TECHNISCHE UNIVERSITÄT DRESDEN

Fakultät Informatik

TUD–FI03–10–September 2003

Norman Feske and Hermann Härtig

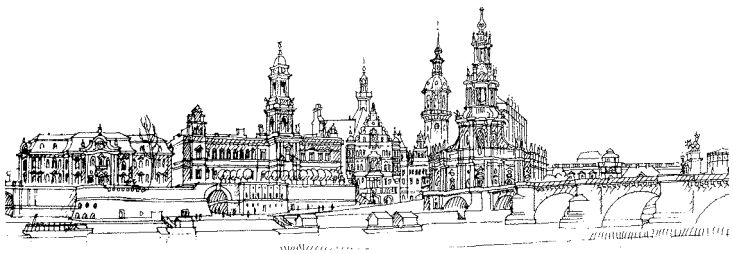
Institute for System Architecture, Operating Systems Group

DOpE — a Window Server for Real-Time and Embedded Systems

Technische Berichte

Technical Reports

ISSN 1430-211X



Technische Universität Dresden
Fakultät Informatik
D-01062 Dresden
Germany

URL: http://www.inf.tu-dresden.de/

DOpE — a Window Server for Real-Time and Embedded Systems

Norman Feske
Hermann Härtig

Dresden University of Technology
Department of Computer Science

{nf2,haertig}@os.inf.tu-dresden.de

Abstract

A window server used in real-time applications should be able to assure previously agreed-upon redrawing rates for a subset of windows while providing best-effort services to the remaining windows and operations such as moving windows. A window server used in embedded systems should be small and require only minimal operating system support, for example just threads and address spaces as provided by micro-kernels.

In this paper, we present the design and an implementation of the DOpE window server that has these properties. The key techniques used are to move redrawing responsibility from client applications to the window server and to devise a simple scheduling discipline for the redrawing subtasks.

1 Introduction

Applications that need graphical representations can roughly be characterized as belonging to one of two domains:

- Interactive applications basically execute a loop where the application waits for user input, changes its internal state, and updates its graphical representation accordingly. Typical representatives of this class are word processors, spreadsheet programs, web-browsers, editors, and other dialog-based applications. The time requirements of these applications are imposed by the user's ability to supply input events fast enough. Thus, they spend most of the time idling. The only requirement with respect to user responsiveness is that the delay between the state change and update of the representation is less than approximately 100 ms.

- Multimedia applications are driven not by user input events but by time. The output of such applications, for example video frames, must be available on the user interface in a periodic manner. Even small delays are perceivable for the user and compromise the quality of service that is expected by the user.

We refer to multimedia applications as real-time (RT) and to other applications as non-real-time (NRT) load.

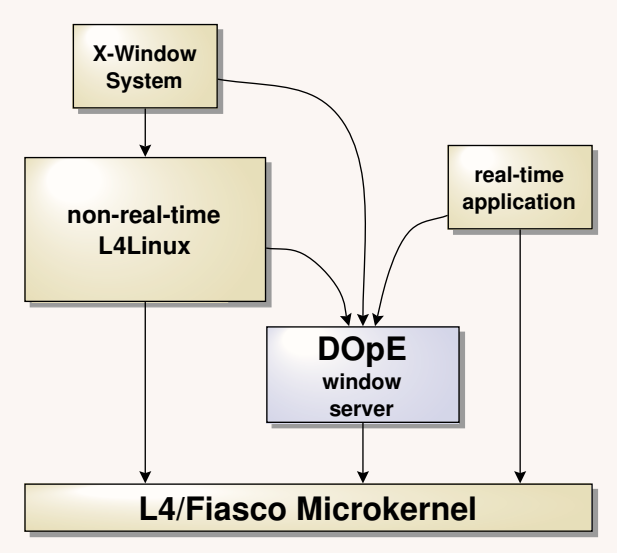


Figure 1: Real-time and non-real-time applications running in one environment

The task of a window server is to multiplex graphical representations of concurrently running client applications onto the screen, which is one single physical resource. The objective is to guarantee the quality of service of real-time client applications even in overload situations. Such overload situations can be induced by massive output of non-real-time applications or by the user who interactively rearranges windows. Window servers that are used in scenarios as described above often exhibit undesirable behaviour. Unawareness of real-time versus non-real-time requirements leads to uniform performance degradation in overload situations, even if there are enough resources available to serve the real-time client applications properly.

If window servers are used in embedded systems with limited resources, they should be small and they should not rely on large operating systems. Even if — for the sake of the argument — we accept that embedded systems needing window server functionality just have to have enough resources for (scaled down) versions of standard operating systems and window servers, reliability and trustworthiness considerations still uphold the requirement for small-sized and self-contained window servers.

The key argument is that currently available operating systems are much too large to be evaluated thoroughly enough to rely on them for critical applications. This argument is especially important for hybrid systems where applications with high real-time or reliability requirements coexist with less critical applications of legacy operating systems on one system.

DROPS[4], the Dresden Real-Time Operating System for which DOpE (Desktop Operating Environment) is made for, is designed to meet that diversified requirement. DROPS is based on the L4/Fiasco [9, 7] microkernel and supports the concurrent operation of real-time applications with L^{4}Linux, a user-level implementation of the Linux kernel[2]. L^{4}Linux and each real-time application run in their own separated address space (Figure 1). Faults occuring in the real-time applications (for example during debugging) do not harm L^{4}Linux, and — much more importantly — L^{4}Linux crashes do not harm matured high-reliability applications after deployment of an embedded system. Detailed evaluations and comparisons with more integrated systems have revealed that the execution and response-time costs of that separation are within small margins acceptable for most applications[2, 10]. To enable sharing of resources other than just CPU and memory, resource managers[5] are being built for other, higher-level resources such as disk and network bandwidth.

DOpE, the window server presented in this paper, is such a resource manager that provides window-redrawing bandwidth. In a general DROPS setting, DOpE supports real-time clients besides L^{4}Linux and XFree86 as non-real-time clients. Since several months, the authors give most of their presentations using DOpE and a simple, non-real-time presentation program running besides L^{4}Linux. The following numbers are given to provide the reader with an idea about the sizes involved: A presentation using DOpE relies on ca 30.000 lines of code (including all supporting software such as L4/Fiasco [9, 7]) in comparison to millions lines of code in other systems that are often used for presentation.

A small window server is of interest in another scenario as well. If applications such as a bank transaction need legacy operating system support, for example for providing network connectivity, and highly trustworthy components, for example for secure storage of keys, then the window server will be part of the trusted computing base of such a system. DOpE is small enough to have the potential for a thorough evaluation.

In this paper, we present the overall architecture (Section 2) of DOpE and then concentrate on the real-time operation. We then give some details about the implementation (Section 3) and compare DOpE wih Artifact [8] (Section 5), the only other real-time window server about which we have non-trivial architectural information.

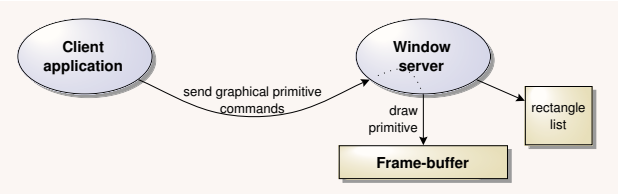


Figure 2: A client application sends graphical primitives to the window server

2 Architecture

2.1 Client-server architecture

A graphical user interface consists of one window server and a set of client applications, which can connect to the window server. The window server stores the information about which screen region belongs to which client application, handles user events, and coordinates redrawing operations.

The window server must enforce the separation of multiple client applications and assure the identity of the client application in the active window. Thus, in environments requiring additional trustworthiness the window server belongs to the trusted computing base.

We make no assumptions about the trustworthiness of a client application. A malicious client application must neither impact concurrently running client applications nor acquire or counterfeir information about other client applications.

2.1.1 Classical approach

Classical window servers, for example the X-Window System, implement a set of graphical primitives and provide an interface used by client applications to send graphical primitive commands.

Figure 2 shows a client application sending graphical primitive commands to the server. The server, in turn, interprets the commands and draws to the physical frame buffer accordingly. Since the window server has the information about the window configuration, it can apply clipping to the drawing functions in the correct way. Thus, the integrity of the user interface can be protected by the window server.

During the execution of redraw operations, for example when the user moves a window, the server needs the active help of its client applications. Only the client application stores the information about its representation on screen. Thus, the client application must provide a description of its representation in the form of graphical primitives to the server for each redraw operation. This is illustrated in Figure 3.

Figure 4 illustrates an exemplary scenario: When the interactive user moves a window, parts of its underlying windows get visible. The window server notifies all affected client applications to trigger a redraw. In return, each of these client applications send a description of their repre-

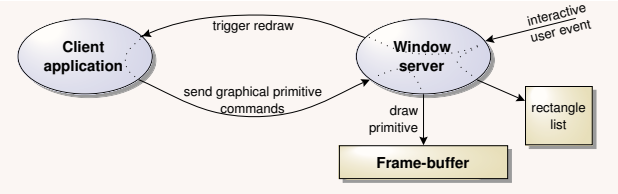


Figure 3: An interactive user event triggers a redraw request

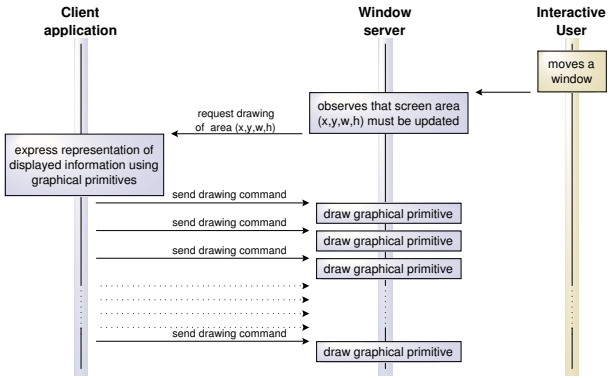


Figure 4: Exemplary scenario: user moves a window

sentation in the form of graphical primitives to the window server who performs the actual drawing operation.

2.1.2 DOpE's approach

DOpE's client-server architecture differs to the classical approach in two ways:

- A client application and the window server share a compact description of the client application's graphical representation in a language with a semantic known to the window server. Thus, the window server can redraw the graphical representation of any client application without the active cooperation of the affected client application.
- Client applications trigger redraw requests at the server instead of supplying graphical primitives directly.

Figure 5 illustrates this architecture. In comparison to the classical approach (see Section 2.1.1), our approach leads

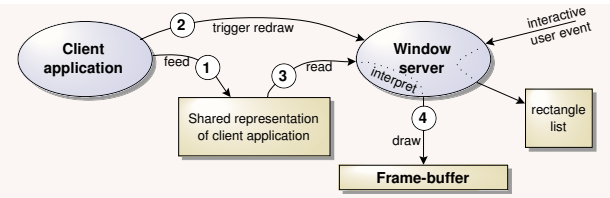


Figure 5: The client application and the window server share a compact description of the client's representation

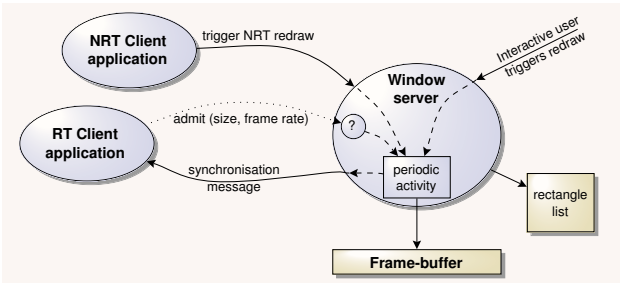


Figure 6: Example scenario showing real-time-admission, synchronization feedback and the triggering of non-real-time redraw requests

to a different mode of operation: A client application updates the shared representation and then triggers a corresponding redraw operation in the window server.

With regard to real-time, our design allows local scheduling of redraw operations within the window server. The redraw operations are designed such that their execution time is known in advance.

2.1.3 Redraw for real-time client applications

For the output of real-time client applications we introduce a periodic activity in the window server that triggers redraw operations in a defined (and predictable) way. We discuss in Section 2.2 how we use this periodic activity in the window server.

To establish a periodical real-time output, a client application has to request an admission at the server by specifying the size and update frequency of the desired output area (widget) on screen. Figure 6 shows an exemplary admission scenario.

After a successful admission, the server refreshes the negotiated widget periodically, based on the known representation of the client application's data, and optionally informs the client about a completed redraw using synchronization messages. It is up to the real-time client application to timely feed its current state to the graphical representation which, in turn, is shared with the window server.

2.2 Real-time redraw engine

As a result of the design presented in Section 2.1 we have the following situation on the server side:

- Non-real-time redraw requests of any complexity can be requested at any time.
- There is a fixed set of real-time redraws to be executed periodically.
- The server can execute redraw operations without the active help of its client applications. Thus, redraw operations can be scheduled based on known execution times.

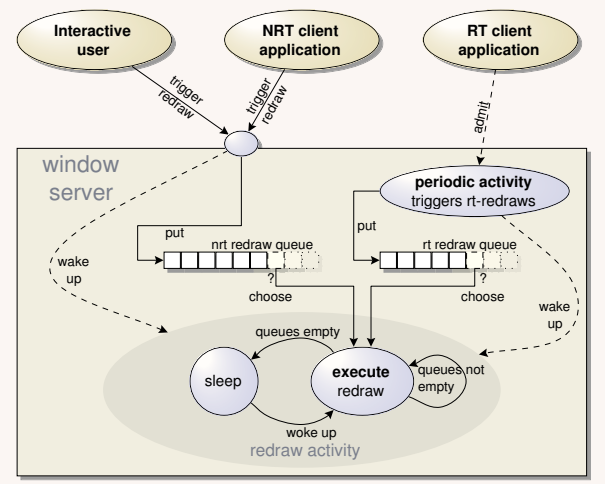


Figure 7: Illustration of redraw queues

- We regard redraw requests triggered by the interactive user as non-real-time redraw requests.

Additionally, we consider the following constraints:

- Redraw operations may be time intensive — for example when pixel data must be sent over the system bus to the graphics subsystem.
- Once started, a redraw must be completed before a new redraw operation can start. However, redraw operations can be split into smaller ones before execution.
- The physical screen is an exclusive resource. Only one activity (thread) can manipulate the screen at a time.

The latter two constrains are necessary to enable the (future) usage of hardware-accelerated graphics cards that can carry out only one graphics operation using one current clipping area at a time.

In this section we will present different strategies for serializing redraw operations leading to a design that obeys the following contraints:

- Real-time widgets are periodically updated at defined update rates.
- The graphical appearance and the stored representation of non-real-time client applications is consistent and current.
- The window server is immune to denial of service attacks and guarantees accessibility.

The simplest way to process redraw requests is shown in Figure 7. All incoming redraw requests are stored in either a real-time redraw queue or a non-real-time redraw queue.

There is one redraw activitiy in the server that processes all queued redraw requests. In Artifact [8] this approach is called kernelized monitor.

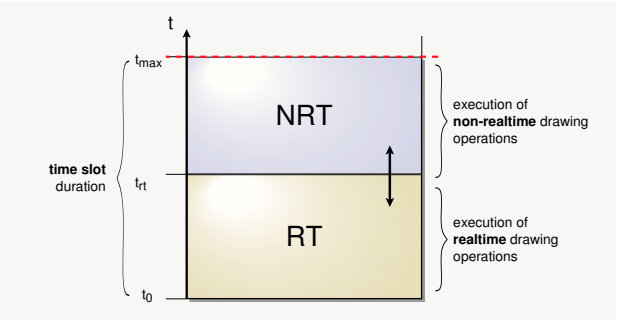


Figure 8: Usage of a time slot for real-time and non-real-time redraw operations

The redraw activity of the window server always processes the real-time redraw operations first, followed by the redraw requests stored in the non-real-time queue.

To prevent overload situations caused by real-time redraw requests, the window server applies admission tests.

However, overload situations caused by non-real-time client applications can lead to unbounded latency of non-real-time redraw operations. We solve this problem with the following technique: Each redraw request refers to one specific window. If a redraw requests comes in for a window for which an entry in the redraw queue already exists, both redraw requests are merged and stay at the queue position of the old request. Thus, the maximum size of the queue equals the number of windows on screen. The sum of the queued pixels to redraw is limited to the number of pixels on screen because all queue entries refer to disjoint screen areas. For a given window configuration, the maximum latency for displaying any kind of information on a screen equals the time that is needed to update the entire screen. This maximum latency defines the guaranteed timeliness of non-real-time client applications.

For determining the chronological order of processing the queued real-time redraw requests an assignment of priorities to real-time redraw requests was proposed in Artifact [8]. The drawback of this method is a priority inversion problem: A currently processing long-taking non-real-time redraw operation or a real-time operation with a lower priority can delay a high-priority real-time redraw operation.

To avoid this problem, we use a time-driven approach for processing real-time redraw operations, which nicely fits with the relatively static periodicity of real-time graphical-user-interface (GUI) client applications. The periodic redraw activity is subdivided in a fixed number of time slots. Each time slot can hold one or more real-time redraw requests. If time is left after the execution of real-time redraw operations it is used to execute requests from the non-real-time redraw queue (Figure 8). Thus, the completion of real-time redraw operations is guaranteed.

When a real-time client application requests a periodic redraw, the window server assigns time slots and allocates the time that is needed for this periodic redraw.

We have chosen a time slot periodicity of 150 Hz to enable redraw rates of 50 Hz, 30 Hz (NTSC), 25 Hz (PAL)

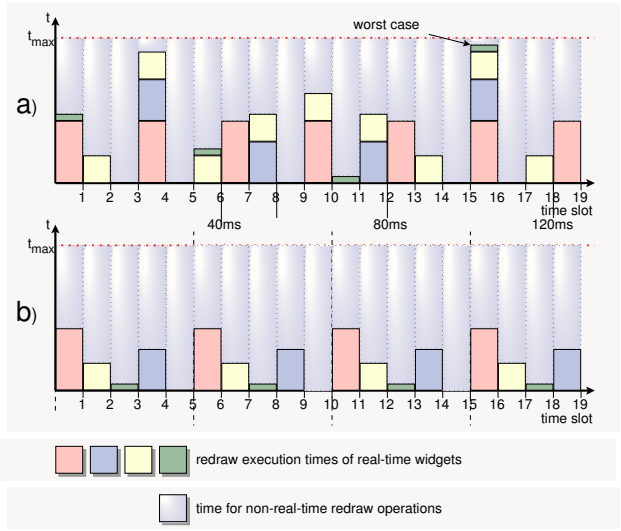


Figure 9: Two opposing real-time redraw strategies

Figure 10: Screenshot of DOpE running two real-time client applications and L4Linux

and so on (any whole numbered fraction of 150 Hz). These update rates are typical for common multimedia applications. Thus, this degree of flexibility is sufficient for the applications we are aiming at.

There are two immediate consequences of this approach:

- The window server must assign real-time redraw operations to time slots at the admission of new real-time client applications.
- To make efficient use of each time slot the space left in time slots must be padded with non-real-time redraw operations. Since redraw operations cannot be preempted, non-real-time redraw operations must be split into smaller ones before starting the execution. We use a runtime-measurement-based heuristic function to estimate the number of pixels that can be drawn in a given time and then split redraw operations into smaller ones that fit in the space left in the time slots.

Two opposing strategies for assigning real-time redraws to time slots and implementing admission tests are illustrated in Figure 9. The figure shows the real-time load per time slot in an exemplary situation with four real-time widgets (marked by different colors). The opposing strategies are:

a) The real-time client application can individually chose any whole-numbered fraction of 150 Hz as the update frequency for each real-time widget. The window server accepts new real-time widgets as long as the sum of all real-time redraw execution times is lower than the time slot duration. Even in the worst case — when all redraws must be executed at one time slot — the sum of all redraws is still lower than the time slot's duration. One drawback for practical use is that the decision about new widgets acts upon a hypothetical worst case. Thus, the admission criterion to decide about new real-time widgets is far too pessimistic.

b) In contrast to the previous strategy the real-time client application can only use one update frequency for real-time widgets. In this case, the window server can execute real-time redraw operations strictly interleaved. The window server accepts new real-time widgets as long as there is a time slot which can hold the new real-time redraw operation without exceeding the time slot's duration. As indicated in Figure 9(b) this strategy is capable of handling more real-time widgets than the first strategy without exceeding the time slot's duration. The drawback of this solution is the lack of flexibility of update frequencies.

3 Implementation

We implemented the conceptual ideas discussed in Section 2 in the window server DOpE (Figure 10).

The DOpE window server runs on the top of the L4/Fiasco [9, 7] microkernel which provides the following functionality:

- Activities (threads)
- Address spaces
- Inter-address-space communication and shared memory [10]

3.1 DOpE server structure

Figure 11 illustrates the structure of the DOpE window server. Currently, we use software rendering routines (Graphics layer) to access a VESA frame buffer. It manages clipping and contains functions for drawing graphical primitives such as scaled images and text.

The mouse and keyboard drivers are ported from Linux to the L4/Fiasco [9, 7] platform. On top of the input device drivers there is an input abstraction layer with support for different keymaps. DOpE handles user input events in a non-blocking way. Thus, the interactive user does not impact the continuous output of real-time client applications

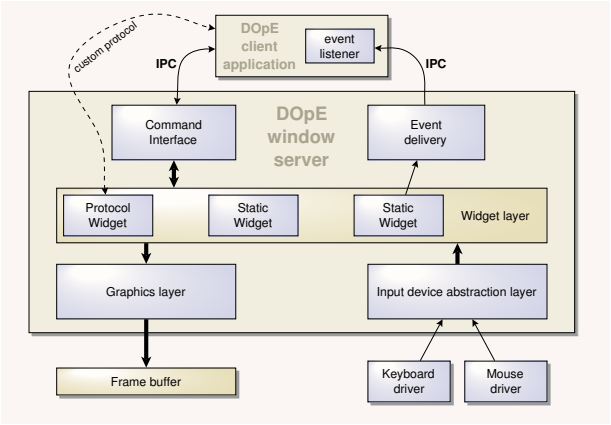


Figure 11: Schematic overview of the DOpE window server

when interacting with the window server — for example by moving a window.

The representation of client applications on the DOpE window server is based on widgets and their relationship to each other (topology). A widget defines the graphical representation and the response to user interactivity of a dedicated type of data or protocol. As illustrated in Figure 11, protocol widgets can establish a dedicated communication channel to the client application, for example via shared memory. This way the representation of a client application can be shared with the window server asynchronously.

The widget layer is built on top of the Graphics and Input abstraction layers and contains the implementation of the following widget types:

- Window, Button, Scrollbar, and Frame as the basic primitives of the window server
- Grid allowing the arrangement of multiple child widgets to be aligned in a rectangular grid
- Container allowing the placement of multiple child widgets via pixel coordinates
- Terminal for textual output with support for a subset of VT100 escape sequences

Additionally, we implemented two protocol widgets with separate communication interfaces to their associated client application:

- pSLIM: pSLIM is a derivate of the SLIM [11] protocol. The client application can send graphical primitives, for example filled rectangles or pixmaps, to a pSLIM widget which acts as a virtual canvas. It supports the display of RGB and YUV color space coded data and also provides support for text output.
- VScreen implements a shared-memory buffer with a pixel-based representation of client data between the client application and the window server. It can be used as a periodically updated real-time widget in a way that is described in Section 2.1.3. The client application and the interactive user can vary the size of each VScreen widget resulting in a scaled output of the represented pixel data.

Similar to the approach of Tcl/Tk [12], user interface widgets are created and configured by using a text-command-based communication interface to the client application.

Input events are primarily handled by the affected widgets. In turn, widgets can forward events to the client application via the event delivery component (see Figure 11).

3.2 L4Linux, XFree86 and other non-real-time client applications

L^{4}Linux [2] is a user level implementation of the Linux kernel on top of the L4/Fiasco [9, 7] microkernel.

We implemented a driver module for the XFree86 [13] X-Window System which forwards the graphical output of the X-Window System to a DOpE widget (pSLIM). Thus, we are able to run the broad range of non-real-time X11 applications together with native real-time applications in one environment.

With Presenter, we implemented a simple presentation program that displays slides as BMP images and let the interactive user chose the currently visible slide.

3.3 Real-time clients

We implemented two real-time client applications that make use of the VScreen widget type to display a continuous stream of pixel data:

- VScrTest is a demonstration program that calculates four different graphical effects: a 3D particle effect, a bumpmapping effect, a voxel landscape and a feedback effect. The effects are rendered into a shared memory buffer which, in turn, is displayed by the DOpE window server at a constant frame rate of 25 frames per second.
- USB WebCam is a client of a Linux USB-stack which was ported to the user level of L4/Fiasco [9, 7]. It constantly displays the image which is sampled from a USB-webcam at a frame rate of 25 frames per second.

Both client applications make use of the real-time features of DOpE. Thus, the rearrangement of windows and the graphical output of concurrently running client applications have no impact on the constant update rate of the real-time clients.

4 Evaluation

4.1 Performance

To put the overhead of the windowing system in perspective, we estimated the needed bandwidth for drawing a scaled image as an instantly used graphics operation (for

action	MByte/s	µs/16-bit
main memory read	176.53	0.0108
main memory write	166.55	0.0114
screen memory read	6.12	0.3116
screen memory write	31.11	0.0614
copy main memory to screen	28.48	0.0670

measured on an AMD Duron 700Mhz PC equipped with a Matrox G450 graphics card

Figure 12: Practical memory access measurements

example the output of a button's background or pSLIM-widget) in DOpE and compared the estimated bandwith with a real-life measurement. Figure 12 shows a table of the measured memory bandwidths on our AMD Duron test computer (700Mhz, 128MByte RAM, AMD chipset).

All transfers were done 16-bit wise because a 16-bit value is equivalent to a pixel with 16-bit 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. Because of this, operations on screen memory must be limited to a minimum. Thus, DOpE uses an offscreen rendering technique (double buffering). A virtual screen buffer is kept in main memory. All graphical operations are quickly applied to the virtual screen. When the graphical operations are finished, the affected area is copied from the virtual screen buffer to the screen memory. Thus, the bottleneck to the screen memory must be passed only once per pixel.

For drawing a scaled image, the interesting values 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 16bit offset from a scale table: $0.0108\mu s$
- reading a 16bit pixel from source image: $0.0108\mu s$
- writing a 16bit pixel to virtual screen: $0.0114\mu s$
- copying a pixel from the virtual screen buffer to the physical screen memory: $0.067\mu s$

The sum of these values is $0.1\mu s$. This, theoretically $1/0.1 = 10$ pixels can be drawn during $1\mu s$. 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. DOpE 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 results of the real-life measurements after working with DOpE for a while were 8 $pixels/\mu s$ average ratio and 4 $pixels/\mu s$ minimum ratio. Thus, the average ratio reaches nearly the estimated value of 10 $pixels$. This is a strong indication for the efficient implementation of the graphical output routines and the low overhead caused by the window server. The minimum ratio occurs when multi-layered widgets such as stacked layout-widgets must be drawn.

4.2 Source-code complexity

In Section 2.1 we claimed that the window server belongs to the trusted computing base. Thus, it is important to keep the source-code complexity of the window server low.

Altogether, the current implementation of DOpE consists of about 10,000 lines of code. This code includes:

- Widgets: Window, Scrollbar, Frame, Grid, Container, Button, Terminal, VScreen, pSLIM
- Support for proportional bitmap fonts
- A keymap component to support different keyboard layouts
- All graphical rendering routines
- A command-interpreter-based client-server communication protocol
- Abstractions for shared memory, screen, threads, timers, input devices

Internally, DOpE is structured in a component-based way, which allows a high degree of customization for special applications. By leaving out higher-level widgets such as Grid-layout, DOpE can be scaled down to a minimalistic but fully working window server with about 7,000 lines of code. In secure system architectures as described in [6] the graphical user interface is part of the trusted computing base. DOpE's extremly low source-code complexity enables an exhausting verification of the window server and makes it viable for secure platforms.

The size of the executable binary of DOpE including input device drivers, graphics routines, graphical data (four bitmap fonts), and all widget types described in Section 3.1 is 250KByte. The core functionality (without input drivers and the L4 Environment) of DOpE enfolds a binary size of 150Kbyte.

5 Related work

The discussion of Artifact [8] in relation to DOpE nicely allows to explain some principle approaches of DROPS in general [5, 1]. DROPS provides resource managers that map underlying, basic resources to higher-level ones. For example, DOpE maps CPU cycles and main memory to rr-edraw bandwidth.Thus, when a new real-time-window is

requested, DOpE checks whether or not the acquired basic resources suffice. This probably leads to less flexibility, but to lower overall admission and scheduling complexity.

New types of resources, such as window-drawing bandwidth, are scheduled and admitted as such locally within the resource manager. Basic resources on the other hand are scheduled and admitted for the resource managers.

Artifact [8] works principally different. It dynamically creates client and server real-time models and then makes global admission decisions (so we understand the principles described in Artifact [8]). In quite some details, there are similar implementation structures: Artifact [8] has (in their first version) a single redraw-execution activity and refers to that version as a kernelized approach. They run it at the highest priority. In later versions they use multiple threads for multiple real-time client applications and derive priorities from real-time client application priorities.

DOpE is based on a localized, simple, essentially time-driven scheduling approach because it gives us a clear perspective to include DROPS' quality assuring scheduling [3] with its mandatory and optional parts of real-time applications as a relatively simple local scheduling discipline. DROPS uses a similar approach for other resources such as disk bandwidth: a disk-bandwidth manager requests its CPU resources as a resource manager as a whole and uses them to perform local disk-request scheduling, a daunting task for a global scheme to schedule basic resources.

6 Conclusion

With DOpE we have a window server which is capable of guaranteeing quality of service for multimedia applications with continuous output while efficiently utilizing the remaining CPU resources to serve non-real-time client applications. It is a foundation for covering those application fields, where graphical user interfaces in combination with real-time demands are needed.

The implementation of DOpE entails a very small memory footprint and a low computational overhead. Its component-based internal structure allows a high degree of customization and thus makes it viable for a broad field of applications, ranging from desktop computers to small portable devices.

Currently, there is work in progress to create a graphics-acceleration driver infrastructure for DOpE to boost the overall performance of the graphical output.

For dialog-based applications the current widget set is not sufficient, yet. We are busy implementing standard widget types such as radio-buttons, menus, text-edit fields as part of the DOpE window server. Then, client applications can use the server-side widgets for common GUI controls rather than providing own pixmap-based widget implementations.

References

[1] Robert Baumgartl, Martin Borriss, Hermann Härtig, Claude-Joachim Hamann, Michael Hohmuth, Lars Reuther, Sebastian Schönberg, and Jean Wolter. Dresden Realtime Operating System. In *Proceedings of the First Workshop on System Design Automation (SDA'98)*, pages 205–212, Dresden, March 1998.

[2] Martin Borriss, Michael Hohmuth, Jean Wolter, and Hermann Härtig. Portierung von Linux auf den μ-Kern L4. In *Int. wiss. Kolloquium*, Ilmenau, September 1997.

[3] C.-J. Hamann, J. Löser, L. Reuther, S. Schönberg, J. Wolter, and H. Härtig. Quality Assuring Scheduling - Deploying Stochastic Behavior to Improve Resource Utilization. In *22nd IEEE Real-Time Systems Symposium (RTSS)*, London, UK, December 2001.

[4] H. Härtig, R. Baumgartl, M. Borriss, Cl.-J. Hamann, M. Hohmuth, F. Mehnert, L. Reuther, S. Schönberg, and J. Wolter. DROPS: OS support for distributed multimedia applications. In *Proceedings of the Eighth ACM SIGOPS European Workshop*, Sintra, Portugal, September 1998.

[5] H. Härtig, L. Reuther, J. Wolter, M. Borriss, and T. Paul. Cooperating resource managers. In *Fifth IEEE Real-Time Technology and Applications Symposium (RTAS)*, Vancouver, Canada, June 1999.

[6] Hermann Härtig. Security Architectures Revisited. In *Proceedings of the Tenth ACM SIGOPS European Workshop*, Saint-Emilion, France, September 2002.

[7] Michael Hohmuth. The Fiasco kernel: System architecture. Technical Report TUD-FI02-06-Juli-2002, TU Dresden, 2002.

[8] Jay K. Strosnider John E. Sasinowski. Artifact: An experimental real-time window system. 1995.

[9] J. Liedtke. On μ-kernel construction. In *15th ACM Symposium on Operating System Principles (SOSP)*, pages 237–250, Copper Mountain Resort, CO, December 1995.

[10] F. Mehnert, M. Hohmuth, and H. Härtig. Cost and benefit of separate address spaces in real-time operating systems. In *Proceedings of the 23th IEEE Real-Time Systems Symposium (RTSS)*, December 2002.

[11] Brian K. Schmidt, Monica S. Lam, and J. Duane Northcutt. The interactive performance of slim: A stateless, thin-client architecture. In *17th ACM Symposium on Operating System Principles (SOSP)*, Kiawah Island, SC, December 1999.

[12] Tcl/tk, scriptics website. URL: `http://www.scriptics.com`.

[13] Xfree86 website. URL: `http://www.xfree86.org`.