RFC 6143: The Remote Framebuffer (RFB) Protocol Analysis

Ariel Stolerman
CS544, Drexel University
ams573@cs.drexel.edu

April 17, 2013

Abstract

The Remote Framebuffer protocol, RFB, is a protocol that works at the framebuffer level and used for remote access to graphical user interfaces in order to view and control a remote window system. RFB is applicable to all windowing systems (X11, Windows, Mac), and used in Virtual Network Computing (VNC). This paper provides an analysis for the RFB protocol, including service, components, security and other characteristics.

Contents

1 Service Description **1**

2 PDU Analysis and Common Themes **1**
2.1 Addressing . . . 1
2.2 Flow Control . . . 2
2.3 PDU . . . 2
2.3.1 Handshake . . . 2
2.3.2 Initialization . . . 3
2.3.3 Client-to-Server Messages . . . 3
2.3.4 Server-to-Client Messages . . . 4
2.3.5 Encoding Types . . . 5
2.4 Error Control . . . 6
2.5 Quality of Service . . . 6

3 Security Issues **6**

4 DFA **6**

5 Extensibility **7**

6 Subjective Analysis **7**

A DFA for the RFB Protocol **8**

1 Service Description

The Remote Framebuffer (RFB) protocol is designed for remote access to windowing-system-independent graphic interfaces [1]. The protocol began to be heavily used when Virtual Network Computing (VNC) systems were developed [2]. VNC systems are platform independent graphical desktop sharing systems designed for remote control by control event transmits followed by screen updates in the other direction, and are the main use of the RFB protocol. In 2002 RealVNC Ltd. [1] was formed to maintain the VNC system and the RFB protocol. The current RFB version is 3.8. In addition to the RFC document, specifications for the latest RFB protocol can be found on the RealVNC website [3].

RFB allows control at the user end-point, known as RFB *viewer* or *client*, consisting of common controls such as screen, keyboard and a pointing device, of a remote system where changes to the framebuffer are applied (the window application), known as the RFB *server*. RFB is a *"thin client"* protocol that demands very little of the client side, allowing flexibility for client implementations and deployment across various hardware and software platforms. RFB client-server communication is illustrated in Fig. 1.

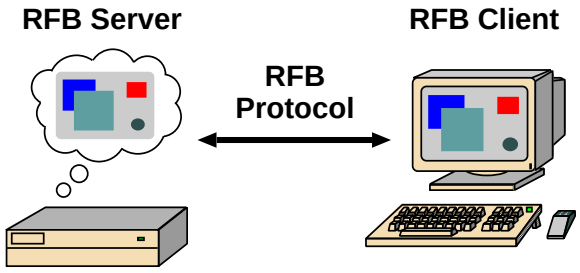

Figure 1: Illustration of the RFB client-server communication. [3]

RFB is stateless on the client side – the state of the user interface is preserved at all times on the server side, regardless of client connectivity. This means a client that disconnects and reconnects will see the exact same graphic state, and even multiple client endpoints can access the same server and see the same state. This location independent uniform view property is one of the main advantages of the RFB protocol.

As mentioned above, the RFB protocol's main usage is by VNC applications, which became a common graphic remote control utility in the industry and can be found as standard in many Unix/Linux distributions. According to the RealVNC website, "VNC has a widespread user base, from individuals to the world's largest multi-national companies, utilizing the technology for a range of applications." [1]

The following sections discuss the properties of the RFB protocol in terms of addressing, flow control, PDU structure and encoding, security issues and others. The details are accompanied with an analysis of the advantages and disadvantages of the protocol scheme.

2 PDU Analysis and Common Themes

2.1 Addressing

RFB communication is usually initiated in the client side, with the server listening on port 5900 (IANA allocated). Systems with multiple RFB servers listen at ports 5900+n (i.e. 5900, 5901, 5902 and so on). RFB servers can also provide HTTP servers on port 5800 for Java browser-based clients. A connection to a RFB server will then be to the server's IP address, on the respective port discussed above.

[1] http://www.realvnc.com/

It may also be the case where the initial connection is done in reverse, where the client listens on port 5500, and the server connects to it, after which the roles are reverted to the standard client-server communication. This option provides the advantage of having the burden of security (firewall/NAT) configuration in the viewer endpoint rather than the server.

The server enumeration supported addressing scheme and flexibility in initiation source are great advantages of RFB, making it convenient and practical for remote control based systems.

2.2 Flow Control

Display Protocol. The building blocks of the RFB display information is pixel data rectangles. Pixel data can be encoded in various ways that allow different tradeoffs between network bandwidth, client drawing and server processing speed. A *framebuffer update*, or simply an *update* is a sequence of such rectangles. A lot of the RFB's efficiency lays in encoding of the rectangles, discussed in Sec. 2.3.5.

RFB is *demand-driven* and asynchronous (after initial handshake), i.e. only explicit requests from the client cause update transmits from the server. This makes RFB adaptive to network speed: lower bandwidth results with less updates. Since consecutive changes tend to be close in the framebuffer (e.g. continuous mouse movements), traffic and client drawing can be reduced by simply ignoring transient states (similar to less frames-per-second in a video).

Input Protocol. Based on a keyboard/pointing device (e.g. mouse) model, key/move events are sent from the client to the server. Inputs can be sent from other I/O devices as well. RFB's event encodings and protocol allow cross-platform mappings (e.g. overcome different keyboard layouts).

Pixel Data Representation. The initial negotiation includes the format and encoding of the pixel data, in favor of the clients preferences (to make drawing on that side easier). Secondary selection is given to the server (choose the encoding easiest to produce).

A pixel is represented by its colors. 16 and 24 bit formats contain RGB (red, green, blue) intensities, and 8 bit format is keys that map to a fixed $2^8 = 256$ RGB table. Rectangle data header includes $\langle x, y \rangle$ position, width, height and pixel encoding type (specified in Sec. 2.3). The various pixel representation formats make RFB adaptive to network speed, in the tradeoff between frequency of updates and quality of transmitted image.

2.3 PDU

RFB can operate over any reliable transport, which is usually TCP/IP. It includes 3 stages:

1. *Handshake.* Agree on protocol version and security type.
2. *Initialization.* `ClientInit` and `ServerInit` messages are exchanged.
3. *Normal Protocol Interaction.* Client sends messages at will, and may receive responses from the server. Messages begin with a type (1 byte) followed by message-specific data.

RFB messages use basic types: `U8, U16, U32, S8, S16, S32`, which stand for unsigned / signed plus number-of-bits messages. Some use arrays of those types, which precede by the array size. Some messages include padding. In terms of delimiters, all messages are either fixed in size or contain a length parameter (e.g. array size). In the rest of the section, PDU chunks are formatted as:

`[`$\langle title | `'value' \rangle$`:` $\langle \#bytes \rangle$ `(`$\langle format \rangle$`)]`

Next, each phase is discussed in detail, including analysis of the approach taken for the current RFB version.

2.3.1 Handshake

This is a synchronous phase for determining version, security type, password (if required) and other initialization information.

In the protocol version phase, the server sends the highest version it controls. The client responds with the decided version, which should not exceed the version supported by the server. Each of these messages consists of 12 bytes of ASCII characters in the format "`RFB xxx.yyy\n`" where `xxx` and `yyy` are the zero-padded major and minor versions, respectively:
`['RFB 003.008\n': 12 (U8 array)]`
Next, the server lists its supported security types in the format:
`[#types: 1 (U8)][ security types: #types (U8 array)]`
If the connection failed (e.g. server does not support requested version), `#types` will be 0 followed by a failure reason string, and the server will close the connection:
`[reason #chars: 4 (U32)][reason: #chars (U8 array)]`
Otherwise the client responds with 1 (U8) byte indicating the selected security type (if any).
Security types and their analysis are discussed in Sec. 3. Security type decision follows with data specific to the selected security, and normally ends with the server sending `SecurityResult` message that indicates whether the security handshake was successful:
`[status: 4 (U32)]`
With the value `0:OK` or `1:failed`. If failed, the server sends a reason message as described above and closes the connection.

The security types discussed in [1] are `0:Invalid` (connection problem), `1:None` (no authentication) and `2:VNC Authentication`. For VNC authentication, the server sends a random 16 bytes challenge. The client encrypts it with DES with a user-defined password, which is adjusted to 8 characters (padded with nulls or truncated), and sends it in a 16 bytes message. This process provides a rather limited security, and discussed in Sec. 3.

2.3.2 Initialization

After security initialization (if such is agreed on), the client sends a `ClientInit` message, a 1 byte (U8) shared-flag that is non-zero (`true`) to indicate the server is sharable, and zero (`false`) if the server should be exclusive and disconnect any connected clients. This is a counterintuitive behavior, giving priority to new clients that can disconnect any existing clients. It would have been more logical to have the protocol manage concurrent clients in a way that prioritize existing ones, in a first-come-first-served manner.

`ClientInit` follows the server sending a `ServerInit`, consisting of the framebuffer's dimensions, pixel format, and desktop name:
`[width: 2 (U16)][height: 2 (U16)][pixel format: 16 (PIXEL_FORMAT)]`
`[name #chars: 4 (U32)][name: name #chars (U8 array)]`
Where the pixel format is the server's default, and will be used unless requested otherwise by the client. `PIXEL_FORMAT` contains a series of 10 U8/U16 fields that define bits-per-pixel (8, 16 or 32), depth, big endian flag, true-color flag (indicates whether the following RGB information is true encoding or keys to color map), and maximums and shifts for each of red, green and blue. A true color is then interpreted by shifting and `AND`'ing with its maximum.
Intermediate Summary. The handshake and initialization process, the synchronous part of RFB, is quite simple and straight forward. However, it suffers from flaws in its built-in security scheme (discussed further in Sec. 3) and a rather poor multi-client management (or more accurately lack thereof).

2.3.3 Client-to-Server Messages

There are 6 message types supported and publicly documented; however extensions can be used after the client confirms the server supports it (via server confirmation). Messages begin with a 1-byte message type.

SetPixelFormat sets the pixel format. If not sent, the server formats pixels as set in `ServerInit` it sent.
`['0x00': 1 (U8)][pad: 3][pixel format: 16 (PIXEL_FORMAT)]`

SetEncodings lists client pixel encoding types in order of preference (server may not comply to preference). Pixel data can always be sent in raw encoding without explicit specification. The client can also request pseudo-encodings, but must get server support confirmation. See Sec. 2.3.5 for encodings and discussion about their properties.
`['0x2': 1 (U8)][pad: 1][#encodings: 2 (U16)]`
`[encoding types: #encodings×4 (S32 array)]`

FramebufferUpdateRequest notifies the server interest in rectangle update of some area in the framebuffer. Answered with `FramebufferUpdate`; one update may cover several requests, so over a slower network update rate can be moderated to save traffic. The updates may be only incremental, unless the client states otherwise. This is a great performance advantage, as the client can regulate incremental requests rate to moderate traffic.
`['0x3': 1 (U8)][incremental: 1 (U8)][x: 2 (U16)][y: 2 (U16)]`
`[width: 2 (U16)][height: 2 (U16)]`

KeyEvent indicates key press/release. Press/release are determined by the down-flag, and the key is specified using the `keysym` values defined by the X Windows system.
`['0x4': 1 (U8][down-flag: 1 (U8)][pad: 2][key: 4 (U32)]`
Guidelines are detailed in [1] to make RFB as interoperable as possible under the complex interpretation of keysyms. For instance, servers are required to simulate shift presses internally to cope with different keyboard layouts (US keyboard uses shift+3 for #, whereas UK keyboard has a # key). This is a great advantage in key events handling, which supports cross-platform and cross-location systems.

PointerEvent indicates pointer movement or button press/release, via position and 8-bit button mask (for 8 buttons: left, middle, right, wheel upward scroll etc.)
`['0x5': 1 (U8)][button-mask: 1 (U8)][x: 2 (U16)][y: 2 (U16)]`

ClientCutText for synchronizing the "cut-buffer" of selected text, this message notifies that the client has new text in its cut-buffer. The text is limited to the Latin-1 charset (ISO 8859-1). This limitation is of course a great disadvantage when using text encoding different than Latin-1.
`['0x6': 1 (U8)][pad: 3][len: 4 (U32)][text: len (U8 array)]`

The client message formats are simple and straight forward, however present some limitations. The most prevalent is probably the Latin-1 encoding limitation for cut-buffers. Different allocation of text buffer messages could have easily handled other encodings, like UTF-8 that requires up to 4 bytes per character rather than 1. [1] mentions other I/O inputs can be synthesized via keyboard events. This provides great flexibility to controls on the client side, which are not dependent only on standard keyboard and pointer device. On the other hand, a standard user-defined approach to define custom I/O events could have been useful, for instance in systems with dedicated hardware that cannot be synthesized via keyboard events alone (this can probably be achieved via pseudo-encoding).

2.3.4 Server-to-Client Messages

There are 4 message types supported and publicly documented. Others can be sent after confirming the clients supports them, usually through a request for some pseudo-encoding.

FramebufferUpdate a sequence of rectangles sent in response to the client's `FramebufferUpdateRequest`. See Sec. 2.3.5 for encoding types. As mentioned, one update can provide response to several requests, which allows adjustment of network traffic.
Header: `['0x0': 1 (U8)][pad: 1][#rectangles: 2 (U16)]`
#rectangles rectangle data: `[x: 2 (U16)][y: 2 (U16)]`
`[width: 2 (U16)][height: 2 (U16)][encoding-type: 4 (S32)]`

SetColorMapEntries Sent upon the first client request, if the pixel format uses a color map. Each entry encodes 16 bits for each of red, green and blue.
Header: `['0x1': 1 (U8)][pad: 1][first-color: 2 (U16)][#colors: 2 (U16)]`
#colors color entries: `[red: 2 (U16)][green: 2 (U16)][blue: 2 (U16)]`

Bell makes an audible signal at the client side.
`['0x2': 1 (U8)]`

ServerCutText Sent when the server has new Latin-1 text in the cut buffer.
`['0x3': 1 (U8)][pad: 3][len: 4 (U32)][text: len (U8 array)]`

Having a custom color map definition message may be useful to focus the palette on the most used 256 colors. On the other hand, having a default color map seems intuitive and saves the need to declare colors at need (which occurs more than once in a single session; tested it in Wireshark). The cut-buffer Latin-1 limitation issue mentioned for the client messages applies here as well.

2.3.5 Encoding Types

[1] includes the eight publicly documented S32 keys and descriptions for the `Raw`, `CopyRect`, `RRE`, `Hextile`, `TRLE` and `ZRLE` encodings and `Cursor` and `DesktopSize` pseudo-encodings. Following is a summary of the different types; detailed PDUs can be found in [1]:

Raw the default and simplest pixel data encoding, consists of width×height×bytes-per-pixel PIXEL array, scanned left-to-right, top-to-bottom.

CopyRect an efficient encoding that gives a $\langle x, y \rangle$ position in the client's framebuffer to copy from. Useful if a window is moved or content is scrolled.

RRE rise-and-run-length encodes a rectangle by a background pixel (most common one) and a set of rectangles defined by position, size and pixel data (different than the background pixel).

Hextile a variation of `RRE` in which rectangles are divided into 16×16 tiles s.t. position and size are encoded in 16 bits in total, and the order of the tiles is indicative of their position in the enclosing rectangle (left-to-right, top-to-bottom). Background and foreground colors may be unset, taking the setup of the previous tile (unless encoded with raw pixel data), saving bits.

TRLE the tiled run-length encoding is more compact than the previous. Similar to `Hextile` it uses tiling, but uses a pixel format `CPIXEL` which is a compressed pixel format that under certain circumstances reduces pixel encoding to 3 bytes.

ZRLE the Zlib [4] run-length encoding is similar to TRLE but uses zlib compression. The rectangle is encoded with a zlib data length, followed by the zlib data. One zlib stream is used per connection, so rectangles are encoded and decoded in order. The inner tiles are 64×64 in size. The transmitted zlib data can be incrementally decoded by the client but encoded tile data is not byte-aligned.

Cursor (pseudo-encoding) client declaring `Cursor` can locally draw pointer cursor, which improves performance on a slow link. The server sends the cursor shape in a rectangle encoded with the hotspot $\langle x, y \rangle$ position, raw pixel data for the rectangle surface and a bitmask to indicate valid cursor pixels.

DesktopSize (pseudo-encoding) client declaring `DesktopSize` can cope with framebuffer size changes. The server will send a `DesktopSize` rectangle last on an update with empty $\langle x, y \rangle$ and new width and height to be set. It is then assumed the old framebuffer should be completely updated, but the client should preserve the overlap between the old and the new size (top-left rectangle portion) for interoperability.

Rectangle encoding, along with pixel encoding, hold key to the majority of efficiency that can be attained for RFB. The publicly defined rectangle encodings above, like `ZRLE` which relays on a known compression protocol, provide good baseline for efficient communication of remote framebuffer data. The pseudo-encoding scheme provide yet more flexibility, by allowing custom defined encodings.

2.4 Error Control

There does not seem to be any corrective error handling in RFB. In the connection initialization phase, two types of errors are mentioned in the documentation: failed agreement on protocol variables (e.g. server does not support client version) or VNC authentication failure. Both are answered with the server closing the connection providing a textual reason. Reattempting connection/authentication, however, is not suggested.

The asynchronous phase of framebuffer requests and updates, which consist most of the communication, is characterized by the server simply ignoring commands and requests it cannot fulfill. In the context of framebuffer transmits this is actually a reasonable handling. For instance, consider the case of a slow network; since changes to the framebuffer tend to be close in time (and usually in space), skipping requests (or accumulating several together into one update) will not harm the service significantly (only an effect of "less frames-per-second"). The server will also ignore encodings and pseudo-encodings it does not support, which will not harm the flow of the protocol. This sort of error handling is simple and seems sufficient for the type of service provided by RFB.

2.5 Quality of Service

As stated in the previous section, the quality of service is adaptive to the network speed, and can be moderated according to resource availability. Level of service is controlled by several parameters, the main being pixel data and rate of updates (this excludes underlying layers overhead, e.g. working over a secure layer). Lowering the rate of updates and using 8-bit color map can reduce network traffic heavily, but still provide a fair service for remote control. A quick review of the RealVNC advanced options reveals some of the parameters that can be tuned, including color scheme (can be degraded to only 8 colors), preferred encoding and encryption level – all affecting the quality of service and the tradeoff between network resources and authenticity of the mapped remote windowing system.

In terms of integrity and trust (which are also considered for QoS), basic implementations of RFB work over TCP, which should provide low-level integrity. However the security standards of RFB (which are mostly none existing) allow tampering with exposed connection, break trust and integrity of data, and by that potentially reduce QoS.

3 Security Issues

The basic supported VNC authentication mentioned in Sec. 2.3.1 is weak and intended to be used in trusted networks. However, there are security extensions (that are not detailed in [1]) which are not publicly documented. The extensibility of RFB allows implementing and applying stronger security like using encrypted channels over IPsec or SSH.

In the VNC authentication option, which provides a DES challenge-response authentication, the channel is not secure. Other supported security types include several RealVNC schemes, TLS, VeNCrypt (communication over TLS with authentication based on X509 certificates), MD5 hash authentication and others. Those options include secure communication that provide authentication, encryption, integrity and trust, however these are not publicly documented and information on their adaptation to RFB is actually quite hard to find. It is clear that future developments of RFB should include a wider range of security choices documented and practiced.

4 DFA

An illustration of a DFA for RFB is given in App. A. Most of the states consist of handshake and initialization, the synchronous phase of the communication. After the main communication is in progress, the

asynchronous phase, there are only two states: the client sent a request and awaits an update (bottom left state), or simply idle (bottom right state). Note that when the client awaits an update, it can continue sending requests; upon the first server response, all requests are answered with a single update (or some are ignored).

The most salient type of states that can be added to the DFA is new security types (e.g. for TLS or VeNCrypt). Other states can be added as well, however more noticeable is that new transitions (edges) can be added. For instance, turn the initialization phase into back-and-forth negotiation, or adding more actions that can be taken by the client or the server after the normal protocol interaction starts.

5 Extensibility

RFB includes 3 official published versions: 3.3, 3.7 and the latest – 3.8. The protocol is extendible, with no need to modify the protocol version. Existing versions can be extended as follows:

New Encodings. Encoding types can easily be added. Servers that won't support it will ignore request for the new encoding, and clients will just not transmit requests with it.

Pseudo-Encodings. With pseudo-encodings the client can declare support in certain protocol extensions, and request to use them. If the server supports it, it will respond with an extension confirmation, otherwise it ignores the request.

New Security Types. Effectively this property means that clients and servers that agree on some new security type can continue following communication in whatever protocol they agree on (which can differ completely from RFB). Examples for different security types are given in Sec. 3.

6 Subjective Analysis

In summary I think this protocol is quite good, applies simple concepts and provides a good solution for remote framebuffer view and control, adjustable to the network characteristics. I have a lot of experience with VNC clients, working on remote servers that cross states and continents – in all cases RFB provides a very reliable, usually fast, convenient desktop control. This is one of the reasons I chose RFB for analysis, to better understand how the challenge of mapping a desktop in real-time is approached.

After taking a close look at the protocol, I can justify my statement on the simplicity and efficiency applied to the protocol in its asynchronous updates, efficient pixel data and rectangle encodings and other tweaks (like incremental update rather than full update). With that, the weaknesses of the protocol should be taken into consideration, including the cut-buffer text encoding limitation, poor multiclient management, and especially the weak basic security scheme, which can be dealt with implementations that use secure connections (or just use VPNs). However, RFB has high QoS and in general is a good protocol.

References

[1] T. Richardson and J. Levine, "The Remote Framebuffer Protocol," RFC 6143 (Informational), Internet Engineering Task Force, Mar. 2011. [Online]. Available: http://www.ietf.org/rfc/rfc6143.txt

[2] Wikipedia, "RFB protocol," 2013. [Online]. Available: http://http://en.wikipedia.org/wiki/RFB_protocol

[3] T. Richardson, "The RFB Protocol Version 3.8," RealVNC Ltd., Nov. 2010. [Online]. Available: http://www.realvnc.com/docs/rfbproto.pdf

[4] P. Deutsch and J.-L. Gailly, "ZLIB Compressed Data Format Specification version 3.3," RFC 1950 (Informational), Internet Engineering Task Force, May 1996. [Online]. Available: http://www.ietf.org/rfc/rfc1950.txt

A DFA for the RFB Protocol