	x-position		x coordinate of the mouse
2	y-position		y coordinate of the mouse
1	button-mask		described below

Indicates either pointer movement or a pointer button press or release. The pointer is now at (x-position, y-position), and the current state of buttons 1 to 8 are represented by bits 0 to 7 of button-mask respectively, 0 meaning up, 1 meaning down (pressed).

On a conventional mouse, buttons 1, 2 and 3 correspond to the left, middle and right buttons on the mouse. On a wheel mouse, each step of the wheel is represented by a press and release of a certain button. Button 4 means up, button 5 means down, button 6 means left and button 7 means right.

6.5.2 KeyInput - TCP/UDP

The *KeyInput* event sends key presses or releases.

Client → Host

Bytes	Name	Value	Description
1	type	5	
1	down-flag	0 or 1	indicates whether the key is now pressed or released
4	key		”keysym”

Details can be found at the RFB Spec

6.6 Clipboard

6.6.1 ClipboardTypeRequest - TCP

Used to request clipboard types the Host supports.

Client → Host

Bytes	**Name**	**Value**
1	type	6

6.6.2 ClipboardTypeResponse - TCP

Response to the *ClipboardTypeRequest*

Host → Client

Bytes	**Name**	**Value**	**Description**
1	type	7	
1	number-of-clipboard-types	0-255	
variable	data (variable bytes)		described below

number-of-clipboard-types is always 0 if *clipboard-readable* is 0.

data contains *number-of-clipboard-types ClipboardTypes*. *ClipboardType* is defined below.

Bytes	**Name**	**Value**	**Description**
1	type-length	1-255	
type-length	type-name		type name in ASCII

6.6.3 CopyRequest - TCP

This is a request for a keyboard contents. It can be made by either the Client or the Host.

Client ↔ Host

Bytes	**Name**	**Value**	**Description**
1	type	8	
1	type-length	1-255	
type-length	type-name		type name in ASCII

6.6.4 CopyResponse - TCP

CopyResponse message is a response to a *CopyRequest*.

Client ↔ Host

Bytes	**Name**	**Value**	**Description**
1	type	9	
1	accepted	0 or 1	
Below only if *accepted* is 1			
1	type-length	1-255	
type-length	type-name		type name in ASCII
3	content-length		the length of the content (maximum 2^{24} bytes or 16MB)
content-length	data		zlib'ed raw data

accepted indicates whether the *CopyRequest* was accepted. If 0, the rest of the message MUST not exist. If *clipboard-readable* is 0, *accepted* is always 0. A Client or Host may send this message without a request. If a *CopyResponse* is unsolicited, then *accepted* MUST be 1.

data is zlib compressed.

6.6.5 A note on Pasting

There is a no paste message. To paste data an unsolicited *CopyResponse* may be sent and then the keyboard shortcut (ctrl+v or cmd+v) should be sent via the *KeyboardMessage*

6.7 FrameData - UDP

The *FrameData* message contains an update of a particular cell on a particular *Display*.

Host →Client

Bytes	**Name**	**Value**
1	type	10
4	frame-number	
1	display-id	0-255
2	cell-number	
2	size	
size	data	

frame-number is a 32 bit counter, initialized with 0 at the begining of the protocol, and incremented once from *FrameData* message sent.
data contains jpeg pixel data of the updated cell.

6.8 Congestion

When sending messages over a network the network may become congested to avoid congesting the network further RVD implements a congestion detection and congestion control mechanism. RVD uses Additive increase/multiplicative decrease or AIMD to control the *OutputMaximum* which is the maximum messages allowed per *CongestionWindow*.