Internet Engineering Task Force (IETF)
Request for Comments: 6143
Category: Informational
ISSN: 2070-1721

T. Richardson
J. Levine
RealVNC Ltd.

March 2011

# The Remote Framebuffer Protocol

## Abstract

RFB ("remote framebuffer") is a simple protocol for remote access to graphical user interfaces that allows a client to view and control a window system on another computer. Because it works at the framebuffer level, RFB is applicable to all windowing systems and applications. This document describes the protocol used to communicate between an RFB client and RFB server. RFB is the protocol used in VNC.

## Status of This Memo

This document is not an Internet Standards Track specification; it is published for informational purposes.

This document is a product of the Internet Engineering Task Force (IETF). It represents the consensus of the IETF community. It has received public review and has been approved for publication by the Internet Engineering Steering Group (IESG). Not all documents approved by the IESG are a candidate for any level of Internet Standard; see Section 2 of RFC 5741.

Information about the current status of this document, any errata, and how to provide feedback on it may be obtained at http://www.rfc-editor.org/info/rfc6143.

Table of Contents

## 1. Introduction

RFB ("remote framebuffer") is a simple protocol for remote access to graphical user interfaces. Because it works at the framebuffer level, it is applicable to all windowing systems and applications, including X11, Windows, and Macintosh. RFB is the protocol used in VNC. The protocol is widely implemented and has had fairly good interoperability.

The remote endpoint where the user sits (typically with a display, keyboard, and pointer) is called the RFB client or viewer. The endpoint where changes to the framebuffer originate (i.e., the windowing system and applications) is known as the RFB server.

RFB is a "thin client" protocol. The emphasis in the design of the RFB protocol is to make very few requirements of the client. In this way, clients can run on the widest range of hardware, and the task of implementing a client is made as simple as possible.

The protocol also makes the client stateless. If a client disconnects from a given server and subsequently reconnects to that same server, the state of the user interface is preserved. Furthermore, a different client endpoint can be used to connect to the same RFB server. At the new endpoint, the user will see exactly the same graphical user interface as at the original endpoint. In effect, the interface to the user's applications becomes completely mobile. Wherever suitable network connectivity exists, the user can access their own personal applications, and the state of these applications is preserved between accesses from different locations. This provides the user with a familiar, uniform view of the computing infrastructure wherever they go.

The RFB protocol has evolved over the past decade, and has been implemented several times, including at least one open source version. This document describes the RFB protocol as actually implemented, so that future implementers can interoperate with existing clients and servers.

## 2. Initial Connection

An RFB server is typically a long-lived process that maintains the state of a framebuffer. RFB clients typically connect, communicate with the server for a period of time to use and manipulate the framebuffer, then disconnect. A subsequent RFB session will then pick up where a prior session left off, with the state of the framebuffer intact.

An RFB client contacts the server on TCP port 5900. On systems with multiple RFB servers, server N typically listens on port 5900+N, analogous to the way that X Window servers listen on port 6000+N.

Some browser-based clients use a Java application to run the RFB protocol. RFB servers sometimes provide a simple HTTP server on port 5800 that provides the requisite Java applet.

In some cases, the initial roles of the client and server are reversed, with the RFB client listening on port 5500, and the RFB server contacting the RFB client. Once the connection is established, the two sides take their normal roles, with the RFB server sending the first handshake message.

Note that the only port number assigned by IANA for RFB is port 5900, so RFB clients and servers should avoid using other port numbers unless they are communicating with servers or clients known to use the non-standard ports.

## 3. Display Protocol

The display side of the protocol is based around a single graphics primitive: "put a rectangle of pixel data at a given x,y position". This might seem an inefficient way of drawing many user interface components. However, allowing various different encodings for the pixel data gives us a large degree of flexibility in how to trade off various parameters such as network bandwidth, client drawing speed, and server processing speed.

A sequence of these rectangles makes a framebuffer update (simply referred to here as "update"). An update represents a change from one valid framebuffer state to another, so in some ways is similar to a frame of video. The rectangles in an update are usually but not always disjoint.

The update protocol is demand-driven by the client. That is, an update is only sent from the server to the client in response to an explicit request from the client. This gives the protocol an adaptive quality. The slower the client and the network are, the lower the rate of updates. With typical applications, changes to the same area of the framebuffer tend to happen soon after one another. With a slow client or network, transient states of the framebuffer can be ignored, resulting in less network traffic and less drawing for the client.

After the initial handshake sequence, the protocol is asynchronous, with each side sending messages as needed. The server must not send unsolicited updates. An update must only be sent in response to a request from the client. When several requests from the client are outstanding, a single update from the server may satisfy all of them.

## 4. Input Protocol

The input side of the protocol is based on a standard workstation model of a keyboard and multi-button pointing device. Input events are simply sent to the server by the client whenever the user presses a key or pointer button, or whenever the pointing device is moved. These input events can also be synthesized from other non-standard I/O devices. For example, a pen-based handwriting recognition engine might generate keyboard events.

## 5. Representation of Pixel Data

Initial interaction between the RFB client and server involves a negotiation of the format and encoding of the pixel data that will be sent. This negotiation has been designed to make the job of the client as easy as possible. The server must always be able to supply pixel data in the form the client wants. However, if the client is able to cope equally with several different formats or encodings, it may choose one that is easier for the server to produce.

Pixel format refers to the representation of individual colors by pixel values. The most common pixel formats are 24-bit or 16-bit "true color", where bit-fields within the pixel value translate directly to red, green, and blue intensities, and 8-bit "color map" (palette) where the pixel values are indices into a 256-entry table that contains the actual RGB intensities.

Encoding refers to the way that a rectangle of pixel data will be sent to the client. Every rectangle of pixel data is prefixed by a header giving the X,Y position of the rectangle on the screen, the width and height of the rectangle, and an encoding type which specifies the encoding of the pixel data. The data itself then follows using the specified encoding.

The encoding types defined at present are: Raw, CopyRect, RRE, TRLE, Hextile, and ZRLE. In practice, current servers use the ZRLE, TRLE, and CopyRect encodings since they provide the best compression for typical desktops. Clients generally also support Hextile, which was often used by older RFB servers that didn't support TRLE. See Section 7.7 for a description of each of the encodings.

## 6. Protocol Versions and Extensions

The RFB protocol has evolved through three published versions: 3.3, 3.7, and 3.8. This document primarily documents the final version 3.8; differences from the earlier versions, which are minor, are described in Appendix A. Under no circumstances should an implementation use a protocol version number other than one defined in this document. Over the years, different implementations of RFB have attempted to use different version numbers to add undocumented extensions, with the result being that to interoperate, any unknown 3.x version must be treated as 3.3, so it is not possible to add a 3.9 or higher version in a backward-compatible fashion. Future evolution of RFB will use 4.x version numbers.

It is not necessary to change the protocol version number to extend the protocol. The protocol can be extended within an existing version by:

New encodings
: A new encoding type can be added to the protocol relatively easily while maintaining compatibility with existing clients and servers. Existing servers will simply ignore requests for a new encoding that they don't support. Existing clients will never request the new encoding so will never see rectangles encoded that way.

Pseudo-encodings
: In addition to genuine encodings, a client can request a "pseudo-encoding" to declare to the server that it supports a certain extension to the protocol. A server that does not support the extension will simply ignore the pseudo-encoding. Note that this means the client must assume that the server does not support the extension until it gets some extension-specific confirmation from the server. See Section 7.8 for a description of current pseudo-encodings.

New security types
: Adding a new security type gives full flexibility in modifying the behavior of the protocol without sacrificing compatibility with existing clients and servers. A client and server that agree on a new security type can effectively talk whatever protocol they like after that -- it doesn't necessarily have to be anything like the RFB protocol.

See Section 8 for information on obtaining an ID for a new encoding or security type.

## 7. Protocol Messages

The RFB protocol can operate over any reliable transport, either byte-stream or message based. It usually operates over a TCP/IP connection. There are three stages to the protocol. First is the handshaking phase, the purpose of which is to agree upon the protocol version and the type of security to be used. The second stage is an initialization phase where the client and server exchange ClientInit and ServerInit messages. The final stage is the normal protocol interaction. The client can send whichever messages it wants, and may receive messages from the server as a result. All these messages begin with a message-type byte, followed by message-specific data.

The following descriptions of protocol messages use the basic types U8, U16, U32, S8, S16, and S32. These represent, respectively, 8-, 16-, and 32-bit unsigned integers and 8-, 16-, and 32-bit signed integers. All multiple-byte integers (other than pixel values themselves) are in big endian order (most significant byte first). Some messages use arrays of the basic types, with the number of entries in the array determined from fields preceding the array.

The type PIXEL means a pixel value of bytesPerPixel bytes, where bytesPerPixel is the number of bits-per-pixel divided by 8. The bits-per-pixel is agreed by the client and server, either in the ServerInit message (Section 7.3.2) or a SetPixelFormat message (Section 7.5.1). See Section 7.4 for the detailed description of the pixel format.

Several message formats include padding bits or bytes. For maximum compatibility, messages should be generated with padding set to zero, but message recipients should not assume padding has any particular value.

### 7.1. Handshake Messages

When an RFB client and server first connect, they exchange a sequence of handshake messages that determine the protocol version, what type of connection security (if any) to use, a password check if the security type requires it, and some initialization information.

#### 7.1.1. ProtocolVersion Handshake

Handshaking begins by the server sending the client a ProtocolVersion message. This lets the client know which is the highest RFB protocol version number supported by the server. The client then replies with a similar message giving the version number of the protocol that should actually be used (which may be different to that quoted by the server). A client should never request a protocol version higher than that offered by the server. It is intended that both clients and servers may provide some level of backwards compatibility by this mechanism.

The only published protocol versions at this time are 3.3, 3.7, and 3.8. Other version numbers are reported by some servers and clients, but should be interpreted as 3.3 since they do not implement the different handshake in 3.7 or 3.8. Addition of a new encoding or pseudo-encoding type does not require a change in protocol version, since a server can simply ignore encodings it does not understand.

The ProtocolVersion message consists of 12 bytes interpreted as a string of ASCII characters in the format "RFB xxx.yyy\n" where xxx and yyy are the major and minor version numbers, left-padded with zeros:

```
RFB 003.008\n (hex 52 46 42 20 30 30 33 2e 30 30 38 0a)
```

#### 7.1.2. Security Handshake

Once the protocol version has been decided, the server and client must agree on the type of security to be used on the connection. The server lists the security types that it supports:

| No. of bytes | Type [Value] | Description |
|---|---|---|
| 1 | U8 | number-of-security-types |
| number-of-security-types | U8 array | security-types |

If the server listed at least one valid security type supported by the client, the client sends back a single byte indicating which security type is to be used on the connection:

| No. of bytes | Type [Value] | Description |
|---|---|---|
| 1 | U8 | security-type |

If number-of-security-types is zero, then for some reason the connection failed (e.g., the server cannot support the desired protocol version). This is followed by a string describing the reason (where a string is specified as a length followed by that many ASCII characters):

| No. of bytes | Type [Value] | Description |
|---|---|---|
| 4 | U32 | reason-length |
| reason-length | U8 array | reason-string |

The server closes the connection after sending the reason-string.

The security types defined in this document are:

| Number | Name |
|---|---|
| 0 | Invalid |
| 1 | None |
| 2 | VNC Authentication |

Other security types exist but are not publicly documented.

Once the security-type has been decided, data specific to that security-type follows (see Section 7.2 for details). At the end of the security handshaking phase, the protocol normally continues with the SecurityResult message.

Note that after the security handshaking phase, it is possible that further communication is over an encrypted or otherwise altered channel if the two ends agree on an extended security type beyond the ones described here.

### 7.1.3. SecurityResult Handshake

The server sends a word to inform the client whether the security handshaking was successful.

| No. of bytes | Type [Value] | Description |
|---|---|---|
| 4 | U32 | status: |
| | 0 | OK |
| | 1 | failed |

If successful, the protocol passes to the initialization phase (Section 7.3).

If unsuccessful, the server sends a string describing the reason for the failure, and then closes the connection:

| No. of bytes | Type [Value] | Description |
|---|---|---|
| 4 | U32 | reason-length |
| reason-length | U8 array | reason-string |

## 7.2. Security Types

Two security types are defined here.

### 7.2.1. None

No authentication is needed. The protocol continues with the SecurityResult message.

### 7.2.2. VNC Authentication

VNC authentication is to be used. The server sends a random 16-byte challenge:

| No. of bytes | Type [Value] | Description |
|---|---|---|
| 16 | U8 | challenge |

The client encrypts the challenge with DES, using a password supplied by the user as the key. To form the key, the password is truncated to eight characters, or padded with null bytes on the right. The client then sends the resulting 16-byte response:

| No. of bytes | Type [Value] | Description |
|---|---|---|
| 16 | U8 | response |

The protocol continues with the SecurityResult message.

This type of authentication is known to be cryptographically weak and is not intended for use on untrusted networks. Many implementations will want to use stronger security, such as running the session over an encrypted channel provided by IPsec [RFC4301] or SSH [RFC4254].

### 7.3. Initialization Messages

Once the client and server agree on and perhaps validate a security type, the protocol passes to the initialization stage. The client sends a ClientInit message. Then, the server sends a ServerInit message.

#### 7.3.1. ClientInit

| No. of bytes | Type [Value] | Description |
|---|---|---|
| 1 | U8 | shared-flag |

Shared-flag is non-zero (true) if the server should try to share the desktop by leaving other clients connected, and zero (false) if it should give exclusive access to this client by disconnecting all other clients.

#### 7.3.2. ServerInit

After receiving the ClientInit message, the server sends a ServerInit message. This tells the client the width and height of the server's framebuffer, its pixel format, and the name associated with the desktop:

| No. of bytes | Type [Value] | Description |
|---|---|---|
| 2 | U16 | framebuffer-width in pixels |
| 2 | U16 | framebuffer-height in pixels |
| 16 | PIXEL_FORMAT | server-pixel-format |
| 4 | U32 | name-length |
| name-length | U8 array | name-string |

Server-pixel-format specifies the server's natural pixel format. This pixel format will be used unless the client requests a different format using the SetPixelFormat message (Section 7.5.1).

## 7.4. Pixel Format Data Structure

Several server-to-client messages include a PIXEL_FORMAT, a 16-byte structure that describes the way a pixel is transmitted.

| No. of bytes | Type [Value] | Description |
|---|---|---|
| 1 | U8 | bits-per-pixel |
| 1 | U8 | depth |
| 1 | U8 | big-endian-flag |
| 1 | U8 | true-color-flag |
| 2 | U16 | red-max |
| 2 | U16 | green-max |
| 2 | U16 | blue-max |
| 1 | U8 | red-shift |
| 1 | U8 | green-shift |
| 1 | U8 | blue-shift |
| 3 | | padding |

Bits-per-pixel is the number of bits used for each pixel value on the wire. This must be greater than or equal to the depth, which is the number of useful bits in the pixel value. Currently bits-per-pixel must be 8, 16, or 32. Big-endian-flag is non-zero (true) if multi-byte pixels are interpreted as big endian. Although the depth should be consistent with the bits-per-pixel and the various -max values, clients do not use it when interpreting pixel data.

If true-color-flag is non-zero (true), then the last six items specify how to extract the red, green, and blue intensities from the pixel value. Red-max is the maximum red value and must be 2^N - 1, where N is the number of bits used for red. Note the -max values are always in big endian order. Red-shift is the number of shifts needed

to get the red value in a pixel to the least significant bit. Green-max, green-shift, blue-max, and blue-shift are similar for green and blue. For example, to find the red value (between 0 and red-max) from a given pixel, do the following:

- Swap the pixel value according to big-endian-flag, e.g., if big-endian-flag is zero (false) and host byte order is big endian, then swap.
- Shift right by red-shift.
- AND with red-max (in host byte order).

If true-color-flag is zero (false), then the server uses pixel values that are not directly composed from the red, green, and blue intensities, but serve as indices into a color map. Entries in the color map are set by the server using the SetColorMapEntries message (See Section 7.6.2).

## 7.5. Client-to-Server Messages

The client-to-server message types defined in this document are:

| Number | Name |
|---|---|
| 0 | SetPixelFormat |
| 2 | SetEncodings |
| 3 | FramebufferUpdateRequest |
| 4 | KeyEvent |
| 5 | PointerEvent |
| 6 | ClientCutText |

Other message types exist but are not publicly documented. Before sending a message other than those described in this document, a client must have determined that the server supports the relevant extension by receiving an appropriate extension-specific confirmation from the server.

### 7.5.1. SetPixelFormat

A SetPixelFormat message sets the format in which pixel values should be sent in FramebufferUpdate messages. If the client does not send a SetPixelFormat message, then the server sends pixel values in its natural format as specified in the ServerInit message (Section 7.3.2).

If true-color-flag is zero (false), then this indicates that a "color map" is to be used. The server can set any of the entries in the color map using the SetColorMapEntries message (Section 7.6.2). Immediately after the client has sent this message, the contents of the color map are undefined, even if entries had previously been set by the server.

| No. of bytes | Type [Value] | Description |
|---|---|---|
| 1 | U8 [0] | message-type |
| 3 | | padding |
| 16 | PIXEL_FORMAT | pixel-format |

PIXEL_FORMAT is as described in Section 7.4.

### 7.5.2. SetEncodings

A SetEncodings message sets the encoding types in which pixel data can be sent by the server. The order of the encoding types given in this message is a hint by the client as to its preference (the first encoding specified being most preferred). The server may or may not choose to make use of this hint. Pixel data may always be sent in raw encoding even if not specified explicitly here.

In addition to genuine encodings, a client can request "pseudo-encodings" to declare to the server that it supports certain extensions to the protocol. A server that does not support the extension will simply ignore the pseudo-encoding. Note that this means the client must assume that the server does not support the extension until it gets some extension-specific confirmation from the server.

See Section 7.7 for a description of each encoding and Section 7.8 for the meaning of pseudo-encodings.

| No. of bytes | Type [Value] | Description |
|---|---|---|
| 1 | U8 [2] | message-type |
| 1 | | padding |
| 2 | U16 | number-of-encodings |

This is followed by number-of-encodings repetitions of the following:

| No. of bytes | Type [Value] | Description |
|---|---|---|
| 4 | S32 | encoding-type |

### 7.5.3. FramebufferUpdateRequest

A FramebufferUpdateRequest message notifies the server that the client is interested in the area of the framebuffer specified by x-position, y-position, width, and height. The server usually responds to a FramebufferUpdateRequest by sending a FramebufferUpdate. A single FramebufferUpdate may be sent in reply to several FramebufferUpdateRequests.

The server assumes that the client keeps a copy of all parts of the framebuffer in which it is interested. This means that normally the server only needs to send incremental updates to the client.

If the client has lost the contents of a particular area that it needs, then the client sends a FramebufferUpdateRequest with incremental set to zero (false). This requests that the server send the entire contents of the specified area as soon as possible. The area will not be updated using the CopyRect encoding.

If the client has not lost any contents of the area in which it is interested, then it sends a FramebufferUpdateRequest with incremental set to non-zero (true). If and when there are changes to the specified area of the framebuffer, the server will send a FramebufferUpdate. Note that there may be an indefinite period between the FramebufferUpdateRequest and the FramebufferUpdate.

In the case of a fast client, the client may want to regulate the rate at which it sends incremental FramebufferUpdateRequests to avoid excessive network traffic.

| No. of bytes | Type [Value] | Description |
|---|---|---|
| 1 | U8 [3] | message-type |
| 1 | U8 | incremental |
| 2 | U16 | x-position |
| 2 | U16 | y-position |
| 2 | U16 | width |
| 2 | U16 | height |

### 7.5.4. KeyEvent

A KeyEvent message indicates a key press or release. Down-flag is non-zero (true) if the key is now pressed, and zero (false) if it is now released. The key itself is specified using the "keysym" values defined by the X Window System, even if the client or server is not running the X Window System.

| No. of bytes | Type [Value] | Description |
|---|---|---|
| 1 | U8 [4] | message-type |
| 1 | U8 | down-flag |
| 2 | | padding |
| 4 | U32 | key |

For most ordinary keys, the keysym is the same as the corresponding ASCII value. For full details, see [XLIBREF] or see the header file <X11/keysymdef.h> in the X Window System distribution. Some other common keys are:

| Key name | Keysym value (hex) |
|---|---|
| BackSpace | 0xff08 |
| Tab | 0xff09 |
| Return or Enter | 0xff0d |
| Escape | 0xff1b |
| Insert | 0xff63 |
| Delete | 0xffff |
| Home | 0xff50 |
| End | 0xff57 |
| Page Up | 0xff55 |
| Page Down | 0xff56 |
| Left | 0xff51 |
| Up | 0xff52 |
| Right | 0xff53 |
| Down | 0xff54 |
| F1 | 0xffbe |
| F2 | 0xffbf |
| F3 | 0xffc0 |
| F4 | 0xffc1 |
| ... | ... |
| F12 | 0xffc9 |
| Shift (left) | 0xffe1 |
| Shift (right) | 0xffe2 |
| Control (left) | 0xffe3 |
| Control (right) | 0xffe4 |
| Meta (left) | 0xffe7 |
| Meta (right) | 0xffe8 |
| Alt (left) | 0xffe9 |
| Alt (right) | 0xffea |

The interpretation of keysyms is a complex area. In order to be as widely interoperable as possible, the following guidelines should be followed:

- The "shift state" (i.e., whether either of the Shift keysyms is down) should only be used as a hint when interpreting a keysym. For example, on a US keyboard the '#' character is shifted, but on a UK keyboard it is not. A server with a US keyboard receiving a '#' character from a client with a UK keyboard will not have been sent any shift presses. In this case, it is likely that the server will internally need to simulate a shift press on its local system in order to get a '#' character and not a '3'.

- The difference between upper and lower case keysyms is significant. This is unlike some of the keyboard processing in the X Window System that treats them as the same. For example, a server receiving an upper case 'A' keysym without any shift presses should interpret it as an upper case 'A'. Again this may involve an internal simulated shift press.

- Servers should ignore "lock" keysyms such as CapsLock and NumLock where possible. Instead, they should interpret each character-based keysym according to its case.

- Unlike Shift, the state of modifier keys such as Control and Alt should be taken as modifying the interpretation of other keysyms. Note that there are no keysyms for ASCII control characters such as Ctrl-A -- these should be generated by clients sending a Control press followed by an 'a' press.

- On a client where modifiers like Control and Alt can also be used to generate character-based keysyms, the client may need to send extra "release" events in order that the keysym is interpreted correctly. For example, on a German PC keyboard, Ctrl-Alt-Q generates the '@' character. In this case, the client needs to send simulated release events for Control and Alt in order that the '@' character is interpreted correctly, since Ctrl-Alt-@ may mean something completely different to the server.

- There is no universal standard for "backward tab" in the X Window System. On some systems shift+tab gives the keysym "ISO_Left_Tab", on others it gives a private "BackTab" keysym, and on others it gives "Tab" and applications tell from the shift state that it means backward-tab rather than forward-tab. In the RFB protocol, the latter approach is preferred. Clients should generate a shifted Tab rather than ISO_Left_Tab. However, to be backwards-compatible with existing clients, servers should also recognize ISO_Left_Tab as meaning a shifted Tab.

- Modern versions of the X Window System handle keysyms for Unicode characters, consisting of the Unicode character with the hex 1000000 bit set. For maximum compatibility, if a key has both a Unicode and a legacy encoding, clients should send the legacy encoding.

- Some systems give a special interpretation to key combinations such as Ctrl-Alt-Delete. RFB clients typically provide a menu or toolbar function to send such key combinations. The RFB protocol does not treat them specially; to send Ctrl-Alt-Delete, the client sends the key presses for left or right Control, left or right

Alt, and Delete, followed by the key releases. Many RFB servers accept Shift-Ctrl-Alt-Delete as a synonym for Ctrl-Alt-Delete that can be entered directly from the keyboard.

### 7.5.5. PointerEvent

A PointerEvent message 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 means up, 1 means 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 upwards is represented by a press and release of button 4, and each step downwards is represented by a press and release of button 5.

| No. of bytes | Type [Value] | Description |
|---|---|---|
| 1 | U8 [5] | message-type |
| 1 | U8 | button-mask |
| 2 | U16 | x-position |
| 2 | U16 | y-position |

### 7.5.6. ClientCutText

RFB provides limited support for synchronizing the "cut buffer" of selected text between client and server. This message tells the server that the client has new ISO 8859-1 (Latin-1) text in its cut buffer. Ends of lines are represented by the newline character (hex 0a) alone. No carriage-return (hex 0d) is used. There is no way to transfer text outside the Latin-1 character set.

| No. of bytes | Type [Value] | Description |
|---|---|---|
| 1 | U8 [6] | message-type |
| 3 | | padding |
| 4 | U32 | length |
| length | U8 array | text |

### 7.6. Server-to-Client Messages

The server-to-client message types defined in this document are:

| Number | Name |
|---|---|
| 0 | FramebufferUpdate |
| 1 | SetColorMapEntries |
| 2 | Bell |
| 3 | ServerCutText |

Other private message types exist but are not publicly documented. Before sending a message other than those described in this document a server must have determined that the client supports the relevant extension by receiving some extension-specific confirmation from the client -- usually a request for a given pseudo-encoding.

#### 7.6.1. FramebufferUpdate

A framebuffer update consists of a sequence of rectangles of pixel data that the client should put into its framebuffer. It is sent in response to a FramebufferUpdateRequest from the client. Note that there may be an indefinite period between the FramebufferUpdateRequest and the FramebufferUpdate.

| No. of bytes | Type [Value] | Description |
|---|---|---|
| 1 | U8 [0] | message-type |
| 1 | | padding |
| 2 | U16 | number-of-rectangles |

This header is followed by number-of-rectangles rectangles of pixel data. Each rectangle starts with a rectangle header:

| No. of bytes | Type [Value] | Description |
|---|---|---|
| 2 | U16 | x-position |
| 2 | U16 | y-position |
| 2 | U16 | width |
| 2 | U16 | height |
| 4 | S32 | encoding-type |

The rectangle header is followed by the pixel data in the specified encoding. See Section 7.7 for the format of the data for each encoding and Section 7.8 for the meaning of pseudo-encodings.

### 7.6.2. SetColorMapEntries

When the pixel format uses a "color map", this message tells the client that the specified pixel values should be mapped to the given RGB values. Note that this message may only update part of the color map. This message should not be sent by the server until after the client has sent at least one FramebufferUpdateRequest, and only when the agreed pixel format uses a color map.

Color map values are always 16 bits, with the range of values running from 0 to 65535, regardless of the display hardware in use. The color map value for white, for example, is 65535,65535,65535.

The message starts with a header describing the range of colormap entries to be updated.

| No. of bytes | Type [Value] | Description |
|---|---|---|
| 1 | U8 [1] | message-type |
| 1 | | padding |
| 2 | U16 | first-color |
| 2 | U16 | number-of-colors |

This header is followed by number-of-colors RGB values, each of which is in this format:

| No. of bytes | Type [Value] | Description |
|---|---|---|
| 2 | U16 | red |
| 2 | U16 | green |
| 2 | U16 | blue |

### 7.6.3. Bell

A Bell message makes an audible signal on the client if it provides one.

| No. of bytes | Type [Value] | Description |
|---|---|---|
| 1 | U8 [2] | message-type |

### 7.6.4. ServerCutText

The server has new ISO 8859-1 (Latin-1) text in its cut buffer. Ends of lines are represented by the newline character (hex 0a) alone. No carriage-return (hex 0d) is used. There is no way to transfer text outside the Latin-1 character set.

| No. of bytes | Type [Value] | Description |
|---|---|---|
| 1 | U8 [3] | message-type |
| 3 | | padding |
| 4 | U32 | length |
| length | U8 array | text |

## 7.7. Encodings

The encodings defined in this document are:

| Number | Name |
|---|---|
| 0 | Raw |
| 1 | CopyRect |
| 2 | RRE |
| 5 | Hextile |
| 15 | TRLE |
| 16 | ZRLE |
| -239 | Cursor pseudo-encoding |
| -223 | DesktopSize pseudo-encoding |

Other encoding types exist but are not publicly documented.

### 7.7.1. Raw Encoding

The simplest encoding type is raw pixel data. In this case, the data consists of width*height pixel values (where width and height are the width and height of the rectangle). The values simply represent each pixel in left-to-right scan line order. All RFB clients must be able to handle pixel data in this raw encoding, and RFB servers should only produce raw encoding unless the client specifically asks for some other encoding type.

| No. of bytes | Type [Value] | Description |
|---|---|---|
| width*height*bytesPerPixel | PIXEL array | pixels |

### 7.7.2. CopyRect Encoding

The CopyRect (copy rectangle) encoding is a very simple and efficient encoding that can be used when the client already has the same pixel data elsewhere in its framebuffer. The encoding on the wire simply consists of an X,Y coordinate. This gives a position in the framebuffer from which the client can copy the rectangle of pixel data. This can be used in a variety of situations, the most common of which are when the user moves a window across the screen, and when the contents of a window are scrolled.

| No. of bytes | Type [Value] | Description |
|---|---|---|
| 2 | U16 | src-x-position |
| 2 | U16 | src-y-position |

For maximum compatibility, the source rectangle of a CopyRect should not include pixels updated by previous entries in the same FramebufferUpdate message.

### 7.7.3. RRE Encoding

Note: RRE encoding is obsolescent. In general, ZRLE and TRLE encodings are more compact.

RRE stands for rise-and-run-length encoding. As its name implies, it is essentially a two-dimensional analogue of run-length encoding. RRE-encoded rectangles arrive at the client in a form that can be

rendered immediately by the simplest of graphics engines. RRE is not appropriate for complex desktops, but can be useful in some situations.

The basic idea behind RRE is the partitioning of a rectangle of pixel data into rectangular subregions (subrectangles) each of which consists of pixels of a single value, and the union of which comprises the original rectangular region. The near-optimal partition of a given rectangle into such subrectangles is relatively easy to compute.

The encoding consists of a background pixel value, Vb (typically the most prevalent pixel value in the rectangle) and a count N, followed by a list of N subrectangles, each of which consists of a tuple <v,x,y,w,h> where v (which should be different from Vb) is the pixel value, (x,y) are the coordinates of the subrectangle relative to the top-left corner of the rectangle, and (w,h) are the width and height of the subrectangle. The client can render the original rectangle by drawing a filled rectangle of the background pixel value and then drawing a filled rectangle corresponding to each subrectangle.

On the wire, the data begins with the header:

| No. of bytes | Type [Value] | Description |
|---|---|---|
| 4 | U32 | number-of-subrectangles |
| bytesPerPixel | PIXEL | background-pixel-value |

This is followed by number-of-subrectangles instances of the following structure:

| No. of bytes | Type [Value] | Description |
|---|---|---|
| bytesPerPixel | PIXEL | subrect-pixel-value |
| 2 | U16 | x-position |
| 2 | U16 | y-position |
| 2 | U16 | width |
| 2 | U16 | height |

### 7.7.4. Hextile Encoding

Note: Hextile encoding is obsolescent. In general, ZRLE and TRLE encodings are more compact.

Hextile is a variation on RRE. Rectangles are split up into 16x16 tiles, allowing the dimensions of the subrectangles to be specified in 4 bits each, 16 bits in total. The rectangle is split into tiles starting at the top left going in left-to-right, top-to-bottom order. The encoded contents of the tiles simply follow one another in the predetermined order. If the width of the whole rectangle is not an exact multiple of 16, then the width of the last tile in each row will be correspondingly smaller. Similarly, if the height of the whole rectangle is not an exact multiple of 16, then the height of each tile in the final row will also be smaller.

Each tile is either encoded as raw pixel data, or as a variation on RRE. Each tile has a background pixel value, as before. The background pixel value does not need to be explicitly specified for a given tile if it is the same as the background of the previous tile. However, the background pixel value may not be carried over if the previous tile was raw. If all of the subrectangles of a tile have the same pixel value, this can be specified once as a foreground pixel value for the whole tile. As with the background, the foreground pixel value can be left unspecified, meaning it is carried over from the previous tile. The foreground pixel value may not be carried over if the previous tile was raw or had the SubrectsColored bit set. It may, however, be carried over from a previous tile with the AnySubrects bit clear, as long as that tile itself carried over a valid foreground from its previous tile.

The data consists of each tile encoded in order. Each tile begins with a subencoding type byte, which is a mask made up of a number of bits:

| No. of bytes | Type [Value] | Description |
|---|---|---|
| 1 | U8 | subencoding-mask: |
| | [1] | Raw |
| | [2] | BackgroundSpecified |
| | [4] | ForegroundSpecified |
| | [8] | AnySubrects |
| | [16] | SubrectsColored |

If the Raw bit is set, then the other bits are irrelevant; width*height pixel values follow (where width and height are the width and height of the tile). Otherwise, the other bits in the mask are as follows:

BackgroundSpecified
If set, a pixel value of bytesPerPixel bytes follows and specifies the background color for this tile. The first non-raw tile in a rectangle must have this bit set. If this bit isn't set, then the background is the same as the last tile.

ForegroundSpecified
If set, a pixel value of bytesPerPixel bytes follows and specifies the foreground color to be used for all subrectangles in this tile.

If this bit is set, then the SubrectsColored bit must be zero.

AnySubrects
If set, a single byte follows and gives the number of subrectangles following. If not set, there are no subrectangles (i.e., the whole tile is just solid background color).

SubrectsColored
If set, then each subrectangle is preceded by a pixel value giving the color of that subrectangle, so a subrectangle is:

| No. of bytes | Type [Value] | Description |
|---|---|---|
| bytesPerPixel | PIXEL | subrect-pixel-value |
| 1 | U8 | x-and-y-position |
| 1 | U8 | width-and-height |

If not set, all subrectangles are the same color -- the foreground color; if the ForegroundSpecified bit wasn't set, then the foreground is the same as the last tile. A subrectangle is:

| No. of bytes | Type [Value] | Description |
|---|---|---|
| 1 | U8 | x-and-y-position |
| 1 | U8 | width-and-height |

The position and size of each subrectangle is specified in two bytes, x-and-y-position and width-and-height. The most significant 4 bits of x-and-y-position specify the X position, the least significant specify the Y position. The most significant 4 bits of width-and-height specify the width minus 1, the least significant specify the height minus 1.

### 7.7.5. TRLE

TRLE stands for Tiled Run-Length Encoding, and combines tiling, palettization, and run-length encoding. The rectangle is divided into tiles of 16x16 pixels in left-to-right, top-to-bottom order, similar to Hextile. If the width of the rectangle is not an exact multiple of 16, then the width of the last tile in each row is smaller, and if the height of the rectangle is not an exact multiple of 16, then the height of each tile in the final row is smaller.

TRLE makes use of a new type CPIXEL (compressed pixel). This is the same as a PIXEL for the agreed pixel format, except as a special case, it uses a more compact format if true-color-flag is non-zero, bits-per-pixel is 32, depth is 24 or less, and all of the bits making up the red, green, and blue intensities fit in either the least significant 3 bytes or the most significant 3 bytes. If all of these are the case, a CPIXEL is only 3 bytes long, and contains the least significant or the most significant 3 bytes as appropriate. bytesPerCPixel is the number of bytes in a CPIXEL.

Each tile begins with a subencoding type byte. The top bit of this byte is set if the tile has been run-length encoded, clear otherwise. The bottom 7 bits indicate the size of the palette used: zero means no palette, 1 means that the tile is of a single color, and 2 to 127 indicate a palette of that size. The special subencoding values 129 and 127 indicate that the palette is to be reused from the last tile that had a palette, with and without RLE, respectively.

Note: in this discussion, the div(a,b) function means the result of dividing a/b truncated to an integer.

The possible values of subencoding are:

0: Raw pixel data. width*height pixel values follow (where width and height are the width and height of the tile):

| No. of bytes | Type [Value] | Description |
|---|---|---|
| width*height*BytesPerCPixel | CPIXEL array | pixels |

1: A solid tile consisting of a single color. The pixel value follows:

| No. of bytes | Type [Value] | Description |
|---|---|---|
| bytesPerCPixel | CPIXEL | pixelValue |

2 to 16: Packed palette types. The paletteSize is the value of the subencoding, which is followed by the palette, consisting of paletteSize pixel values. The packed pixels follow, with each pixel represented as a bit field yielding a zero-based index into the palette. For paletteSize 2, a 1-bit field is used; for paletteSize 3 or 4, a 2-bit field is used; and for paletteSize from 5 to 16, a 4-bit field is used. The bit fields are packed into bytes, with the most significant bits representing the leftmost pixel (i.e., big endian). For tiles not a multiple of 8, 4, or 2 pixels wide (as appropriate), padding bits are used to align each row to an exact number of bytes.

| No. of bytes | Type [Value] | Description |
|---|---|---|
| paletteSize*bytesPerCPixel<br>m | CPIXEL array<br>U8 array | palette<br>packedPixels |

where m is the number of bytes representing the packed pixels. For paletteSize of 2, this is div(width+7,8)*height; for paletteSize of 3 or 4, this is div(width+3,4)*height; or for paletteSize of 5 to 16, this is div(width+1,2)*height.

17 to 126: Unused. (Packed palettes of these sizes would offer no advantage over palette RLE).

127: Packed palette with the palette reused from the previous tile. The subencoding byte is followed by the packed pixels as described above for packed palette types.

128: Plain RLE. The data consists of a number of runs, repeated until the tile is done. Runs may continue from the end of one row to the beginning of the next. Each run is represented by a single pixel value followed by the length of the run. The length is represented as one or more bytes. The length is calculated as one more than the sum of all the bytes representing the length. Any byte value other than 255 indicates the final byte. So for

example, length 1 is represented as [0], 255 as [254], 256 as [255,0], 257 as [255,1], 510 as [255,254], 511 as [255,255,0], and so on.

| No. of bytes | Type [Value] | Description |
|---|---|---|
| bytesPerCPixel | CPIXEL | pixelValue |
| div(runLength - 1, 255) | U8 array | 255 |
| 1 | U8 | (runLength-1) mod 255 |

129: Palette RLE with the palette reused from the previous tile. Followed by a number of runs, repeated until the tile is done, as described below for 130 to 255.

130 to 255: Palette RLE. Followed by the palette, consisting of paletteSize = (subencoding - 128) pixel values:

| No. of bytes | Type [Value] | Description |
|---|---|---|
| paletteSize*bytesPerCPixel | CPIXEL array | palette |

Following the palette is, as with plain RLE, a number of runs, repeated until the tile is done. A run of length one is represented simply by a palette index:

| No. of bytes | Type [Value] | Description |
|---|---|---|
| 1 | U8 | paletteIndex |

A run of length more than one is represented by a palette index with the top bit set, followed by the length of the run as for plain RLE.

| No. of bytes | Type [Value] | Description |
|---|---|---|
| 1 | U8 | paletteIndex + 128 |
| div(runLength - 1, 255) | U8 array | 255 |
| 1 | U8 | (runLength-1) mod 255 |

### 7.7.6. ZRLE

ZRLE stands for Zlib (see [RFC1950] and [RFC1951]) Run-Length Encoding, and combines an encoding similar to TRLE with zlib compression. On the wire, the rectangle begins with a 4-byte length field, and is followed by that many bytes of zlib-compressed data. A single zlib "stream" object is used for a given RFB protocol connection, so that ZRLE rectangles must be encoded and decoded strictly in order.

| No. of bytes | Type [Value] | Description |
|---|---|---|
| 4 | U32 | length |
| length | U8 array | zlibData |

The zlibData when uncompressed represents tiles in left-to-right, top-to-bottom order, similar to TRLE, but with a tile size of 64x64 pixels. If the width of the rectangle is not an exact multiple of 64, then the width of the last tile in each row is smaller, and if the height of the rectangle is not an exact multiple of 64, then the height of each tile in the final row is smaller.

The tiles are encoded in exactly the same way as TRLE, except that subencoding may not take the values 127 or 129, i.e., palettes cannot be reused between tiles.

The server flushes the zlib stream to a byte boundary at the end of each ZRLE-encoded rectangle. It need not flush the stream between tiles within a rectangle. Since the zlibData for a single rectangle can potentially be quite large, clients can incrementally decode and interpret the zlibData but must not assume that encoded tile data is byte aligned.

## 7.8. Pseudo-Encodings

An update rectangle with a "pseudo-encoding" does not directly represent pixel data but instead allows the server to send arbitrary data to the client. How this data is interpreted depends on the pseudo-encoding.

### 7.8.1. Cursor Pseudo-Encoding

A client that requests the Cursor pseudo-encoding is declaring that it is capable of drawing a pointer cursor locally. This can significantly improve perceived performance over slow links. The server sets the cursor shape by sending a rectangle with the Cursor

pseudo-encoding as part of an update. The rectangle's x-position and y-position indicate the hotspot of the cursor, and width and height indicate the width and height of the cursor in pixels. The data consists of width*height raw pixel values followed by a shape bitmask, with one bit corresponding to each pixel in the cursor rectangle. The bitmask consists of left-to-right, top-to-bottom scan lines, where each scan line is padded to a whole number of bytes, the number being div(width+7,8). Within each byte, the most significant bit represents the leftmost pixel; a bit set to 1 means the corresponding pixel in the cursor is valid.

| No. of bytes | Type [Value] | Description |
|---|---|---|
| width*height*bytesPerPixel | PIXEL array | cursor-pixels |
| div(width+7,8)*height | U8 array | bitmask |

### 7.8.2. DesktopSize Pseudo-Encoding

A client that requests the DesktopSize pseudo-encoding is declaring that it is capable of coping with a change in the framebuffer width and height. The server changes the desktop size by sending a rectangle with the DesktopSize pseudo-encoding as the last rectangle in an update. The rectangle's x-position and y-position are ignored, and width and height indicate the new width and height of the framebuffer.

There is no further data associated with the rectangle. After changing the desktop size, the server must assume that the client no longer has the previous framebuffer contents. This will usually result in a complete update of the framebuffer at the next update. However, for maximum interoperability with existing servers the client should preserve the top-left portion of the framebuffer between the old and new sizes.

## 8. IANA Considerations

IANA has allocated port 5900 to the RFB protocol. The other port numbers mentioned in Section 2 are called out for historical context and do not match IANA allocations.

Future assignments to the IANA registries created by this specification are to be made through either "Expert Review" or "IESG Approval" (if there is no currently appointed expert) as defined in [RFC5226].

### 8.1. RFB Security Types

#### 8.1.1. Registry Name

The name of this registry is "Remote Framebuffer Security Types".

#### 8.1.2. Registry Contents

IANA established a registry for security types that are used with the RFB protocol.

The initial entries in the registry are:

| Number | Name | References |
|---|---|---|
| 0 | Invalid | (this document) |
| 1 | None | (this document) |
| 2 | VNC Authentication | (this document) |
| 3 to 15 | RealVNC | (historic assignment) |
| 16 | Tight | (historic assignment) |
| 17 | Ultra | (historic assignment) |
| 18 | TLS | (historic assignment) |
| 19 | VeNCrypt | (historic assignment) |
| 20 | GTK-VNC SASL | (historic assignment) |
| 21 | MD5 hash authentication | (historic assignment) |
| 22 | Colin Dean xvp | (historic assignment) |
| 128 to 255 | RealVNC | (historic assignment) |

### 8.2. Client-to-Server Message Types

#### 8.2.1. Registry Name

The name of this registry is "Remote Framebuffer Client-to-Server Message Types".

#### 8.2.2. Registry Contents

IANA established a registry for client-to-server message types that are used with the RFB protocol.

The initial entries in the registry are:

| Number | Name | References |
|---|---|---|
| 0 | SetPixelFormat | (this document) |
| 2 | SetEncodings | (this document) |
| 3 | FramebufferUpdateRequest | (this document) |
| 4 | KeyEvent | (this document) |
| 5 | PointerEvent | (this document) |
| 6 | ClientCutText | (this document) |
| 127 | VMWare | (historic assignment) |
| 128 | Nokia Terminal Mode Spec | (historic assignment) |
| 249 | OLIVE Call Control | (historic assignment) |
| 250 | Colin Dean xvp | (historic assignment) |
| 251 | Pierre Ossman SetDesktopSize | (historic assignment) |
| 252 | tight | (historic assignment) |
| 253 | gii | (historic assignment) |
| 254 | VMWare | (historic assignment) |
| 255 | Anthony Liguori | (historic assignment) |

### 8.3. Server-to-Client Message Types

#### 8.3.1. Registry Name

The name of this registry is "Remote Framebuffer Server-to-Client Message Types".

#### 8.3.2. Registry Contents

IANA established a registry for server-to-client message types that are used with the RFB protocol.

The initial entries in the registry are:

| Number | Name | References |
|---|---|---|
| 0 | FramebufferUpdate | (this document) |
| 1 | SetColourMapEntries | (this document) |
| 2 | Bell | (this document) |
| 3 | ServerCutText | (this document) |
| 127 | VMWare | (historic assignment) |
| 128 | Nokia Terminal Mode Spec | (historic assignment) |
| 249 | OLIVE Call Control | (historic assignment) |
| 250 | Colin Dean xvp | (historic assignment) |
| 252 | tight | (historic assignment) |
| 253 | gii | (historic assignment) |
| 254 | VMWare | (historic assignment) |
| 255 | Anthony Liguori | (historic assignment) |

### 8.4. RFB Encoding Types

#### 8.4.1. Registry Name

The name of this registry is "Remote Framebuffer Encoding Types".

#### 8.4.2. Registry Contents

IANA established a registry for encoding types that are used with the RFB protocol.

The initial entries in the registry are:

| Number | Name | References |
|---|---|---|
| 0 | Raw | (this document) |
| 1 | CopyRect | (this document) |
| 2 | RRE | (this document) |
| 5 | Hextile | (this document) |
| 16 | ZRLE | (this document) |
| -239 | Cursor pseudo-encoding | (this document) |
| -223 | DesktopSize pseudo-encoding | (this document) |
| 4 | CoRRE | (historic assignment) |
| 6 | zlib | (historic assignment) |
| 7 | tight | (historic assignment) |
| 8 | zlibhex | (historic assignment) |
| 15 | TRLE | (this document) |
| 17 | Hitachi ZYWRLE | (historic assignment) |
| 1024 to 1099 | RealVNC | (historic assignment) |
| -1 to -222 | tight options | (historic assignment) |
| -224 to -238 | tight options | (historic assignment) |
| -240 to -256 | tight options | (historic assignment) |
| -257 to -272 | Anthony Liguori | (historic assignment) |
| -273 to -304 | VMWare | (historic assignment) |
| -305 | gii | (historic assignment) |
| -306 | popa | (historic assignment) |
| -307 | Peter Astrand DesktopName | (historic assignment) |
| -308 | Pierre Ossman ExtendedDesktopSize | (historic assignment) |
| -309 | Colin Dean xvp | (historic assignment) |
| -310 | OLIVE Call Control | (historic assignment) |
| -412 to -512 | TurboVNC fine-grained quality level | (historic assignment) |

| | | |
|---|---|---|
| -523 to -524 | Nokia Terminal Mode Spec | (historic assignment) |
| -763 to -768 | TurboVNC subsampling level | (historic assignment) |
| 0x574d5600 to 0x574d56ff | VMWare | (historic assignment) |

## 9. Security

The RFB protocol as defined here provides no security beyond the optional and cryptographically weak password check described in Section 7.2.2. In particular, it provides no protection against observation of or tampering with the data stream. It has typically been used on secure physical or virtual networks.

Security methods beyond those described here may be used to protect the integrity of the data. The client and server might agree to use an extended security type to encrypt the session, or the session might be transmitted over a secure channel such as IPsec [RFC4301] or SSH [RFC4254].

## 10. Acknowledgements

James Weatherall, Andy Harter, and Ken Wood also contributed to the design of the RFB protocol.



## 11. References

### 11.1. Normative References

[RFC1950] Deutsch, L. and J-L. Gailly, "ZLIB Compressed Data Format Specification version 3.3", RFC 1950, May 1996.

[RFC1951] Deutsch, P., "DEFLATE Compressed Data Format Specification version 1.3", RFC 1951, May 1996.

[XLIBREF] Nye, A., "XLIB Reference Manual R5", June 1994.

### 11.2. Informative References

[RFC4254] Ylonen, T. and C. Lonvick, "The Secure Shell (SSH) Connection Protocol", RFC 4254, January 2006.

[RFC4301] Kent, S. and K. Seo, "Security Architecture for the Internet Protocol", RFC 4301, December 2005.

[RFC5226] Narten, T. and H. Alvestrand, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 5226, May 2008.

## Appendix A. Differences in Earlier Protocol Versions

For maximum interoperability, clients and servers should be prepared to fall back to the earlier 3.3 and 3.7 versions of the RFB protocol. Any version reported other than 3.7 or 3.8 should be treated as 3.3.

All of the differences occur in the initial handshake phase. Once the session reaches the ClientInit and ServerInit messages, all three protocol versions are identical. Even within a protocol version, clients and servers may support different subsets of the encoding and pseudo-encoding types.

### A.1. Differences in the Version 3.3 Protocol

The ProtocolVersion message is:

RFB 003.003\n (hex 52 46 42 20 30 30 33 2e 30 30 33 0a)

In the security handshake (Section 7.1.2), rather than a two-way negotiation, the server decides the security type and sends a single word:

| No. of bytes | Type [Value] | Description |
|---|---|---|
| 4 | U32 | security-type |

The security-type may only take the value 0, 1, or 2. A value of 0 means that the connection has failed and is followed by a string giving the reason, as described in Section 7.1.2.

If the security-type is 1, for no authentication, the server does not send the SecurityResult message but proceeds directly to the initialization messages (Section 7.3).

In VNC Authentication (Section 7.2.2), if the authentication fails, the server sends the SecurityResult message, but does not send an error message before closing the connection.

### A.2. Differences in the Version 3.7 Protocol

The ProtocolVersion message is:

RFB 003.007\n (hex 52 46 42 20 30 30 33 2e 30 30 37 0a)

After the security handshake, if the security-type is 1, for no authentication, the server does not send the SecurityResult message but proceeds directly to the initialization messages (Section 7.3).

In VNC Authentication (Section 7.2.2), if the authentication fails, the server sends the SecurityResult message, but does not send an error message before closing the connection.

## Authors' Addresses

Tristan Richardson
RealVNC Ltd.
Betjeman House, 104 Hills Road
Cambridge CB2 1LQ
UK

Phone: +44 1223 310400
EMail: standards@realvnc.com
URI: http://www.realvnc.com

John Levine
RealVNC Ltd.

Phone: +44 1223 790005
EMail: standards@taugh.com
URI: http://jl.ly