SPICE
Project

Spice remote computing protocol definition v1.0

Draft 1

Table of Contents

1. Introduction

Spice protocol defines a set of protocol messages for accessing, controlling, and receiving inputs from remote computing devices (e.g., keyboard, video, mouse) across networks, and sending output to them. A controlled device can reside on either side, client and/or server. In addition, the protocol defines a set of calls for supporting migration of a remote server from one network address to another. Encryption of transported data, with one exception, was kept out of the protocol for maximum flexibility in choosing an encryption method. Spice uses simple messaging and does not depend on any RPC standard or a specific transport layer.

Spice communication session is split into multiple communication channels (e.g., every channel is a remote device) in order to have the ability to control communication and execution of messages according to the channel type (e.g. QoS encryption), and to add and remove communication channels during run time (which is supported by spice protocol definition). The following communication channels are defined in the current protocol definition: a). the main channel serves as the main spice session connection b). display channel for receiving remote display updates c). inputs channel for sending mouse and keyboard events d). cursor channel for receiving pointer shape and position e). Playback channel for receiving audio stream, and f). Record channel for sending audio capture. More channel types will be added as the protocol evolves. Spice also defines a set of protocol definitions for synchronizing channels` execution on the remote site.

2. Common Protocol definition

2.1. Endianness

Unless stated otherwise, all data structures are packed and byte and bit order is in little endian format.

2.2. Data types

a) UINT8 – 8 bits unsigned integer

b) INT16 – 16 bits signed integer

c) UINT16 – 16 bits unsigned integer

d) UINT32 – 32 bits unsigned integer

e) INT32 - 32 bits signed integer

f) UINT64 – 64 bits unsigned integer

g) ADDRESS - 64 bits unsigned integer, value is the offset of the addressed data from the beginning of spice protocol message body (i.e., data following RedDataHeader or RedSubMessage).

h) FIXED28_4 – 32 bits fixed point number. 28 high bits are signed integer. Low 4 bits is unsigned integer numerator of a fraction with denominator 16.

i) POINT

   INT32 x
   INT32 y

j) POINT16

   INT16 x
   INT16 y

k) RECT

   INT32 top
   INT32 left
   INT32 bottom
   INT32 right

l) POINTFIX

   FIXED28_4 x
   FIXED28_4 y

2.3. Protocol Magic number UINT8[4]

RED_MAGIC = { 0x52, 0x45, 0x44, 0x51}

2.4. Protocol version

Protocol version defined as two UINT32 values, major protocol version and minor protocol version. Server and client having the same major version must keep compatibility regardless of minor version (i.e., incrementing the major version brakes compatibility). Major protocol version with "huge" value are reserved for development purposes and are considered unsupported and unreliable. "huge" values are defined as having bit 31 set. The minor protocol version is incremented on every protocol change that does not break compatibility. It is set to zero on major protocol version increment.

Current Protocol version

- RED_VERSION_MAJOR   = 1
- RED_VERSION_MINOR   = 0

2.5. Compatibility – UINT32[]

In order to allow some degree of flexibility in client and server implementation and in order to improve compatibility, spice protocol supports bidirectional exchange of channels compatibilities. Compatibilities are expressed in UINT32 vector that is split into two groups: common compatibilities and channels compatibilities. Common compatibilities stands for compatibilities shared by all channels, and channels compatibilities stands for channel specific compatibilities. Splitting the vector into two types allows us to add channel compatibilities independently. Each compatibility is expressed using one or more bits in the compatibilities vector.

2.6. Channel types – UINT8

▪ RED_CHANNEL_MAIN	= 1
▪ RED_CHANNEL_DISPLAY	= 2
▪ RED_CHANNEL_INPUTS	= 3
▪ RED_CHANNEL_CURSOR	= 4
▪ RED_CHANNEL_PLAYBACK	= 5
▪ RED_CHANNEL_RECORD	= 6

2.7. Error codes UINT32

▪ RED_ERROR_OK	= 0
▪ RED_ERROR_ERROR	= 1
▪ RED_ERROR_INVALID_MAGIC	= 2
▪ RED_ERROR_INVALID_DATA	= 3
▪ RED_ERROR_VERSION_MISMATCH	= 4
▪ RED_ERROR_NEED_SECURED	= 5
▪ RED_ERROR_NEED_UNSECURED	= 6
▪ RED_ERROR_PERMISSION_DENIED	= 7
▪ RED_ERROR_BAD_CONNECTION_ID	= 8
▪ RED_ERROR_CHANNEL_NOT_AVAILABLE	= 9

2.8. Warning codes

▪ RED_WARN_GENERAL	= 0

2.9. Information codes

▪ RED_INFO_GENERAL	= 0

2.10. public key buffer size.

- RED_TICKET_PUBKEY_BYTES = 162, size needed for holding 1024 bit RSA public key in X.509 SubjectPublicKeyInfo format.

2.11. Channel link: establishing a channel connection.

a) Connection process

The channel connection process is initiated by the client. The client sends RedLinkMess. In response, the server sends RedLinkReply. When the client receives RedLinkReply, it examines the error code and in case there is no error it encrypts its password with public key received in RedLinkReply and sends it to the server. The server receive the password and sends the link result to the client. The client examines the link result, and in case the result equals to RED_ERROR_OK, a valid connection is established.

Channel connection for channel types other then RED_CHANNEL_MAIN is allowed only after the client has active RED_CHANNEL_MAIN channel connection. Only one RED_CHANNEL_MAIN connection is allowed, and this channel connection establishes spice session with the remote server.

b) Ticketing

Ticketing is a mechanism implemented in spice to ensure connections are opened only from authorized sources. To enable this mechanism a ticket is set in spice server consisting of a password and time validity. After time validity passes, the whole ticket is expired. The ticket is encrypted. To encrypt, server generates a 1024 bit RSA key and send the public part to the client (via RedLinkInfo). Client uses this key to encrypt the password and send it back to server (after RedLinkMess). Server decrypt the password, compare it to ticket and ensure it was received within the allowed time-frame.

c) RedLinkMess definition.

UINT32 magic -	value of this fields must be equal to RED_MAGIC
UINT32 major_version -	value of this fields must be equal to RED_VERSION_MAJOR.
UINT32 minor_version -	value of this fields must be equal to RED_VERSION_MINOR.
UINT32 size -	number of bytes following this field to the end of this message.
UINT32 connection_id -	In case of a new session (i.e., channel type is

RED_CHANNEL_MAIN) this field is set to zero, and in response the server will allocate session id and will send it via the RedLinkReply message. In case of all other channel types, this field will be equal to the allocated session id.

UINT8 channel_type - one of RED_CHANNEL_?

UINT8 channel_id - channel id to connect to. This enables having multiple channels of the same type.

UINT32 num_common_caps - number of common client channel capabilities words

UINT32 num_channel_caps - number of specific client channel capabilities words

UINT32 caps_offset - location of the start of the capabilities vector given by the bytes offset from the " size" member (i.e., from the address of the "connection_id" member).

d) RedLinkReply definition

UINT32 magic - value of this field must be equal to RED_MAGIC

UINT32 major_version - server major protocol version.

UINT32 minor_version - server minor protocol version.

UINT32 size - number of bytes following this field to the end of this message.

UINT32 error - Error codes (i.e., RED_ERROR_?)

UINT8[RED_TICKET_PUBKEY_BYTES] pub_key – 1024 bit RSA public key in X.509 SubjectPublicKeyInfo format.

UINT32 num_common_caps - number of common server channel capabilities words

UINT32 num_channel_caps - number of specific server channel capabilities words

UINT32 caps_offset - location of the start of the capabilities vector given by the bytes offset from the " size" member (i.e., from the address of the "connection_id" member) .

e) Encrypted Password

Client sends RSA encrypted password, with public key received from server (in RedLinkReply). Format is EME-OAEP as described in PKCS#1 v2.0 with SHA-1, MGF1 and an empty encoding parameter.

f) Link Result UINT32

The server sends link result error code (i.e., RED_ERROR_?)

2.12. Protocol message definition

All messages transmitted after the link stage have a common message layout. It begins with RedDataHeader which describes one main message and an optional sub messages list.

a) RedDataHeader

UINT64 serial – serial number of the message within the channel. Serial numbers start with a value of 1 and are incremented on every message transmitted.

UINT16 type – message type can be one that is accepted by all channel (e.g., RED_MIGRATE), or specific to a channel type (e.g., RED_DISPLAY_MODE for display channel).

UINT32 size – size of the message body in bytes. In case sub_list (see below) is not zero then the actual main message size is sub_list. The message body follows RedDataHeader.

UINT32 sub_list – optional sub-messages list. If this field is not zero then sub_list is the offset in bytes to RedSubMessageList from the end of RedDataHeader. All sub-messages need to be executed before the main message, and in the order they appear in the sub-messageslist.

b) RedSubMessageList

UINT16 size – number of sub-messages in this list.

UINT32[] sub_messages – array of offsets to sub message, offset is number of bytes from the end of RedDataHeader to start of RedSubMessage.

c) RedSubMessage

UINT16 type - message type can be one that is accepted by all channel (e.g., RED_MIGRATE), or specific to a channel type (e.g., RED_DISPLAY_MODE for display channel).

UINT32 size - size of the message body in bytes. The message body follows RedSubMessage.

2.13. Common messages and messaging naming convention

Messages types and message body structures are prefixed according to the source of the message. The prefixes for messages sent from the server to the client are RED for types and Red for structures. For messages sent from the client the prefixes are REDC and Redc.

2.14. Server messages that are common to all channels

RED_MIGRATE	= 1
RED_MIGRATE_DATA	= 2
RED_SET_ACK	= 3
RED_PING	= 4
RED_WAIT_FOR_CHANNELS	= 5
RED_DISCONNECTING	= 6
RED_NOTIFY	= 7
RED_FIRST_AVAIL_MESSAGE	= 101

Specific channel server messages start from RED_FIRST_AVAIL_MESSAGE. All message types from RED_NOTIFY + 1 to RED_FIRST_AVAIL_MESSAGE – 1 are reserved for further use.

2.15. Client messages that are common to all channels

REDC_ACK_SYNC	= 1
REDC_ACK	= 2
REDC_PONG	= 3
REDC_MIGRATE_FLUSH_MARK	= 4
REDC_MIGRATE_DATA	= 5
REDC_DISCONNECTING	= 6
REDC_FIRST_AVAIL_MESSAGE	= 101

Specific channel client messages start from REDC_FIRST_AVAIL_MESSAGE. All message types from REDC_ACK_SYNC+ 1 to REDC_FIRST_AVAIL_MESSAGE – 1 are reserved for further use.

2.16. Messages acknowledgment.

Spice provides a set of messages for requesting an acknowledgment on every one or more messages that the client consumes. In order to request acknowledgment messages, the server sends RED_SET_ACK with the requested acknowledgment frequency – after how many received messages the client sends acknowledgment. . In response, the client sends REDC_ACK_SYNC. From this point, for every requested number of messages that the client receive, it will send a REDC_ACK message.

a) RED_SET_ACK, RedSetAck

UINT32 generation – the generation of the acknowledgment sequence. This value will be sent back by REDC_ACK_SYNC. It is used for acknowledgment accounting synchronization.

UINT32 window – the window size. Spice client will send acknowledgment for every "window" messages. Zero window size will disable messages acknowledgment.

b) REDC_ACK_SYNC, UINT32

UINT32 – Spice client sends RedSetAck.generation in response to RED_SET_ACK

c) REDC_ACK, VOID

Spice client sends REDC_ACK message for every RedSetAck.window messages it consumes.

2.17. Ping

Spice protocol provides ping messages for debugging purpose. Spice server sends RED_PING and the client responses with REDC_PONG. The server can measure round trip time by subtracting current time with the time that is returned in REDC_PONG message.

a) RED_PING, RedPing

UINT32 id – the id of this message

UINT64 time – time stamp of this message

b) REDC_PONG, RedPong

UINT32 id – Spice client copies it from RedPing.id

UINT64 time – Spice client copies it from RedPing.time

2.18. Channel migration

Spice supports migration of Spice server. The following common messages combined with specific main channel messages is used for migrating channels connections between spice servers. We will refer these servers as source and destination. Main channel is used for initiating and controlling the migration process. The following describes the actual channel migration process.

Channel migration process starts with sending RED_MIGRATE message from the server. The client receives the message, examine the attached flags and:

- if the server requests messages flush (i.e., RED_MIGRATE_NEED_FLUSH flag is on), the client sends REDC_MIGRATE_FLUSH_MARK message to the server. This procedure can be used to ensure safe delivery of all mid air messages before performing the migration action.

- if the server requests data transfer (i.e., RED_MIGRATE_NEED_DATA_TRANSFER flag is on), the client expects to receive one last message from the server before migrating to destination. This message type must be RED_MIGRATE_DATA type. The content of the received message will be transmitted to the destination on connection swap.

Afterward, the client swaps communication channels (i.e., starts using the connection with the destination server). The client can close connection with the source server only after all other channels also have finished the migration process. If the server side has requested data transfer, the client first transmits REDC_MIGRATE_DATA message containing the data received on RED_MIGRATE_DATA.

a) Migration flags

RED_MIGRATE_NEED_FLUSH	= 1
RED_MIGRATE_NEED_DATA_TRANSFER	= 2

b) RED_MIGRATE, RedMigrate

UINT32 flags – combination of red migration flags.

c) RED_MIGRATE_DATA, UINT8[]

Server migrate data, body of this message is variable length raw data that is determined by each channel type independently

d) REDC_MIGRATE_FLUSH_MARK, VOID

This messages mark completion of client communication channel flushing.

e) REDC_MIGRATE_DATA, UINT8[]

Post migration data, sent by client to the destination, containing the data sent by the source using the RED_MIGRATE_DATA message.

2.19. Channel synchronization

Spice provides mechanism for synchronizing channels message execution on the client side. The server sends RED_WAIT_FOR_CHANNELS message which contains a list of channels messages to wait for (i.e., RedWaitForChannels). The Spice client will wait for completion of all the messages that are in that list before executing any more messages.

a) RedWaitForChannel

UINT8 type – channel type (e.g., RED_CHANNEL_INPUTS)

UINT8 id – channel id.

UIN64 serial – message serial id (i.e, RedDataHeader.serial) to wait for

b) RED_WAIT_FOR_CHANNELS, RedWaitForChannels

UINT8 wait_count – number of items in wait_list

RedWaitForChannel[] wait_list – list of channels to wait for.

2.20. Disconnect reason

The following messages are used for notification about orderly disconnection of the server or client.

a) RED_DISCONNECTING, RedDisconnect

UINT64 time_stamp – time stamp of disconnect action on the server.

UINT32 reason – disconnect reason, RED_ERROR_?

b) REDC_DISCONNECTING, RedcDisconnect

UINT64 time_stamp – time stamp of disconnect action on the client.

UINT32 reason – disconnect reason, RED_ERROR_?

2.21. Server notification

Spice protocol defines message for delivering notifications to the client using RED_NOTIFY message. Messages are categorized by severity and visibility. The later can be used as hint for the way the message is displayed to the user. For example high visibility notifications will trigger message box and low visibility notifications will be directed to the log.

a) RED_NOTIFY, RedNotify

UINT64 time_stamp – server side time stamp of this message.

UINT32 severity – one of RED_NOTIFY_SEVERITY_?

UINT32 visibility - one of RED_NOTIFY_VISIBILITY_?

UINT32 what – one of RED_ERROR_?, RED_WARN_? Or RED_INFO_?, depending on severity.

UINT32 message_len – size of message

UINT8[] message – message string in UTF8.

UINT8 0 – string zero termination

3. Main Channel definition

3.1. Server messages

RED_MAIN_MIGRATE_BEGIN	= 101
RED_MAIN_MIGRATE_CANCEL	= 102
RED_MAIN_INIT	= 103
RED_MAIN_CHANNELS_LIST	= 104
RED_MAIN_MOUSE_MODE	= 105
RED_MAIN_MULTI_MEDIA_TIME	= 106
RED_MAIN_AGENT_CONNECTED	= 107
RED_MAIN_AGENT_DISCONNECTED	= 108
RED_MAIN_AGENT_DATA	= 109
RED_MAIN_AGENT_TOKEN	= 110

3.2. Client messages

REDC_MAIN_RESERVED	= 101
REDC_MAIN_MIGRATE_READY	= 102
REDC_MAIN_MIGRATE_ERROR	= 103
REDC_MAIN_ATTACH_CHANNELS	= 104
REDC_MAIN_MOUSE_MODE_REQUEST	= 105
REDC_MAIN_AGENT_START	= 106
REDC_MAIN_AGENT_DATA	= 107
REDC_MAIN_AGENT_TOKEN	= 108

3.3. Migration control

Spice migration control is performed using the main channel messages. Spice server initiates migration process by sending RED_MAIN_MIGRATE_BEGIN message. Once the client has completed its pre-migrate procedure it notifies the server by transmitting REDC_MAIN_MIGRATE_READY message. In case of pre-migrate procedure error, the client sends REDC_MAIN_MIGRATE_ERROR. Once the server receives REDC_MAIN_MIGRATE_READY he can commence the migration process. The server can send RED_MAIN_MIGRATE_CANCEL in order to instruct the client to cancel the migration process.

a) RED_MAIN_MIGRATE_BEGIN, RedMigrationBegin

   UINT16 port – port of destination server

   UINT16 sport – secure port of destination server

   UINT8[] host_name – host name of destination server

a) RED_MAIN_MIGRATE_CANCEL, VOID

   Instruct the client to cancel migration process

a) REDC_MAIN_MIGRATE_READY, VOID

   Notify the server of successful completion of the pre-migrate stage

a) REDC_MAIN_MIGRATE_ERROR, VOID

   Notify the server of pre-migrate stage error

3.4. Mouse modes

Spice protocol specifies two mouse modes, client mode and server mode. In client mode, the affective mouse is the client side mouse: the client sends mouse position within the display and the server sends mouse shape messages. In server mode, the client sends relative mouse movements and the server sends position and shape commands. Spice main channel is used for mouse mode control.

a) Modes

   RED_MOUSE_MODE_SERVER = 1
   RED_MOUSE_MODE_CLIENT = 2

b) RED_MAIN_MOUSE_MODE, RedMouseMode

   Spice server sends this message on every mouse mode change

   UINT32 supported_modes – current supported mouse mode, this is any combination of RED_MOUSE_MODE_?

   UINT32 current_mode – the current mouse mode. Can be one of RED_MOUSE_MODE_?

c) REDC_MAIN_MOUSE_MODE_REQUEST, UINT32

Spice client sends this message to request specific mouse mode. It is not guarantied that the server will accept the request. Only on receiving RED_MOUSE_MODE message, the client can know of actual mouse mode change.

UINT32 – requested mode, one of RED_MOUSE_MODE_?

3.5. Main channel init message

Spice server must send RedInit as the first transmitted message t and is disallowed to send it at any other point.

a) RED_MAIN_INIT, RedInit

UINT32 session_id – session id is generated by the server. This id will be send on every new channel connection within this session (i.e., in RedLinkMess.connection_id).

UINT32 display_channels_hint – optional hint of expected number of display channels. Zero is defined as an invalid value

UINT32 supported_mouse_modes – supported mouse modes. This is any combination of RED_MOUSE_MODE_?

UINT32 current_mouse_mode – the current mouse mode, one of RED_MOUSE_MODE_?

UINT32 agent_connected – current state of Spice agent (see Section 3.8), 0 and 1 stand for disconnected and connected state respectively.

UINT32 agent_tokens – number of available tokens for sending messages to Spice agent.

UINT32 multi_media_time – current server multimedia time. The multimedia time is used for synchronizing video (for more information see “Multimedia time”).

UINT32 ram_hint - optional hint for help in determining global LZ compression dictionary size (for more information see section “Spice Image” in “Display Channel”).

3.6. Server side channels notification

In order to have the ability to dynamically attach to the server side channels, Spice protocol includes RED_MAIN_CHANNELS_LIST message. This massage informs the client of available channels in the server side. In response to this message the client can decide to link with the new available channel(s). The server must receive REDC_MAIN_ATTACH_CHANNELS before sending any RED_MAIN_CHANNELS_LIST message.

a) RED_MAIN_CHANNELS_LIST, RedChannels

UINT32 num_of_channels – number of channels in this list

RedChanneID[] channels – vector of "num_of_channels" channel ids

b) RedChanneID

UINT8 type – channel type, one of RED_CHANNEL_? channel types, except for RED_CHANNEL_MAIN

UINT8 id – channel id

3.7. Multimedia time

Spice defines messages for setting multimedia time for synchronization of video and audio streams. Two methods for updating multimedia time are supported. The first method uses the time stamp of data that arrives on the playback channel.The second method uses the main channel RED_MAIN_MULTI_MEDIA_TIME message. The latter method is used when no active playback channel exist.

a) RED_MAIN_MULTI_MEDIA_TIME, UINT32

UINT32 – multimedia time

3.8. Spice agent

Spice protocol defines a set of messages for bidirectional communication channel between Spice client and spice client agent on the remote server. Spice provides a communication channel only, the actual transferred data content is opaque to the protocol. This channel can be used for various purposes, for example, client-guest clipboard sharing, authentication and display configuration.

Spice client receives notifications of remote site agent connection as part of the RED_MAIN_INIT message or by a specific server RED_MAIN_AGENT_CONNECTED. Remote agent disconnection notification is delivered by RED_MAIN_AGENT_DISCONNECTED message. A bidirectional tokens mechanism is used in order to prevent blocking of the main channel with agent messages (e.g., in case the agent stops consuming the data). Each side is not allowed to send more messages than the tokens allocated to it by the other side. The number of tokens that are allocated for the client is initialized from RED_MAIN_INIT message, and farther allocation of tokens is done using RED_MAIN_AGENT_TOKEN. Server tokens initial count is delivered in REDC_MAIN_AGENT_START message. This message must be the first agent related message that the client sends to the server. Farther tokens allocation for the server is done using REDC_MAIN_AGENT_TOKEN. Actual data packets are delivered using RED_MAIN_AGENT_DATA and REDC_MAIN_AGENT_DATA.

a) Although agent messages are opaque for the protocol, agent data stream is defined by Spice protocol in order to delineate messages. Still, the client-server communication is independent from the agent channel, e.g., agent protocol conflicts don't affect the rest of the channels. Agent stream is defined as a run of messages having the following format:

    UINT32 protocol - unique protocol of this message. The protocol id must be registered in order to prevent conflicts.

    UINT32 type – protocol dependent message type.

    UINT64 opaque - protocol dependent opaque data.

    UINT32 size – size of data in bytes.

    UINT8 data[0] – data of this message.

   Client and server must continue processing unknown protocols messages or messages having unknown type (i.e., receive and dump).

b) RED_MAIN_AGENT_CONNECTED, VOID

c) RED_MAIN_AGENT_DISCONNECTED, UINT32

   UINT32 – disconnect error code RED_ERROR_?

d) RED_AGENT_MAX_DATA_SIZE = 2048

e) RED_MAIN_AGENT_DATA, UINT8[]

   Agent packet is the entire message body (i.e. RedDataHeader.size). The maximum packet size is RED_AGENT_MAX_DATA_SIZE.

f) RED_MAIN_AGENT_TOKEN, UINT32

   UINT32 – allocated tokens count for the client

g) REDC_MAIN_AGENT_START, UINT32

   UINT32 – allocated tokens count for the server

h) REDC_MAIN_AGENT_DATA, UINT8[]

   Agent packet is the entire message body (i.e. RedDataHeader.size). The maximum packet size is RED_AGENT_MAX_DATA_SIZE.

i) REDC_MAIN_AGENT_TOKEN, UINT32

   UINT32 – allocated tokens count for the server

4. Inputs channel definition

Spice Inputs channel controls the server mouse and the keyboard.

4.1. Client messages

REDC_INPUTS_KEY_DOWN	= 101
REDC_INPUTS_KEY_UP	= 102
REDC_INPUTS_KEY_MODIFAIERS	= 103
REDC_INPUTS_MOUSE_MOTION	= 111
REDC_INPUTS_MOUSE_POSITION	= 112
REDC_INPUTS_MOUSE_PRESS	= 113
REDC_INPUTS_MOUSE_RELEASE	= 114

4.2. Server Messages

RED_INPUTS_INIT	= 101
RED_INPUTS_KEY_MODIFAIERS	= 102
RED_INPUTS_MOUSE_MOTION_ACK	= 111

4.3. Keyboard messages

Spice supports sending keyboard key events and keyboard leds synchronization. The client sends key event using REDC_INPUTS_KEY_DOWN and REDC_INPUTS_KEY_UP messages. Key value is expressed using PC AT scan code (see KeyCode). Keyboard leds synchronization is done by sending RED_INPUTS_KEY_MODIFAIERS message by the server or by sending REDC_INPUTS_KEY_MODIFAIERS by the client, these messages contain keyboard leds state. Keyboard modifiers is also sent by the server using RED_INPUTS_INIT, this message must be sent as the first server message and the server mustn't send it at any other point.

a) Keyboard led bits

RED_SCROLL_LOCK_MODIFIER	= 1
RED_NUM_LOCK_MODIFIER	= 2
RED_CAPS_LOCK_MODIFIER	= 4

b) RED_INPUTS_INIT, UINT32

UINT32 – any combination of keyboard led bits. If bit is set then the led is on.

c) RED_INPUTS_KEY_MODIFAIERS, UINT32

   UINT32 – any combination of keyboard led bits. If bit is set then the led is on.

d) REDC_INPUTS_KEY_MODIFAIERS, UINT32

   UINT32 – any combination of keyboard led bits. If bit is set then the led is on.

e) KeyCode

   UINT8[4] - the value of key code is a PC AT scan code. The code is composed by up to four bytes for supporting extended codes. A code is terminated by a zero byte.

f) REDC_INPUTS_KEY_DOWN, KeyCode

   KeyCode – client sends this message to notify of key press event.

g) REDC_INPUTS_KEY_UP, KeyCode

   KeyCode – client sends this message to notify of key release event.

4.4. Mouse messages

Spice support two modes of mouse operation: client mouse and server mouse (for more information see "Mouse modes"). In server mouse mode the client sends mouse motion message (i.e., REDC_INPUTS_MOUSE_MOTION), and in client mouse mode it sends position message (i.e., REDC_INPUTS_MOUSE_POSITION). Position message holds the position of the client mouse on the display and the id of the display channel, which is derived from RedLinkMess.channel_id. In order to prevent flood of mouse motion/position events, the server sends RED_INPUTS_MOUSE_MOTION_ACK message on every RED_MOTION_ACK_BUNCH messages it receive. This mechanism allows the client to keep track on the server's messages consumption rate and to change the event pushing policy according to it. Mouse button events are sent to the server using REDC_INPUTS_MOUSE_PRESS and REDC_INPUTS_MOUSE_RELEASE messages.

a) Red Button ID

REDC_MOUSE_LBUTTON	= 1, left button
REDC_MOUSE_MBUTTON	= 2, middle button
REDC_MOUSE_RBUTTON	= 3, right button
REDC_MOUSE_UBUTTON	= 4, scroll up button
REDC_MOUSE_DBUTTON	= 5, scroll down button

b) Buttons masks

REDC_LBUTTON_MASK	= 1, left button mask

REDC_MBUTTON_MASK	= 2, middle button mask
REDC_RBUTTON_MASK	= 4, right button mask

c) RED_MOTION_ACK_BUNCH = 4

d) REDC_INPUTS_MOUSE_MOTION, RedcMouseMotion

   INT32 dx – number of pixels the mouse had moved on x axis

   INT32 dy - number of pixels the mouse had moved on y axis

   UINT32 buttons_state – any combination of buttons mask. Set bit describe pressed button and clear bit describe unpressed button.

e) REDC_INPUTS_MOUSE_POSITION, RedcMousePosition

   UINT32 x – position on x axis

   UINT32 y - position on y axis

   UINT32 buttons_state - any combination of buttons mask. Set bit describe pressed button and clear bit describe unpressed button.

   UINT8 display_id – id of the display that client mouse is on.

f) REDC_INPUTS_MOUSE_PRESS, RedcMousePress

   UINT32 button_id – one of REDC_MOUSE_?BUTTON

   UINT32 buttons_state - any combination of buttons masks. Set bit describes pressed button, and clear bit describes unpressed button.

g) REDC_INPUTS_MOUSE_RELEASE, RedcMouseRelease

   UINT32 button_id – one of REDC_MOUSE_?BUTTON

   UINT32 buttons_state - any combination of buttons mask. Set bit describes pressed button and clear bit describes unpressed button.

5. Display channel definition

Spice protocol defines a set of messages for supporting rendering of the remote display area on the client display. The protocol supports rendering of graphics primitives (e.g., lines, images) and video streams. The protocol also supports caching of images and color palettes on the client side. Spice display channel supports several images compression methods for reducing network traffic.

5.1. Server messages

RED_DISPLAY_MODE	= 101
RED_DISPLAY_MARK	= 102
RED_DISPLAY_RESET	= 103
RED_DISPLAY_COPY_BITS	= 104
RED_DISPLAY_INVAL_LIST	= 105
RED_DISPLAY_INVAL_ALL_IMAGES	= 106
RED_DISPLAY_INVAL_PALETTE	= 107
RED_DISPLAY_INVAL_ALL_PALETTES	= 108
RED_DISPLAY_STREAM_CREATE	= 122
RED_DISPLAY_STREAM_DATA	= 123
RED_DISPLAY_STREAM_CLIP	= 124
RED_DISPLAY_STREAM_DESTROY	= 125
RED_DISPLAY_STREAM_DESTROY_ALL	= 126
RED_DISPLAY_DRAW_FILL	= 302
RED_DISPLAY_DRAW_OPAQUE	= 303
RED_DISPLAY_DRAW_COPY	= 304
RED_DISPLAY_DRAW_BLEND	= 305
RED_DISPLAY_DRAW_BLACKNESS	= 306
RED_DISPLAY_DRAW_WHITENESS	= 307
RED_DISPLAY_DRAW_INVERS	= 308
RED_DISPLAY_DRAW_ROP3	= 309
RED_DISPLAY_DRAW_STROKE	= 310
RED_DISPLAY_DRAW_TEXT	= 311
RED_DISPLAY_DRAW_TRANSPARENT	= 312
RED_DISPLAY_DRAW_ALPHA_BLEND	= 313

5.2. Client messages

REDC_DISPLAY_INIT	= 101

5.3. Operation flow

Spice server sends to the client a mode message using RED_DISPLAY_MODE for specifying the current draw area size and format. In response the client creates a draw area for rendering all the followed rendering commands sent by the server. The client will expose the new remote display area content (i.e., after mode command) only after it receives a mark command (i.e., RED_DISPLAY_MARK) from the server. The server can send a reset command using RED_DISPLAY_RESET to instruct the client to drop its draw area and palette cache. Sending mode message is allowed only while no active draw area exists on the client side. Sending reset message is allowed only while active draw area exists on client side. Sending mark message is allowed only once, between mode and reset messages.

Draw commands, copy bits command and stream commands are allowed only if the client have an active display area (i.e., between RED_DISPLAY_MODE to RED_DISPLAY_RESET).

On channel connection, the client optionally sends an init message, using REDC_DISPLAY_INIT, in order to enable image caching and global dictionary compression. The message includes the cache id and its size and the size of the dictionary compression window. These sizes and id are determined by the client. It is disallowed to send more then one init message.

Color pallets cache are manged by the server.
Items cache insertion commands are sent as part of the rendering commands. Cache items removal are sent explicitly using RED_DISPLAY_INVAL_LIST or RED_DISPLAY_INVAL_LIST server messages. Resetting client caches is done by sending RED_DISPLAY_INVAL_ALL_IMAGES or RED_DISPLAY_INVAL_ALL_PALETTES server messages.

5.4. Draw area control

a) RED_DISPLAY_MODE, RedMode

UINT32 width – width of the display area

UINT32 height - height of the display area

UINT32 depth – color depth of the display area. Valid values are 16bpp or 32bpp.

b) RED_DISPLAY_MARK, VOID

Mark the beginning of the display area visibility

c) RED_DISPLAY_RESET, VOID

Drop current display area of the channel and reset palette cache

5.5. Raster operation descriptor

The following defines a set of flags for describing raster operations that can be applied on a source image, source brush, destination and the result during a rendering operation. Combination of those flags defines the necessary steps that are needed to be preformed during a rendering operation. In the the following definitions of rendering commands this combination is referred to by 'rop_descriptor'.

ROPD_INVERS_SRC = 1
Source Image need to be inverted before rendering

ROPD_INVERS_BRUSH	= 2

Brush need to be inverted before rendering

ROPD_INVERS_DEST	= 4

Destination area need to be inverted before rendering

ROPD_OP_PUT	= 8

Copy operation should be used.

ROPD_OP_OR	= 16

OR operation should be used.

ROPD_OP_AND	= 32

AND operation should be used.

ROPD_OP_XOR	=64

XOR operation should be used.

ROPD_OP_BLACKNESS	= 128

Destination pixel should be replaced by black

ROPD_OP_WHITENESS	= 256

Destination pixel should be replaced by white

ROPD_OP_INVERS	= 512

Destination pixel should be inverted

ROPD_INVERS_RES	= 1024

Result of the operation needs to be inverted

- OP_PUT, OP_OR, OP_AND, OP_XOR, OP_BLACKNESS, OP_WHITENESS, and OP_INVERS are mutually exclusive

- OP_BLACKNESS, OP_WHITENESS, and OP_INVERS are exclusive

5.6. Raw raster image

The following section describes Spice raw raster image (Pixmap). Pixmap is one of several ways to transfer images in Spice protocol (for more information see “Spice Image”).

a) Pixmap format types

PIXMAP_FORMAT_1BIT_LE	= 1

1 bit per pixel and bits order is little endian. Each pixel value is an index in a color table. The color table size is 2.

PIXMAP_FORMAT_1BIT_BE = 2
1 bit per pixel and bits order is big endian. Each pixel value is index in a color table. The color table size is 2.

PIXMAP_FORMAT_4BIT_LE = 3
4 bits per pixel and nibble order inside a byte is little endian. Each pixel value is an index in a color table. The color table size is 16.

PIXMAP_FORMAT_4BIT_BE = 4
4 bits per pixel and nibble order inside a byte is big endian. Each pixel value is an index in a color table. The color table size is 16.

PIXMAP_FORMAT_8BIT = 5
8 bits per pixel. Each pixel value is an index in a color table. The color table size is 256.

PIXMAP_FORMAT_16BIT = 6
pixel format is 16 bits RGB555.

PIXMAP_FORMAT_24BIT = 7
pixel format is 24 bits RGB888.

PIXMAP_FORMAT_32BIT = 8
pixel format is 32 bits RGB888.

PIXMAP_FORMAT_RGBA = 9
pixel format is 32 bits ARGB8888.

b) Palette

UINT64 id - unique id of the palette

UINT16 table_size – number of entries in the color table

UINT32[] color_table – each entry is RGB555 or RGB888 color depending on the current display area mode. If display area mode color depth is 32, the effective format is RGB888. If display area mode color depth is 16 the effective format is RGB555.

c) Pixmap flags

PIXMAP_FLAG_PAL_CACHE_ME = 1
Instruct the client to add the palette to cache

PIXMAP_FLAG_PAL_FROM_CACHE = 2
Instruct the client to retrieve palette from cache.

PIXMAP_FLAG_TOP_DOWN = 4
Pixmap lines are ordered from top to bottom (i.e., line 0 is the highest line).

d) Pixmap

UINT8 format – one of PIXMAP_FORMAT_?

UINT8 flags - combination of PIXMAP_FLAG_?

UINT32 width – width of the pixmap

UINT32 height – height of the pixmap

UINT32 stride – number of bytes to add for moving from the beginning of line n to the beginning of line n+1

union {
    ADDRESS palette – address of the color palette. Must be zero if no color table is required for *format*.

    UINT64 palette_id – id of the palette, valid if FLAG_PAL_FROM_CACHE is set
}

ADDRESS data – address of line 0 of the pixmap.

5.7. LZ with palette

This section describes a data structure that is combination of a color palette and a compressed pixmap data. The pixmap is compressed using our implementation of LZSS algorithm (see next section). Each decoded pixel value is an index in the color palette.

a) LZPalette Flags

LZPALETTE_FLAG_PAL_CACHE_ME	= 1

Instruct the client to add the palette to the cache

LZPALETTE_FLAG_PAL_FROM_CACHE	= 2

Instruct the client to retrieve palette from the cache.

LZPALETTE_FLAG_TOP_DOWN	= 4

pixmap lines are ordered from top to bottom (i.e. line 0 is the highest line).

b) LZPalette

UINT8 flags – combination of LZPALETTE_FLAG_?

UINT32 data_size – size of compressed data

union {
    ADDRESS palette - address of the color palette (see Palette section in “Raw raster image”). Zero value is disallowed.

UINT64 palette_id - id of the palette, valid if FLAG_PAL_FROM_CACHE is set.
}

UINT8[] data – compressed pixmap

5.8. Spice Image

The following section describes Spice image. Spice image is used in various commands and data structures for representing a raster image. Spice image supports several compression types in addition to the raw mode: Quic, LZ and GLZ. Quic is a predictive coding algorithm. It is a generalization of SFALIC from gray-scale to color images withe addition of RLE encoding. By LZ we refer to the our implementation of the LZSS algorithm, which was adjusted for images in different formats. By GLZ we refer to an extension of LZ that allows it to use a dictionary that is based on a set of images and not just on the image being compressed.

a) Image types

IMAGE_TYPE_PIXMAP	= 0
IMAGE_TYPE_QUIC	= 1
IMAGE_TYPE_LZ_PLT	= 100
IMAGE_TYPE_LZ_RGB	= 101
IMAGE_TYPE_GLZ_RGB	= 102
IMAGE_TYPE_FROM_CACHE	= 103

b) Image flags

IMAGE_FLAG_CACHE_ME = 1, this flag instruct the client to add the image to image cache, cache key is ImageDescriptor.id (see below).

c) ImageDescriptor

UINT64 id – unique id of the image

UINT8 type – type of the image. One of IMAGE_TYPE_?

UINT8 flags - any combination of IMAGE_FLAG_?

UINT32 width - width of the image

UINT32 height - height of the image

d) Image data

Image data follows ImageDescriptor and its content depends on ImageDescriptor.type:

In case of PIXMAP – content is Pixmap.

In case of QUIC – content is Quic compressed image. Data begins with the size of the compressed data, represented by UINT32, followed by the compressed data.

In case of LZ_PLT – content is LZPalette.

In case of LZ_RGB – content is LZ_RGB – LZ encoding of an RGB image. Data begins with the size of the compressed data, represented by UINT32, followed by the compressed data.

In case of GLZ_RGB – content is GLZ_RGB – GLZ encoding of an RGB image. Data begins with the size of the compressed data, represented by UINT32, , followed by the compressed data.

In case of FROM_CACHE – No image data. The client should use ImageDescriptor.id to retrieve the relevant image from cache.

5.9. Glyph String

Glyph string defines an array of glyphs for rendering. Glyphs in a string can be in A1, A4 or A8 format (i.e., 1bpp, 4bpp, or 8bpp alpha mask). Every glyph contains its rendering position on the destination draw area.

a) RasterGlyph

POINT render_pos – location of the glyph on the draw area

POINT glyph_origin– origin of the glyph. The origin is relative to the upper left corner of the draw area. Positive value on x axis advances leftward and positive value on y axis advances upward.

UINT16 width - glyph's width

UINT16 height - glyph's height

UINT8[] data – alpha mask of the glyph. Actual mask data depends on the glyph string's flags. If the format is A1 then the line stride is ALIGN(width, 8) / 8. If the format is A4, the line stride is ALIGN(width, 2) / 2. If the format is A8, the line stride is width.

b) Glyph String flags

GLYPH_STRING_FLAG_RASTER_A1 = 1
Glyphs type is 1bpp alpha value (i.e., 0 is transparent 1 is opaque)

GLYPH_STRING_FLAG_RASTER_A4 = 2
Glyphs type is 4bpp alpha value (i.e., 0 is transparent 16 is opaque)

GLYPH_STRING_FLAG_RASTER_A8 = 4
Glyphs type is 4bpp alpha value (i.e., 0 is transparent 256 is opaque)

GLYPH_STRING_FLAG_RASTER_TOP_DOWN = 8
Line 0 is the top line of the mask

c) GlyphString

UINT16 length - number of glyphs
UINT16 flags – combination of GLYPH_STRING_FLAG_?
UINT8[] data - glyphs

5.10. Data Types

a) RectList

UINT32 count – number of RECT items in rects

RECT[] rects – array of <count> RECT

b) Path segment flags

PATH_SEGMENT_FLAG_BEGIN = 1
this segment begins a new path

PATH_SEGMENT_FLAG_END = 2
this segment ends the current path

PATH_SEGMENT_FLAG_CLOSE = 8
this segment closes the path and is invalid if PATH_SEGMENT_FLAG_END is not set

PATH_SEGMENT_FLAG_BEZIER = 16
this segment content is a Bezier curve

c) PathSeg

UINT32 flags – any combination of PATH_SEGMENT_FLAG_?

UINT32 count – number of points in the segment

POINTFIX[] points – segment points

d) PathSegList

List of PathSeg items. End of the list is reached if the sum of all previous PathSegs' sizes is equal to list_size. Address of next segment is the address of

PathSeg.points[PathSeg.count]

UINT32 list_size – total size of in bytes of all PathSegs in the list,

PathSeg seg0 – first path segment.

e) Clip types

CLIP_TYPE_NONE = 0
no clipping

CLIP_TYPE_RECTS = 1
data is RectList and union of all rectangles in RectList is the effective clip

CLIP_TYPE_PATH = 2
data is PathSegList and the figure described by PathSegList is the effective clip

f) Clip

UIN32 type – one of CLIP_TYPE_?

ADDRESS data – address of clip data. The content depends on <type>

g) Mask flags

MASK_FLAG_INVERS = 1, the effective mask is the inverse of the mask

h) Mask

UINT8 flags – flags of the mask, combination of MASK_FLAG_?

POINT position - origin of the mask in bitmap coordinates

ADDRESS bitmap – address of the mask's image, the format of the image must be 1bpp. If the bitmap is zero then no masking operation needs to be preformed.

- In all rendering commands, the mask must be big enough to cover the destination rectangle

i) Brush types

BRUSH_TYPE_NONE	= 0, the brush is invalid.
BRUSH_TYPE_SOLID	= 1, the brush is solid RGB color
BRUSH_TYPE_PATTERN	= 2, the brush is a pattern.

j) Pattern

ADDRESS image – address of the pattern's Image

POINT position – origin coordinates of the pattern in the image

k) Brush

UINT32 type – one of BRUSH_TYPE_?

Union {

UINT32 color – RGB color. The format of the color depends on current draw area mode.

Pattern pattern;
}

l) Image scale mode

The following defines the method for scaling image

IMAGE_SCALE_INTERPOLATE = 0
The client is allowed to INTERPOLATE pixel color.

IMAGE_SCALE_NEAREST = 1
The client must use the nearest pixel.

m) LineAtrr flags

LINE_ATTR_FLAG_START_WITH_GAP = 4
first style segment if gap (i.e., foreground)

LINE_ATTR_FLAG_STYLED = 8
style member of LineAtrr is valid and contains effective line style for the rendering operation.

n) LineAtrr join style

LINE_ATTR_JOIN_ROUND	= 0
LINE_ATTR_JOIN_BEVEL	= 1
LINE_ATTR_JOIN_MITER	= 2

o) LineAtrr cap style

LINE_ATTR_CAP_ROUND	= 0
LINE_ATTR_CAP_SQUARE	= 1
LINE_ATTR_CAP_BUTT	= 2

p) LineAttr

UINT8 flags – combination of LINE_ATTR_?

UINT8 join_style – one of LINE_ATTR_JOIN_?

UINT8 cap_style - one of LINE_ATTR_CAP_?

UINT8 style_num_segments – number of style segments in line style

FIXED28_4 width – width of the line in pixels

FIXED28_4 miter_limit – miter limit in pixels

ADDRESS style – address of line style
line style is array of FIXED28_4. The array defines segments that each represents length of foreground or background pixels in the style. If FLAG_START_WITH_GAP is defined then the first segment in the style is background, otherwise it is foreground. Renderer uses this array of segments repeatedly during rendering operation.

5.11. Rendering command

a) RedDrawBase

Common field to all rendering command

RECT bounding_box – the affected area on the display area

Clip clip – the effective clip to set before rendering a command

b) RED_DISPLAY_COPY_BITS

RedDrawBase
POINT source_position

Copy bits from the draw area to bounding_box on the draw area. Source area left top corner is source_position and its height and width is equal to bounding_box height and width. Source and destination rectangles can overlap.

c) RED_DISPLAY_DRAW_FILL

RedDrawBase
Brush brush
UINT16 rop_descriptor
Mask mask

Fill bounding_box using brush as the fill pattern and rop_descriptor instructions. If the mask is valid, it will limit the modified area (i.e., only pixels on the destination area that their corresponding bits are set will be affected).

d) RED_DISPLAY_DRAW_OPAQUE

RedDrawBase
ADDRESS source_image
RECT source_area
Brush brush
UINT16 rop_descriptor
UINT8 scale_mode
Mask mask

Combine pixels from source_area in source_image with the brush's pattern using rop_descriptor instructions. The result image will be rendered into bounding_box. In case scaling of source image is required it will be performed according to scale_mode and before the combination with brush pixels. If mask is valid it will limit the modified area.

e) RED_DISPLAY_DRAW_COPY

RedDrawBase
ADDRESS source_image
RECT source_area
UINT16 rop_descriptor
UINT8 scale_mode
Mask mask

Copy pixels from source_area in source_image to bounding_box using rop_descriptor instructions. In case scaling of source image is required it will be performed according to scale_mode and before the copying to the draw area. If mask is valid it will limit the modified area.

f) RED_DISPLAY_DRAW_BLEND

RedDrawBase
ADDRESS source_image
RECT source_area
UINT16 rop_descriptor
UINT8 scale_mode
Mask mask

Mixing pixels from source_area in source_image with bounding_box pixels on the draw area using rop_descriptor instructions. In case scaling of source image is required it will be performed according to scale_mode and before the mixing with the draw area. If mask is valid it will limit the modified area.

g) RED_DISPLAY_DRAW_BLACKNESS

RedDrawBase
Mask mask

Fill bounding_box with black pixels. If mask is valid it will limit the modified area.

h) RED_DISPLAY_DRAW_WHITENESS

RedDrawBase
Mask mask

Fill bounding_box with white pixels. If mask is valid it will limit the modified area.

i) RED_DISPLAY_DRAW_INVERS

RedDrawBase
Mask mask

Inverse all pixels in bounding_box. If mask is valid it will limit the modified area.

j) RED_DISPLAY_DRAW_ROP3

RedDrawBase
ADDRESS source_image
RECT source_area
Brush brush
UINT8 rop3
UINT8 scale_mode
Mask mask

Mix pixels from source_area in source_image, bounding_box pixels in the draw area, and the brush pattern. The method for mixing three pixels into the destination area (i.e., bounding_box) is defined by rop3 (i.e., *ternary raster operations)*. In case scaling of source image is required it will be performed according to scale_mode and before the mixing. If mask is valid it will limit the modified area.

k) RED_DISPLAY_DRAW_TRANSPARENT

RedDrawBase
ADDRESS source_image
RECT source_area
UINT32 transparent_color
UINT32 transparent _true_color

Copy pixels from source_area on source_image to bounding_box on the draw area. In case scaling of source image is required it will use IMAGE_SCALE_NEAREST. Pixels with value equal to the transparent color will be masked out. Transparent color is provided in two forms: true color (i.e., RGB888) and the color in the original format (i.e., before compression) .

l) RED_DISPLAY_DRAW_ALPHA_BLEND

RedDrawBase
UINT8 alpha

ADDRESS source_image
RECT source_area

Alpha blend source_area of source_image on bounding_box of draw area using alpha value or alternatively per pixel alpha value. In case scaling of source image is required, it will use IMAGE_SCALE_INTERPOLATE mode. Alpha value is defined as 0 is full transparency and 255 is full opacity. Format of source image can be pre-multiplied ARGB8888 for per pixel alpha value.

New RGB color is defined as:
color' = (source_color * alpha) / 255
alpha' = (source_alpha * alpha) / 255
new_color = color' + ((255 - alpha') * destination_color) / 255

m) RED_DISPLAY_DRAW_STROKE

RedDrawBase
ADDRESS path – address of the PathSegList that defines the path to render
LineAttr attr
Bush brush
UINT16 fore_mode - foreground rop_descriptor
UINT16 back_mode – background rop_descriptor

Render path using brush line attribute and rop descriptors. If the line is styled (i.e., LINE_ATTR_FLAG_STYLED is set in attr.falgs) then background (i.e., inverse of the style) is drawn using back_mode and the foreground is drawn using fore_mode. If the line is not styled, the entire path is rendered using fore_mode.

n) RED_DISPLAY_DRAW_TEXT

RedDrawBase
ADDRESS string – address of GlyphString
RECT back_area
Brush fore_brush
Brush back_brush
UINT16 fore_mode
UINT16 back_mode

Render string of glyph on the display area using brush fore_brush and the rop_descriptor fore_mode. If back_area is not empty the renderer fill back_area on the display area prior to rendering the glyph string. back_area is filled using back_brush and the rop_descriptor back_mode.

5.12. Video streaming commands

Spice supports the creation of video streams by the server for rendering video content on the client display area. Unlike other rendering commands, the stream data can be compressed using lossy or video specific compression algorithms. It is not required to render video frames as they arrive and it is also allowed to drop video frames. This enables using video frames buffering for having smoother playback and audio synchronization. Audio synchronization is achieved by using time stamp that is attached to audio and video streams. By using video streaming the network traffic can be dramatically reduced. When the stream is created, the server sends create message using RED_DISPLAY_STREAM_CREATE. After the server creates a stream he can send data using RED_DISPLAY_STREAM_DATA, or set new stream clipping by sending clip message using RED_DISPLAY_STREAM_CLIP. Once the server no longer needs the stream, he can send destroy command using RED_DISPLAY_STREAM_DESTROY. The server can also destroy all active streams by sending destroy all message using RED_DISPLAY_STREAM_DESTROY_ALL.

a) Stream flags

   STREAM_FLAG_TOP_DOWN = 1, stream frame line order is from top to bottom

b) Codec types

   STREAM_CODEC_TYPE_MJPEG = 1, this stream uses motion JPEG codec

c) RED_DISPLAY_STREAM_CREATE, RedStreamCreate

   UINT32 id – id of the new stream. It is the server's responsibility to manage stream ids

   UINT32 flags – flags of the stream, any combination of STREAM_FLAG_?

   UINT32 codec_type – type of codec used for this stream, one of STREAM_CODEC_TYPE_?

   UINT64 reserved – must be zero

   UINT32 stream_width - width of the source frame.

   UINT32 stream_height - height of the source frame

   UINT32 source_width – actual frame width to use, must be less or equal to stream_width.

   UINT32 source_height - actual frame height to use, must be less or equal to stream_height.

   RECT destination – area to render into on the client display area

   Clip clip – clipping of the stream

d) RED_DISPLAY_STREAM_DATA, RedStreamData

   UINT32 id – stream id (i.e., RedStreamCreate.id)

UINT32 multimedia_time – frame time stamp

UINT32 data_size – stream data size to consume in bytes

UINT32 pad_size – additional data padding in bytes

UINT8[] data – stream data depending on RedStreamCreate.codec_type. Size of data is (data_size + pad_size)

e) RED_DISPLAY_STREAM_CLIP, RedStreamClip

   UINT32 id - stream id (i.e., RedStreamCreate.id)

   Clip clip – new clipping of the stream

f) RED_DISPLAY_STREAM_DESTROY, UINT32

   UINT32 – id of stream to destroy

g) RED_DISPLAY_STREAM_DESTROY_ALL, VOID

   Destroy all active streams

5.13. Cache control

a) Resource type

   RED_RES_TYPE_IMAGE = 1

b) RedResourceID

   UINT8 type – type of the resource, one of RED_RES_TYPE_?

   UINT64 id – id of the resource

c) RedResourceList

   UINT16 count – number of items in resources

   RedResourceID[] resources - list of resources id

d) RED_DISPLAY_INVAL_LIST, RedResourceList

   RedResourceList – list of resources to remove from cache

e) RED_DISPLAY_INVAL_ALL_IMAGES, RedWaitForChannels

Remove all images from the image cache. The client must use RedWaitForChannels (for more info see 2.19 Channel synchronization) to synchronize with other channels before clearing the cache.

f) RED_DISPLAY_INVAL_PALETTE, UINT64

UINT64 – id of palette, client needs to remove palette with that id from the cache

g) RED_DISPLAY_INVAL_ALL_PALETTES, VOID

Remove all palettes from palette cache

6. Cursor channel definition

Spice protocol defines a set of messages for controlling cursor shape and position on the remote display area, cursor position messages are irrelevant for client mouse mode (see “Mouse mode”). Spice protocol also defines a set of messages for managing cursor shape cache on the client site. Client must strictly obey all such instructions. The server sends RED_CURSOR_INIT to set current pointer state (i.e., shape, position, visibility etc.) and to clear shape cache. After the server sends init message it can send any other cursor command except for RED_CURSOR_INIT. The server can send RED_CURSOR_RESET message - this will disable the cursor and reset the cursor cache. After this message the only valid message the server can send is RED_CURSOR_INIT. The relevant remote display area for a cursor channel is the one of the display channel that has the same channel id (i.e., RedLinkMess.channel_id).

6.1. Server messages

RED_CURSOR_INIT	= 101
RED_CURSOR_RESET	= 102
RED_CURSOR_SET	= 103
RED_CURSOR_MOVE	= 104
RED_CURSOR_HIDE	= 105
RED_CURSOR_TRAIL	= 106
RED_CURSOR_INVAL_ONE	= 107
RED_CURSOR_INVAL_ALL	= 108

a) Cursors types

▪ CURSOR_TYPE_ALPHA	= 0
▪ CURSOR_TYPE_MONO	= 1
▪ CURSOR_TYPE_COLOR4	= 2
▪ CURSOR_TYPE_COLOR8	= 3
▪ CURSOR_TYPE_COLOR16	= 4

- CURSOR_TYPE_COLOR24 = 5
- CURSOR_TYPE_COLOR32 = 6

b) CursorHeader

UINT64 unique – unique identifier of the corresponding cursor shape. It is used for storing and retrieving cursors from the cursor cache.

UINT16 type – type of the shape, one of CURSOR_TYPE_?

UINT16 width – width of the shape

UINT16 height - height of the shape

UINT16 hot_spot_x – position of hot spot on x axis

UINT16 hot_spot_y - position of hot spot on y axis

c) Cursor flags

CURSOR_FLAGS_NONE = 1
set when RedCursor (see below) is invalid

CURSOR_CURSOR_FLAGS _CACHE_ME = 2
set when the client should add this shape to the shapes cache. The client will use CursorHeader.unique as cache key.

CURSOR_FLAGS_FROM_CACHE = 4
set when the client should retrieve the cursor shape, using CursorHeader.unique as key, from the shapes cache. In this case all fields of CursorHeader except for 'unique' are invalid.

d) RedCursor

UINT32 flags – any valid combination of RED_CURSOR_?

CursorHeader header

UINT8[] data – actual cursor shape data, the size is determine by width, height and type from CursorHeader. Next we will describe in detail the shape data format according to cursor type:

- ALPHA, alpha shape – data contains pre-multiplied ARGB8888 pixmap. Line stride is is <width * 4>.

- MONO, monochrome shape - data contains two bitmaps with size <width> * <height>. The first bitmap is AND mask and the second is XOR mask. Line stride is ALIGN(<width>, 8) / 8. Bits order within every byte is big endian.

- COLOR4, 4 bits per pixel shape - First data region is pixmap: the stride of the pixmap is ALIGN(width , 2) / 2; every nibble is translated to a color usingthe color palette; Nibble order is big endian. Second data region contain 16 colors palette: each entry is 32 bit RGB color. Third region is a bitmap mask: line stride is ALIGN(<width>, 8) / 8; bits order within every byte is big endian.

- COLOR4, 8 bits per pixel shape - First data region is pixmap: the stride of the pixmap is <width>; every byte is translated to color using the color palette. Second data region contain 256 colors palette: each entry is 32 bit RGB color. Third region is a bitmap mask: line stride is ALIGN(<width>, 8) / 8; bits order within every byte is big endian.

- COLOR16, 16 bits per pixel shape - First data region is pixmap: the stride of the pixmap is <width * 2>; every UINT16 is RGB_555. Second region is a bitmap mask: line stride is ALIGN(<width>, 8) / 8; bits order within every byte is big endian.

- COLOR24, 24 bits per pixel shape - First data region is pixmap: the stride of the pixmap is <width * 3>; every UINT8[3] is RGB_888. Second region is a bitmap mask: line stride is ALIGN(<width>, 8) / 8; bits order within every byte is big endian.

- COLOR32, 32 bits per pixel shape - First data region is pixmap: the stride of the pixmap is <width * 4>,;every UINT32 is RGB_888. Second region is a bitmap mask: line stride is ALIGN(<width>, 8) / 8; bits order within every byte is big endian.

For more deatails on drawing the cursor shape see Section 6.2.

e) RED_CURSOR_INIT, RedCursorInit

POINT16 position – position of mouse pointer on the relevant display area. Not relevant in client mode.

UINT16 trail_length – number of cursors in the trail excluding main cursor.

UINT16 trail_frequency – millisecond interval between trail updates.

UIN8 visible – if 1, the cursor is visible. If 0, the cursor is invisible.

RedCursor cursor – current cursor shape

f) RED_CURSOR_RESET, VOID

g) RED_CURSOR_SET, RedCursorSet

POINT16 position - position of mouse pointer on the relevant display area. not relevant in client mode.

UINT8 visible – if 1, the cursor is visible. If 0, the cursor is invisible.

RedCursor cursor – current cursor shape

h) RED_CURSOR_MOVE, POINT16

   POINT16 – new mouse position. Not relevant in client mode. This message also implicitly sets cursor visibility to 1.

i) RED_CURSOR_HIDE, VOID

   Hide pointer on the relevant display area.

j) RED_CURSOR_TRAIL

   UINT16 length - number of cursors in the trail excluding main cursor.

   UINT16 frequency - millisecond interval between trail updates

k) RED_CURSOR_INVAL_ONE, UINT64

   UINT64 – id of cursor shape to remove from the cursor cache

l) RED_CURSOR_INVAL_ALL, VOLD

   Clear cursor cache

6.2. Drawing the cursor shape according to the cursor type

This section is relevant only for server mouse mode. Cursor shape positioning on the display area is done by placing cursor hot spot on the current cursor position.

a) Alpha - no spacial handling, just bland the shape on the display area.

b) Monochrome -
   - For each cleared bit in the AND mask clear the corresponding bits in the relevant pixel on the display area
   - For each set bit in the XOR mask reverse the corresponding bits in the relevant pixel on the display area

c) Color -
   - If the source color is black and mask bit is set, NOP.
   - Else, if the source color is white and the mask bit is set, reverse all bits in the relevant pixel on the display area.
   - Else, put source color.

7. Playback channel definition

Spice supports sending audio streams for playback on the client side. An audio stream is sent by the server in an audio packet using RED_PLAYBACK_DATA message. The content of the audio packet is controlled by the playback mode that the server sends using RED_PLAYBACK_MODE message. The server can start and stop the stream using RED_PLAYBACK_START and RED_PLAYBACK_STOP messages. Sending audio packet is allowed only between start and stop messages. Sending start message is allowed only in stop state and after at least one mode message was sent. Sending a stop message is allowed only during a start state.

7.1. Server messages

RED_PLAYBACK_DATA	= 101
RED_PLAYBACK_MODE	= 102
RED_PLAYBACK_START	= 103
RED_PLAYBACK_STOP	= 104

7.2. Audio format

RED_PLAYBACK_FMT_S16 = 1, each channel sample is a 16 bit signed integer

7.3. Playback data mode

Two types of data mode are available: (1) raw PCM data and (2) compressed data in CELT 0_5_1 format.

RED_PLAYBACK_DATA_MODE_RAW	= 1
RED_PLAYBACK_DATA_MODE_CELT_0_5_1	= 2

7.4. Playback channel capabilities

RED_PLAYBACK_CAP_CELT_0_5_1 = 0

Spice client needs to declare support of CELT_5_1 in channel capabilities in order to allow the server to send playback packets in CELT_0_5_1 format.

7.5. RED_PLAYBACK_MODE, RedPlaybackMode

UINT32 time – server time stamp

UINT32 mode – one of RED_PLAYBACK_DATA_MODE_?

UINT8[] data – specific data, content depend on mode

7.6. RED_PLAYBACK_START, RedRecordStart

UINT32 channels – number of audio channels

UINT32 format – one of RED_PLAYBACK_FMT_?

UINT32 frequency – channel samples per second

7.7. RED_PLAYBACK_DATA, RedPlaybackPacket

UINT32 time - server time stamp

UINT8[] data – playback data , content depend on mode

7.8. RED_PLAYBACK_STOP, VOID

Stop current audio playback

8. Record Channel definition

Spice supports transmitting of audio captured streams from the client to the server. Spice server starts audio capturing using RED_RECORD_START message. This message instructs the client to start transmitting captured audio . In response, the client sends time stamp of the stream start using REDC_RECORD_START_MARK. After the client sends start mark it can start transmitting audio stream data using REDC_RECORD_DATA. One mode message must be sent by the client before any other message using REDC_RECORD_MODE. This, in order to inform the server on what type of data will be transferred. Mode message can also be transmitted at any other time in order to switch the data type delivered by REDC_RECORD_DATA. The Server can send RED_RECORD_STOP for stopping captured audio streaming. Sending a start message is allowed only while the stream is in stop state. Sending a stop message and data messages is allowed only while the stream is in start state. Sending mark message is allowed only between start message and the first data message.

8.1. Server messages

RED_RECORD_START	= 101
RED_RECORD_STOP	= 102

8.2. Client messages

REDC_RECORD_DATA	= 101
REDC_RECORD_MODE	= 102
REDC_RECORD_START_MARK	= 103

8.3. Audio format

RED_RECORD_FMT_S16 = 1, each channel sample is a 16 bit signed integer

8.4. Record data mode

Two types of data mode are available: (1) raw PCM data (2) compressed data in CELT 0.5.1 format.

RED_RECORD_DATA_MODE_RAW	= 1
RED_RECORD_DATA_MODE_CELT_0_5_1	= 2

8.5. Record channel capabilities

RED_PLAYBACK_CAP_CELT_0_5_1 = 0

Spice server needs to declare support of CELT_5_1 in channel capabilities in order to allow the client to send recorded packets in CELT_0_5_1 format.

8.6. REDC_RECORD_MODE, RedcRecordMode

UINT32 time – client time stamp

UINT32 mode – one of RED_RECORD_DATA_MODE_?

UINT8[] data – specific data, content depend on mode

8.7. RED_RECORD_START, RedRecordStart

UINT32 channels – number of audio channels

UINT32 format – one of RED_AUDIO_FMT_?

UINT32 frequency – channel samples per second

8.8. REDC_RECORD_START_MARK, UINT32

UINT32 – client time stamp of stream start

8.9. REDC_RECORD_DATA, RedcRecordPacket

UINT32 time - client time stamp

UINT8[] data – recorded data , content depend on mode

8.10. RED_RECORD_STOP, VOID

Stop current audio capture