B A C H E L O R
T H E S I S

VNC-Interface for Java X86-Emulator Dioscuri

Evgeni Genev

Matr.-Nr. 2151451

advisor

Dr. Dirk von Suchodoletz

Prof. Dr. Gerhard Schneider

at the

Chair of Communication Systems
University of Freiburg

Declaration

Table of Contents

Abstract

Dioscuri ist ein in Java geschriebener X86-Emulator für die digitale Langzeitarchivierung. Es gab im Zuge des PLANETS Project eine Reihe von Experimenten für die Automatisierung von typischerweise interaktiven Abläufen. Dieses wurde bisher mit dem Emulator QEMU gemacht, der über ein eingebautes VNC-Interface verfügt. Dieser Ansatz soll verallgemeinert werden und dafür auch Dioscuri um ein solches VNC-Interface erweitert werden. Danach soll dessen Funktion getestet werden.

1 Introduction

In the past fifty years a real boom in the development of computer hardware and software has been witnessed. On the one hand, this trend has a positive influence on the software development. On the other hand, it is almost impossible to keep digital equipment older than twenty years running, either because the spare parts are very expensive or because they are almost impossible to be found. As a consequence, many digital objects are loosing their natural hardware and software environment. Literally, this could mean for some of them the end of their life. This makes the virtual reproduction of this working environment, the emulation, an important part of the preservation of digital objects.

The emulation is not the only possible preservation strategy; another one is the migration. Both are rather complementary than contradictory to each other. The migration, for example, can benefit from the emulation. The emulation makes it possible to convert digital objects in the future, even if the particular working environment is no longer available. Another positive effect of the emulation is that the object can stay unchanged and, as already mentioned, can be viewed in its original environment.

The migration of software in such an emulated environment poses another question: how to create a generalized automated migration approach that works for any combination of emulated hardware and software? The main observation is that the most objects to be kept are objects, which were created and managed using some text screen or a graphical user interface. Another fact is that any user interaction can be interpreted as a sequence of mouse and keyboard events. An automated GUI or command line sequence can be created recording these user interactions and running them afterward, proving not only the end result but also the in-between states.

As a part of the PLANETS[1] project a handful of experiments was made trying to automate typically interactive procedures. These tests have shown that the remote control software VNC[2] is a very promising solution. VNC builds upon RFB[3] - a simple remote access protocol, which works at the framebuffer level and is consequently platform independent.

The above mentioned tests were made using the QEMU[4] emulator, which has an integrated VNC interface. The resulting automation technique should be generalized and tested with other suitable emulators. The main problem is that many of them do not have an integrated VNC interface. There exists also the possibility to run a VNC server inside of the emulated environment, but the solution with a VNC interface as a part of the emulator is the general one.

The last paragraph almost states the goal of this work: it is the implementation of an integrated VNC server for another suitable emulator, thus help generalizing the above technique. As such

[1]See PLANETS [2010]
[2]See RealVNC [2010]
[3]See Richardson [2009]
[4]See QEMU [2010]

an emulator Dioscuri[5] is chosen. The first two sentences from the Dioscuri's homepage outline the reasons for this choice:

> „Dioscuri is an x86 computer hardware emulator written in Java“, and therefore platform independent.
>
> „It is designed by the digital preservation community to ensure documents and programs from the past can still be accessed in the future. “

[5]See Dioscuri [2010]

2 State of the art

The aim of this chapter is to state the preconditions for the development of a VNC interface for Dioscuri. It starts with a brief explanation of what an emulator is, then describes the current development state of the emulator, and goes shortly into details on how it functions. Afterwards the terms VNC and RFB will be explained and at the end an overview of the Java library VNCj[6] will be supplied.

2.1 The hardware emulator Dioscuri

There are many definitions of the emulator term. One of them can be found in Rothenberg [2000]:

> „The computer science term 'emulation' denotes a process in which one computer is used to reproduce the behavior of another computer with such fidelity that the emulation can be used in place of the original computer..."

From the digital preservation point of view an emulator is a tool used to keep an obsoleted software running, without needing the respective hardware. Not the performance but the flexibility and the durability of the emulator are important:

> „As Moore's law is still applicable today, it is likely that computer systems in the future will turn out to be faster. Speculating on this line of thought, building an emulator for digital preservation doesn't have to be focused on performance too much." [7]

These specific requirements led the National Library of the Netherlands[8] and the Nationaal Archief of the Netherlands[9] to the idea of the creation of Dioscuri, a new emulator to suite exactly the needs of the long time preservation of digital objects.

Dioscuri is a relatively young project. Its development started in 2006 and was led by the software company Tessella Support Services plc.[10] „From 2008 onwards, Dioscuri is being further developed by the National Library of the Netherlands within the European project PLANETS and later on also the European project KEEP."[11] The main goal is the creation of a full 32-bit emulator, but, as stated on the project's homepage, the intermediate result is focused on the creation of a „16-bit emulator capable of executing MS DOS and earlier versions of MS Windows (3.x)."

[6]See Liron [2002]
[7]See van der Hoeven [2007]
[8]The homepage of the library `http://www.kb.nl/`
[9]The homepage of the archive `http://www.nationaalarchief.nl/`
[10]The homepage of the Tessella company `http://www.tessella.com/`
[11]Dioscuri [2010]

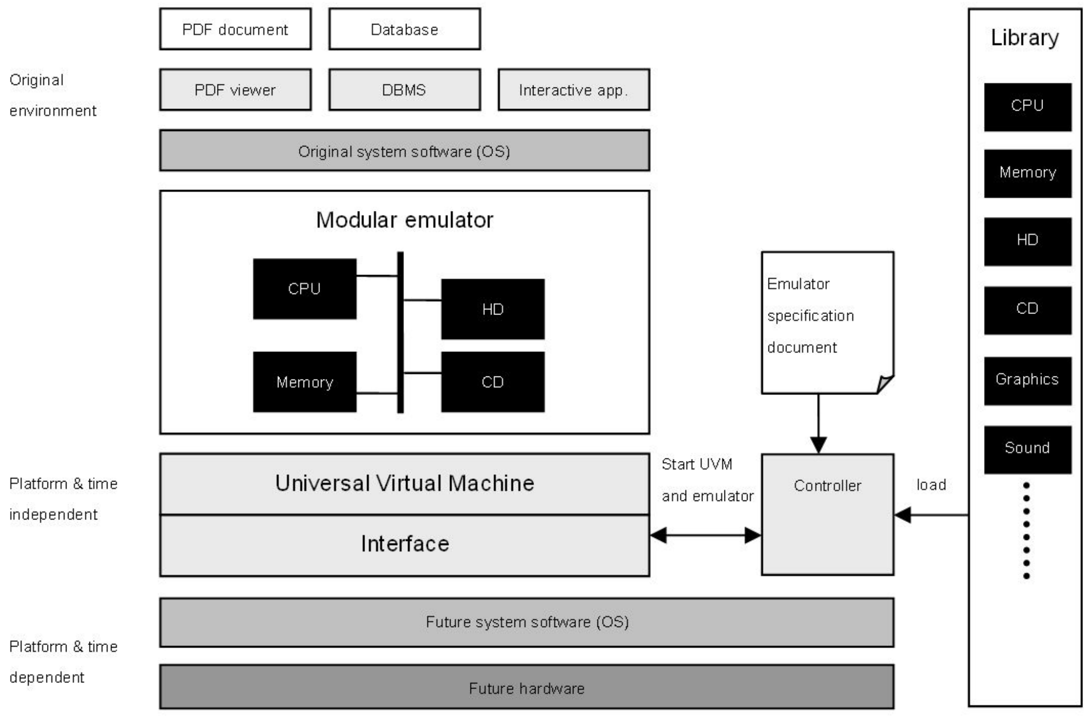


Figure 1: The Dioscuri design[12]

Dioscuri is an open source software published under the GNU General Public License (GPL).[13] It is written in Java, hence it can be moved to any computer platform which supports the Java Virtual Machine (JVM). It is the portability that is crucial for the long-term availability of the emulator. Software already running on multiple platforms is supposed to continue running in the future too.

Dioscuri is completely component-based. From the users point of view this is a very convenient characterization; they can put different hardware modules together building a very specific virtual machine suiting their needs. This flexibility eases the life of the developers too, making the development of new or updated future modules simpler.

The structure of the emulator's GUI is quite simple. It consists of three main components – the menu, the screen panel (with its underlying emulation screen) and the status bar. The GUI is built using the Swing API, which is a part of the Java Foundation Classes.[14]

Of major importance for this work the screen panel plus the underlying emulation screen as well as the GUI itself are. The entire output of the underlying emulation is redirected to the

[12]Source: http://www.kb.nl/hrd/dd/dd_projecten/projecten_emulatieproject-en.html
[13]The GNU General Public License http://www.gnu.de/documents/gpl.en.html
[14]A good start using JFC/Swing is „The Swing Tutorial“ at http://download.oracle.com/javase/tutorial/uiswing/

screen panel. In other words, the content of the screen panel represents the output to be carried over the network. The input is composed of keyboard and pointer events. The pointer events are handled in the emulation screen whereas the keyboard input is handled in the GUI. The protocol which takes care of how the input and the output are to be transported over the network is the Remote Framebuffer protocol.

2.2 RFB and VNC

„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). “[15]

The first important feature of RFB is its simplicity, which makes the creation of a new client or the porting of an existing one an easy process. The next distinguishing marks, which make RFB a good tool for the uses of the digital preservation are its portability and flexibility. RFB can be used to supply access to any windowing system or to a single application. This turns RFB and partly VNC to a very usable tool, when the intention is to allow remote access to the graphical user interface of particular desktop application.

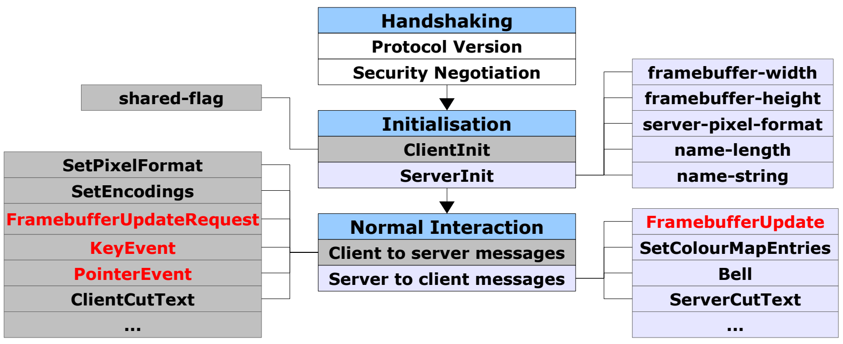


Figure 2: The RFB phases

Two communication endpoints can be distinguished: the RFB server and the RFB client or viewer. The RFB server is the machine where the changes to the framebuffer originate, i.e. where the accessed GUI application runs. The RFB client is the other endpoint, where the keyboard and the pointer input are generated.[16]

[15]The most citations in this work are from RFB 3.8 because of the better explanation of the terms and the not so huge differences between the versions.

[16]The client and the server can be a single machine like in the case, where the VNC is only used to create an extra abstraction level between the user interface and the real GUI hidden behind the VNC interface.

As shown in *Figure 2*, the protocol has three subsequent stages. In the *Negotiation Phase* the protocol version and the security are negotiated. If and only if both sides agree about them, the *Initialization Phase* happens. It is about setting the standards for the further communication. The third and last stage is the communication itself or the *Normal Interaction Phase*.

The steps inside the *Communication Phase* look in general as follows (see also *Figure 3*):

1. The user generates input typing on his keyboard or moving his mouse.
2. The input is sent from the RFB client to the server whenever the user presses a key or a button or whenever he moves a pointing device
3. The server software registers the events and relays them to the windowing system.
4. The windowing system updates its framebuffer if necessary.
5. On client request the server sends the framebuffer updates to the client.

In the Implementation chapter it will be accentuated on the *Normal Interaction Phase*, especially on the way the *key* and *pointer events* are translated to the underlying application layer and on the different ways the *framebuffer updates* are generated and encoded.

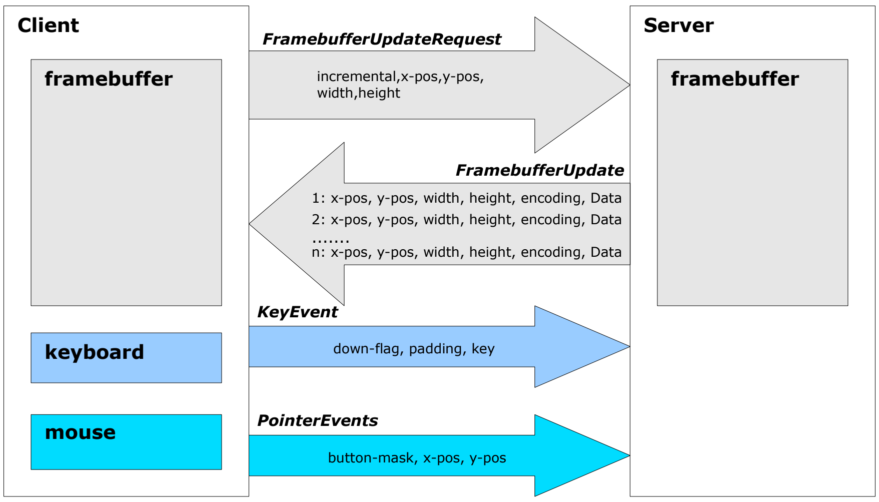

Client
framebuffer
keyboard
mouse
FramebufferUpdateRequest
incremental,x-pos,y-pos, width,height
FramebufferUpdate
1: x-pos, y-pos, width, height, encoding, Data
2: x-pos, y-pos, width, height, encoding, Data
....
n: x-pos, y-pos, width, height, encoding, Data
KeyEvent
down-flag, padding, key
PointerEvents
button-mask, x-pos, y-pos
Server
framebuffer

Figure 3: the RFB's communication phase

2.3 The Java library VNCj

VNCj[17] is a Java library, created in the late 2000, lastly updated in 2002 (despite some small changes in 2003) and published under the LGPL[18]. At the present day the development of the library is discontinued. Its latest release is designed for JDK 1.4 and the standard today is JDK 6.[19]

According to its homepage VNCj is a Java library „for developing VNC servers“. The library implements the first version of the RFB, namely the version 3.3 from January 1998. However, the differences between RFB 3.3 and 3.8 are not so immense. For example, some new pixel encodings were added, but the manner the client and the server communicate is almost unchanged.

The library allows six different models of use: „Swing“, „AWT“, „Lightweight“, „Pixel“, „Console“ and the most general of all – the one that all others implement – the „RFB“ model. For every model a small example of use exists. As shown in the Implementation phase, the „Swing Model“ and the underlying „RFB Model“ are highly important for the purposes of this work.

The library has also a tiny integrated web server, which can serve the Java applet to any Java-enabled browser.

Another benefit of the library is its uniqueness. Java is mainly used on the client-side to supply a platform independent RFB Viewers. The server side applications are developed using compiled languages like C++ for example.[20] This strategy leads to speed advantages but at the same time limits the portability of the implemented VNC server software. As already stated, not the speed but the portability is of greater importance for the long time preservation of digital objects.

The project's homepage claims that „using VNCj, you can create a full graphical user interface in Java [...] and immediately export it to anyone on the network with a VNC viewer“. The next chapter demonstrates that an updated version of the library can also be used to export an already existing Java GUI.

[17]See Liron [2002]

[18]The GNU Lesser General Public License `http://www.gnu.org/licenses/lgpl.html`

[19]JDK 6 is sometimes also referred as JDK 1.6. In this thesis the Oracle notation is used, namely JDK 6 Update 21

[20]See `http://libvncserver.sourceforge.net/`

3 Implementation

The goal of this chapter is to show the development of a fully functional VNC interface for the modular emulator Dioscuri. A brief look inside the source code of Dioscuri and of VNCj helps outlining the rough implementation idea.

On the one hand, Dioscuri is a Java Swing application and his top level container, the *DioscuriFrame*, is an extended *JFrame*. The *DioscuriFrame* also registers the mouse and keyboard events and redirects them to the emulation module. The emulation's screen output occurs inside the GUI's *screenPane* which is an instance of the *JScrollPane* class.

On the other hand, the VNCj's *VNCJFrame* is also an extension of the *JFrame*. Creating an instance of the *VNCJFrame* and adding the already instantiated *screenPane* as a *contentPane* of it solves the *output* problem. The solution of the *input* problem is also straightforward – capture the keyboard and the mouse events in the top level *VNCJFrame* and send them to the Dioscuri's event manager.

In the first part of this chapter the update of the VNCj library is briefly represented. An explanation of the event translation and the *framebuffer update* implementation follow. The last subpart covers the creation of the VNC interface.

The development and the testing were made under JDK 6 Update 21[21] using the free IDE Eclipse Galileo.[22]

3.1 Updating VNCj

The latest version of VNCj requires JDK 1.4.0, the actual version of the JDK is JDK 6 Update 21. Initial tests under JDK 6 have shown, that the Swing and AWT parts of the library are not working correctly. The main problems were caused by interface changes. Fortunately, the most missing methods were almost automatically added by Eclipse.

Another problem was that the *gnu.awt.virtual.VirtualLightweightPeer*, needed both from the Swing and the AWT part, was not a *Swing Container*. Adding an extra *„implements ContainerPeer“*-directive and the corresponding methods solves this problem too.

3.2 Events

From the RFB point of view the key events are part of the *Client to server messages* and build the server-side input. This input is generated on the client side. It reaches the server in the form

[21]See `http://www.oracle.com/technetwork/java/javase/overview/index.html`
[22]See `http://www.eclipse.org/galileo/`

of machine-independent byte stream. To ease the decoding every message begins with a single byte, holding its unique identity number. The message-specific data follows.

From Java's point of view this messages form a *Data Stream*. The part of VNCj, that breaks this input data stream into useful data chunks is the *gnu.rfb.server.RFBSocket* class. Every message is afterwards resend for processing to the respective part of the library. The postprocessing of the key and the pointer events, for example, is done in the *gnu.vnc.awt.VNCEvents* class.

3.2.1 Keyboard events

The key events as defined in the RFB Protocol are „A key press or release". A key event generates an 8 byte long message where the second byte is non-zero, if the key was pressed, and zero on key released. The identity of the key is coded in the last 4 bytes using the key's *keysym* value.[23] Pressing the left *Shift* key, for example, followed by the *'a'* key generates two key pressed events - one with keysym value ***0xffe1*** and another with ***0x0041*** – the *keysym* of the Latin capital letter *'A'*(the *keysym* for *'a'* is ***0x0061***). The simple observation is, that the pressing of a capital *'A'* can be identified only by its *keysym*. According to RFB a „shift state" is only to be used as a hint and „the difference between upper and lower case *keysyms* is significant". Vice versa, the *modifier keys* like *Control* and *Alt* „should be taken as modifying the interpretation of other *keysyms*".

Java's interpretation of the key events looks a little bit different. Java distinguishes between three types of key events: *key-pressed*, *key-released* and *key-typed*. There are also three key types:

1. Character keys – letters, numbers and all symbols
2. Action keys – the function keys (*F1*, *F2*,...), *Page Up*, *Page Down*,...
3. Modifiers – *Shift*, *Control*, *Alt* etc.

The character keys generate all three event types. According to the *KeyEvent* documentation the key-typed event is the preferred way to find out about character input. The action keys and the modifiers generate only key-pressed and key-released events. The modifiers also modify the way the character and action keys are interpreted.

Typing for example the *'a'* key in a Java GUI produces the following event sequence:

```
java.awt.event.KeyEvent[KEY_PRESSED,keyCode=65,
    keyText=A,keyChar='a',keyLocation=KEY_LOCATION_STANDARD,
    rawCode=65,primaryLevelUnicode=97,scancode=30] on frame0

java.awt.event.KeyEvent[KEY_TYPED,keyCode=0,
    keyText=Unknown keyCode: 0x0,keyChar='a',keyLocation=KEY_LOCATION_UNKNOWN,
    rawCode=0,primaryLevelUnicode=0,scancode=0] on frame0
```

[23] A list of the *keysym* values can be found under `http://cgit.freedesktop.org/xorg/proto/x11proto/plain/keysymdef.h`

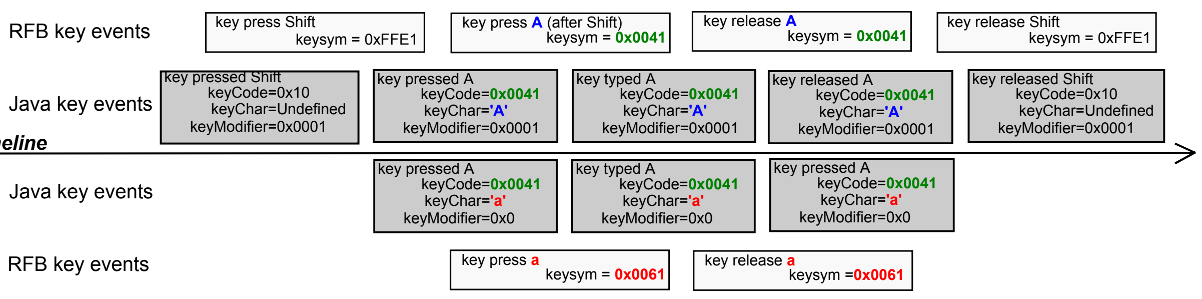

RFB key events
key press Shift keysym = 0xFFE1
key press A (after Shift) keysym = 0x0041
key release A keysym = 0x0041
key release Shift keysym = 0xFFE1
Java key events
key pressed Shift keyCode=0x10 keyChar=Undefined keyModifier=0x0001
key pressed A keyCode=0x0041 keyChar='A' keyModifier=0x0001
key typed A keyCode=0x0041 keyChar='A' keyModifier=0x0001
key released A keyCode=0x0041 keyChar='A' keyModifier=0x0001
key released Shift keyCode=0x10 keyChar=Undefined keyModifier=0x0001
timeline
Java key events
key pressed A keyCode=0x0041 keyChar='a' keyModifier=0x0
key typed A keyCode=0x0041 keyChar='a' keyModifier=0x0
key pressed A keyCode=0x0041 keyChar='a' keyModifier=0x0
RFB key events
key press a keysym = 0x0061
key release a keysym =0x0061

Figure 4: The difference between a Latin capital letter 'A' and a Latin small letter 'a'

```
java.awt.event.KeyEvent[KEY_RELEASED, keyCode=65,
      keyText=A, keyChar='a', keyLocation=KEY_LOCATION_STANDARD,
      rawCode=65, primaryLevelUnicode=97, scancode=30] on frame0
```

For the key-pressed and key-released events the values of *keyCode* and *keyChar* are sufficient to determine which key was pressed or released. The things are getting even simpler with the key-typed event – it only needs a valid *keyChar*.

It is quite interesting how the capital letters are interpreted. They have the same *keyCode* as the small letters, only the *keyChar* is different. Another difference is the registration of the pressed *Shift* as a modifier (*keyModifier*=**0x1**). The other modifiers are similarly registered – the left *Alt* key results in *keyModifier=**0x8***, the left *Control* in *keyModifier=**0x2***, and the combination *Shift+Control* in *keyModifier=**0x3*** (***0x1+0x2***).

All these differences make clear, that the key events cannot be consumed from the underlying Java application as they are received from the RFB server. The translation function implemented in VNCj can be described as follows:

Starting with a *keysym* and *down-flag*, first check if the key is a modifier. If the key is a modifier and if it is pressed, increment the *keyModifier* with its value, if it is a modifier and is released – decrease the *keyModifier*. Translate the *keysym* into a *keyCode*.[24] Finally, if the *down-flag* is non-zero, i.e. the key is being pressed, generate a key-pressed event:

```
new KeyEvent( container, KeyEvent.KEY_PRESSED,
      System.currentTimeMillis(), keyModifier,
      keyCode, keyChar, KeyEvent.KEY_LOCATION_UNKNOWN );
```

If the *down-flag* is zero, generate a key-released event and a key-typed event only for character keys.

```
new KeyEvent( container, KeyEvent.KEY_RELEASED,
      System.currentTimeMillis(), keyModifier,
      keyCode, keyChar,    KeyEvent.KEY_LOCATION_UNKNOWN );
```

[24]For a list of the virtual keys see the Java KeyEvent Documentation `http://download-llnw.oracle.com/javase/6/docs/api/java/awt/event/KeyEvent.html`

```
new KeyEvent( container , KeyEvent.KEY_TYPED,
     System.currentTimeMillis() , keyModifier ,
     0x0 , keyChar , KeyEvent.KEY_LOCATION_UNKNOWN ) ;
```

The keyboard event translation was one of the main areas for improvement. For instance, the event translation in the original version assumed that a character key only generates a key-typed event, whereas the modifiers themselves fire no events at all.

3.2.2 Mouse (Pointer) events

The next form of client to server messages are the pointer events. The RFB Protocol distinguishes between two types of pointer events – *pointer movement* and *pointer button press or release*. *Figure 5* shows how a single pointer event message looks like. It is 6 bytes long and the value of the first byte, its *message-type*, equals *5*. The next byte holds information about which button was pressed or released. The mapping between the single bits and the respective mouse buttons is also shown in *Figure 5*. Special attention requires the interpretation of the mouse wheel – every pressed-released sequence of the fourth/fifth mouse button results in a single upwards/downwards step of the mouse wheel. The last two message bytes are holding the *current* position of the pointer.

By contrast, Java distinguishes between three different *pointer events*:

1. *Mouse events* – the cursor enters/exits component's onscreen area or a mouse button was pressed/released

2. *Mouse-motion events* – take notion of the onscreen cursor movement

3. *Mouse-wheel events* – wheel up/down

Every type of pointer event has its own Java interface – a subinteface of the *EventListener*. To track mouse events, the MouseListener should be registered, for the mouse-motion events – the *MouseMotionListener*, and for the mouse-wheel events – the *MouseWheelListener*. The first two are also combined in the *MouseInputListener*.

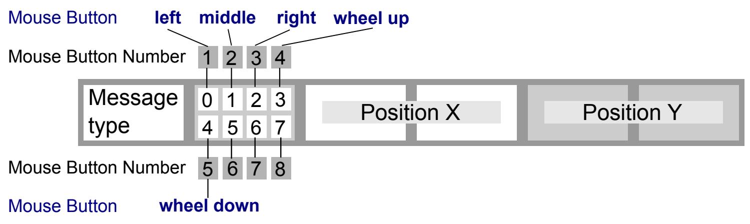


Figure 5: PointerEvent message format

MouseEvent	**PointerEvent or Pointer Event combination**
mouse moved	pointer movement
mouse entered	t $_{-1}$: pointer was outside component t $_0$: pointer is inside component
mouse exited	t $_{-1}$: pointer was inside component t $_0$: pointer is outside component
mouse pressed	button 1 - 3 press
mouse released	button 1 - 3 release
mouse clicked	t $_{-1}$: button 1 - 3 press t $_0$: button 1 - 3 release
mouse dragged	t $_0$: button press at position (x $_0$, y $_0$) t $_1$: pointer movement - position (x $_1$, y $_1$) ... t $_n$: button release at position (x $_n$, y $_n$)
mouse wheel	button 4 / 5 press button 4 / 5 release

Table 1: Relation Mouse-Pointer events

The table above displays the dependance between the *PointerEvents* as received from the RFB server and the *MouseEvents* as expected from the underlying Java application.

An invisible fact from the above table is that the key modifiers also affect the way a mouse click is interpreted. A key modifier reflects in changing the value of the modifiers variable of every pointer event.

The need of pointer event translation is obvious. It resides in the same module as the key-event-translation, namely in the *gnu.vnc.awt.VNCEvents* and can be basically outlined as follows:

Starting with a *buttonMask* and *x-* and *y-position*, determine if and which button was *pressed* and save it as a modifier in the variable *mouseModifiers*. An initial state of zero indicates that no button is down at the time. The press of the fourth or fifth button indicates a mouse-wheel rotation. In this case fire a *mouse-wheel* event:

```
new MouseWheelEvent(component, MouseEvent.MOUSE_WHEEL,
    System.currentTimeMillis(), keyModifiers | mouseModifiers,
    x, y, clicks, false,
    MouseWheelEvent.WHEEL_UNIT_SCROLL, 3, rotationDirection );
```

If the last state of the variable *mouseModifiers* is unchanged and if it is non-zero, i.e. a mouse button stays pressed, then it is a *dragging*:

```
new MouseEvent( component, MouseEvent.MOUSE_DRAGGED,
    System.currentTimeMillis(), keyModifiers | mouseModifiers,
    x, y, 0, false );
```

else fire a *mouse-moved* event:

```
new MouseEvent( component, MouseEvent.MOUSE_MOVED,
    System.currentTimeMillis(), keyModifiers | mouseModifiers,
    x, y, 0, false );
```

If the value of *mouseModifiers* has changed, it indicates a *mouse-pressed* plus *mouse-clicked* event or a *mouse-released* event. The difference makes the *pressed* state. If *pressed* is true:

```
new MouseEvent( component, MouseEvent.MOUSE_PRESSED,
    System.currentTimeMillis(), keyModifiers | mouseModifiers,
    x, y, numClicks, false );

new MouseEvent( component, MouseEvent.MOUSE_CLICKED,
    System.currentTimeMillis(), keyModifiers | mouseModifiers,
    x, y, numClicks, false );
```

otherwise:

```
new MouseEvent( component, MouseEvent.MOUSE_RELEASED,
    System.currentTimeMillis(), keyModifiers | mouseModifiers,
    x, y, 0, false );
```

And lastly, when the current component is not the same as from the last time period, fire a *mouse-exited* and *mouse-entered* events:

```
new MouseEvent( component, MouseEvent.MOUSE_EXITED,
    System.currentTimeMillis(), keyModifiers | mouseModifiers,
    oldx, oldy, 0, false );

new MouseEvent( component, MouseEvent.MOUSE_ENTERED,
    System.currentTimeMillis(), keyModifiers | mouseModifiers,
    x, y, 0, false );
```

To the changes in this area the adding of the mouse-wheel events and some other fine adjustments count.

3.3 Framebuffer update requests and framebuffer updates

Another part of the RFB client-server communication are the *framebuffer update requests* and the *framebuffer updates*. A *framebuffer update request* is a 10 byte long message consisting of *message-type*, *incremental flag*, *the position* of the upper left corner of the region of interest and its *width* and *length*. The incremental flag tells the server, if the client needs only an incremental update of the pixel data in the region or the pixel data itself.

The *framebuffer update* is not so trivial. It consists of a sequence of rectangles of pixel area encoded in some way. It starts with a header message, holding the number of the rectangles followed by a rectangle array in a specific from the RFB protocol defined encoding.[25]

[25] See Richardson [2009] for complete list of the supported encodings

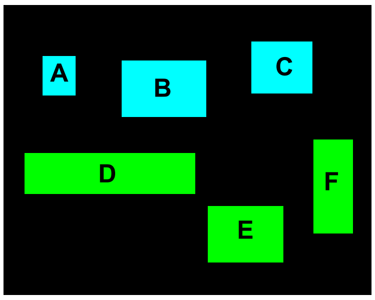
The framebuffer update, especially the encodings, constitutes the region, where the difference between RFB 3.3 and RFB 3.8 gets very clear. RFB 3.3 defines five encoding types and RFB 3.8 seven. RFB 3.8 contains an extra list of registered encoding, whose use is also permitted.

As already mentioned, the current version of VNCj is built upon the version 3.3 of the protocol. In the following a brief overview of the five from RFB 3.3 defined encodings will be presented. These are *Raw*, *CopyRect*, *RRE*, *CoRRE* and *Hextile*.

Raw is the simplest encoding and also the default one. Every server should be able to send raw data, and every client should accept it. Raw means that the server sends every pixel value from the rectangle area in left-to-right scan-line order. Therefore, the size of the update depends on the size of the rectangle and on the color depth.

CopyRect is possibly the most network efficient encoding. It consists of 4 bytes, holding the position of the framebuffer region, wherefrom the client can copy the pixel data. It can be used for example by text typing or by redrawing regions affected by mouse scrolling.

RRE stands for *rise-and-run-length-encoding*. Its layout can be easily explained using the very basic example from the figure below:



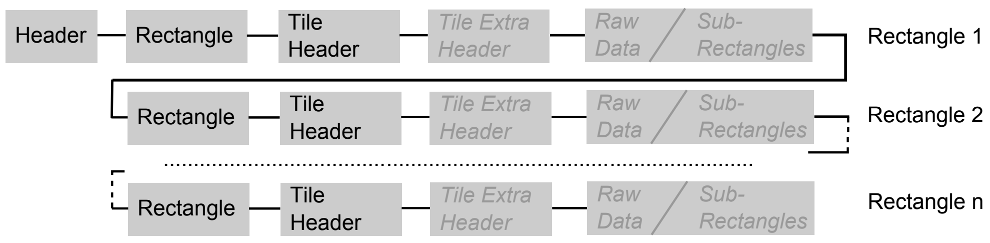
The main region is subdivided in subregions. The first protocol message holds the number of the subrectangles in the region, in this case 6, and its background color value, black. This message header is followed by 6 consequent messages holding the color, the position and the dimensions of each subrectangle from A to F.

CoRRE is a minor variation of RRE, which limits the size of the largest rectangle to 255x255 pixels. This allows using a single byte for the dimensions of the subrectangles.

„Hextile is a variation on the RRE idea. The rectangles are split up into 16x16 tiles, allowing the dimensions of the subrectangles to be specified in 4 bits each, 16 bits in total.“[26] The structure of framebuffer update using only the hextile encoding can be synthesized as follows:

The *Tile Header* is a one byte field. Every *Tile Header* bit indicates if property of the tile is set. The most significant bit is the first from right to left, namely the *Raw* bit.

[26]Richardson [2009]



If the *Raw* bit is non-zero, the other bits are neglected, the *Tile Extra Header* is not set, and the raw pixel data follows.

The next bit is the *BackgroundSpecified* bit. If it is set, the first part of the *Tile Extra Header* is a bits-per-pixel long field which holds the background color. If it is not set, the current tile gets the last defined background color.

If the next switch, *ForegroundSpecified*, is set, then the last one, *SubrectColoured*, must be zero. The subsequent part of the *Tile Extra Header* is then the foreground color for every subrectangle in the tile.

AnySubrects shows if there are subrectangles at all; if it is not set, neither the *Raw Data* nor the *SubRectangles* data follows. If it is set, a number-of-subrectangles value is put into the *Tile Extra Header*, and a number of subrectangles follows it.

If the last *Tile Header* bit is non-zero, then each subrectangle should define its own foreground color. A single subrectangle field non specifying its own foreground color is 2 bytes long; by declaring a foreground color the size of the field grows by the color depth value.

A very detailed explanation of the framebuffer update implementation would make this chapter unnecessary large. For this reason, only a brief overview follows. The text below treats mainly the Swing part of VNCj.

To be able to send the draw data over the wire, a custom *RepaintManager* must be implemented which resides in *gnu.vnc.awt.swing.VNCRepaintManager*. This manager holds a queue of all clients, declares regions as dirty (i.e. needing a repaint), invalidates components and executes the framebuffer updates. The queue management and the further proceeding of the encoding are done in the *gnu.vnc.VNCQueue*. The namespace, where the encodings reside, is *gnu.rfb*. The part translating the encoding requests to the according encoding class is *gnu.rfb.Rect*. At the time only the RFB 3.3 encodings are implemented. The structure of the library indeed, leaves the door for future encodings wide open.[27] [28]

[27]See http://www.couchpotato.net/vcm-source/ for example implementation of the ZRLE encoding
[28]This is also the part of VNCj where almost no changes were made during this work.

3.4 A VNC server for Dioscuri

The last part of the implementation chapter is the building of the internal Dioscuri's VNC server itself. As visualized in the last subchapter, VNCj already implements all the mechanisms and functions of a working VNC server. The creation of an integrated VNC server reduces to the integration of the library into the Dioscuri project.

The first question about the integrated server is, where to position it in the class structure of Dioscuri. It is not important what such a server is, but what it is not. It is not a hardware module, therefore it will be by all means false putting him among the hardware modules. It is also not a GUI, even though they are very similar in some particular aspects. It is an interface, allowing the export of a GUI or a part of it over the network. Therefore, it should be either positioned in the namespace of the GUI itself or in its own namespace. The second alternative makes the later extension of the VNC handier and by some means it is the better one.

The next question is about the configuration options. The emulator's configuration menu resides in the main menu under „configure ⇒ edit config“. The configuration classes itself are all put into the *dioscuri.config* namespace. Adding the *VncPanel* module to the same namespace contradicts a little with the initial idea to keep the VNC apart from the existing module structure of the emulator. Besides, all the Dioscuri's configuration resides there. Another fact in favor of this alternative is the usage of the already existing configuration mechanisms, resulting in an enormous time saving during the implementation process.

The redirection of the emulator's in- and output was already outlined at the beginning of this chapter. The first step is to create an extra class in the *dioscuri.vnc* namespace, namely the *VNCTopFrame*, which extends the already existing *gnu.vnc.awt.swing.VNCJFrame*.

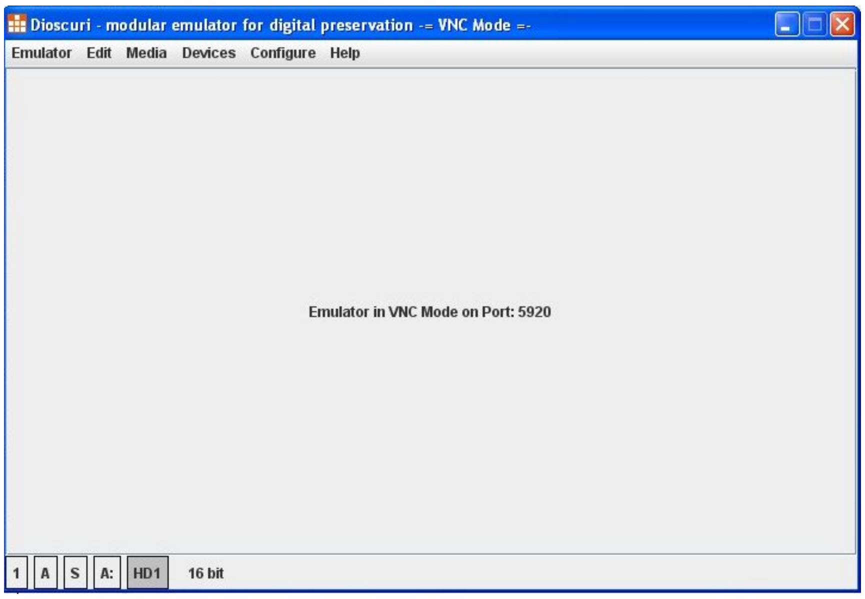
To start and stop the server, two new methods were added to the top level *DioscuriFrame*: *startVncServer* and *stopVncServer*. The first method reads the VNC configuration parameters, starts a new *RFBHost*[29], moves the *DioscuriFrame.screenPane* to the created *VncTopFrame* instance and attaches the emulator's *KeyListener* and *MouseInputListener* to the *VNCTopFrame*'s *GlassPane*. The reason for the use of *GlassPane* needs a short explanation.

Usually all the key and pointer events are delegated to the last component in the GUI's hierarchy. This makes the control of what happens slightly complicated. Adding a *GlassPane* to the *VNCTopFrame* breaks the event propagation and eases the event control. To correctly redirect the mouse and keyboard events to the underlying emulation module, the *GlassPane* should be the same size as the emulation *screen* and both must overlap perfectly.

The last paragraph gives an idea of when the server should be started. It makes sense to move the *screen* as soon as its dimensions are defined. This happens normally on emulation start. This is why *startVncServer* is executed when the emulation starts and *stopVncServer* when the emulation stops.

[29]The RFBHost constructor creates a VNCTopFrame instance

A message appears in the emulation screen area of the emulator showing that Dioscuri is running in VNC mode on a specific port.


Dioscuri - modular emulator for digital preservation -= VNC Mode =-
Emulator
Edit
Media
Devices
Configure
Help
1
A
S
A:
HD1
16 bit

Emulator in VNC Mode on Port: 5920

Figure 6: Dioscuri running in VNC mode

4 Testing

The main idea behind the testing is to demonstrate that the implemented internal VNC server for Dioscuri is a fully functional VNC server. The accent is put on the same modules as in the implementation chapter: the keyboard and pointer events, and the framebuffer updates.

The potential extra CPU or memory overload caused from the server constructs another test area. It is to be shown that these overloads are not so immense and do not impact the functionality of the underlying emulation module.

The tests were made under Windows XP SP3 using the TightVNC viewer[30] and the test scenario looks as follows:

I: Run Dioscuri in **non VNC mode**

1. Start Dioscuri on node **A**
2. Test the keyboard and the mouse
3. Take screenshots
4. Take notion of CPU and memory usage

II: Run Dioscuri in **VNC mode**

1. Start Dioscuri on node **A**
2. Connect to the VNC server from the same node
3. Test the keyboard and the mouse
4. Take screenshots
5. Take notion of CPU and memory usage on node **A**

6. Connect to the VNC server from node **B**
7. Test the keyboard and the mouse
8. Take screenshots
9. Take notion of CPU and memory usage on node **A**

10. Connect simultaneously from node **C** and repeat steps 7, 8 and 9
11. Compare the display of the RFB viewer on node B with the output on node **A** and **C**

III: Compare the results from phase I and II.

By the mouse and keyboard testing the Dioscuri debug messages were very useful. The very slight disparity in the debug messages is a clean proof that the mouse and keyboard events are correctly transported through the VNC layer.

[30]TightVNC is a part of the TightVNC package, which is a free software. TightVNC Viewer is a Java application, and therefore extremely portable. See TightVNC [2010] for more information.

A proof that the framebuffer update requests and the framebuffer updates reach the other side is the update of the TightVNC viewer's display.

The test results are as follows:

- The VNC server runs on the specified port
- RFB clients can access the server, using its IP address and the specified port
- Multiple connections are possible
- Mouse and keyboard events are transported correctly, i.e. they are not affected from the VNC layer
- Framebuffer update requests reach the server
- Framebuffer updates reach the client
- The CPU and memory overload is not so immense as feared

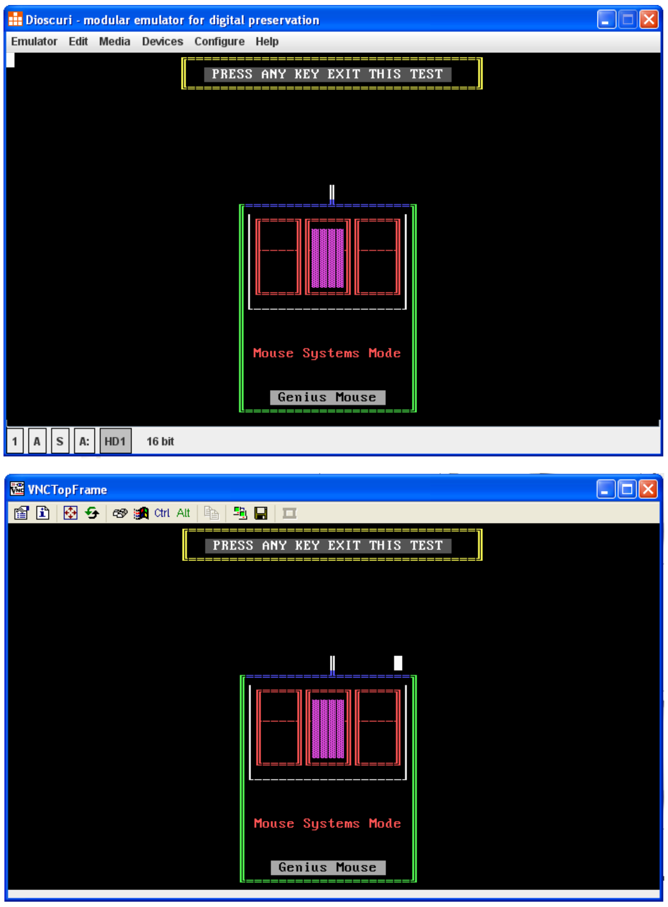

Dioscuri - modular emulator for digital preservation
Emulator Edit Media Devices Configure Help
PRESS ANY KEY EXIT THIS TEST
Mouse Systems Mode
Genius Mouse
1 A S A: HD1 16 bit
VNCTopFrame
Ctrl Alt
PRESS ANY KEY EXIT THIS TEST
Mouse Systems Mode
Genius Mouse

Figure 7: Mouse testing in normal (top) and in VNC mode (bottom)

5 Conclusion

The goal of this work is the implementation of an integrated VNC-interface for the Java x86-emulator Dioscuri.

In the first part of the current thesis a research over the state of the involved software is made. The aim is to outline the single development steps. This results in an additional attempt of clarifying terms like RFB and VNC and in an explanation of the function of a VNC server.

The second part, the implementation, consists of two phases. The first one is the preparation of the VNCj library for the actual development of the VNC interface. It also continues the attempt from the first part and explains some important terms like events and framebuffer update. The second phase is the integration of VNCj in Dioscuri. This is the part where the achievement of the main goal is presented, namely the development of a fully functional VNC server for Dioscuri.

Despite the positive test results in the last chapter, some further improvements in different areas are possible. The first one is the implementation of the newest RFB 3.8 protocol. This will result in slight changes in some modules – authentication and framebuffer updates for example.

Because of the scope of this work, only parts of the VNCj library were enhanced. A lot of future work is to be put in this direction too.

Very useful, currently not implemented feature is the GUI-less start of the emulator in VNC mode. This arises a need of a GUI management console to facilitate the usability of the emulator for the purposes of the long time preservation of digital objects.

Literature

[Dioscuri 2010] DIOSCURI: *Dioscuri - the modular emulator*. Online, *http://dioscuri.sourceforge.net*. 2010. – URL `http://dioscuri.sourceforge.net`

[van der Hoeven 2007] HOEVEN, J.R. van der: *Dioscuri's Object Design Document*. Online, *http://dioscuri.sourceforge.net/docs/ODD_Dioscuri_KBNA_v1_1_en.pdf*. 2007

[Liron 2002] LIRON, Tal: *VNCj*. Online, *http://emblemparade.net/projects/vncj*. 2002. – URL `http://emblemparade.net/projects/vncj`. – Source code: http://sourceforge.net/projects/vncjlgpl

[Philipps 2010] PHILIPPS, Mario: *Entwurf und Implementierung eines Softwarearchivs für die digitale Langzeitarchivierung*, Albert-Ludwigs-Universität Freiburg, Diplomarbeit, July 2010. – URL `http://eprints.rclis.org/18994/`

[PLANETS 2010] PLANETS: *PLANETS - Digital Preservation Research and Technology*. Online, *http://www.planets-project.eu*. 2010. – URL `http://www.planets-project.eu`

[QEMU 2010] QEMU: *QEMU - open source processor emulator*. Online, *http://wiki.qemu.org*. 2010. – URL `http://wiki.qemu.org`

[RealVNC 2010] REALVNC: *RealVNC*. Online, *http://www.realvnc.com*. 2010. – URL `http://www.realvnc.com`

[Richardson 2009] RICHARDSON, Tristan: *The RFB Protocol*. Online, *http://www.realvnc.com/docs/rfbproto.pdf*. 2009

[Rothenberg 2000] ROTHENBERG, Jeff: *Using Emulation to Preserve Digital Documents*. Online, *http://www.kb.nl/pr/publ/usingemulation.pdf*. 2000

[TightVNC 2010] TIGHTVNC: *TightVNC Software - Free, Lightweight, Fast and Reliable Remote Control Software*. Online, *http://www.tightvnc.com*. 2010. – URL `http://www.tightvnc.com`