Diploma
DOpE - a graphical user interface for DROPS

Norman Feske
nf2@inf.tu-dresden.de
Dresden University of Technology
Operating Systems Group

17th October 2002

Contents

List of Figures

1 Introduction

DROPS is a research project of the 'Operating Systems Group' at the Dresden University of Technology . It is an acronym for 'The Dresden Real-Time Operating System Project'. The research of the 'Operating Systems Group' aims mainly at improvements of current micro-kernel technologies in connection with 'Quality of Service' requirements.

One of the biggest efforts is the port of Linux to the DROPS platform. The so called L^4*Linux* runs as user-level application on the top of the DROPS kernel - namely Fiasco (an implementation of the L^4 kernel API).

Up to now, the only user interface of DROPS is a console application ('Dropscon'-server). It provides a variety of features to comfortably interact with DROPS applications. Every application can occupy one or more 'virtual consoles'. While only one 'virtual console' is visible at a time, the user can choose the currently visible 'virtual console' by using keyboard shortcuts.

While 'Dropscon' has sufficient functionality for the comfortable handling of a number of applications, it can not be called intuitively usable. A lot of today's commonly used applications require more advanced user interactions. Today's expectations of a user interface raise beyond pure functionality - ergonomic aspects and visual issues tend to become very important when attracting computer users. Every mainstream desktop operating system features windowed graphical user interfaces - even on portable devices windowed user interfaces gain more and more popularity. Consequently, a windowed user interface featuring the today's standards (and even more) would increase the application area of DROPS dramatically.

The aim of this work is to create such a user interface while keeping the real-time facilities of DROPS available at the user interface level.

Due to the complexity of this task, not all of its facets can be discussed within this document. Therefore, the document is focused only at the most interesting aspects of the windowing system, which are namely: its real-time capability and its command interface.

1.1 Acknowlegdements

A lot of thanks have to go to Christian Helmuth for being a great tutor and Frank Mehnert for his excellent support. Furthermore, the author of this document wants to thank the following persons: Christin Geyer (for her technical and moral support), Prof. Dr. Hermann Härtig (for showing that great interest in *DOpE* and for his valuable hints), Leon O'Reilly (for creating God-

Pey and for checking my spelling), Michael Hohmuth (for the introduction to the OS Group at the TU-Dresden and for maintaining the MiNT kernel in his prior life), Lars Reuther, Adam Lackorzynski and the other members of the OS Group at the TU-Dresden.

1.2 Declaration

I declare that all parts of this work were autonomously written by me while using only legal resources. All resources, that were used within this work are explicitely announced. To the best of my knowledge the content of this work is original and was not published before by me or another author.

Norman Feske, Dresden, 02-09-26

2 Basics

This section is meant to show up the background and motivation behind the work on the windowing system *DOpE* ('Desktop Operating Environment'). Furthermore, it substantiates the given task of creating a windowing system for DROPS.

Firstly, some terminology is introduced. Later sections of this document will frequently refer to the term 'real-time'. Section 2.1 will clear up its meaning in the context of this work.

The section 2.2 will put this work into the context of current developments at the 'Operation Systems Group' at the Dresden University of Technology. It is followed by a section, presenting the living space of *DOpE* - namely DROPS and points out its features and restrictions from *DOpE*'s point of view.

Afterwards the position of this work in the range of existing solutions in the field of graphical user interfaces is defined.

2.1 What is 'real-time'?

The term 'real-time' plays a very central role within this work. So it is important to define this term in the context of a graphical user interface.

The two tasks for a graphical user interface are:

- the handling of user input events (communication from the user to the application)
- the graphical representation of applications (communication from the application to the user)

The term 'real-time' is focused at the second point.

User input events, such as mouse movements or key press events, should be handled as 'fast as necessary' - what does that mean? The user input handling is very closely related to ergonomic issues. So 'fast as necessary' means that response times of the user interface should allow comfortable interactions between the user and an application.

The second point (the graphical representation of applications) is far more complicated. Firstly, because graphical output consumes a lot of processing time, and secondly, because there can be multiple applications, that intend to communicate to the user at the same time.

To face the term 'real-time', it makes sense to divide the application's needs into three groups:

- dialogs: the application waits for user-events and reacts to the user's actions. This is the case for the majority of classic applications, such as text editors, terminals etc.
- continuous data output: A screen area must be updated at a fixed frame-rate to provide a fluent output. This is typical for many kinds of graphical multimedia applications, such as movie players.
- sudden data output: a graphical output must be made visible abruptly (e.g. a very important error message) without an indeterministic delay.

'Dialogs' can be seen as counterpart to 'user input events'. There are no well defined constraints how 'dialogs' should behave as long as ergonomic aspects are met. Within this work the term 'real-time' is focused on 'continuous data output' at a fixed update rate. Anyhow, a way to provide the ability of 'sudden data output' is presented in section 3.5.11.

An application can request a continuous update of a certain screen area ('widget') at a fixed update rate. When the user interface accepts the request, it ensures that the widget is 'redrawn' at the specified update rate.

2.2 Related work

During the introduction section the console application of DROPS - namely 'Dropscon' - was introduced. This section provides some additional information about 'Dropscon' and associated applications. The long-term objective of *DOpE* is to replace 'Dropscon' completely. This can only succeed if *DOpE* offers at least equal functionality:

'Dropscon' provides a small set of functions, that can be used by client applications to draw graphical primitives onto a virtual console screen. This set of functions (also called *pSLIM*-protocol) consists of:

- 'fill': draws a filled rectangular area using a specified color
- 'copy': copies a specified rectangular area to another specified screen position
- 'bmap': paints an 1bit image (bitmap) to a specified position on screen
- 'set': paints an image (with the same color depth as the console) to a specified position on screen
- 'cscs': paints an YUV-coded image to a specified position on screen

The *pSLIM*-protocol was slightly enhanced by two functions to speed up textual output:

- 'puts': paints a given string using specified fore- and background color
- 'puts_attr': paints a given string, that features implicit color information per character

All currently existent 'Dropscon' applications make use of the *pSLIM*-protocol to implement their graphical and textual outputs. If a windowing system provided the *pSLIM*-interface all console applications would be able to run happily under this windowed environment - including L^4*Linux*.

2.3 Demands and Constraints

The primary platform of *DOpE* will be the DROPS operating system. This section will show up the implications of this fact for designing *DOpE*.

As stated in section 2.1 the main tasks of a user interface are the handling of graphical output and user input.

The base of the graphical output is described by the GRUB-bootloader, which provides information about the currently selected VESA screen mode including color-depth, screen size and the base address of the frame buffer. So the desired screen mode must be set up via GRUB while booting DROPS. All graphical output is done by writing data to the physical frame buffer of the graphics card.

For the handling of user input devices, a port of the Linux device driver concept is used. The L^4-port of the device driver concept comes in the shape of a library. The advantages of this input-library are the wide range of supported input device hardware and a well designed generic interface to the application.

One of the most important features of DROPS is its real-time capability. Consequently, this feature should be provided at the user interface level. This is the reason of the big role of term 'real-time' within this work (see section 2.1).

The usage of DROPS implies some restrictions affecting the design of the windowing system:

- no UNIX infrastructure: the comfortable UNIX infrastructure is not available under DROPS. This encloses file-i/o, devices, processes, standard libraries, dynamic loading etc. As shown in section 2.4 this fact complicates the port of existing solutions, which are using a lot of the UNIX infrastructure.

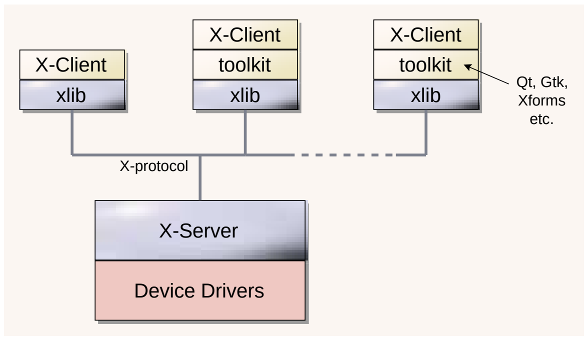


Figure 1: structure of X-windows

- no file-system: Although there exists a simple file provider on DROPS the windowing system should not depend on the existence of a file system. Thus, *DOpE* should allow to link all image and font data to the main program. Optionally, it should be possible to use a file provider to load the desired data in runtime.

2.4 Review of existing solutions

Before starting a project from scratch it is worthwhile to take a look at existing approaches and solutions.

If there already exists a perfect solution for the problem, the solution could be ported to the final platform (L^4). Of course, this is only possible for projects with freely available source-codes and a suitable license. Otherwise it is still possible to learn from existing solutions.

This section reviews existent windowing systems and rates their usability as graphical user interface for DROPS.

2.4.1 X-windows

An early idea was to port X-windows to L^4. The obvious reason is the big amount of applications, that are available for X-windows.

X-window's popularity gains from a great feature: its network transparency allows to run clients on a remote computer while displaying the client's user interface on the local machine. Figure 1 shows the general structure of X-windows. Technically, a X-windows terminal provides a comprehensive set of drawing primitives via the X-protocol. A remotely running program can draw its output by sending the corresponding X-protocol commands and parameters (via xlib) to the displaying terminal. The X-server has a passive nature and has no knowledge about the 'meaning' of the displayed data. It just executes graphical operations.

Common applications use additional 'toolkits' (such as Qt, Gtk, GnuStep, Tcl/Tk or XForms) as an abstraction layer between the application and 'xlib'. While 'xlib' provides only functions to draw graphical primitives a 'toolkit' provides the management of more abstract user interface elements (buttons, frames, pop-ups etc.). So the look, feel and functionality of X-windows applications highly depend on the used toolkits. Due to the number of available 'toolkits' there is no established common look and feel among X-windows applications.

When it comes to the porting of X-windows to DROPS some heavy weighted problems appear:

- source-code complexity: It would be much more work to dive into the depths of an X-server such as XFree86, than to create a completely new windowing system. The central point of this work is to create a concept for a user interface, which reflects the real-time capabilities of the underlying OS. It would be a very hard and doubtful work to modify an existing X-server to meet the needs of a real-time concept.

- missing infrastructures on L^4: without a real file-system it makes only limited sense to port X-windows/XFree. It uses the file system for a lot of things, such as config files, fonts, pixmaps etc. Beside the file-system X-windows also needs other parts of a UNIX-infrastructure, that is not available under L^4.

- The main design criterion of X-windows is its network transparency - this plays only a minor role for our needs.

There are real-time-solutions which use X-servers. One example is IRIX, the operating system of Silicon Graphics. The IRIX-kernel features real-time facilities. As graphical user interface a X-server is used. Due to the fact that IRIX is closed-source a port to L^4 is impossible.

2.4.2 EmbeddedQt

The argument for EmbeddedQt is very similar to the argument for X-windows: There already exists a wide range of applications for Qt. Qt-applications can be ported to EmbeddedQt just by recompiling. EmbeddedQt is designed to be used on portable devices running Linux.

There are some arguments against the usage of EmbeddedQt:

- It requires a Linux-infrastructure.
- Not real-time capable: Real-time aspects had to be implemented into an already existing concept. It would be much easier to address real-time issues, when freely designing a new concept.
- source code complexity: Quoted from Trolltech website: "Qt offers the functionality you need to create professional applications. There are around 250 C++ classes in the Qt API. Most of these classes are GUI specific; however, Qt also provides template-based collections, serialization, file and a general I/O device, directory management, date/time classes, regular expression parsing and more."
- It is a commercial product. When developing a commercial application using EmbeddedQt license fees must be paid to Trolltech.
- Not secure: The frame-buffer is not handled by a server.

Even so, there exists already a port of Qt 2.3 for L^4 by Simon Kagstrom. This port does not include the implementation of real-time facilities into Qt. It aims rather at getting existing Qt applications to run under L^4 than at creating a graphical environment for real-time applications.

2.4.3 GtkFB

As for Qt there exists an embedded version of Gtk. Sadly the drawbacks of both embedded solutions are similar. Additionally, GtkFB runs only one application at a time, which is not sufficient for our needs.

2.4.4 Microwindows

Microwindows is a small windowing system, running under DOS or Linux. Sadly, the windowing system gets really slow with just a few windows. While testing Microwindows it tended to produce redraw errors or even crashed. Its speed makes multimedia applications unviable.

2.4.5 QNX

QNX is a closed source real-time operating system featuring a windowed user interface called 'Photon'. It deals with nearly the same problems as a windowing system under L^4. Sadly, the available documentation does not cover how 'Photon' solves the real-time demands. Due to the fact that QNX deals with the same problems as we do, its widget set is a good orientation for the needs of a graphical user interface in general.

2.4.6 Artifact

Artifact is an experimental real-time windowing system. This windowing system solves the real-time demands in a way that is discussed in more depth in the section 3.5.2. Artifact does not provide a complete user interface concept, which would make it usable in 'real life'.

2.4.7 Consequence

Most of the existing solutions lack either of platform independence or simplicity. Implementing real-time facilities into an existing project can be a difficult task, because it is a fundamental design criterion.

Since the author of this work had already some experiences in programming user interfaces on small platforms, it was easier to start a new project from scratch than digging in existing source codes. This also gave the chance to give DROPS a unique user interface.

3 Concept

3.1 Conceptual design criteria

3.1.1 Real-time capability

The most interesting aspect of the DROPS operating system is its real-time capability. Thus, this aspect need to be constantly kept in mind during the design process. A description of the term ’real-time’ can be found in the section 2.1. At the end, the user interface shall reflect the advantages of the real-time abilities of DROPS for the end user (e.g. when playing an animation at a constant frame rate).

3.1.2 Small interfaces

Interfaces are mostly designed with a concrete application in mind. While these interfaces work perfectly with such kind of applications, they show their weakness in future use. A good example is the X-protocol. It consists of a number of graphical primitives to draw user interfaces. When this protocol was defined, nobody imagined that hardware could support 3D acceleration features, alpha transparency etc. So the protocol was not sufficient for modern needs anymore and had to be expanded.

Another approach is the definition of generic protocols without predefined limitations. A good example is Tcl/Tk, where the script language Tcl can be considered as ’protocol’. It has a very clean and generic structure and does not limit itself to a set of predefined functions. The way in which commands can easily be added, ensures ’future compatibility’.

So the aim is to keep interfaces (especially between client and server) as small and generic as possible.

Another important reason to keep interfaces small is to verify its functions. In contrast, a highly complex protocol is difficult to test thoroughly.

3.1.3 Expandability

At the time of software planning the developer is hardly be aware of all the needs that could pop up in the future. Expandability is the keyword to give the new ’baby’ a shiny future. This point is very closely related to the previous one (section 3.1.2). Only generic interfaces can handle slight changes of the application field. The whole application should be based on a modular concept while allowing functionality to expand by adding new modules transparently.

Practically, expanding functionality often means the expansion or introduction of communication protocols.

All this must be possible without harming the relationship of the user interface to its clients (keeping up compatibility).

3.1.4 Security

As denoted in [9] security requirements can aim at three aspects:

- confidentiality: This aspect comes into play when running user interfaces via network (like X-windows). Since this is not the primary aim of this work, confidentiality plays no major role here. When considering the use of this concept via network, encryption mechanisms should be implemented at RPC-level.
- availability: While speaking of graphical user interfaces, availability means accessibility. The concept has to ensure the users ability to control the applications whenever he wants. Depending on the used applications, this aspect concerns ergonomy or security. Of course, accessibility can not be guaranteed for every single application. But the user interface can make sure that no application can harm the accessibility of any other running application.
- integrity: This is surely the most interesting security aspect. As stated in [8] thrust-worthy user interfaces have to assure the perfectly identification of the user's 'communication partner' - meaning the application he interacts with. Applications must not be able to draw on screen areas, that do not belong to the application. If this was possible, it would smooth the way for Trojan horses. As a consequence the access of clients to the physical frame-buffer must be absolutely restricted.

3.1.5 Ergonomic aspects

As indicated in the introduction of this document, ergonomic issues are arguments to attract potentially new users.

Generally it is no problem to include fancy and colorful user interface elements - but this is not meant with the term 'ergonomy'. A lot of today's graphical user interfaces offer exchangeable skins - but the buzzword 'look&feel' consists of two parts. While the look of other user interfaces can be perfectly emulated the 'emulation' lacks mostly of the corresponding 'feel'.

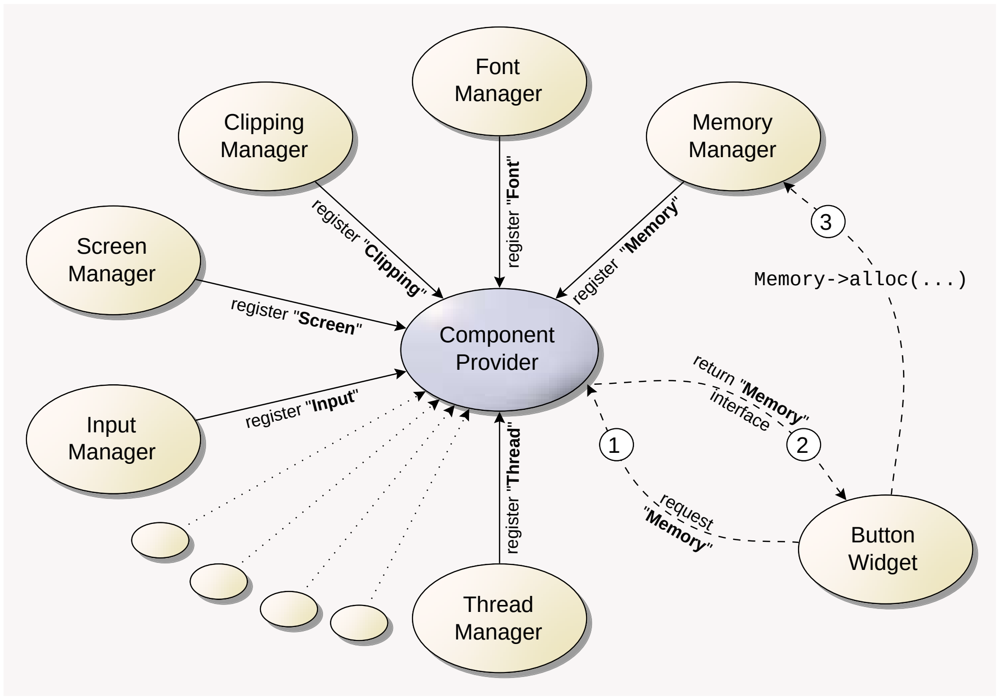


Figure 2: exemplary scenario of the component architecture

So the stated long term goal of *DOpE* is the offering of a unique 'feel' of the user interface while providing an appealing look.

3.2 General design

3.2.1 Component architecture

As expandability is an important design goal the windowing system is structured in a component based way. A component is a functional entity, that provides a well defined interface. Each component holds an unique identifier and a version number.

As shown in figure 2 the central role of this concept plays a 'component provider'. Components can register their existence at the 'component provider' and export their interfaces with their associated identifiers. In return - components can request interfaces of other components. Thereby, the 'component provider' knows exactly which component makes use of which other components. The aim of this construction is that components can easily be added to the whole system through a simple interface. Components could be exchanged even in runtime by other ones, that feature the same interfaces. Since all depencies between components are known by the 'component provider' it can force a reinitalisation if the affected components.

A simple version management shall avoid problems caused by changing interfaces. Each component provides its current version number and the version number of the earliest version with the same interface. When another component requests a certain version of this component, the 'component provider' can check the compatibility of the requested and available component, easily.

There are no limitations of what a component can be and what functionality it offers. Despite of that, the components of the windowing system can be categorised into three groups:

- 'Integral parts' are components for the handling of memory, threads, client-server communication, input- and output devices. They are fundamental for the overall work of the windowing system. Most of them are platform dependent because they implement hardware abstraction layers. Porting the windowing system to another platform means exchanging the platform dependent components while keeping their interfaces.
- 'Widgets' represent the basic elements of the user interface as described in the following section. The application of the windowing system appoints which widgets are needed. That means that the source code complexity depends on the desired functionality. While a minimalistic user interface for security applications requires only a very basic set of widgets, a windowing system for the usage on a desktop computer can feature much more complex widgets. The majority of widgets are platform independent. Anyway, it is possible to implement platform dependent widgets for taking over non-ordinary tasks for a special application.
- Helper functions such as font handling, graphical primitives, cache handling, hash table handling etc. are mostly used by widgets to keep themself small and handy. These components usually work 'on the top' of 'integral parts' and are implemented in a platform independent way.

A further advantage of the components architecture is the easy revisability of the overall system. Since each component has a well defined interface and its desired functionality it can be individually tested.

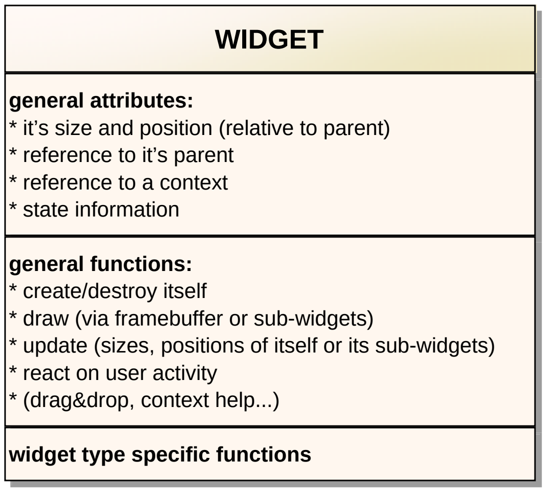

WIDGET
general attributes:
* it's size and position (relative to parent)
* reference to it's parent
* reference to a context
* state information
general functions:
* create/destroy itself
* draw (via framebuffer or sub-widgets)
* update (sizes, positions of itself or its sub-widgets)
* react on user activity
* (drag&drop, context help...)
widget type specific functions

Figure 3: general attributes and methods of widgets

3.2.2 Widgets

Widgets are the building bricks of which a user interface is made of. Each widget takes care of a certain rectangular region of the screen. Within this region there is a graphical representation of 'data' or 'protocols'.

Figure 3 provides an exemplary overview over the general attributes and methods of widgets.

The term 'data' refers to static elements ranging from simple labels (representing text strings) over buttons (representing states) to more complex objects, such as images. Even other widgets can be considered as data by 'layout'-widgets, which organise a number of 'child'-widgets within itselfs. All 'data'-widgets represent static information. When it comes to the representation of dynamic information 'protocol'-widgets come into play.

While a 'data'-widget handles an associated data type, a 'protocol'- widget handles an associated protocol providing a specific communication interface to the users application. In this way certain screen areas can be assigned to different protocols. Examples for such protocols are VT100, *pSLIM*, X-protocol, Glx or PostScript.

The option to transparently add new protocols or data types to the windowing system fulfills the design criterion 'expandability'.

A widget does not only take care about the graphical representation of its associated data or protocol - it also handles user interactivity. The way which a widget reacts to users activity is primary defined by the widget itself - not by the application. This ensures a consistent way of how widgets appear and behave across different applications.

The advantages of this approach are:

- There is a high degree of independence between the application and its user interface. Since the windowing system has the information of how a certain widget is represented on the screen it can handle its redraw without interacting with the application. In contrast to other windowing systems (e.g. X-windows) windows can be moved or resized without the help of the applications. Consequently, the communication bandwidth between the client and the window server is reduced to a minimum.

- The windowing system 'knows' the data the user is working with. In classical windowing systems only the applications have the information about the 'meaning' of the displayed information. When exchanging data between applications active help of the applications is needed. If the windowing system can interpret the represented data it can provide basic functionality by itself. E.g. for any Image-widget the windowing system could provide standard functions such as copy, save, drag&drop, zoom, brightness control etc. The applications scope can be reduced to its core functionality.

3.2.3 Structure of the window server

The central component of the windowing server is a command interpreter, which is used by applications to control their user interfaces. Applications can create and configure widgets and define relationships between the widgets using an object oriented command language. More information about the command language can be found in section 3.3.1.

Figure 4 illustrates an example constellation of an application, the windowing server and some exemplary widgets. The application instructs the command interpreter to create the widgets of its user interface. As seen in the figure there are different types of widgets. The *pSLIM*-widget exemplifies a 'protocol'-widget, which provides an independent communication interface to the application. This enables the representation of dynamically changing data. In contrast to that, the other widgets represent 'static data' such as texts or images. The Image-widget is a good example of what is possible when the windowing server 'knows' the data, the user deals with. It can provide standard operations, that can be applied to any widget of this type. This offered functionality need not to be inevitably built in the widget - the widget can uti-

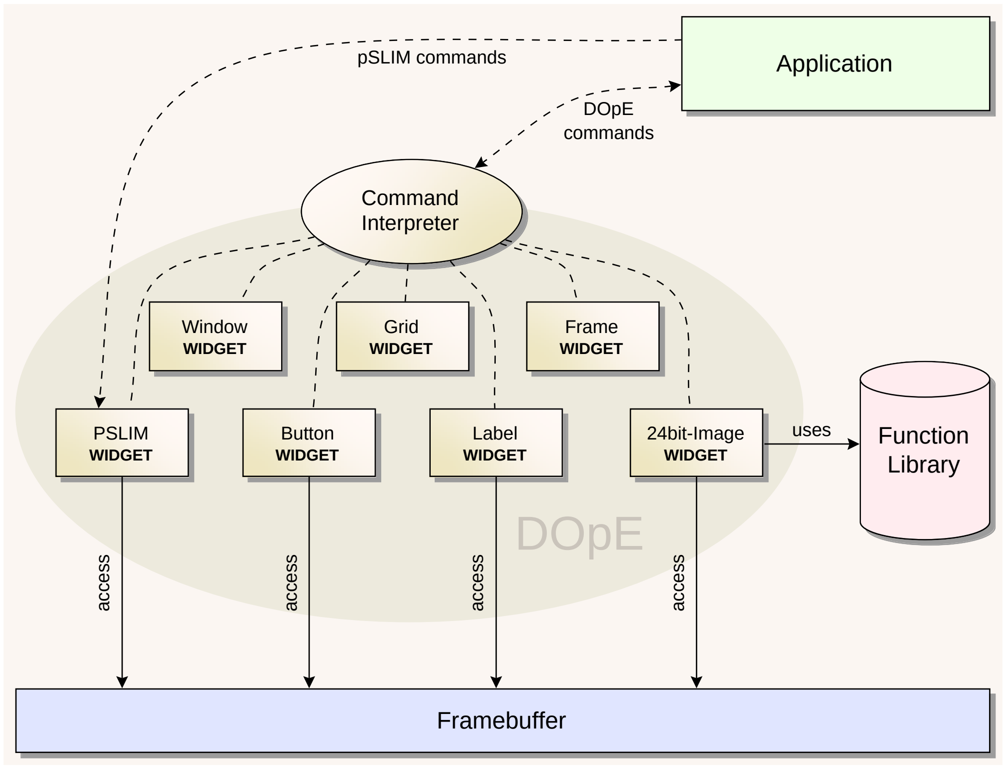


Figure 4: relationship between an application and the windowing server

lize even an external function library, that runs in a separate address space. The library can be assembled by the user without changing or configuring the windowing server.

3.3 Client-server communication

This section addresses the communication concept between client applications and the windowing server.

The client's view on the windowing server is at a high abstraction level. Since the server handles widgets and their topological structure autonomously, clients do not need to take care about the physical representation of widgets (e.g. type information of their attributes etc.). The only remaining responsibility of a client is the definition of the widgets properties and how widgets relate to each other (widget topology), leading to a layout of the user interface. Due to the client's abstract perception of the windowing server, an abstract way is advisable to express the widget's properties and topology.

The problem of a compact description of a user interface was solved already in a very elegant way by the developers of Tcl / Tk. Because of the lingual limitations of C / C++ a script language was invented, which was optimised for describing user interfaces. One great feature of Tcl/Tk is the ’tag-value’ notation for specifying widget’s attributes. The following Tcl/Tk command creates a Button-widget with its initial properties:

```
button .b -text "Quit" -bg red -command exit
```

An equivalent expression of this action would take at least 4 lines of C/C++-code (the creation of the widget, the definitions of each attribute). Even worse: The fact that the 4 C/C++ instructions belong to one action gets lost. Thus, when updating multiple widget attributes, the widget’s appearance must either be updated after each attribute assignment (although only one redraw is needed) or explicitely by calling an ’update’ function (resulting in 5 lines of code).

Besides the compactness, the usage of a command language for this purpose implies some other advantages:

- The communication interface is very simple - it must only be capable of transferring ASCII-strings.
- The command language can be extended without any change of the communication interface. The presence of a new widget type enhances the command language but has no effect on existing applications.
- A user interface can be designed using a command shell - a big advantage of Tcl/Tk is the easiness of experimenting.

An obvious drawback of this approach is the overhead caused by the parsing of the commands. Practically, this drawback does not come to the fore. Client-server communication happens only during the initialisation and reconfiguration of the user interface’s widgets - in contrast to X-windows, where clients get involved in every graphics operation (e.g. when moving or resizing windows).

3.3.1 Lingual scope and syntax

Each command is one entity, which can be executed completely without taking the previous and next commands into account. Thus, there are no control structures needed. The control flow is the client’s responsibility and must be implemented on the client’s side.

Although Tcl/Tk was mentioned as a clever way to describe user interfaces, its syntax is not used here. The chosen syntax reminds strongly on

C/C++ but features advanced lingual expressions. The following syntax definitions are expressed in EBNF notation - they are not exhausting but reduced to the essential rules.

There is only one keyword 'new', which is always used in connection with an assignment:

```
<create> := <refname> '=' 'new' <type> '(' { <tagval> } ')'
```

It is used to create a new object of the specified <type>. After the execution of such a command, the object can be referenced using the <refname>. Both <refname> and <type> are identifiers:

```
<digit>  := '0'|'1'| ... |'9'
<letter> := 'a'|'b'| ... |'z'|'A'|'B'| ... |'Z'
<ident>  := <letter>|'_' { <letter>|<digit>|'_' }
<refname>:= <ident>
<type>   := <ident>
```

Optionally, object attributes can be set up inside the brackets. The attributes are specified by a sequence of tags and values:

```
<tag> := <ident>
<value> := <int>|<float>|<ident>|<string>
<tagval> := '-'<tag> <value>
```

Tags specify the attribute the <value> is assigned to. Once an object is created its <refname> can be used to invoke methods within the object:

```
<method> := <ident>
<methodcall> := <refname>'.'<method>'('<params>')'
```

The parameter sequence consists of an mandatory and optional part:

```
<manpar> := <value>{ ',' <value> }
<optpar> := <tagval>{ <tagval> }
<params> := [<manpar>] | [<optpar>] | <manpar> ',' <optpar>
```

Mandatory parts must be specified likewise function parameters in C/C++. Additionally, optional parameters can be passed using the 'tag-value' notation. For each optional parameter a default value is predefined, which is used when the optional parameter is not specified. If a method returns an object reference, the return value can be assigned to a reference identifier.

```
<invoke> := [<refname> '='] <methodcall>
```

While attributes are set via method invocations, they can be requested by using:

```
<attribute> := <ident>
<request> := <refname>'.'<attribute>
```

So a command is either an object creation, a method invocation or an attribute request:

```
<command> := <create> | <invoke> | <request>
```

3.4 Event handling

The previous section covered the communication from the client to the *DOpE*-server. This section deals with the other way: How does *DOpE* notifies its clients about events?

The basic idea of the event handling of *DOpE* is derivated from the event concept of Tcl/Tk. This concept allows the binding of events to individual widgets. The kinds of events, that are notified to the application can be defined for every widget individually.

Figure 5 shows an overview over the used event concept. After the application created its widgets it can bind certain event types to its widgets. This is done in step 1 using the method "bind" of the corresponding widget. This method gets two arguments: the event-type and an associated event-message. Both arguments are strings to prevent built-in limitations of the concept. So the event-message can be any kind of data (identifiers, function pointers etc.). It is up to the client to define what such a message represents. The available event types depend on the widget, the event should be bound to. Widgets can provide custom event types (e.g. a change of the text of an entry field, a change of a button's state, synchronisation message of a real-time widget etc.).

The script interpreter of *DOpE* interprets the command (see section 3.3) and instructs the widget to activate the desired binding (step 2). Each widget applies a filter to incoming input events, which only allows bounded events to be forwarded. When an input event occurs (step 3) the widget checks its bindings and passes a 'notification' message to the 'Messenger'-component (step 4). The 'Messenger'-component is the communication interface to the client application and forwards the notification to the client via an IPC call. On the client's side there is a dedicated thread - the 'Action Listener' - for

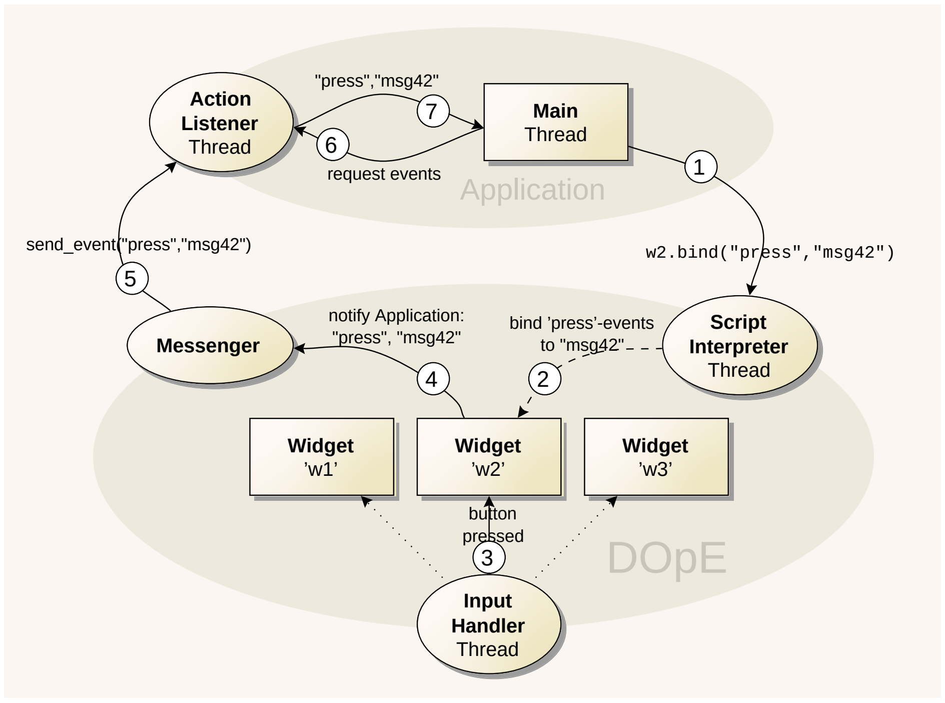


Figure 5: schematic overview over the event handling of DOpE

receiving event messages from the windowing server (step 5). The client is responsible of what to do, when the 'Action Listener' receives an event. Figure 5 shows one way to handle events on the client's side by queuing them. The main-thread can request the queued events explicitely (steps 6 and 7).

Another way to handle events on the client's side is the registration of callback functions. In this case the event message represents a pointer to a callback routine. Everytime, when the action listener receives an event it also gets the event message and can execute the corresponding callback routine without interfering the main thread.

Even classical single-threaded event-loop based applications are possible by merging the action-listener and the main-thread into one thread on the client's side.

With respect to the design criteria "small interfaces" (section 3.1.2) the interface for transferring events from *DOpE* to its clients is very simple and generic. The mechanism allows the implementation of various event delivery paradigms such as callback functions, event-loops or more powerful ones on the client's side.

3.5 Redraw/real-time concept

The central point of the real-time concept is the redraw handling. Since drawing operations are very time consuming, it is very important to organize them in a clever way.

Firstly, the problem of executing redraw operations within windowing systems is outlined. Secondly, a possible solution is shown and discussed briefly. The following sections show a progress to achieve the goal of real-time capability at the user interface level.

3.5.1 Clipping

The organisation of a window based graphical user interface is based on rectangular areas. Every element of the user interface uses a rectangular shape as internal representation (window, button etc.). Also redraw operations refer to rectangular shapes. So the basic problem of (re-)drawing a part of the screen can be described as follows:

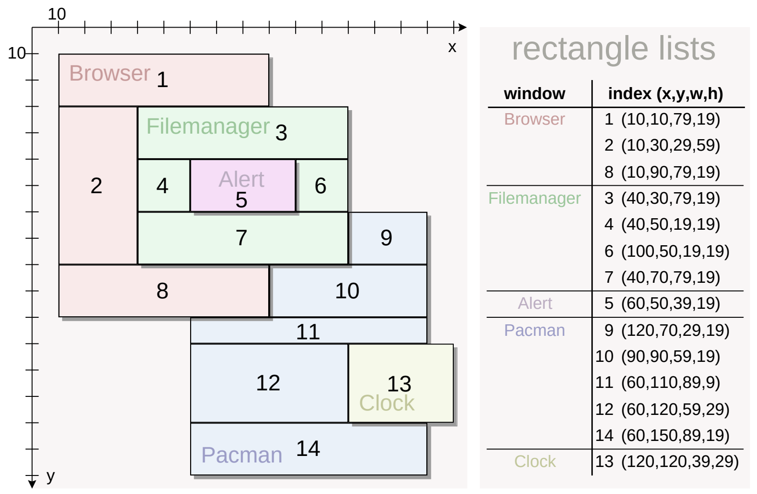
A window can be described by using four attributes (x,y,width,height). Every window configuration is an ordered set of windows w[0],w[1],w[2]...w[n]. The index describes the 'depth' of a window. The index 0 represents the nearest window (top window). The index n stands for the background window (desktop).

Redrawing a given screen area (r_x,r_y,r_width,r_height) - called 'dirty area' - means firstly, to find out which window is visible at which part of this screen area.

This is done by evaluating the intersections of the dirty area with the windows - beginning with window w[0]. If there is an intersection of the dirty area and a window w[i] the area is divided into two sub-areas:

- the intersection area, where w[i] is visible - this area can be drawn immediately
- the rest, where the windows w[i+1]..w[n] are potentially visible. We have to go on with the intersections of this area with w[i+1]..w[n] recursively. (the remaining area is usually not rectangular and must be subdivided into rectangles)

Obviously, the drawing operations within a window must be limited to its rectangular area (clipping). So every time, when a drawing operation is executed a clipping area must be set to limit the drawing to the valid area.



window	index (x,y,w,h)
Browser	1 (10,10,79,19)
	2 (10,30,29,59)
	8 (10,90,79,19)
Filemanager	3 (40,30,79,19)
	4 (40,50,19,19)
	6 (100,50,19,19)
	7 (40,70,79,19)
Alert	5 (60,50,39,19)
Pacman	9 (120,70,29,19)
	10 (90,90,59,19)
	11 (60,110,89,9)
	12 (60,120,59,29)
	14 (60,150,89,19)
Clock	13 (120,120,39,29)

Figure 6: simplified example of using rectangle lists

The common way to assure the limiting of the drawing operations to the clipping area is the implementation of clipping checks into the drawing routine of every graphical primitive.

3.5.2 A straight forward approach

This section discusses an idea of managing redraw operations as described in [1]. It deals with terms like ’window’ or ’moving a window’ in the meaning of their common senses.

For a given window configuration the windowing system determines, which visible rectangular screen area belongs to which window and stores this information in so called ’rectangle lists’. Every window has an associated ’rectangle list’ holding the information about the visible parts of the window. Figure 6 illustrates an example window configuration and the corresponding rectangle lists.

Each time, when the window configuration changes (windows are moved, resized, they appear or disappear) the ’rectangle lists’ must be updated by the windowing system.

When a redraw is needed it is up to the client to check the ’rectangle list’ of the affected window and draw the corresponding rectangular areas. The client has to take care about applying clipping to its drawing routines (see section 3.5.1).

This approach gives the clients full control about the points in time when redraws are done. Multiple clients can update their screen areas without blocking each other. It can be seen as a direct mapping of the underlying scheduling scheme to a graphical user interface.

The biggest advantage of this method is flexibility: a client has full control over his redraw operations. The windowing system need not to care about 'update rates' because this is the responsibility of the clients and the scheduler.

So, is there a reason not to go for this easy solution?

Since the clients deal with the redraw operations they need access to the physical frame-buffer and must implement drawing operations in a correct way. The windowing system has no chance to forbid a client to paint on other client's screen areas. One client can mess up the whole screen. This conflicts heavily with the security design criterion described in section 3.1.4!

Due to the assignment of a huge part of the GUI's responsibility to the clients they need a lot of information about the system (screen size, color depth etc.) and grow very complex. The clients have to deal with a lot of things (painting of elements of the user interface), that do not belong to their actual scope.

It is hard to establish an uniform look and feel among the client applications if every client is completely responsible for its appearance and behavior.

This problem could partly be solved with a library, that provides standard functions for the drawing and handling of the GUI's primitives.

Still, the big security drawback remains.

3.5.3 Further demands

The previous section covered an easy way to manage multiple windowed client applications on one screen. This surely satisfies the meaning of the term 'windowing system' but it is far behind the needs of a modern graphical user interface.

This section examines which demands are made to a graphical user interface. it should impart a feeling about the range of problems, which must be respected while designing the concept.

Firstly, the results of section 3.5.2 are taken into account. After that, another important property of modern user interfaces - hierarchical widgets - is introduced. Subsequently, the view on the user interface from three different perspectives (user, real-time-application, non-real-time-application) is illuminated.

Shrinking client's responsibility

The consequence of section 3.5.2 is that drawing operations should only be executed by one trusted instance - the window manager. The responsibility of the clients should be set to a minimum to meet the security and ergonomic goals described in section 3.1.

Hierarchical widgets - clipping stack

Typically, a graphical user interface does not only features windows as primitives (as done in section 3.5.2) but more advanced widgets. Widgets can be organised hierarchically to build complex dialogs out of them. The hierarchical structure of widgets leads also to a hierarchical structure of clipping areas. A 'layout'-widget (a widget which handles one or more 'child' widgets) has to limit its output to its own area before drawing its children. A child can be a 'layout'-widget again etc.

Examples for layout widgets are: Windows (a Window consists of the window elements and its content - which is a widget), Frames (where its content can be placed independently by using Scrollbars), Scrollbars (which consists of arrows and a slider), Grids (child widgets are organised in rows and columns on a grid) etc.

The hierarchically clipping is done using a clipping-stack. Each stack element represents a rectangular screen area. When pushing a new area to the clipping-stack, the current clipping area (the current top element of the clipping stack) is shrunk to be the intersection of the new and current clipping area.

Since only one instance (the window manager) is responsible for drawing operations only one clipping stack exists. As a consequence only one redraw operation can be executed at any one time and must be finished before any other redraw operation can be started. (otherwise it would be hardly possible to keep the user interface consistent during window configuration changes)

In contrast to that, classical windowing systems - such as X-windows - consider the window's content as 'flat'. The handling of hierarchically structured widgets is the client's responsibility. X-windows has no information about the 'meaning' of the window's content.

The users demands

The one, for whom all this work is primary done is the user. So what are the most important things about the user interface from the user's viewpoint?

- low reaction times / ergonomic issues: The usage of the user interface should be sensed as fluently and comfortable. A user interface can only be attractive to a user when giving him the attention he needs.

- accessibility: Even under hard circumstances (very low free CPU time) the user interface still needs to be usable. The user should never gain the feeling not to be master of the system.

User generated redraws can occur at any moment because no predictions about the user's actions can be made. It is important to execute this kind of redraw immediately (or at least: keep the delay of the redraw in a tolerable range) to guarantee the accessibility of the user interface.

Demands of non-real-time processes

The only important thing about non-real-time processes concerning the user interface is topicality. The visible appearance on the user interface must represent the current state of the running application.

The generation of redraws of non-real-time widgets is done explicitely by the associated program. When considering 'hard' situations (low free CPU time) delays of redraw executions are acceptable. (either way, the user can not decide if the client or the windowing system causes the delay - so this does not harm his comfort)

Demands of real-time processes

The strictest conditions about redraws are made by the real-time widgets. Their redraws must be executed in a very controlled way to make sure that the needed service (a constant frame rate) can be guaranteed. Non-controlled delays of redraws must be obviated.

There is one nice thing about redraw operations caused by real-time widgets in contrast to the users or non-real-time redraws: They are predictable!

3.5.4 A naive way

The simplest way to let the user interface work follows the rule: first come - first served.

A 'draw' method inside the window manager is called each time, when a redraw is needed. If the window manager is already busy with another redraw the caller has to wait until the current redraw is finished (section 3.5.3).

Each real-time widget has a dedicated thread running at *DOpE*'s user space. It continuously produces redraw events for its corresponding widget. So basically any desired constant update rate of a real-time widget could be realised.

When assuming a lightening fast execution speed with redraw operations taking only a very small fraction of the available CPU time this method could work well. So the user interface would only become active from time to time to do a redraw - which is done instantly.

Sadly, such assumptions can not be made. Practically, screen drawing operations are very time consuming on account to a small screen memory access bandwidth (see also section 5.1). So multiple redraw requests can easily collide with each other. Such collisions cause a delay of the redraw operation which can have fatal consequences for concurrent real-time threads or even the system's overall accessibility.

Even worse - it is not predictable if the window manager can bear an additional real-time widget in a certain situation. So the only way to discover that is just to try it out and risk a harm of real-time conditions or even a collapse of the system. This hardly meets the understanding of 'Quality of Service'.

3.5.5 Introducing the Redraw Manager

The first idea coming in mind to outplay at least a part of the arising problems is the introduction of priorities. They are assigned to redraw operations to reduce the blocking of important (real-time) redraws by unimportant (non-real-time) redraws.

Instead of performing redraw operations directly by instrumenting the 'window manager' (as described in section 3.5.4) redraw requests have to be registered at a dedicated component - namely Redraw Manager. The Redraw Manager stores incoming redraw requests together with their priorities into a 'redraw queue'. Figure 7 shows an illustration of how the Redraw Manager works. Each party (user-thread, real-time-widget, non-real-time-widget) can register redraw requests. When executing the stored redraw requests the Redraw Manager can take their associated priorities into account to determine the chronological order of the redraw request executions.

The decoupling of the creator and the executor of redraw operations leads to a much more structured way to handle them. Since real-time widgets can produce redraws at high priorities their needs are respected much more than previously (section 3.5.4).

For an estimation of this strategy, the demands of the user, the non-real-time clients and the real-time clients have to be taken into account. The following hassles show up:

real-time clients viewpoint:

- Each real-time widget needs a dedicated thread that constantly produces redraw requests.

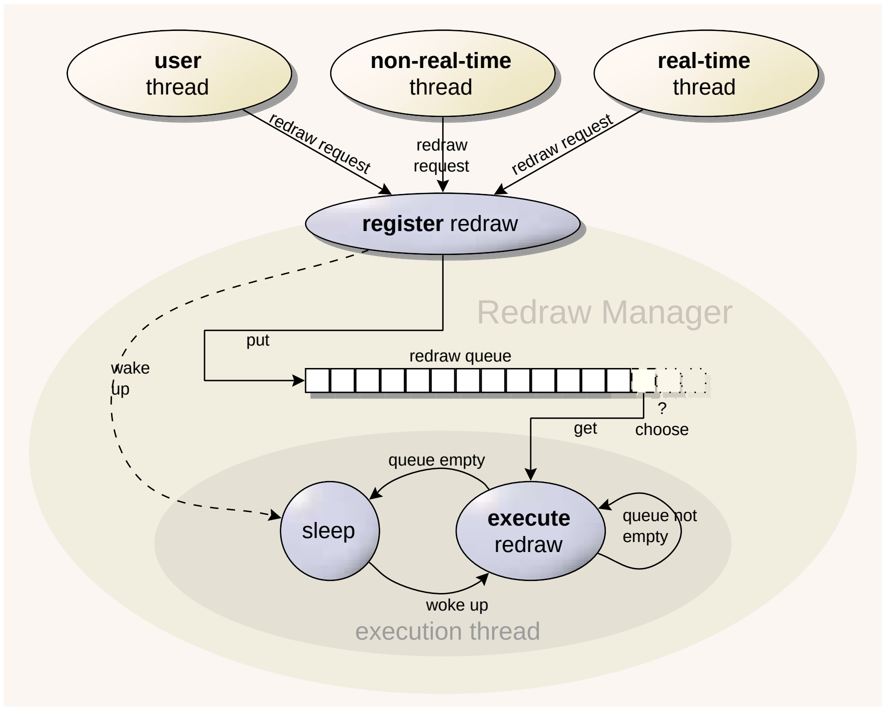


Figure 7: Redraw Manager

- Every real-time redraw producing thread (one for each real-time widget) produces redraw requests independently. No assumptions can be made that these requests are equally distributed in time. The redraw requests are produced in a more or less chaotic way (especially when multiple widgets use different update frequencies). As a consequence, unpredictable temporary hard situations can emerge.
- The core of the problem remains: A once started long running redraw operation prevents other redraw operations from execution until it is finished. So a low priority redraw can cause unswallowable delays of high priority redraw operations.

users and non-real-time clients viewpoint:

- A permanently too low amount of free CPU time rises the queue size and can cause a queue overflow - this can lead to ergonomic drawbacks and even to the loose of accessibility. There is no indication of what 'too low free CPU time' means.
- The effect of a newly appearing real-time widget is not predictable.

3.5.6 An extreme approach as direction sign

One word, that immediately arises when engaging real-time operating systems is 'period'. A typical real-time application uses a fixed time to produce or consume a fixed amount of data (e.g. a video / audio stream) during one period. The execution of periods is clocked. The big advantage of these constraints is the predictability of the system's behavior. Until now, the concept of the user interface does not fit very well into this scheme.

As 'real-time' was declared as an important design goal it would make sense to let the user interface work periodically.

A simple periodically working user interface could force a redraw of the whole screen periodically. The needed time for the redraw could be estimated by a heuristic function dependent on the used widgets or by measuring the drawing speed. The needed time for such a redraw could be reserved using a real-time-scheduler.

The bad thing about this idea is the dramatical decrease of the overall system performance because of the big amount of unnecessarily executed redraws of passive screen areas.

Another disadvantage is the need of a global screen update frequency. This screen update frequency sets the limit for the speed of all actions on the screen (frames per second for animations as well as the speed of window movements etc.). However, it could be specified by the user dependent on the performance of the platform and the used applications to fit his needs.

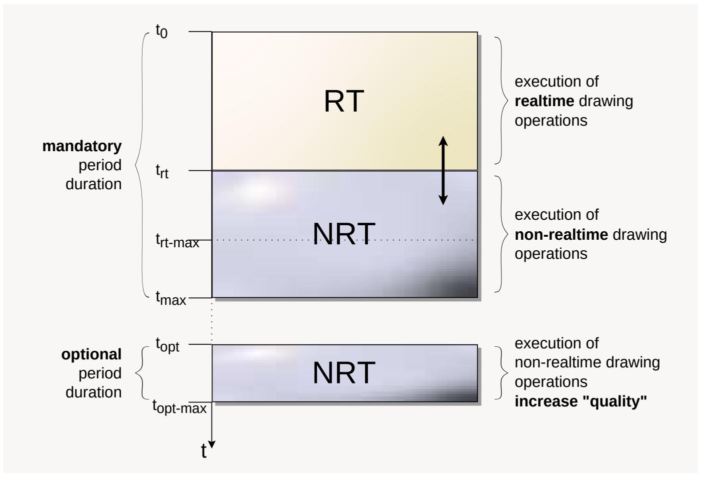
Obviously, the waste of CPU time makes this construction not usable for practical needs. But it shows the direction to go for.

3.5.7 A periodically working concept

The point of this section is to combine the ideas of the previous sections. The result should be a periodically working concept, which respects the different needs of the user, real-time clients and non-real-time clients.

The upper part of figure 8 displays a single period and its usage for real-time and non-real-time operations. The condition for these operations is not to exceed the time of the period (t_max). If this happened the execution of a drawing operation would rise up into the beginning of the next period.

The field of duty of the Redraw Manager is shrunk to handle only the users and non-real-time redraws. Real-time redraws are handled by a new component, called Realtime Manager (which should actually be called Realtime Redraw Manager).



t_0
RT
execution of **realtime** drawing operations
mandatory period duration
t_{rt}
NRT
execution of **non-realtime** drawing operations
$t_{rt\text{-}max}$
t_{max}
optional period duration
t_{opt}
NRT
execution of non-realtime drawing operations **increase "quality"**
$t_{opt\text{-}max}$
t

Figure 8: period with a mandatory and optional part

The drawing of the real-time widgets must be done without any conditions. Thus, this is done at the beginning of the period by calling a corresponding function of the Realtime Manager. Since the CPU time requirements of real-time widgets are predictable the exceed of the period's length can be prevented.

The remaining time of the period can be used to perform the execution of the Redraw Manager's drawing operations, generated by the user and non-real-time clients. Again, an exceeding of the period's length must be strictly prevented. For this reason the remaining time is passed to the Redraw Manager as an argument. The residual question of how the Redraw Manager ensures to keep the redraw operations time into the given range is covered in the following section.

The lower part of Figure 8 shows an additional feature of modern real-time scheduling models: optional parts. While the upper (mandatory) part implements the essential functionality, an optional part has the task to increase "the quality" of the functionality. In contrast to the well defined execution of mandatory parts, optional parts are only executed sometimes to fill CPU time gaps.

The usage of optional parts for non-real-time redraws is a good example of how to increase the "quality" of the user interface at times, when there is some CPU time left. E.g. section 3.5.9 will show the effect of optional parts on the 'smoothness' of the user interface.

3.5.8 Subdividing redraw operations

As stated in section 3.5.3 redraw operations must be completely processed at once. A long continuing redraw operation delays other redraw requests until it is finished. If such a redraw exceeded the length of a period (see the previous section) it would delay the real-time redraws of the next period. Such a behaviour must strictly be circumvented.

The key is, to split large redraw requests into smaller ones before executing them.

Firstly, a function 'Redraw_Pixels' is introduced, that takes a number of pixels as argument and executes queued redraw operations with the specified number of pixels.

The function takes always the first redraw queue element and tries to redraw it. If the size of the redraw operation is higher than the maximum amount of pixels to draw, it is divided into a part that fits into the left amount of pixels to draw (this part is drawn immediately) and a part that stays at the top of the redraw queue.

Additionally, a heuristic function is needed to determine how many pixels can be drawn within a given time. This heuristic needs not to be exact. It should be chosen pessimistic to make sure that execution time of a 'Redraw_Pixels' function call does not exceed the 'left time' of the period. Although, a too pessimistic heuristic function would cause an overhead by needless 'redraw splittings'.

The following pseudocode illustrates the idea:

```
WHILE ((redraw_queue_size>0) AND (left_time>0)) DO BEGIN
  beg_time   := Get_Time()
  num_pixels := Pix_Per_Usec(left_time)

  Redraw_Pixels(num_pixels)

  { determine how much time is left }
  end_time  := Get_Time()
  left_time := left_time - (end_time - beg_time)
END
```

It is recommended to base the heuristic function on performance measurements. Two meaningful values, that can be taken into account, are the write access speeds to the screen memory (on the graphics card) and main memory.

Another way to define the heuristic function can be based on real-life measurements. During redraw operations of windowing system the pixel / time ratio can be measured to influence the heuristic function dynamically.

3.5.9 Reducing non-real-time redraw operations

In order to shrink the needed CPU time for dealing with non-real-time widgets to a minimum, redundant drawing operations must be prevented. The 'redraw queuing' concept makes away free to perform some great optimisations.

The first optimisation aims at the prevention of redundant redraw operations.

The second optimisation utilises the queue length as a criterion to decide whether redraw operations should be generated or not.

Redraw dropping

Since the execution of redraw operations is a time intensive task there is a great chance that multiple redraw requests for a certain screen area occur before this area is completely updated. Such a situation can even cause an overflow of the 'redraw queue'. Consequently, the Redraw Manager has to ensure that all redraw queue elements refer to disjoint screen areas.

The idea of redraw dropping is based on the following advisement: For widgets, which change their state faster than redraws can be executed, only the most recent state must be made visible on the screen. Add sub-states in between two redraw operations can be skipped. Every time a widget changes its state a redraw request is generated - but only some of them induce an actual redraw operation.

For each incoming redraw request its intersections with the currently queued redraw requests are determined. If such an intersection occurs the intersection is 'cut out' of the new redraw requests area. Obviously, new redraw requests that are completely covered by an already queued redraw request are completely skipped (e.g. a typical situation is a scrolling screen area).

Technically, the 'cut out' is performed by creating one or multiple rectangles out of the remaining area of the original redraw request. The intersection tests with the rest of the queue must be applied for each of these rectangles. This leads to recursive computations with branching factors ranging from 0 (new redraw request is completely covered by an existing one) to 4 (new redraw request contains an existing one completely - a rectangular hole of the redraw request must be 'cut out').

Since overlapping drawing operations can be prevented by this technique the maximum amount of queued ’dirty’ pixels is limited by the screen size.

On the other hand there exists a drawback due to the needed recursive computations.

So there is trade-off between the expense of redraw operations and the expense of computations of the optimisation. Practically, the amount of recursive operations can be held very low by assigning dedicated widgets to redraw operations. Thus, the intersection tests must only be applied to queue elements referring to the same widget.

Adaptive user input handling

The previous section dealt with optimising the ’redraw queue’ using recursive computations. This optimisation only works until the queue size enlarges to a size, where the these computations get more expensive than the redraw operations itself.

So the aim of this section is the reduction of the amount of redraw requests. This is done by taking the current ’redraw queue’ size into account. A high queue size is a sign for a ’hard situation’.

The Redraw Manager handles only non-real-time redraws. So redraw requests can be caused either by non-real-time application or by user actions (e.g. the user moves a big window). The priorities in such ’hard situations’ are:

- ’Quality of Service’ - real-time redraws must not be harmed
- accessibility - the user interface must remain completely accessible by the user

The ’smoothness’ of non-real-time redraws (e.g. the smoothness of a moving window) plays only a minor role. So the current queue size can be used as an indication to skip non-essential redraws caused by the user.

Naturally, this technique can be applied to limit the generation of redraw requests of non-real-time clients, too. The execution of a non-real-time client could even be delayed until the ’hard situation’ is over.

Optional parts (as described in section 3.5.7) can help to additionally decrease the ’redraw queue’ size. In return, the number of skipped non-essential redraws decreases, too. This raises the ’smoothness’ of the user interface in such situations.

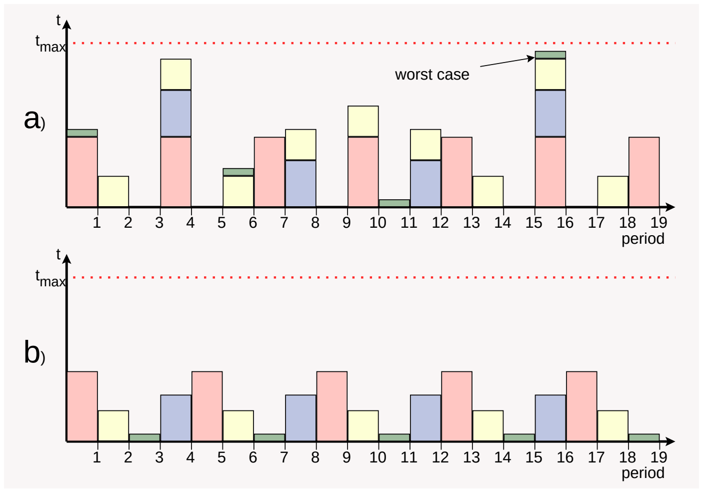


Figure 9: two opposing real-time redraw strategies

3.5.10 Handling of real-time widgets

The redraw handling of real-time widgets is much straighter than the handling of non-real-time redraws because their constraints limit the ways to handle them a lot.

Section 3.5.7 already broached the problem of integrating real-time widgets into the periodically working concept. The described concept shares one disadvantage with the 'extreme concept' (section 3.5.6): there exist global period parameters (frequency and duration time). The problem of supporting different update frequencies (frame rates) for different widgets arises immediately. A way to provide a certain degree of flexibility is, not to draw each real-time widget at every period but leaving out a widget-specific number of periods between its redraws. E.g. when a global period frequency of 100Hz is given, real-time widgets could be updated at frequencies of 100Hz, 50Hz, 33.3Hz, 25Hz etc. (any whole-numbered fraction of the global period frequency). These 'frame rates' are typical for common multimedia applications. Thus, this degree of flexibility is sufficient for the applications this work is aiming at.

Given a situation, where multiple real-time widgets at different frame rates must be handled, the windowing system must decide, which widget must be drawn at which period. The drawing time of each real-time widget is known. Obviously, the needed time for all real-time-redraws at a certain period is the sum of redraw operations that become due in this period. The windowing system has to ensure that this sum does not transcend the whole duration time (t_max) of the period.

Different strategies are possible to fulfill this condition. The windowing system does not provide one predefined solution but offers a general interface for implementing such strategies. To display the wide range of possible ways, two opposing examples are illustrated in figure 9:

a) Any whole-numbered fraction of the global period frequency can individually be chosen for each real-time widget. New real-time widgets are accepted as long as the sum of all real-time redraw durations is lower than the length of the period. Even in the worst case - when all redraws must be executed at one period - the sum of all redraws is still lower than the period's duration time. So the condition is fulfilled for sure. There is only one important drawback for practical uses: The decision about new widgets acts upon a hypothetical worst case - so the criterion to decide about new real-time widgets is far too pessimistic.

b) In contrast to the previous strategy only one update frequency for real-time widgets is allowed (e.g. 25Hz). Periods are grouped into time slots (e.g. 4 time slots when 100Hz periods are used). In this way, real-time redraws can be executed interleaved. New real-time widgets are accepted as long as there is a time slot, which can hold the new widget's redraw without exceeding the period's length. As indicated by figure 9 this strategy is capable of handling more real-time widgets without exceeding the period's length than the first strategy. The drawback of this solution is the lack of flexibility of update frequencies.

Obviously, the golden way for practical uses is somewhere in between these two exemplary strategies.

3.5.11 The next level - multiple clipping stacks

Although the previously described concept provides enough flexibility for common multimedia applications and fulfills the today's needs, it is rather limited compared to the freedom of the 'Artifact' windowing system [1] (see section 3.5.2).

As stated in section 3.1.3 'future compatibility' is a strong design criterion - so limiting the ways to handle real-time redraws to the previously described concept can be a stumbling block for future developments.

A way to ensure the maximum of flexibility is the support of concurrent redraws. In contrast of the sequential processing of all redraws (as done in the previous sections) more than one redraw operations at a time must be allowed.

In section 3.5.3 a clipping stack was introduced to handle the clipping of hierarchical widgets. The consequence was, that only one drawing operation can take happen at a time. The key to support concurrent redrawing operations is the handling of multiple clipping stacks. Every thread of the window manager, that concurrently accesses the screen has to manage one clipping stack for its current drawing operation. Consequently, the corresponding clipping stack must be passed as parameter to the draw functions of the widgets - in contrast of the common use of one global clipping stack as done before.

Each window has to hold a semaphore to assure the consistency of the window's positions and sizes. It is locked during the usage of the window's properties (size and position). So a window can not change its properties during its involvement in a drawing operation.

The usage of concurrent redraws by using multiple clipping stacks is neither essential for todays needs nor used by the currently available widget types. Concurrent redraws are just a further option to provide a maximum of flexibility for the future. Widgets, that utilise concurrent redraws have the same degree of freedom and responsibility as applications on the 'Artifact' windowing system.

4 Implementation

The conceptual ideas, which were discussed in the previous sections, were implemented in the form of the windowing system *DOpE*. *DOpE* stands for 'Desktop Operating Environment'.

This section will firstly provide general information about the implementations philosophy. Subsequently, the current state of development is summarised. After that, several interesting aspects about the implementation get outlined.

4.1 Implementation related design criteria

In section 3.1 general design criteria were discussed. When it comes to a concrete implementation, even more aspects arise. This section will show additional goals and criteria, that were taken into account when implementation related decisions were made.

Often there is a trade-off between these criteria. Optimising the code to a maximum (and taking processor types etc. into account) obviously conflicts with the goal 'portability'.

4.1.1 Performance and memory usage

The economical usage of CPU and memory resources can be seen as a general rule. As much as possible CPU and memory resources should be kept free for the usage by applications.

Basic rules were:

- no multiple storage of data
- compact data structures
- very sensible use of external libraries: even libc is not used by *DOpE*

4.1.2 Portability

All code it written in 100% ANSI-C.

As expected, *DOpE* has to deal with a number of platform dependent things, such as mouse, keyboard, screen drivers etc. All platform dependent elements are encapsulated into separate components with sensibly defined interfaces. For all platform dependent functions (memory allocation, thread creation), abstractions were created and consistently used. This makes it possible to keep the majority of the code generic.

All inter-process-communication interfaces were defined using IDL ('Interface Description Language"). So *DOpE* can easily be ported to platforms, where an IDL compiler exists. Additionally, all parts of *DOpE*, that deal with inter-process-communication were encapsulated into separate components. So the communication technique can easily be exchanged by implementing new communication components featuring the same interfaces. (e.g. to let *DOpE* run over a network)

4.1.3 Object orientation - encapsulating

The code is structured in an object oriented fashion. It is consequently divided into functional entities with clearly defined interfaces (components). Originally, the modular design aimed at the runtime-exchangeability of single modules. Only modules, that are currently needed should be kept in memory. Since L^4 does not feature the needed infrastructure for this (file-system, dynamic loader library), this functionality is not fully implemented but prepared.

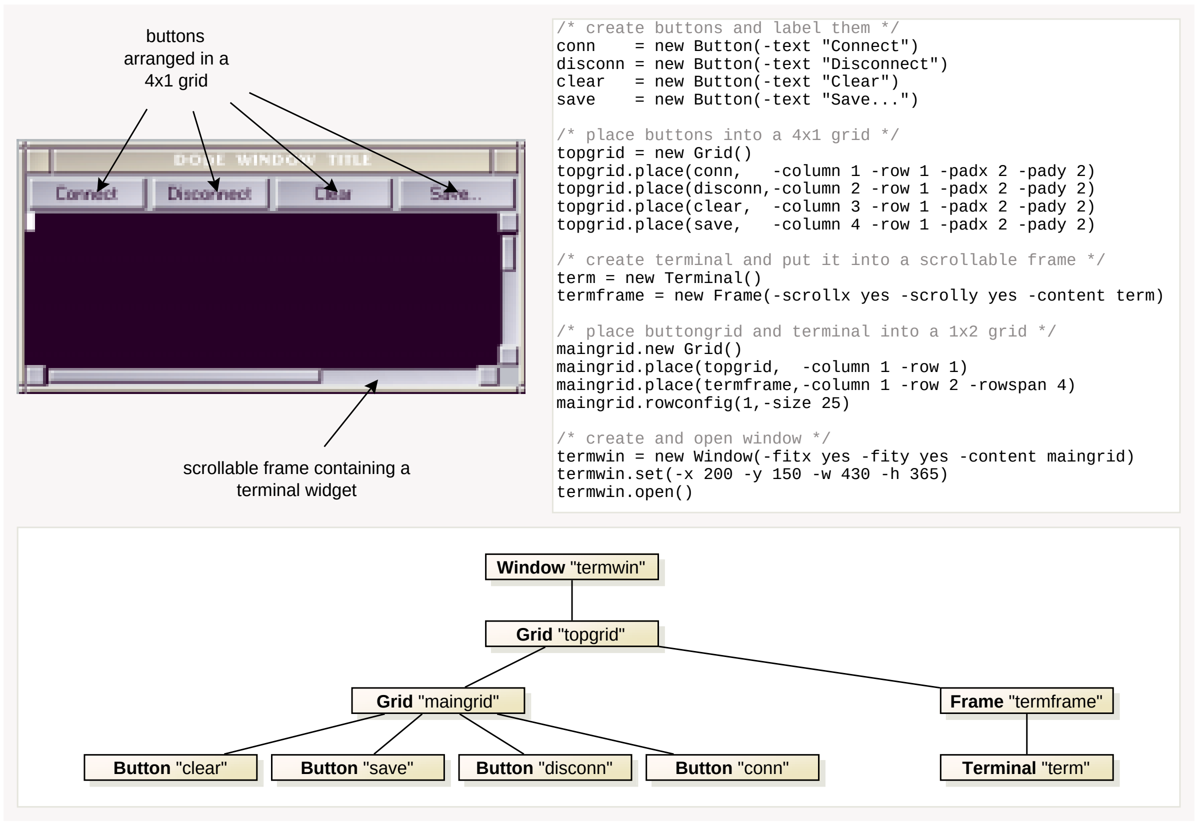
4.1.4 Easy client development

The acceptability of *DOpE* within the OS community highly depends on the ease of client development. For this reason a library was implemented, which features an easy-to-use interface to create *DOpE*-clients.

Even without this library it is no hard deal to create a *DOpE* client.

4.2 Current features of *DOpE*

- fully functional, non-blocking windowing system
- double buffered output
- font engine, that supports proportional bitmap fonts
- redraw concept supporting non-real-time and real-time widgets
- client-server communication via Flick (L^4) or Orbit (Linux)
- command interpreter
- event handling concept
- keymaps to support different keyboard layouts
- a library for easy client development



```
/* create buttons and label them */
conn    = new Button(-text "Connect")
disconn = new Button(-text "Disconnect")
clear   = new Button(-text "Clear")
save    = new Button(-text "Save...")

/* place buttons into a 4x1 grid */
topgrid = new Grid()
topgrid.place(conn,   -column 1 -row 1 -padx 2 -pady 2)
topgrid.place(disconn,-column 2 -row 1 -padx 2 -pady 2)
topgrid.place(clear,  -column 3 -row 1 -padx 2 -pady 2)
topgrid.place(save,   -column 4 -row 1 -padx 2 -pady 2)

/* create terminal and put it into a scrollable frame */
term = new Terminal()
termframe = new Frame(-scrollx yes -scrolly yes -content term)

/* place buttongrid and terminal into a 1x2 grid */
maingrid.new Grid()
maingrid.place(topgrid,  -column 1 -row 1)
maingrid.place(termframe,-column 1 -row 2 -rowspan 4)
maingrid.rowconfig(1,-size 25)

/* create and open window */
termwin = new Window(-fitx yes -fity yes -content maingrid)
termwin.set(-x 200 -y 150 -w 430 -h 365)
termwin.open()
```

Figure 10: example of using different widgets, corresponding source code and widget hierarchy

4.2.1 Widget-set

The current implementation of *DOpE* provides a basic widget set. Due to the modular concept new widget types can easily be added. The following subsections are brief descriptions of the implemented widget types.

Figure 10 shows the user interface of an example 'terminal' application. It makes use of different widget types, which are provided by the current implementation of *DOpE*. The following subsections briefly describe the most interesting widget types. Primitive widget types such as Button or Background are not explained because they are considered as being common sense.

Window

Window-widgets are the basic elements of the windowing system. A Window consists of window control elements and its content. Window control elements can be used to arrange windows, to resize them or to change their stacking order. The provided elements are:

- ’border’, that surrounds the window. It allows the resizing of the window by dragging the border with the mouse.
- ’title-bar’, which contains a window title. It is used to move the window on the screen (by dragging the title-bar) or bring the window to top (by only clicking on the title bar).
- ’closer’ for closing the window
- ’fuller’ for setting the windows size to its maximum

The presence of the window elements is configurable per window.

Container

A Container-widget is a ’layout’-widget, which can hold multiple ’children’ widgets. The children can be placed relative to the Container’s position by using pixel coordinates (the origin of the coordinates is the top-left corner of the Container).

Frame

Frame-widgets can be used to hold a ’child’-widget (content) of any size inside a definable rectangular area. If the content is bigger than the Frame itself the viewport at the content is freely definable. Additionally, a Frame can provide full scrollbar functionality to let the user define the viewport freely.

Grid

Grid-widgets allow the arrangement of multiple ’child’-widgets on a grid. The rows and columns of the grid can be configured to fixed or weighted sizes. A ’child’-widget can not only be placed into a grid cell - it can even cover multiple grid cells. Additionally, horizontal and vertical spaces around ’child’-widgets can be specified. The outcome of this is a very powerful tool to layout dialog windows.

pSLIM

The *pSLIM*-widget is an implementation of the *pSLIM*-protocol, which was mentioned in section 2.2. It is the key to port ’Dropscon’ applications to *DOpE* easily. An example for such an application is *L^4Linux*. Each *pSLIM*-widget manages a separate virtual screen buffer. The size of the widget can be chosen independently from the size of the virtual screen buffer. In this case the output is scaled to fit into the desired widget’s dimensions.

PSLIM-widgets can be configured to act as real-time widget. In this case it is periodically updated at a specified frame rate by copying the virtual *pSLIM* screen-buffer to the physical screen. Otherwise a non-real-time redraw is requested after each processed graphics operation.

Terminal

The Terminal-widget allows the output of text. It provides a subset of VT100 functionality. Additionally, back and foreground colors are supported (via standard escape sequences).

4.3 Parallel development for Linux and L4

While *DOpE* is primary designed for L^4 it also runs under Linux. This cross-platform development is a good way to verify the portability design criterion. The more important reason is, that the development under Linux for Linux is far more comfortable compared to the permanent use of a test system. The compile-test-evaluation cycle is significantly lower, because no test computer must be rebooted.

4.3.1 Hardware abstractions under Linux

As abstraction for the real hardware 'libSDL' was used. It provides hardware abstractions for graphical output and user input. The graphical output properties can be freely chosen (any color depth, screen size). The screen output appears either as normal X-window on screen or uses a full-screen mode. The X-window mode is very comfortable for testing.

4.3.2 Thread and memory-management abstractions

The thread abstraction is also based on 'libSDL', which provides an interface to create and handle threads. Internally, these threads are POSIX-threads.

For memory allocation and deallocation the corresponding functions of 'stdlib' were used.

4.3.3 Client-server-communication

Since the inter-process-communication interfaces were defined in IDL language, a suitable IDL-compiler (Orbit) was used, to generate the client- and server stubs for Linux.

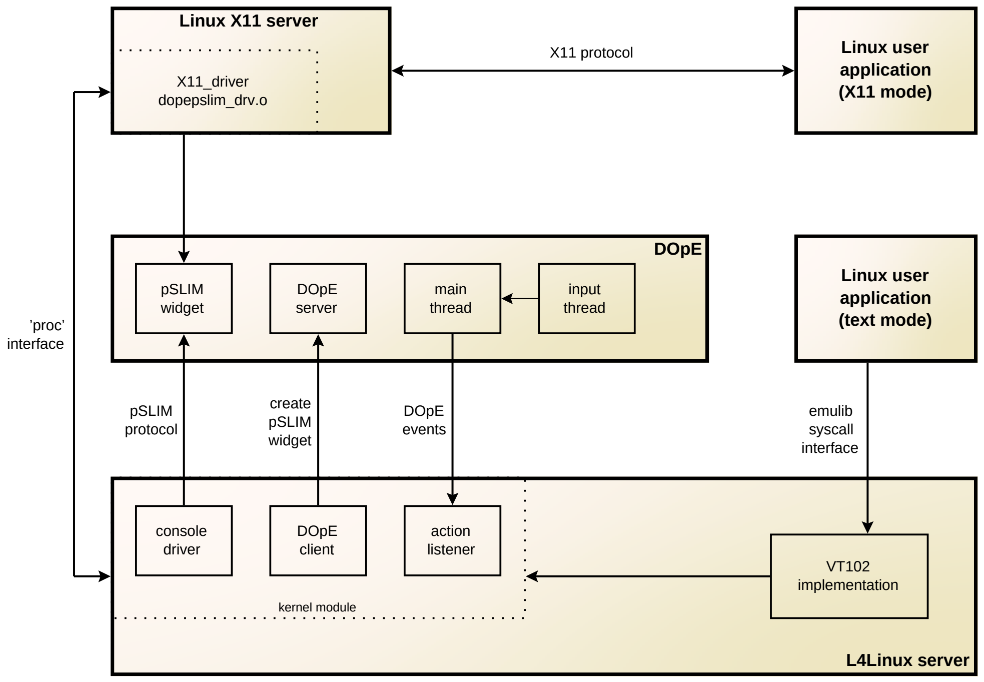


Figure 11: Linux running with DOpE

Sadly, the used IDL compilers Flick (for L^4) and Orbit (for Linux) have a quite different behavior when it comes to string references or the handling of data structures.

Finally, only the intersection of the features of both IDL compilers could be used for interface definitions.

4.4 Linux gets *DOpE*d

The previous section dealt with running *DOpE* inside a X-window. Now lets turn it around! - Running Linux inside a *DOpE*-window under L^4.

Figure 11 illustrates how L^4*Linux* and *DOpE* are related to each other. From *DOpE*'s viewpoint L^4*Linux* behaves just like a normal application, that uses the *pSLIM*-protocol to display pixel data. L^4*Linux* deals with *DOpE* using a kernel module - the L^4*Linux*-stub. The L^4*Linux*-stub plays the role of a *DOpE* application and provides an appropriate interface to the Linux kernel. Furthermore, the L^4*Linux*-stub provides a '/proc'-interface containing status information of the *pSLIM*-widget such as the virtual screen size, color depth and absolute mouse position.

The following two subsections describe the two ways of information flow:

- passing input event information into the Linux kernel

- displaying the Linux output inside a *DOpE* window

They are followed by a brief description of how to run XFree86 on L^4*Linux*.

4.4.1 Handling input events

When starting up the L^4*Linux*-stub it initiates an 'Action Listener' thread for receiving mouse and keyboard events. All input events referring the L^4*Linux* *DOpE*-window are immediately forwarded to this Action Listener. For passing the input events into the Linux kernel it provides two pseudo-device-drivers (keyboard and mouse) to the kernel. All events, that are received by the Action Listener are translated to corresponding Linux kernel event messages and are reported to the Linux kernel.

4.4.2 Displaying Linux output

While initialising the L^4*Linux*-stub a *DOpE*-window containing a *pSLIM*-widget is created via sending the corresponding *DOpE*-commands to the windowing server. As described in section 2.2 a *pSLIM*-widget provides a virtual screen area on which a set of graphical primitives can be applied. For running Linux in text mode a Linux console driver interface is provided by the L^4*Linux*-stub. So the L^4*Linux*-stub receives textual output from the Linux kernel and formulates convenient *pSLIM*-commands to paint the textual output onto the *pSLIM*-screen. For speeding up colored textual output the *pSLIM*-protocol was slightly enhanced by the 'puts_attr' command. This command paints an ASCII-string with interleaved color information to a specified position on the *pSLIM*-widget.

4.4.3 Running X-windows

Due to the open architecture of XFree4, display and input drivers can be transparently added to the X-Server. For forwarding the graphical output of X-windows to the *pSLIM*-widget a XFree4 display driver was implemented, that handles a virtual screen buffer and transfers updated screen regions to the *pSLIM*-widget via the 'set'-command. The thread-id of the *pSLIM*-widget is requested via the '/proc'-interface, which is provided by the L^4*Linux*-stub.

As for the mouse driver, basically the standard XFree mouse driver featuring the PS/2 protocol can be used in connection with the '/dev/input/mice' interface. Since the L^4*Linux*-stub provides a mouse driver the information about mouse input events is available at '/dev/input/mice'. There is only one drawback of this solution: The '/dev/input/mice' interface utilises the PS/2 protocol, which deals only with relative movements. Consequently, the positions of *DOpE*'s mouse cursor and X-windows' mouse cursor are not consistent. Even worse - the XFree mouse driver accelerates the mouse movements. So even the relative movements of both mouse cursors are varying.

To solve the mouse consistency problem, the XFree mouse driver was slightly modified to read the actual absolute mouse position from the '/proc' device rather than using the relative movement information provided by '/dev/input/mice'. The '/dev/input/mice' interface is still being used for handling the mouse buttons and for the information about the points in time, when mouse movements occur.

action	MB/sec	usec/byte
main memory read	176,53	0,0054
main memory write	166,55	0,0057
screen memory read	6,12	0,1558
screen memory write	31,11	0,0307
copy main memory to screen	28,48	0,0335

measured on an AMD Duron 700Mhz PC equipped with a Matrox G440 graphics card

Figure 12: practical memory access measurements

5 Evaluation

5.1 Performance

Without any doubt: the transfer bandwidth from the CPU to the screen memory of the graphics card is the most limiting bottleneck of the performance of *DOpE*. Figure 12 displays a table of the measured memory bandwidths on an AMD Duron test computer. All measurements were done 16bit wise because a 16bit value is equivalent to a pixel with 16bit color-depth - as used by *DOpE*. The low transfer bandwidth to the graphics card is distinctive when compared to the access speed to the main memory. Due to that fact, operations on screen memory must be shrunk to a minimum. Thus, *DOpE* uses an offscreen rendering technique (double buffering). A virtual screen buffer is kept in the main memory. All graphics operations are quickly applied to the virtual screen. When the graphical operations are finished, the result is copied to the screen memory only once. So the bottleneck to the screen memory must be passed only one time per pixel.

A constantly used graphics operation of *DOpE* is the output of scaled images (e.g. output of a button's background or *pSLIM*-widget).

Consequently, the most interesting values for this practical use are 'main memory read' (for reading a source image), 'main memory write' (for writing to the virtual screen) and 'copy main memory to screen' (for transfering the pixels from the virtual screen buffer to the screen memory).

The needed times for the involved memory transfer operations are (see figure 12):

- reading a 32bit offset from a 'scale table': 4*0,0054 usec = 0,0216 usec

- reading a 16bit pixel from source image: 2*0,0054 usec = 0,0108 usec
- writing a 16bit pixel to virtual screen: 2*0,0057 usec = 0,0114 usec
- copying a pixel from the virtual screen buffer to the physical screen memory: 2*0,0335 usec = 0,067 usec

The sum of these values is 0,1108 usec. So theoretically 1 / 0,1108 = 9,025 pixels can be drawn during 1 usec. The memory accesses for loading the CPU instructions are not taken into account because they are kept in the processor's cache.

For the real-life measurement the Redraw Manager of *DOpE* was enhanced by statistical computations. The 'exec_redraw' function of the Redraw Manager determines the number of drawn pixels per redraw operation and measures the corresponding processing time. It keeps track of two values: 'average pixel / usec ratio' and 'minimum pixel / usec ratio'. The 'average pixel / usec ratio' is computated using a sliding mean algorithm with a constant learning rate of 0.05. The computation is done via:

```
current := current*0.95 + new*0.05
```

The results of the real-life measurements after working with *DOpE* for a while were 8 pix / usec average ratio and 4 pix / usec minimum ratio. So the average ratio reaches nearly the theoretically possible value of 9,025 pixels. This is a strong indication for the efficient implementation of the graphical output routines and the low overhead caused by the windowing system. The minimum ratio occurs when multi layered widgets such as stacked 'layout'-widgets must be drawn.

The statistical measurements are also used as heuristic function in the current implementation of the redraw concept (see section 3.5.8).

5.2 Memory requirements

DOpE was designed to fulfill the needs of desktop computers as well as portable systems. This fact is reflected by the low memory consumption of both - the windowing server and its clients. The core of *DOpE* requires less than 400 kilobytes of memory. The stripped binary of the windowing server has a size of less than 250 kilobytes - fonts and other required data included! The remaining 150 kilobytes are dynamically allocated. When clients connect to the windowing server and create widgets, additional memory for holding the widget's data is required. The 'terminal' example of section 4.2.1 enfolds one Window, four Buttons, two Grids, one Terminal and two Scrollbar-widgets. Anyhow, the windowing server allocates less than 10 kilobytes of memory to handle these widgets.

When using the offscreen rendering technique (see previous section) additional memory for holding the virtual screen buffer is needed. E.g. a 1024x768 screen mode with 16bit color-depth requires 1024*768*2 = 1536 kilobytes. The same applies on *pSLIM*-widgets, which store their content as raster image.

5.3 Source code complexity

When it comes to security applications - the user must be able to thrust in the windowing server because it is the only instance with access to the screen-buffer (see section 3.1.4). For this reason, the windowing server is structured into independent revisable entities (components) of manageable sizes.

Altogether, the current implementation of *DOpE* consists of circa 9000 lines of code.

Since the assembly of a set of *DOpE*'s components defines the overall functionality of the windowing system it can be adapted to certain applications. E.g. *DOpE* can be down-scaled to a minimalistic but full working windowing system with circa 7000 lines of code (without *pSLIM* and Grid). This low source code complexity enables an exhausting verification of the windowing system's functionality.

6 Conclusions

The concept of *DOpE* respects the different application's needs, that were presented in section 2.1:

- 'Dialogs' can be implemented using non-real-time widgets. They are drawn by the Redraw Manager (section 3.5.7).
- 'Continuous data output' can be displayed using real-time widgets, which are cyclically updated at certain frame rates with the help of the Realtime Manager (section 3.5.10).
- 'Sudden data output' can be achieved by implementing a concurrent drawing real-time-widget (section 3.5.11), which acts alongside the Realtime Manager.

Both presented strategies for the handling of real-time redraws by the Realtime Manager (section 3.5.10) are suboptimal. For practical needs, a reasonable compromise between flexibility and efficiency must be found.

As stated in section 2.2, the long term objective of *DOpE* is the replacement of 'Dropscon'. Although, *DOpE* implements the *pSLIM*-protocol completely, this goal is not reached, yet. Still, *DOpE* is behind 'Dropscon' when it comes to the following aspects:

- *DOpE* is restricted to use the color depth of 16bit. The principle design envisions the support of other color depths than 16bit but it is not implemented, yet. Unlike *DOpE*, 'Dropscon' can handle all hi/true color depths that are supported by the graphics card.
- 'Dropscon' features hardware acceleration support for commonly used graphics cards to speed up drawing operations. Up to now, 'DOpE' only utilises software rendering for its graphical output. As shown in section 5.1 the transfer bandwidth to the screen memory of the graphics card is relatively slow and limits the output speed a lot. When using hardware acceleration functions of the graphics card the pixel transfer from the virtual screen buffer the the graphics card can be circumvented - leading to a rapid speed increase.
- When using *pSLIM* for graphical output under *DOpE* - as done for *L^4Linux*, the content of the *pSLIM* image buffer must be stored twice: at the client's address space (where drawing operations are applied) and at *DOpE*'s address space (from where it is copied to screen when needed).

Both buffers must be held consistent by copying image data from the client to the *pSLIM*-widget of *DOpE*. A faster and more memory-friendly way to display image data is provided by 'Dropscon': The usage of one shared memory block for both parties (server and client). Consequently, no pixel data must be transfered from the client's to the server's address space to keep both buffers consistent. For catching up in this 'discipline' *DOpE* has to be extended by a new 'image mapping'-widget.

There are a lot of ideas for a further development of *DOpE*. The following examples point out the wide range of possible future developments:

- expanding the widget set: The currently available widget types are not sufficient to built common applications out of them. It should be expanded by a variety of standard widgets such as sliders, menus, popups, group frames, radio buttons, edit fields, text fields etc.

- making widgets more intelligent: As stated in section 3.2.2, widgets can provide standard functionality due to their knowledge of the displayed data. So adding standard functionality and drag&drop capability to certain widgets would be a great effort for the end user.

- X-widget: A protocol-widget type, providing the X-protocol could integrate *L^4Linux* applications better into *DOpE*'s desktop.

- 'themes' support: Modern user interfaces mostly support the ability of exchanging their look. *DOpE* could offer a higher degree of customization by providing a 'theme' concept.

- hardware acceleration support: While *DOpE* renders all graphical output to a virtual screen buffer, the hardware drawing routines of certain graphics cards can do this job much faster. A viable concept for integrating hardware acceleration into *DOpE* could be developed.

- desktop environment: A graphical desktop for launching DROPS-programs and accessing the system's components would be a big step forward to the practical use of DROPS for end users.

7 Summarisation

With *DOpE* a slick and extensible windowed graphical user interface is available for the DROPS operating system. It is a foundation for capturing those application fields, where comfortable graphical user interfaces in connection with real-time demands are needed. While being extremely flexible it can be easily tweaked to fit the needs of a wide range of different uses. Its efficient implementation entails only a small memory footprint and a low computational overhead. This fact makes it ideally usable on small devices as well as on desktop computers.

References

[1] "ARTIFACT: An Experimental Real-Time Window System"
1995, John E. Sasinowski, Jay K. Strosnider

[2] "Programming in Tcl/Tk"
1999, Brent Welch
Prentice-Hall, Inc. ISBN: 0-13-022028-0

[3] "The Embedded Linux GUI/Windowing Quick Reference Guide"
http://www.linuxdevices.com/articles/AT9202043619.html

[4] "Real-Time systems"
2000, Liu, Jane W.
Prentice-Hall, Inc. ISBN 0-13-099651-3

[5] "L4 Reference Manual"
Jochen Liedtke
http://os.inf.tu-dresden.de/L4/l4doc.html

[6] "QNX System Architecture"
1999, QNX Software Systems GmbH
http://www.qnx.de/literatur/qnx4_sysarch_de/index.html

[7] "Aqua Human Interface Guidelines"
2002 Apple Computer, Inc.
http://developer.apple.com/techpubs/macosx/Essentials/AquaHIGuidelines/

[8] "Security Architectures Revisited"
Hermann Härtig
http://os.inf.tu-dresden.de/drops/doc.html

[9] "Sicherheit in Rechnernetzen"
2000, Andreas Pfitzmann
http://dud.inf.tu-dresden.de/~pfitza/DSuKrypt.html

[10] "The GRUB manual"
Gordon Matzigkeit and OKUJI Yoshinori
http://www.gnu.org/manual/grub-0.90/html_mono/grub.html

[11] "DirectFB Overview"
2001, Andreas Hundt
http://www.directfb.org/documentation/DirectFB_overview_V0.1.pdf

[12] "Evas Presentation"
David Odin
http://www.enlightenment.org/pages/docs.html

[13] "SDL Library Documentation"
2001, Sam Lantinga, Martin Donlon
http://www.libsdl.org

[14] "Qt Reference Documentation"
2002 Trolltech
http://doc.trolltech.com/3.0/

Glossary

ANSI American National Standards Institute: primary organisation for defining technology standards in the United States of America.

EBNF Extended Backus-Naur Form: standard notation for the description of the syntax of programming languages

FB Frame Buffer: the memory on the graphics card, that stores the physically displayed image data

GLX: a protocol for transfering 3D graphics commands over an X-window's client-server connection

GRUB GNU GRand Unified Bootloader

IDL Interface Description Language: a standard language for describing network transparent interfaces between applications

IPC Inter Process Communication: transfering data between processes running in different address spaces

PostScript: a programming language for describing printed pages. It is commonly used as communication interface to printers, that can execute PostScript code.

RPC Remote Procedure Call

VT100 Virtual Terminal protocol standard: Additional to the output of plain text, VT100 includes a defined set of commands (escape sequences) for controlling the terminal.

YUV color encoding scheme: A color is defined by the components luminance (Y) and chrominance (UV). While the luminance component is stored at the full bandwidth the chrominance components are stored at a lower bandwidth.