The RFB Protocol

Tristan Richardson
RealVNC Ltd
(formerly of Olivetti Research Ltd / AT&T Labs Cambridge) *

Version 3.8
Last updated 5 October 2006

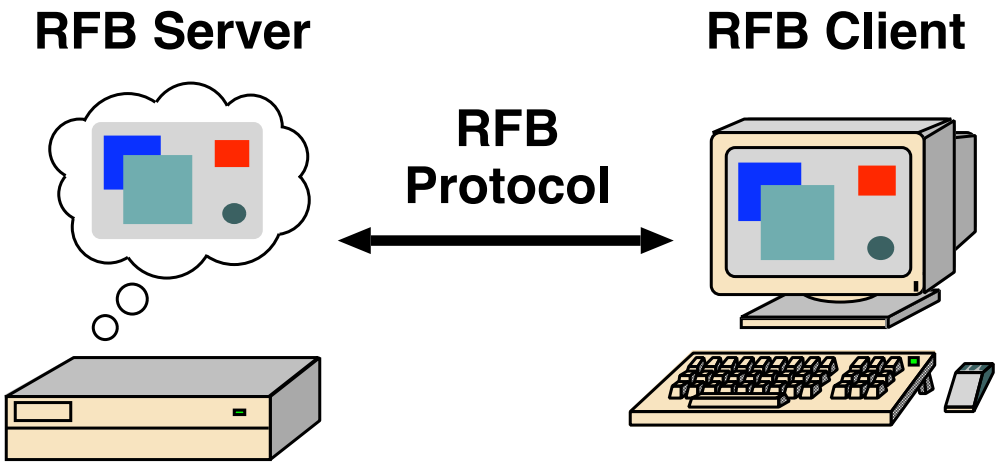
Contents

*James Weatherall, Andy Harter and Ken Wood also helped in the design of the RFB protocol

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 (Virtual Network Computing).

The remote endpoint where the user sits (i.e. the display plus keyboard and/or 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 truly 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.

2 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"*. At first glance 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* (or simply *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 disjoint but this is not necessarily the case.

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 becomes. With typical applications, changes to the same area of the framebuffer tend to happen soon after one another. With a slow client and/or network, transient states of the framebuffer can be ignored, resulting in less network traffic and less drawing for the client.

3 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 synthesised from other non-standard I/O devices. For example, a pen-based handwriting recognition engine might generate keyboard events.

4 Representation of pixel data

Initial interaction between the RFB client and server involves a negotiation of the *format* and *encoding* with which pixel data will be sent. This negotiation has been designed to make the job of the client as easy as possible. The bottom line is that 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 which is easier for the server to produce.

Pixel *format* refers to the representation of individual colours by pixel values. The most common pixel formats are 24-bit or 16-bit "true colour", where bit-fields within the pixel value translate directly to red, green and blue intensities, and 8-bit "colour map" where an arbitrary mapping can be used to translate from pixel values to the RGB intensities.

Encoding refers to how a rectangle of pixel data will be sent on the wire. 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*, *Hextile* and *ZRLE*. In

practice we normally use only the *ZRLE*, *Hextile* and *CopyRect* encodings since they provide the best compression for typical desktops. See section 6.6 for a description of each of the encodings.

5 Protocol extensions

There are a number of ways in which the protocol can be extended:

New encodings A new encoding type can be added to the protocol relatively easily whilst maintaining compatibility with existing clients and servers. Existing servers will simply ignore requests for a new encoding which 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 which 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 6.7 for a description of current pseudo-encodings.

New security types Adding a new security type gives the ultimate flexibility in modifying the behaviour of the protocol without sacrificing compatibility with existing clients and servers. A client and server which 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.

Under no circumstances should you use a different protocol version number. Protocol versions are defined by the maintainers of the RFB protocol, RealVNC Ltd. If you use a different protocol version number then you are not RFB / VNC compatible. To ensure that you stay compatible with the RFB protocol it is important that you contact RealVNC Ltd to make sure that your encoding types and security types do not clash. Please see the RealVNC website at http://www.realvnc.com for details of how to contact us; sending a message to the VNC mailing list is the best way to get in touch and let the rest of the VNC community know.

6 Protocol Messages

The RFB protocol can operate over any reliable transport, either byte-stream or message-based. Conventionally it is used 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 initialisation 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 any message-specific data.

The following descriptions of protocol messages use the basic types `U8`, `U16`, `U32`, `S8`, `S16`, `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).

The type `PIXEL` is taken to mean a pixel value of $bytesPerPixel$ bytes, where $8 \times bytesPerPixel$ is the number of *bits-per-pixel* as agreed by the client and server – either in the *ServerInit* message (section 6.3.2) or a *SetPixelFormat* message (section 6.4.1).

6.1 Handshaking Messages

6.1.1 ProtocolVersion

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 which 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, 3.8 (version 3.5 was wrongly reported by some clients, but this should be interpreted by all servers as 3.3). 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, padded with zeros.

No. of bytes	Value
12	`"RFB 003.003\n"` (hex 52 46 42 20 30 30 33 2e 30 30 33 0a)

or

No. of bytes	Value
12	`"RFB 003.007\n"` (hex 52 46 42 20 30 30 33 2e 30 30 37 0a)

or

No. of bytes	Value
12	`"RFB 003.008\n"` (hex 52 46 42 20 30 30 33 2e 30 30 38 0a)

6.1.2 Security

Once the protocol version has been decided, the server and client must agree on the type of security to be used on the connection.

Version 3.7 onwards The server lists the security types which 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*.

Version 3.3 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 above.

The security types defined in this document are:

Number	Name
0	*Invalid*
1	***None***
2	***VNC Authentication***

Other registered security types are:

Number	Name
5	***RA2***
6	***RA2ne***
16	Tight
17	Ultra
18	TLS
19	VeNCrypt

Once the *security-type* has been decided, data specific to that *security-type* follows (see section 6.2 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 protocol data is over an encrypted or otherwise altered channel.

6.1.3 SecurityResult

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 initialisation phase (section 6.3).

Version 3.8 onwards 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*

Version 3.3 and 3.7 If unsuccessful, the server closes the connection.

6.2 Security Types

6.2.1 None

No authentication is needed and protocol data is to be sent unencrypted.

Version 3.8 onwards The protocol continues with the *SecurityResult* message.

Version 3.3 and 3.7 The protocol passes to the initialisation phase (section 6.3).

6.2.2 VNC Authentication

VNC authentication is to be used and protocol data is to be sent unencrypted. 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, and sends the resulting 16-byte response:

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

The protocol continues with the *SecurityResult* message.

6.3 Initialisation Messages

Once the client and server are sure that they're happy to talk to one another using the agreed security type, the protocol passes to the initialisation phase. The client sends a *ClientInit* message followed by the server sending a *ServerInit* message.

6.3.1 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, zero (false) if it should give exclusive access to this client by disconnecting all other clients.

6.3.2 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*
2	`U16`	*framebuffer-height*
16	`PIXEL_FORMAT`	*server-pixel-format*
4	`U32`	*name-length*
name-length	`U8` array	*name-string*

where `PIXEL_FORMAT` is

No. of bytes	Type [Value]	Description
1	`U8`	*bits-per-pixel*
1	`U8`	*depth*
1	`U8`	*big-endian-flag*
1	`U8`	*true-colour-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*

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 6.4.1).

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 — less than 8-bit pixels are not yet supported. *Big-endian-flag* is non-zero (true) if multi-byte pixels are interpreted as big endian. Of course this is meaningless for 8 bits-per-pixel.

If *true-colour-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 ($= 2^n - 1$ where n is the number of bits used for red). Note this value is 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* and *blue-max*, *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-colour-flag* is zero (false) then the server uses pixel values which are not directly composed from the red, green and blue intensities, but which serve as indices into a colour map. Entries in the colour map are set by the server using the *SetColourMapEntries* message (section 6.5.2).

6.4 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 registered message types are:

Number	Name
255	Anthony Liguori

Note that before sending a message not defined in this document a client must have determined that the server supports the relevant extension by receiving some extension-specific confirmation from the server.

6.4.1 SetPixelFormat

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 6.3.2).

If *true-colour-flag* is zero (false) then this indicates that a "colour map" is to be used. The server can set any of the entries in the colour map using the *SetColourMapEntries* message (section 6.5.2). Immediately after the client has sent this message the colour map is empty, 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*

where `PIXEL_FORMAT` is as described in section 6.3.2:

No. of bytes	Type	[Value]	Description
1	`U8`		*bits-per-pixel*
1	`U8`		*depth*
1	`U8`		*big-endian-flag*
1	`U8`		*true-colour-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*

6.4.2 SetEncodings

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 which 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 6.6 for a description of each encoding and section 6.7 for the meaning of pseudo-encodings.

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

followed by *number-of-encodings* repetitions of the following:

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

6.4.3 FramebufferUpdateRequest

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*. Note however that a single *FramebufferUpdate* may be sent in reply to several *FramebufferUpdateRequest*s.

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.

However, if for some reason the client has lost the contents of a particular area which 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 *FramebufferUpdateRequest*s to avoid hogging the network.

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*

6.4.4 KeyEvent

A key press or release. *Down-flag* is non-zero (true) if the key is now pressed, zero (false) if it is now released. The *key* itself is specified using the "keysym" values defined by 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 The Xlib Reference Manual, published by O'Reilly & Associates, or see the header file <X11/keysymdef.h> from any X Window System installation. Some other common keys are:

Key name	Keysym value
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

Key name	Keysym value
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 used:

- The "shift state" (i.e. whether either of the Shift keysyms are 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 "fake" a shift press on its local system, in order to get a '#' character and not, for example, 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 which treats them as the same. For example, a server receiving an uppercase 'A' keysym

without any shift presses should interpret it as an uppercase 'A'. Again this may involve an internal "fake" 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 viewers sending a Control press followed by an 'a' press.

- On a viewer where modifiers like Control and Alt can also be used to generate character-based keysyms, the viewer 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 viewer needs to send "fake" release events for Control and Alt in order that the '@' character is interpreted correctly (ctrl-alt-@ is likely to 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. Viewers should generate a shifted Tab rather than ISO_Left_Tab. However, to be backwards-compatible with existing viewers, servers should also recognise ISO_Left_Tab as meaning a shifted Tab.

6.4.5 PointerEvent

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 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*

6.4.6 ClientCutText

The client has new ISO 8859-1 (Latin-1) text in its cut buffer. Ends of lines are represented by the linefeed / newline character (value 10) alone. No carriage-return (value 13) is needed. There is currently 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*

6.5 Server to client messages

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

Number	Name
0	*FramebufferUpdate*
1	*SetColourMapEntries*
2	*Bell*
3	*ServerCutText*

Other registered message types are:

Number	Name
255	Anthony Liguori

Note that before sending a message not defined 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.

6.5.1 FramebufferUpdate

A framebuffer update consists of a sequence of rectangles of pixel data which 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 is followed by *number-of-rectangles* rectangles of pixel data. Each rectangle consists of:

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

followed by the pixel data in the specified encoding. See section 6.6 for the format of the data for each encoding and section 6.7 for the meaning of pseudo-encodings.

6.5.2 SetColourMapEntries

When the pixel format uses a “colour map”, this message tells the client that the specified pixel values should be mapped to the given RGB intensities.

No. of bytes	Type [Value]	Description
1	U8 1	*message-type*
1		*padding*
2	U16	*first-colour*
2	U16	*number-of-colours*

followed by *number-of-colours* repetitions of the following:

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

6.5.3 Bell

Ring a bell on the client if it has one.

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

6.5.4 ServerCutText

The server has new ISO 8859-1 (Latin-1) text in its cut buffer. Ends of lines are represented by the linefeed / newline character (value 10) alone. No carriage-return (value 13) is needed. There is currently 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*

6.6 Encodings

The encodings defined in this document are:

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

Other registered encodings are:

Number	Name
4	CoRRE
6,7,8	zlib, tight, zlibhex
-272 to -257	Anthony Liguori
-256 to -240	
-238 to -224	
-222 to -1	tight options

6.6.1 Raw encoding

The simplest encoding type is raw pixel data. In this case the data consists of $width \times height$ pixel values (where $width$ and $height$ are the width and height of the rectangle). The values simply represent each pixel in left-to-right scanline order. All RFB clients must be able to cope with 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
$width \times height \times bytesPerPixel$	`PIXEL` array	*pixels*

6.6.2 CopyRect encoding

The *CopyRect* (copy rectangle) encoding is a very simple and efficient encoding which 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 obvious of which are when the user moves a window across the screen, and when the contents of a window are scrolled. A less obvious use is for optimising drawing of text or other repeating patterns. An intelligent server may be able to send a pattern explicitly only once, and knowing the previous position of the pattern in the framebuffer, send subsequent occurrences of the same pattern using the *CopyRect* encoding.

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

6.6.3 RRE encoding

RRE stands for *rise-and-run-length encoding* and 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 which can be rendered immediately and efficiently 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, V_b (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 ($\neq V_b$) 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*

6.6.4 Hextile encoding

Hextile is a variation on the RRE idea. 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. However, 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. 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.

So 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	***SubrectsColoured***

If the ***Raw*** bit is set then the other bits are irrelevant; $width \times height$ 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 follows which specifies the background colour for this tile:

No. of bytes	Type [Value]	Description
$bytesPerPixel$	PIXEL	*background-pixel-value*

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 follows which specifies the foreground colour to be used for all subrectangles in this tile:

No. of bytes	Type [Value]	Description
$bytesPerPixel$	PIXEL	*foreground-pixel-value*

If this bit is set then the SubrectsColoured bit must be zero.

AnySubrects - if set, a single byte follows giving the number of subrectangles following:

No. of bytes	Type [Value]	Description
1	U8	*number-of-subrectangles*

If not set, there are no subrectangles (i.e. the whole tile is just solid background colour).

SubrectsColoured - if set then each subrectangle is preceded by a pixel value giving the colour 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 colour, the foreground colour; 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 four bits of *x-and-y-position* specify the X position, the least-significant specify the Y position. The most-significant four bits of *width-and-height* specify the width minus one, the least-significant specify the height minus one.

6.6.5 ZRLE encoding

ZRLE stands for Zlib[1] Run-Length Encoding, and combines zlib compression, tiling, palettisation and run-length encoding. 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 of 64x64 pixels in left-to-right, top-to-bottom order, similar to hextile. 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.

ZRLE makes use of a new type CPIXEL (compressed pixel). This is the same as a PIXEL for the agreed pixel format, except where *true-colour-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. In this 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 seven bits indicate the size of the palette used - zero means no palette, one means that the tile is of a single colour, 2 to 127 indicate a palette of that size. The possible values of $subencoding$ are:

0 - Raw pixel data. $width \times height$ pixel values follow (where $width$ and $height$ are the width and height of the tile):

No. of bytes	Type [Value]	Description
$width \times height \times bytesPerCPixel$	CPIXEL array	$pixels$

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

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

2 to 16 - Packed palette types. Followed by the palette, consisting of $paletteSize(= subencoding)$ pixel values. Then the packed pixels follow, each pixel represented as a bit field yielding an index into the palette (0 meaning the first palette

[1] see http://www.gzip.org/zlib/

entry). 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, 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
$paletteSize \times bytesPerCPixel$	`CPIXEL` array	$palette$
m	`U8` array	$packedPixels$

where m is the number of bytes representing the packed pixels. For $paletteSize$ of 2 this is $floor((width + 7)/8) \times height$, for $paletteSize$ of 3 or 4 this is $floor((width+3)/4)\times height$, for $paletteSize$ of 5 to 16 this is $floor((width+1)/2) \times height$.

17 to 127 - unused (no advantage over palette RLE).

128 - Plain RLE. 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 a 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`	$pixelValue$
$floor((runLength - 1)/255)$	`U8` array 255	
1	`U8`	$(runLength - 1)\%255$

129 - unused

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

No. of bytes	Type [Value]	Description
$paletteSize \times bytesPerCPixel$	`CPIXEL` array	$palette$

Then as with plain RLE, consists of 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`		$paletteIndex + 128$
$floor((runLength - 1)/255)$	`U8` array	255	
1	`U8`		$(runLength - 1)\%255$

6.7 Pseudo-encodings

6.7.1 Cursor pseudo-encoding

A client which requests the *Cursor* pseudo-encoding is declaring that it is capable of drawing a mouse cursor locally. This can significantly improve perceived performance over slow links. The server sets the cursor shape by sending a pseudo-rectangle with the *Cursor* pseudo-encoding as part of an update. The pseudo-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 $width \times height$ pixel values followed by a bitmask. The bitmask consists of left-to-right, top-to-bottom scanlines, where each scanline is padded to a whole number of bytes $floor((width + 7)/8)$. Within each byte the most significant bit represents the leftmost pixel, with a 1-bit meaning the corresponding pixel in the cursor is valid.

No. of bytes	Type [Value]	Description
$width \times height \times bytesPerPixel$	`PIXEL` array	*cursor-pixels*
$floor((width + 7)/8) * height$	`U8` array	*bitmask*

6.7.2 DesktopSize pseudo-encoding

A client which requests the *DesktopSize* pseudo-encoding is declaring that it is capable of coping with a change in the framebuffer width and/or height. The server changes the desktop size by sending a pseudo-rectangle with the *DesktopSize* pseudo-encoding as the last rectangle in an update. The pseudo-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 pseudo-rectangle.