Belegarbeit

The L4 Font Server

Henrik Malecha

25th July 2005

Technische Universität Dresden
Department of Computer Science
Operating Systems Group

Supervising Professor: Prof. Dr. rer. nat. Hermann Härtig
Supervisor: Dipl.-Inf. Norman Feske, Dipl.-Inf. Christian Helmuth

Erklärung

Hiermit erkläre ich, dass ich diese Arbeit selbstständig erstellt und keine anderen als die angegebenen Hilfsmittel benutzt habe.

Dresden, den 25th July 2005

Henrik Malecha

Contents

List of Figures

1 Introduction

1.1 Overview

Font management and electronic typesetting is a widespread field. Different fonts for different languages, like English, Greek or more complex types like Arabic or Asian one's with a huge amount of letters make it a big challenge for programmers that want to localize their programs to other languages. Fonts must be scalable and should always look the same, regardless of their size. Take for example the traditional X fonts used in terminals. They always have the same distance and are only fixed font sizes, and that is not desirable in a contemporary typesetting.

To the humans eyes, font files with variable symbol distances are more readable and comfortable. Furthermore, a today's font has to be scalable to any size and modern font renderers are often required to support anti-aliasing for letter smoothing. Font formats like Adobe's Type 1 or Apple's True-Type present such more sophisticated ways of font handling.

Nowadays, computer systems are often required to use these scalable, variable-spaced fonts, because fonts are an integral part of a computer system, as they are used to present information to the user. When reading or creating documents, like manuals or letters, surfing in the Internet or writing e-mails or even if you work with the operating system, you are always confronted with fonts.

Moreover, as there are many different languages on the world, using various scripts that vary highly in their count of symbols, it is a real challenge to cope with internationalization. In the early years, computers generally supported the ASCII character set. This provides the user with Latin characters, necessary for typing in the English language. ASCII only consists of 127 characters, which means that 7 bits of a byte are used in that code. This has historical reasons, because, in the beginning of the usage of ASCII, there were computers that handled 7-bit words. Later, the eighth bit was often used as a parity bit.

The problem is that just the fewest languages have such a small number of letters as English has. Perhaps, one of the most extreme example is the Chinese type with its over 70000 symbols. Alternative methods of character representation had to come up, which resulted in the *Unicode Project* and the *ISO 10646* standard, for example.

My thesis deals with the font support for the *L4 environment*. The *L4* system has been developed by the OS Group of the Departure of Computer Science at the Dresden University of Technology. It is build on top of a so-called microkernel, *Fiasco* by name. For more information about the system, look at [L4e01].

The goal of the thesis is to develop a server that provides clients with fonts. Concretely, this means that a client can request rendered bitmaps of characters in a font he chooses out of a font list, the server offers. Moreover, the server shall be able to handle international character sets, not only the standard ASCII sets.

1.2 Layout

Section 2 deals with related work in the field of fonts. It starts with a short overview about the history of fonts that is followed by an introduction into the font terminology, which is necessary, because there are some specific terms. Afterwards, some basic technologies and concepts, like *FreeType2* and *Unicode* are introduced, that will be used within the *L4 Font Server*. The section closes with a look at font solutions under Linux.

In Section 3, I describe the design alternatives and the final decisions about the concept of a font server implementation. The internal and external architecture are introduced and basic questions of how to use the server are answered.

On the base of that, the Section 4 is a more in-depth description of several important aspects of the implementation.

Section 5 tries to evaluate the implemented server. Questions of “what is good” and “where are the drawbacks” are considered.

Finally, a conclusion in Section 6 summarizes the features already included and what has to be done in the future development.

2 State of the art

In this chapter I give an overview of related work and state-of-the-art solutions in the field of font handling and rendering. After a short overview about the historical origins of fonts and type-setting, an introduction in the terminology and the basics of fonts follows. The concept of True-Type fonts has to be clarified, because it is necessary to understand the work of the font server. As well an explanation of *FreeType2*, an important core element of my implementation is given. For the internationalization task, I provide an insight into the *Unicode Project*. Afterwards, I describe how the Linux system deals with font management and challenges like scalable fonts and internationalization.

2.1 History of fonts

Like many concepts that found their way into computer technology, fonts and typesetting have already been existing for a long time. As mentioned in [Fon05], the term font is derivated from the French word *fonte* (i. e., something that has been melt).

The oldest verified notes about printing with movable type were found in China. In the 11th century, a Chinese called Bi Sheng invented movable letters. Later, in Europe, the German printer *Johann Gutenberg* also invented the movable type or *foundry type* and thereby revolutionized the printing technology. In *foundry type* each character was cast into a block of metal, which was an alloy of lead. Every block contains one letter, number or any other symbol. The look of the images on that blocks was called *typeface*, which is still used today. By combining the blocks, one could form pages that were inked and then used for fast reproduction. By that, books and information spread all over Europe and layed one of the cornerstones of renaissance.

This technology was used for hundreds of years, until the middle of the 20th century. In the 1950s a new method came up by the *photographic typesetting*, which had the advantage to be scalable. Therefore, the fonts were saved on discs or rolls of film.

But the era didn't take that long, since, in the 1980s, the digital typography began their relentless conquest of typesetting. Thus, nowadays, digital typesetting is widespread and, in smaller dimensions, usable by everyone, who owns a computer and a printer.

Nowadays, when we speak of fonts, we usually mean computer fonts saved in a file. We use them to present information on screens, to print documents and so on. Many terms in digital type-setting have their origin in the historical technologies. In the next section, the most important of them are explained.

2.2 Font Terminology

On computers, fonts are an important part of the system, as they are used to display information in text-only environments as well as in systems with a graphical user interface. Furthermore,

more sophisticated use of fonts, for example WYSIWYG[1] text editors or desktop publishing software, is widely spread and quasi-standard. Because of that, an efficient representation guaranteeing high output quality is desired. Therefore, font designs are defined in files, which are used by diverse font handlers, which can read these files and use the fonts for character rendering. This section gives a short overview about the subject, for more details, consider [FTy05a] and [MST05].

Several font formats are available for use in the computational type-setting. There are classic bitmap fonts, used in old terminal programs, as well as formats with a vectorized representation of the font's characters. In this section, I will describe the structure of these formats and the important terms to deal with them.

Basically, a font is a collection of images that represent letters. Such an image is called *glyph*. A single font holds glyphs with common attributes, for instance the *look*, the *style* or *weight* (a bold font has a higher *weight* than a normal font). The *Courier* font, for example, contains glyphs of characters that all look like the script of a typewriter.

A font has a family, for example Roman, Sans or Courier. Everybody who has already worked with a WYSIWYG text editor knows that such a font may have more than one style (i. e., bold, italic). So, there has to be a way to carry more than one glyph set in a font. The so-called faces take care of that aspect. There can be more than one face for a font family, thus, a font file is often referred to as a font collection. They can differ in the style, or in other attributes, in a way that they still can be recognized as members of one family (i. e., the look is similar).

Depending on the format, a single font file can include one or more faces. But it is also possible to spread a font family's faces over more than one file. That's why it is often slightly imprecise, when people talk about *fonts*. Usually, they use *font* to circumscribe the term *font family*. For instance, when you select *courier* in a common text editor, you select the font family and think of it as the font. But, in another situation, you have a single font file, with only one face of the family and speak of this face as a font, too. Although there can be other font files that also carry faces of the same family, but maybe of another style. To avoid misunderstandings in my thesis, I will use the term *font* as a synonym for *font family*.

In the most font formats, a font file contains a set of glyphs and several character maps. The glyph set consists of representations for characters that are supported by the font (e. g., Latin characters and Arabic figures). For a single letter there can be more than one glyph in a file, depending on the context and script. In several fonts there may be pre-rasterized bitmaps for the most-used sizes. The *character maps* assign character codes to glyph images. The character codes differ between representations like ASCII or Unicode encodings. A single font file may contain more than one character map.

A glyph representation also holds geometric information, like a bounding box and advance values, necessary for displaying them so that they are written in one line and have the correct distance between them.

2.2.1 Outlines and geometric aspects

This section focuses primarily on scalable fonts, like True-Type or Type1. Non-scalable fonts have the grave disadvantage that they contain only rasterized bitmaps of glyphs for fixed sizes.

[1] WYSIWYG: abbreviation for *What You See Is What You Get*

Within such font files, there may be more than one bitmap for one glyph representing different, but fixed sizes. If you wish to use another size than available, an existing bitmap has to be resized, which reduces quality by giving it a more or less grainy look.

Scalable fonts, on the other hand, use so-called outlines, which are vectorized representations of symbols. Curves of different types are used for that (e. g., B-Splines). The advantage of vector images is that the image is not represented in a pixel-fixed grid, that means in the image it is not said that pixel has the color black. Instead the curves are defined by control points in an abstract coordinate system. The curves can be scaled to any size and afterwards, a renderer *converts* it into a bitmap image of a certain size.

Resolution Between devices like monitors and printers, the resolutions are not the same. Pixels on one screen have another size than on another display or a printed paper. Device resolutions are given in *dots per inch*, short: *dpi*. For instance, a printer has a resolution of 300x600 dpi (i.e., 300 dpi horizontal and 600 dpi vertical) and a monitor may have 96x96 dpi. Because of the differing resolutions, it is not useful to specify font sizes in pixels and that is why another measuring is introduced: the point size. The point size is different from the pixel size and 1 point equals 1/72th of an inch. So, the point size is not device-specific and we can easily calculate the pixel size (i.e., the count of pixel used to draw the point) by the help of the point size and the device resolution:

```
pixel_size = point_size * dev_resolution / 72.
```

Since the horizontal and vertical device resolution are not necessarily the same, a device resolution value consists of values for both dimensions and size conversion have to be done individual for both of them.

Vector Outlines An outline is a collection of so-called *contours*, which are closed paths of *line segments* and *Bezier arcs* delimiting inner and outer regions of the glyph. The Bezier arcs can be defined by quadratic polynomials, as done in the True-Type format, or in cubic curves as in Type 1. A simple glyph has only one contour (7), more complex ones may have more contours (B). Control characters will be mapped to a glyph without contours and composite characters are combinations of two or more basic glyphs.

Contours are sequences of consecutive points. A straight line is defined by two points laying directly on the outline. Bezier arc definitions use two types of points - the normal points laying on the curve and the control points that don't lay on the curve. As the name indicates, they are used to control a curve in its direction and concavity, you can think of it as parameters. Following a contour by increasing point numbers, it is stated that the right side of it will be the filled side (i. e., the body).

EM squares The *em square* has its origins in the traditional type-setting. Formerly, it was a tablet on which the glyphs were drawn. Now, in the computer font terminology, it is almost the same, but imaginary. It works as an orientation for the size and alignment of glyphs. In the *em square*, the smallest unit of measure is the so-called *font unit*.

The dimensions of the em square usually include the whole body height of a font and a little extra space below and above, so that lines of text not collide with each other. It is important to mention that the font designer is not restricted to the em square. It is thus possible to extend a glyph beyond the square's size.

Grid The *font units* define a grid in the em square that is used to address the points of a glyph outline. The grid is a two-dimensional coordinate system in which the x axis describes horizontal movement and the y axis the vertical movement. Each point of the outline must be defined within the range of -16384 and +16383 font units. The size of the points is determined by the resolution that is chosen.

The granularity of the em square is given in font units per em. By that, the em square is divided into small units. The higher the number of units, the higher is the precision of the addressing. The font units within the em square define a relative addressing not depending on the point size. That means, if the point size is increased, the em square will be scaled to its new size but the count of units in that square will not change. Because of that, the whole outline of the glyph will be scaled relatively to its new size. For conversion, the following formulas can be used:

```
pixel_size  = point_size * resolution / 72
pixel_coord = grid_coord * pixel_size / EM_size
```

Hinting In the file of a scalable font, the glyph is stored as an *outline* with its dimensions defined in font units. It has to be scaled to the given resolution before it can be rasterized into a bitmap. Unfortunately, the scaling comes along with undesirable side-effects, like stems with different sizes in height and width.

Therefore, the scaled glyphs need to be aligned along the pixel grid, which is referred to as *grid-fitting* or *hinting*. Three different techniques are established:

- explicit grid-fitting:

  True-Type is a format that uses the so-called *explicit grid-fitting*. For that purpose, in True-Type there is a stack- based virtual machine defined. A font designer can write small programs with about 200 opcodes. As a consequence, each glyph definition contains an outline and a small program for control of the grid-fitting. Thus, the grid-fitting is left to this font-embedded program.

- implicit grid-fitting:

  On the other hand, font formats like Type 1 add additional tags to a glyph definition, called *hints*. They define features, used for grid-fitting. In contrast to the explicit grid-fitting, the font renderer has to interpret these features and take care of the grid alignment, which may lead to a different hint-feature interpretation.

- automatic grid-fitting:

  For automatic grid-fitting, no extra hint information is needed. The renderer will guess hinting features, thus making the quality of the grid-fitting highly dependent from the renderer's automatic hinting abilities.

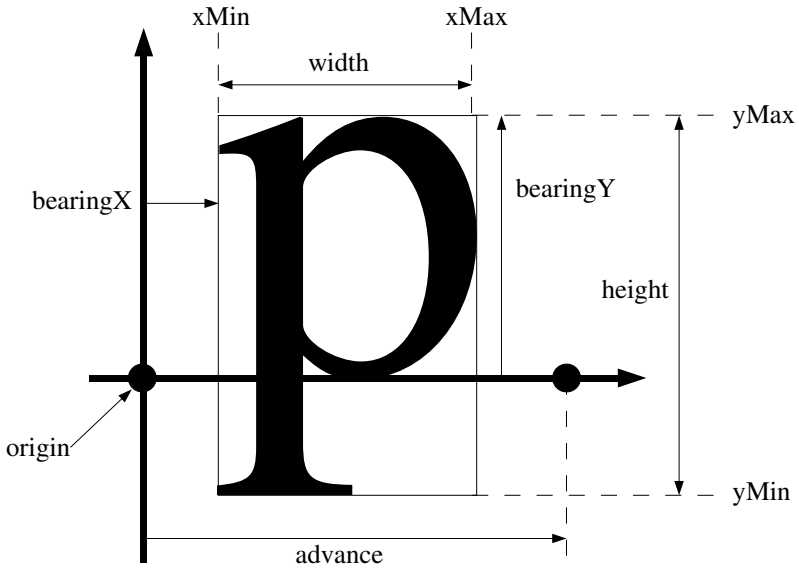

Figure 2.1: Glyph metrics
[Glyph metrics]

The differences of these three approaches lie in the performance and resulting quality. Explicit grid-fitting provides a high output quality, but, on the other side, faces bad performance, because running a virtual machine is time-expensive. The other two approaches cause inconsistencies in the resulting output bitmap, because hinting relies on the font renderer and not on a glyph-specific mini-program.

2.2.2 Glyph metrics

Drawing a collection of glyphs as lines of text, the corresponding bitmaps are placed on the so-called *baseline*. It depends on the language, whether the baseline is horizontal or vertical. In the following explanations, I will narrow the view to the horizontal case.

All glyphs of a line of text rest on this baseline. The pivotal point for glyph drawing is the *pen position* on the baseline. The bitmap will be drawn from this point. Incrementing between two consecutive glyphs will be done by the *advance width*, which is a glyph-specific value.

The following paragraphs give an overview about important terms for dealing with glyph metrics (see also Figure 2.1):

Typographic metrics The *ascent* and the *descent* are the distances from the baseline to the highest, respectively to the lowest outline point in the font. The *line-gap* is the distance that must be placed between two consecutive lines of text.

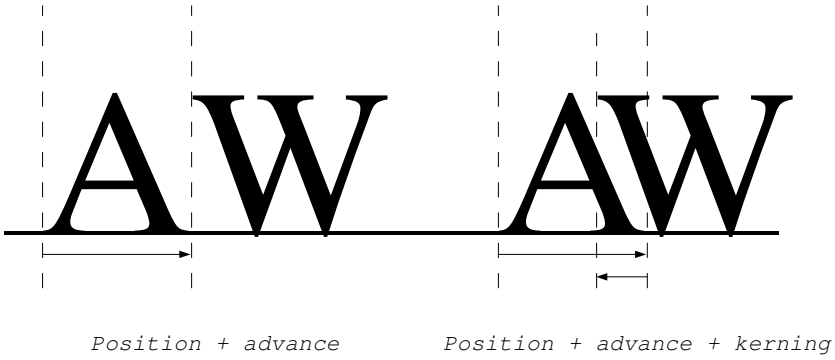

Figure 2.2: Without and with kerning
[Kerning]

A *bounding box* (*bbox*) encloses a glyph as tight as possible. It is defined by the values `xMin`, `xMax`, `yMin`, `yMax`, which can be defined in font units or in pixel values.

Bearings and advances `bearingX` defines the distance from the current pen position to the left side (`xMin`) of the glyph's *bbox*. Analogously, the distance from the baseline to the *bbox's* upper edge is referred to as `bearingY`.

The value by which the pen position is incremented in the drawing process, is the so-called `advanceX`. There is also an `advanceY`, which is used for vertical text layouts.

2.2.3 Kerning

Kerning is used to adjust the positions of neighboring glyphs according to their outlines that gives the output string a more natural look. Take for example the characters "A" and "W". Both characters have angular edges. When kerning is not applied, the "W" will be put on the right side of the "A" in a way that we can cut a vertical line between the displayed glyphs. But it has proved by common habits that it looks much better, if successive characters are displayed interleaved.

As a consequence, many font faces contain a table of kerning pairs. They are ordered (i. e., a pair "A,W" is different to a pair "W,A") and are defined in grid units, since the kerning pairs are assigned directly to glyphs (and, thus, their outlines) and not to characters.

To apply the kerning, the particular kerning value of a glyph pair has just to be added to the pen position, before drawing the second glyph of a pair. By the kerning, the advance value may be increased or decreased.

2.3 Unicode

In the beginning, most computers only supported the basic ASCII character set represented by 7 bits. But, in fact, this set only contains the basic symbols of the Latin script necessary to output text in English language. However, there is a huge amount of other scripts all over the world with more or less complex character sets. Take for example the Asian or Arabic scripts. Because it is desirable to localize software for specific languages, there is a need of a unified standard to represent all these scripts.

Therefore, in the late 1980s, two attempts for standardization were made. One was the *ISO 10646 project* of the *International Organization for Standardization (ISO)* and the other one was the *Unicode Project*, a consortium of multi-lingual software manufacturers. Later, in 1991, they joined their efforts on creating a single unified code table. While *ISO 10646* is not much more than a character set table, the Unicode definition contains more semantic definitions and is thus more useful in practice.

Unicode claims to contain a representation for characters and symbols of most of the used languages in the so-called *Universal Character Set (UCS)*. Basically, the ASCII set is covered. Furthermore, sets like Turkish or Greek are integrated, as well as Asian scripts and even extensions like the artificially created language of J.R.R. Tolkien. ISO 10646 originally defined a 31-bit character set, which has been constricted to 21-bit set, belatedly. This was done because 31 bits offers more space than will be needed in the nearer future.

Because Unicode is just a standard and not an encoding scheme there are different concrete encodings available:

UCS-2 As the name indicates, this encoding uses 2 bytes for specifying character codes. Although ISO 10646 uses more than 16 bits for the character set, almost 99% of the symbols, a program will ever encounter, are covered. Thus, it is mostly sufficient to use this Unicode encoding. On Windows platforms, the String Library defaults to UCS-2 use, if the STR_-UNICODE symbol is defined.

UCS-4 UCS-4, on the other hand uses 4 bytes for character codes. It is the standard Unicode encoding in Linux. UCS-4 is used in the String library, if STR_UNICODE is defined.

UTF-8 *UTF-8* is an abbreviation for *8-bit Unicode Transformation Format* and stands for a variable length encoding scheme for Unicode. Each character is encoded in 1 to 6 bytes. It is standardized in the RFC 3629. This encoding is most compatible to older UNIX tools, which are used to handle ASCII files. It would require modifications to these tools to enable them to deal with UCS-2, for example. UTF-8 is transparent to standard C legacy functions like `strncpy` or `strcat` that are accustomed to 8-bit characters.

In UTF-8, the ASCII characters (UCS code U+0000 to U+007F) are encoded as bytes `0x00` to `0x7F`, thus guaranteeing ASCII compatibility. That means that 7-bit ASCII strings have the same encoding under UTF-8.

UCS character code	byte sequence
U-00000000 - U-0000007F:	0xxxxxxx
U-00000080 - U-000007FF:	110xxxxx 10xxxxxx
U-00000800 - U-0000FFFF:	1110xxxx 10xxxxxx 10xxxxxx
U-00010000 - U-001FFFFF:	11110xxx 10xxxxxx 10xxxxxx 10xxxxxx
U-00200000 - U-03FFFFFF:	111110xx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx
U-04000000 - U-7FFFFFFF:	1111110x 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx

Table 2.1: The UTF-8 encoding scheme

UCS characters >U+007F are encoded in a sequence of several bytes (up to 6 bytes). The first byte of a sequence is always between `0xC0` and `0xFD` and it indicates the number of used bytes. Table 2.1 demonstrates the UTF-8 encoding scheme. Furthermore, a good overview about Unicode is given in [Kuh99].

2.4 FreeType2

FreeType2 ([FTy05b]) is a font library that offers basic functionality in the handling of fonts. As it will be an important core part of the font server design I will propose later in this document, I will give a short introduction into the concept of *FreeType2*.

FreeType2 is a popular software library that enables all kinds of software to access font files. It can handle several font formats, of both bitmap and scalable vector-based, for instance, TrueType, Type1, Open-Type and Windows FON/FNT.

The internal library structure is module based. Because of that, every format is supported via an extra module, as well as the different renderers. Currently, in the FreeType library there are render modules that can generate 1-bit monochrome bitmaps or bitmaps with multiple gray levels (256).

The FreeType library itself is very portable. It is thus a candidate for the use in an L4 font management system.

Another important point to mention is that the FreeType2 library is not thread-safe, but instead, it uses no global variables. If a program wants to use FreeType2 it has to create a `FT_Library`-object, at first. This object will be used for storing variables necessary for the work with FreeType2.

2.5 Fonts in Linux

In order to design and implement a font manager, it is useful to look at existing solutions. In the Linux environment several attempts promise to be a helpful inspiration. In this section I will give a basic overview of these solutions.

2.5.1 XLFD

The XLFD (`X Logical Font Description`) is a string of characters that describes the properties of a font a user want to select. It contains 15 fields that are parted by hyphens. The fields are:

`FontNameRegistry-Foundry-FamilyName-WeightName-Slant-SetwidthName-AddStyleName-PixelSize-PointSize-ResolutionX-ResolutionY-Spacing-AverageWidth-CharSetRegistry-CharSetCodin`

By that, we can specify the font we would like to use. Take for example the following string:

```
-jis-fixed-bold-r-normal--16-150-75-75-c-160-jisx0208.1983-0
```

The string describes a *fixed* font with a weight of *bold* and a point size of *16*. The font is created for a display (i. e., a screen or a printer) resolution of 75 * 75 dpi.

To select a font the user has to define such a string. It is up to the user, whether he specifies all fields or leaves fields as wildcards.

2.5.2 Font Systems

In Linux, there are basically two font systems ([Chr03]) that are used by the *X Server : the original core X11 font system* and Xft fonts system.

The core *X11 font system* exists since 1987 and supports only 1-bit monochrome bitmaps. As a result, it also lacks support for anti-aliasing.

It can handle bitmap fonts, as well as scalable fonts. They are placed in a fonts directory that contains a special file, “fonts.dir”. This file enlists the names of the font files and a font description in XLFD. A typical entry would look like the following:

```
bens.pfb -softmaker-benjaminsans-medium-r-normal--0-0-0-0-p-0-iso8859-1
```

The `fonts.dir` file can be created with the tool `mkfontdir`. If scalable fonts are used, an extra file “fonts.scale” is needed, since `mkfontdir` cannot automatically recognize scalable fonts. This file can be created manually or with another program, called `mkfontscale`.

In order to handle gray-scaled high-quality fonts, the *Xft* (X FreeType interface library) fonts system has been developed. It uses the FreeType library in the background and supports scalable fonts like *TrueType* and gray-scaled glyph bitmaps. *Xft* is the base of most WYSIWYG applications in Linux.

2.5.3 Xfontsel

Xfontsel ([Moh02]) is a tool for the work with fonts in a Linux system. It is a X client and enables the user to select fonts by several attributes, to view examples of a font and to get a full XLFD Description (see Section 2.5.1) of a font. The user can interact with the program via a graphical user interface.

As a client of the X Window System, which includes the font system of Linux, Xfontsel does the font selection by the font information retrieved from the font system.

2.5.4 Fontconfig

The idea behind *Fontconfig* ([Pac05]) is to provide a unified system-wide way to select and configure fonts. It basically consists of two modules: A configuration module that builds an internal configuration by the help of loaded XML files and a matching module, which returns matching fonts, when a request has been made.

Within the XML files, mappings can be defined by which the fonts loaded are gathered in groups. For example, fonts like *Times* or *Bitstream Vera Serif* are so-called *serif* faces (i.e., fonts with hooks or serifs on the letters) or *sans-serif* faces like *Arial* or *Verdana* without the hooks. Differences in the denotions of font properties, like the family, are tried to be mapped to the same name. So, *sans serif* and *sans* are both mapped to *sans-serif*. By that, the fonts are grouped together and discontinuities are disposed. The user can select fonts by defining a pattern that he hands over to the matching module. Within that, the *distance* between the pattern and all provided fonts is calculated and, finally, the font with the nearest distance to the pattern is returned. The distance is calculated by the match or dismatch of the particular font attributes. To do that, there is a certain priority, as the family is more important as the style, for example.

The benefit of the tool is the system-wide availability for font selection, which delivers static results to the programs working with this system.

3 Design

This section gives a requirement specification and the alternatives in design and the resulting structure of the *L4_Font_Server*. First, there is an overview about the design goals, followed by the decisions I made.

3.1 Design Goals

A font server has to fulfill several requirements. It is quite plain that it should be able to handle fonts and is flexible in respect to different font formats like Type1, True-Type, simple bitmap fonts and, at best, is extendable for further formats. It must have the ability to render glyphs of these fonts and return bitmaps of them to the clients. Furthermore, it should supply the using programs with geometric information, such as a bounding boxes or kerning information for each glyph bitmap, which is needed to draw the bitmap glyph to the screen in a proper way.

In order to handle the fonts we will need a font library. For instance, the *FreeType2* library, already introduced in Section 2.4 is a sophisticated font library, which is able to cope with both fixed-size bitmap fonts and scalable fonts. FreeType2 is the font library of choice for the use in the L4 font system.

It is necessary to select and configure fonts. Clients need, at least, knowledge of offered fonts and possible information about the font's capabilities, like size, style or the character set (i. e. UTF-8). Clients can make a selection of attributes, the selected font should satisfy. We have already seen, in which way the Linux font selection (Section 2.5) works.

Furthermore, the font rendering mechanism will have to satisfy the needs of a variety of programs with different habits. One client may want to render whole pages of text and thus, making the rendering a complex, time-critical task and another one just wants a few strings (e. g., for displaying his own GUI). Resulting goals are *efficiency* and *flexibility*.

For the sake of internationalization, Unicode support is a major requirement. With Unicode, the font server will be able to render a huge variety of characters of different languages from all over the world. The result is that we are not limited to the ASCII set.

As the title of the thesis says, the goal of implementation is a font server, but in the decision process we have to look at all imaginable options. In the case of the L4 environment, there are two alternatives to integrate font rendering engines. I will present both of them however it will soon become clear that only one of them is really useful. The basic question is where to put the whole font management, including the rendering engine. Put it all into an extra server or entirely into an all-in-one library? In the following subsections, I will discuss these two options and develop a concrete design of the option of choice.

3.1.1 Library-based design

One option is to write a library, which acts as the font management system. This means that the font rendering engine is also included in the library. Every program that needs font support has to run an extra instance of the font system.

There may be advantages to this concept, for example for the sake of security: if every client use its own instance of a font manager instead of a central server, then there can't be a flow of information or, in other words, the probability of hidden channels is smaller than a system with a central server version could assure. Servers always run the risk of being attacked by one of their clients, thus possibly affecting other clients that are waiting for a font server's response.

Furthermore, font management requires an intense information exchange between a client and a font server and thus implies high traffic on the communication channel and risking a bottleneck. But, if the font management is embedded in the client as a library, one doesn't have to take the indirection via IPC.

These may be convincing arguments, but nevertheless the disadvantages carry a higher weight. The use of an all-in-one library linked to each program that needs fonts would require that each library instance has to initialize the available fonts. This means, that the library has to read the font files to get face descriptors holding attribute information. Consider a system with about 100 fonts or more. Every instance would have to initialize these descriptors (i.e., reading the font files) and store them in its own address space.

3.1.2 Server-based design

The alternative to a all-in-one-library approach is a centralized font system. In this case, only one server is running an instance of the font management. Clients that are in the need of font services just have to register at the server and then are able to use the font system. Unlike the previous approach, the memory is not wasted by loading a font more than one time or running multiple instances of the rendering system. It is obvious that this should be more resource saving.

On the other side, there is the bottleneck problem described in Section 3.1.1. Due to the fact that the fonts are loaded in the server, all the meta information lies there. For example, if you need kerning information between two letters it is usual to ask *FreeType2* with a method taking these two letters as a pair. Now imagine, you want to display a text and already have rendered bitmaps of the necessary glyphs cached in your client's memory. Because the kerning distance is different between each pair of glyphs, it is not efficient to save the information attached to the rendered glyphs. One way is to ask the server for the kerning of each glyph pair, it wants to display. This produces traffic, because the server has to transmit the kerning pairs back to the client. Such a kerning pair is small, but if you want to display a whole text, then you have to get a lot of kerning pairs, which results in high communication costs. The point is that it is a drawback indeed. But there are options to deal with it and to reduce communication costs, through caching and well thought-out interface design.

To summarize, the Server-based design initializes the font list only once, in contrast to the library-based design, where the initialization is done in every program using that font library. Furthermore, the font face descriptors are hold at one centralized place, which is comparable to *Xft* in Linux (see Section 2.5.2). By this centralization, the initialization time and the amount of

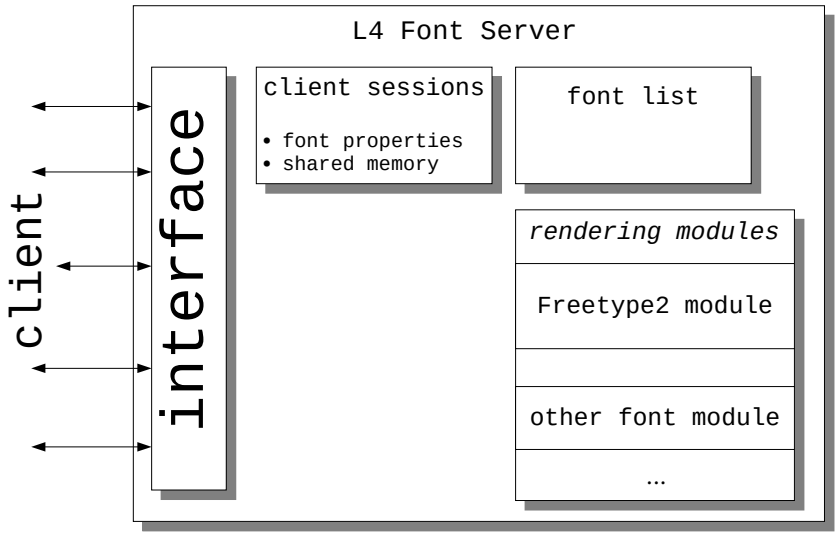

Figure 3.1: Design scheme of the L4 Font Server

used memory is reduced. Because of that, the advantages of the server approach outweigh the drawbacks. That is why the concrete implementation is based on a server based design.

3.2 Design Decisions

The concept of a server environment my be divided into three components: the internal server representation, the IDL interface, and the client library. The internal structure is the most complex part. One of the main goals to reach was flexibility in respect to the supported font formats. *FreeType2* already supports several formats (See in Section 2.4). With its capabilities of including additional modules for font support and rendering it was predestinated for use. The result is that a internal design should consider the use of *FreeType2* and possibly other font libraries.

3.2.1 Internal Server Structure

The central tasks of the server are font rendering and font management, like loading fonts, selecting fonts and holding an info-list of them. Moreover the fonts come up with different formats. Figure 3.1 shows basic components of the font server:

- A structure that organizes registered clients
- A list of available font faces (i.e., the face's descriptor is stored in that list)
- A collection of registered font handling modules

We have seen that *FreeType2* can handle these formats. So, why not just build the server atop of *FreeType2* use its interface throughout the server? As I saw it, this was the straightest way to make the rendering services available to the clients. But, during the process of development I came to the opinion that it would suit better to introduce modules for holding diverse font renderers. And *FreeType2* is just one among of them.

The advantage of this approach is the less dependency on one technology. For instance, if there is another font renderer library, overtaking *FreeType2* in its quality and efficiency or maybe it offers other wanted features, then you can simply write a new wrapper module for that library. The fixed-freetype alternative would make it necessary to change existing code. That's why the font server uses the module scheme.

For the modules there has to be an interface, which provides the following methods:

1. load a font / get a font descriptor
2. configure a font
3. render requested characters to glyphs
4. get kerning information

It has to be mentioned, that further methods may be added in the process of implementation.

The information of fonts registered at the server, which has to be stored in a structure that contains the font description, is intended to be centralized. This means that the font information is kept in one place in the server, not caring about the format and to which render module a font is assigned. The result is that there is a need of a somewhat generic representation of fonts, so that we can handle them outside of their modules.

3.2.2 Interface

The communication interface is one of the first things to think about. What interface methods are required? How will a typical font server session look like? What about efficiency and how to keep the communication costs low? Important questions when you deal with security objections and the avoidance of bottlenecks.

Start/End Sessions The word *session* already implies that a client working with the server will have to register at the font server. Therefore commands like `start_session` and `close_session` are needed. The reason use sessions is, that the server has to hold client specific information, like font configurations, shared memory spaces and so on. We will address that later. The point is that the server will keep such information only as long as the session is active.

Font Configuration During a session a client has the ability to select and configure fonts. One way to achieve this, is to send the explicit font properties each time you call the render service. That means that everytime you want to render glyphs, you would have to transmit the whole font properties, even if you call the render service a hundred times with the same configuration. That would be too much overload and doesn't meet the aim of efficiency.

The better choice is an extra interface method (i. e., `set_propid`) that supplies a configuration possibility. Thus, we can configure fonts, thereby registering the configuration at the server. The interface will return a value that represents a so-called *property id*. To call the rendering service - `render_str`, the client just has to transmit this id and the string to render, of course, which results in smaller message sizes.

Render Strings The already mentioned *render_str*_-call takes a string and a property id. The server will render the requested string in glyph bitmaps of the defined mode (e. g., monochrome or gray-scaled) and, after that has been done, it will return it back to the client.

Load Fonts Furthermore, an option to load fonts, for example by transferring a buffer that contains a font file, may be desirable. The number of fonts loadable is limited to a certain maximum in order to avoid overloads, be it intended or not. The fonts a server has loaded will be visible to all clients. This means that there is no extra font list for each client, only showing fonts a client is allowed to see. However, this approach comes along with security lacks that has to be discussed, which will be done in 3.2.4.

Get Font Information Last but not least, a client needs a list of the available font faces and the according face attributes. Take for example the kerning information. The functions `get_flist` and `get_kerning` will meet these goals.

3.2.3 Client Library

The client library is the part of the server environment a client includes to make the server's interface available to him. It works as a wrapper for the interface calls. Moreover, the library can provide transparency and more convenience to the client. Beyond that it is on one side of the client/server communication interface, giving it a role in the process of cost reduce. For example, it is a good place to organize a caching for often used data.

The library functions are `start_session`, `close_session`, `set_propid`, `render_str` and `get kerning`. Their operational aspects have already been explained in Section 3.2.2.

Another task for the client library are caches saving kerning pairs and glyph bitmaps to reduce communication. Therefore, the library functions shall at first look up in these caches and only if that fails IPC calls have to be made. With the cache subject another design decision came up.

The question was, which kind of clients the font server will use. On the one hand there are clients needing just a few strings for displaying few program specific information and on the other hand, clients with a potentially high frequency of glyph requests, for instance text writers. Keeping the design aim of flexibility in mind, we have to discuss the alternatives.

Such clients that only need a few strings to be rendered don't need that much information as text writing tools that must be able to cope with whole pages full of text. For example, a text writer will often need rendered glyphs, as well as kerning information of differing combinations of glyphs. Other programs that just need glyphs for the graphical user interface, on the other hand, will only request the server once, at startup. By that, they are satisfied with a thin client library only fulfilling the task of communication between client and server.

Now, assume a text writer using no caching. In the text writer, we open a document with more than one page (although one page would be enough to demonstrate the disastrous effect). For displaying the text, the program has to make calls for all the words in the text and everytime the server has to repeatedly render the glyphs, wasting time.

What it comes down to is the necessity of caches on the client side. The best way is to use a cache for holding rendered glyph bitmaps and one for kerning information. By these, the amount of IPC calls and repeated renderings of the same character with identical properties should be reduced drastically. But who is responsible for caching? Leave the caching task to the clients themselves or integrate it into the client library? There are the following thinkable approaches:

- **Thick client library approach** A *thick client* takes care of the point that it is not useful to constantly reinvent the wheel. It implements caches within the library functions that wraps the IPC calls. Because of that, the library would cache bitmaps and kerning information all the time even though it is not wished. However, we can introduce a switch to turn the caches on or off, but, nonetheless, we still have a relatively complex library layer.
- **Thin client library approach** The *thin client library* ignores caches completely. Such a library only serves as a communication point to the font server. In that case, the design goals are slenderness and straightforwardness. Caching is left to the clients.
- **Library layers approach** The first two approaches stand for two opposite attitudes. With the approach of multiple library layers we have a compromise between them. In this design, the client is given the choice of different levels at which it can use the library. While the lowest level is equal to the *thin client library*, the upper level supports caching and uses the low level layer functions for server communication.

In the final implementation, the *L4 Font server* client library will use the layers approach, since there is caching offered, while usage is not enforced. Furthermore, through the layers, the program code is divided into logic partitions and thus not mixing tasks in one layer.

3.2.4 Font Loading

The font server needs font files that it can load. But how will we get the font files loaded into the server? And who determines the fonts that will be loaded, especially when? To answer all these question, we have to look at the alternatives:

- **Load fonts at startup time**

  One option is to load fonts already at the startup of the font server. Several aspects make that a reasonable approach. First, in that case the server is able to work when it has been started. That means, if somebody wants to use the server, there is already a selection of fonts offered. It can select a font or just use the default one. Second, if we restrict the server to just allow font loading at startup time, nobody can influence the server negatively at runtime (by having a *load fonts* function as possible point-of-attack) and thus the security may be increased. But, nonetheless, this would confine the flexibility, of course.

- **Load fonts by clients**

  Another way is to give clients the opportunity to load fonts, maybe by passing a buffer containing the font file to the server or just transferring a reference to a file provided by an external file server. This approach has the advantage that fonts can be loaded at runtime.

  A drawback of enabling clients to load fonts is the already mentioned security aspect. When everybody has the right to load new fonts into the server there may be adversaries who try to attack the server by making a high count of load requests. To avoid that, there will be a maximum count of fonts that can be loaded.

  What is even worse, is the probability of unauthorized communication via direct or hidden channels. For example, clients could communicate by naming font filenames so that they transport information. This is the reason why clients very probably won't be able to load fonts. I will come back to that later, in Section 5.2.

For the sake of flexibility, it would be desirable to use both ways in the final font server. But, on the other hand, the second option has security lacks, making it a doubtful option.

Loading fonts, first of all, requires the availability of fonts. Somehow we have to supply the server with common font files. For instance, think of a file server providing font files or a buffer that is handed over to the server (e. g., by a client). The most elegant way is a file server, like the *simple file server*. I will further discuss this in Section 4.5.1.

When the server retrieves a load request, be it on startup or later by a client, the server will be given a buffer that contains the font file. As already mentioned in 2.2, a font file may contain more than one font face. The server will create a data structures for these faces in a list of font faces that consists of the file buffer and the face descriptor, that holds attributes, like the font family and the style of the font. This generic information has to be extracted out of the file and afterwards, it is inserted into a global *face list*.

Additionally, for the internal font selection it is the best to sort the fonts by families, in a *family list* as shown in Figure **??**. Thus, a single family list entry can contain more than one face descriptors. These descriptors can belong to faces out of different files. For example, the entry `Nimbus Roman` holds faces of the files `Nim_roman.ttf` and `any_fnt_coll.ttf` that equals in the face's family name. Furthermore, as you can see in Figure **??**, the file `any_fnt_coll.ttf` contains font faces of different families, one *Roman* and the other one *Courier*.

Because the font server will support more than one font management module (i. e., the Freetype module will be one of them), the extraction of general information has to be done by the module that is assigned to the particular font format. For example, a *True-Type* file will be handled by the Freetype module. The modules have to return descriptors that represents the faces within the font file. Every descriptor will be inserted into the font family list by its own family.

Assigning of a particular font format to a module will be done by a mapping. At the beginning, when all the font management modules are initialized they will have to register their supported font formats at the mapper. Different formats will be distinguished by the endings. It is important that a font loaded into a file server is known with its correct ending. By that, a file with the ending ".ttf" will be recognized as a *True-Type* file.

The references to the buffers of registered (i. e., loaded fonts) font files are stored in an extra font-file-structure list. Besides the buffer pointers, Such a file structure contains the name of the

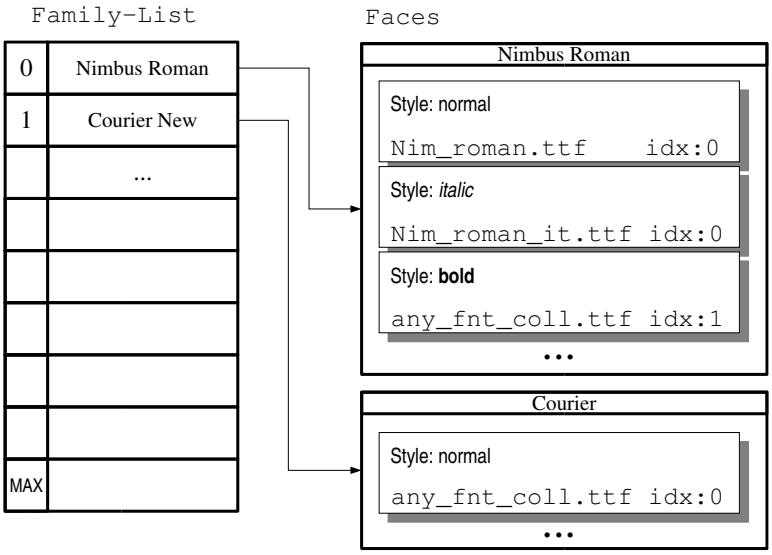

Figure 3.2: Font family list

file and a format tag, that will be used for the mapping of a font-file (with a certain format) to a font module.

3.2.5 Font Selection

The font server needs a method to select fonts. We can think of different ways to do that. For example, the user can choose a font simply by specifying a font family name. Or it can request a font with a certain style, be it bold or italic.

As explained in Section 3.2.1, the server provides the client with a font configure function. This function acts as a configuration method and as the font selector.

The following are several attributes the user will be able to hand over to the server (the list is not complete):

- face id
- file name
- font family
- font style
- font size (width & height)
- resolution of the output device
- transformation (rotation & translation)

The client can specify any of these attributes and can leave others unspecified (which will be mapped to default values). The selection mechanism will use the first, most-fitting font (if there is no perfect-fitting font). For example, if the client requests a font of the family *Times* with the style *bold*, but the only available styles for that family are *normal* and *italic* then the first entry out of the list will be used. However, a not-so-naive approach may choose *normal*, as a user that primarily requests *bold* may be more satisfied with *normal* as the substitute then with *italic*.

To not leave the client blind when choosing a font family, it will be able to do a `get_file_list` call. This function provides the user with a list of the available font families. Furthermore, the client can be provided with any information that will be useful in the client-side selection process. The question is, what we will transfer:

1. **Just the family name**

   The server just returns a list of selectable font families. This option has not much transfer costs, but, too, the client doesn't get information about the available styles or weights of the font family. To just transfer the font family names is almost as helpful as not offering a *get_fontlist*-function.

2. **A collection of face descriptors**

   As we already have a collection of the font face descriptors in the server-internal face list, we could transfer these descriptors to the client (e. g., via shared memory). This approach implies that we can think of passing the font selection to the client, like it is done in Linux, with *Xfontsel* (see Section 2.5.3).

   The drawback of this option is the higher amount of data, because we have to transfer all the face descriptors of all available fonts. But, on the other side, with the face descriptor collection delivered to the client we have the option to leave the font selection policy to the client. By that, a client may use a standard client-library selection function or use an own, more or less sophisticated font selection algorithm.

3. **XLFD-like string representation**

   I introduced *XLFD* in Section 2.5.1. With XLFD, we just have to transfer strings that represents the font's attributes. However, these XLFD-strings have to carry the same information like the face descriptors contains, which means in fact, that the amount of data will be equal to the descriptor-transmission approach.

I decided to use the 2nd option, because it enables the client to do the font selection. The XLFD approach would have required extra efforts in conversion and parsing. Nevertheless, it is an option to offer a function in the client library that takes care of XLFD, but it is not intended to implement it in the context of my thesis.

3.2.6 A Session

To make the whole workflow with the *L4 Font Server* more understandable, a typical session is described in the next paragraphs.

First of all, imagine a client that requests glyph bitmaps of a set of strings. A conceivable candidate is *DOpE*, which uses character bitmaps for displaying titles or contents of its widgets.

Such a client may require fonts of a different style, size or even family. He decides to use the *font server*.

At first, the client has to register at the font server, when it starts a session, and it has to unregister, at the end. The reason for that is that the server has to instantiate several data structures to store client specific information (e. g., font property structures, shared memory data-space, etc.).

During a session, a client has the choice between several calls. If it doesn't want to use the default font configuration it may request a list of available fonts. By that, it can select one of them and make additional choices in style, size, rotation and so on. Having made these font configuration, the client has obtained a data structure that contains all these properties. Therewith, it makes a *set property id* call and thus receiving a *property-id*. The property-id represents a certain font configuration on the server side.

By the help of that property id, the client is in the position to render strings with the rendering call. The font server will return rendered glyph bitmaps and additional meta information (more specified in Chapter 4. These bitmaps can be saved or cached on the client side, for example by the on-top layer of the client library of the font server.

Furthermore, existing font property configuration will be changeable in most of the attributes. The user will just has to repeat the *set property id* call with the old *property id*. As already mentioned, the kerning information has to be handled separately because one kerning tag is assigned to two glyphs.

3.2.7 Unicode Support

Unicode is supported by *Freetype2*. We can use it to support Unicode in the font server, as well. As I described in Section 2.2, a single font file contains several character maps that map a character code to a particular glyph image. In most of the font files, there exists a character map for Unicode mapping.

It is desirable for the font server to work with conventional programs (i. e., clients) that use classic 8-bit character encoding while, too, being able to cope with Unicode strings. Because it is comfortable and compatible, the UTF-8 encoding is the method of choice. In the default mode, the server will recognize strings as UTF-8 strings. By that, normal ASCII-strings will be interpreted as what they are (see in Section 2.3) because of its compatibility. Internally, these strings will be converted into a UCS-4 representation, that is accepted by *FreeType*, for example.

However, there are other encodings like the ISO 8859 encodings ([Bre97]) that extend the common ASCII set by use of the 8th bit of a byte (that was is used in 7-bit ASCII), for example by adding language-specific symbols or control characters. To differentiate from the default UTF-8 mode, there has to be an option to choose between these encodings that is transmitted together with the rendering call. Furthermore, by this option, the font server will accept UCS-4, too.

3.2.8 Example Tools

This section gives an overview about the concepts of example tools using the font server. These tools are aimed to show the abilities and features of the concrete fonts server implementation.

The first tool is called *dfontsel*, which will demonstrate basic features of the font server. A second example is aimed to test the performance of the font server.

3.2.9 Dfontsel

The basic idea behind the font manager tool *Dfontsel* is, as already mentioned, feature demonstration. It is clear that, during the implementation, I will need a testing tool showing the provided functionality. So I can see, if the results are the desired ones, or not.

As Dfontsel will be more comfortable with a GUI, it will be realized as a *DOpE*-tool. The program will use two windows - one for configuration purposes and one for displaying text.

Intended functionalities are font selection and configuration, a text display and transformation options. However, because of its testing purpose, more features than proposed here will be implemented.

3.2.10 SPet - Simple Performance Tester

To evaluate the performance of the current font server implementation, I will need a tool for performance testing. To do this testing, we have to look at the time-critical components of the font server. For that, the tool will have to do rendering of a longer text. This will be done in different modes.

One mode will make a naive text rendering, and thus, not caching anything. Another mode will use caching of the kerning information, but still has no caching of rendered glyph bitmaps on the client side. Finally, a third mode will, at first, cache all necessary glyph bitmaps and thus should be the version with the best performance. The question is: How much better it will do it?

4 Implementation

In this chapter, I want to describe some implementation-related aspects in detail. These are decisions to be made and problems that occurred while implementing the *L4_Font_Server*.

4.1 Porting Freetype2

The *Freetype2* library is aimed to be highly portable, making it easy to port it for the use inside the *L4 Font Server*. As it is independent of the operating system in the background it just depends on the standard functionality, provided through system-specific libraries (e. g., open-, read-calls, etc.). *Freetype2* offers font loading via buffer transfer. Because of that it is even unnecessary to take care of these system library calls.

In the L4 environment, *Freetype2* is an integrated part of the font server and is not intended to be used directly by other clients or servers (Although it is possible).

4.2 Glyph Bitmap Transfer

When doing a render call, the client has to send a string to the font server. This is done by delivering a direct string IPC to the server. This implies, that the maximum length of a transmittable string is 256, as it is the length supported by a direct string IPC.

Afterwards, the server renders the glyphs and returns them to the client. The glyph bitmap transfer is a bottleneck in the client-server system. For example, when dealing with a longer string or, say, a string with a huge character point size, a large amount of data has to be delivered to the client. So how to deal with that? When I implemented the server, I had two basic alternatives for solving that problem:

- a shared memory approach
- the use of IPC transferring

4.2.1 IPC vs. Dataspace

One way to transfer the bitmaps is via IPC[1]. For such huge amount of data, the L4 interface supports so-called *indirect string* IPC's, by which we can send areas of memory by specifying a reference to it. For that, the sender defines:

- *send address* and
- *send size*

[1] Interprocess communication [L4e03]

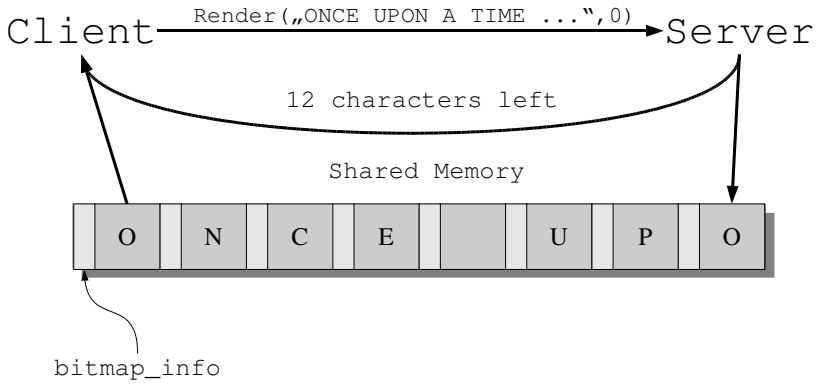

Figure 4.1: Shared memory scheme

of the buffer, while the receiver defines:

- *receive address* and
- *receive size*

While the kernel is delivering the IPC, the buffer is copied from the sender's memory into the receiver's one. String IPC's can transmit up to 4 megabytes of memory. However, there is a drawback, that convinced me to not use this approach. Imagine a render request arriving at the server. The render function has to allocate a buffer to keep the glyph bitmaps before returning them to the client. And finally, during the transfer process, the allocated buffer has to be copied into the client's memory. By the help of another way, we can economize the whole transmission.

The solution is an approach with shared memory. Therefore, the sharing is realized through *dataspaces*. Data-spaces are unstructured data containers that are handled by *dataspace managers* and can be used for sharing memory. Such a shared memory space is depicted in figure 4.1. When a client starts a session with a font server, he first creates such a dataspace by calling a dataspace manager. You can think of a dataspace as memory that is attached to regions of the address space. The dataspace manager acts as a pager for that memory region.

By doing the start`session_call`, the font server is supplied with the client's dataspace (that is created within the client library, transparent to the client) information and is then in the position to attach it to its own address space. By doing this, both the server and the client, are supplied with the same memory region, establishing a space of shared memory. Both sides can write and read data within that memory region.

In this solution, the server doesn't have to allocate an extra buffer, as the other alternative would have to. He just writes the glyph into the shared memory. When returning to the client,

only the offsets of the data in the shared dataspace has to be copied from server to client. As a result, we have less costs caused by copying.

4.2.2 Glyph Buffer Structure

A rendered glyph bitmap does not only consist of the pixel data that will be used for drawing. It also contains associated meta-information. First, the client needs information about the content of the buffer for correct interpretation, like the *pitch* (i. e., the number of bytes taken by one row) or the height and the width in pixel. Moreover, he needs to know several glyph-specific attributes, for instance, an advance value for the pen position and bearing values in x and y direction. They are used for the type-setting on the screen or any other output medium.

For a particular glyph, both the bitmap and the corresponding meta-information have to be transferred to the client, as you can see in Figure 4.1, where the data of the "O" consists of the bitmap and a `bitmap_info`-structure. The previous section has showed that I chose a shared dataspace as the *communication channel*. My solution writes the glyphs as a sequence of *glyph data packages* into the buffer. Every package holds the data for one glyph.

By that, a client receives a continuous buffer containing these packages. Furthermore, the related offsets to the particular data packages within the buffer are transmitted, too. The client can now use these packages by interpreting them and drawing the contained bitmaps or he copies the data into his own memory space, thus, storing them.

4.2.3 Larger Strings

The shared dataspace, which is used for glyph bitmap transfer has a fixed size. It should fit for strings of average length and size. But what is “average”? Most fixed-size solutions come along with the problem that there may be a situation where the fixed size is not enough. So, in the case of the font server, it is possible that a client wants to render a string of a length or size that would result in a necessary amount of memory that is higher than the offered shared dataspace size. This has to be cushioned somehow.

My solution was to split such requests into more than one, so a client has to do more than one call in order do retrieve all glyph bitmaps. The first attempt, I started to implement, was to completely cover the splitting within the client library. The aim of this was to make the whole rendering process transparent to the client. In this approach, a client would make one render call and the result would be a buffer, containing the bitmaps.

Within the library’s rendering function was a loop that repeatedly would call the server until all glyph bitmaps had been transmitted. In the first call of that loop, the server would have rendered the glyphs and stored them in an appropriate temporary buffer within its own memory. When returning to the client, the shared memory will be filled until it is full would be passed back, together with the count of bytes that haven’t been transferred yet.

In the following loop runs, the server would just copy memory from the saved buffer into the shared dataspace, as long as there is memory left. Afterwards the buffer in the server’s memory would be erased. But, as I already indicated, it was my first attempt. In fact, there was a non-negligible count of drawbacks, especially in efficiency.

The main problem were the number of necessary copies to be made. On server side, all glyph bitmaps were pre-rendered into a buffer and from there, they were copied to the shared

memory. After that, on the client side, again, the content of the shared memory was copied into a buffer (that would be returned to the client). What the whole problem made even worse was that the concrete size that would be necessary for the buffer is not quantifiable. It was because the rendered bitmaps have different sizes, especially in proportional fonts, where an “I” needs a *thinner* bitmap than a “W”, for example. So the library only could guess. In the case of a wrong guess, the library had to reallocate a larger buffer and thus we had to do another copy.

The other, more efficient attempt that is now implemented, is comparable to a common read call for files. The server only renders the amount of characters that will fit into the shared memory and returns it together with a number of characters that has been rendered and delivered. Unlike the other approach, the client library won’t run a loop, internally, until all glyphs have been returned. Instead, the client will be supplied with the count of glyphs already rendered and by that, it knows the count of characters it has still to render by a repeated call. As a consequence, the client himself is now in charge to do the loop.

The decisive advantage is the significant reduction of necessary copies. Look at the server, for example. Instead of rendering the glyphs and putting them into a temporary buffer, they will be put into the shared memory, directly. The client library simply hands out the shared memory to the client. The client has to decide whether he just uses that buffer, drawing the glyph out of it, or, on the other hand, copies the bitmaps from the shared memory into another buffer (e. g., a cache). By that, the costs of the rendering process have been reduced and more responsibility is left to the client. That fits well with the design goal to construct a low-level rendering library, with the option to build a high-level library on top of it.

4.3 Handling Clients

As a server, the font management system has to cope with several clients, which will request diverse services. As already indicated in Section 3.2.6, a client that wants to work with the server has to start a *session*, at first. Within such a session, the server handles information about the client:

- thread id of the client
- dataspace (shared memory)
- list of property id’s and the assigned font properties

To start a session, a client call the `fonts_start_session()` function, which thereupon leads to the creation of a `fonts_client` structure. A server can handle `MAX_CLIENTS` of clients (e. g., currently, the number is set to 30). As depicted in the preceding section, the client instantiates a dataspace in his own address space and forwards the dataspace to the server, which stores the dataspace information in the according `fonts_client` element.

One important aspect is that the dataspace is created on client side and is afterwards just attached to the server’s address space. The reason for that is, if the server has to provide own memory everytime a client starts a session, it would be possible to overload the server’s memory by registering a high amount of clients and thus compromising the security of the server by a *Denial-Of-Service* attack, which is not desirable, of course.

Finally, a `fonts_client` object carries a list of font properties. That is because every client can configure font properties. Every created font property is identified by a *property-id* (i. e., `fonts_propid`) and contains information like size, style or a pointer to the configured font face.

The client structure is intended to be extendable for further extensions in the future. All client-specific session information shall be stored within it.

4.4 Supporting Font Modules

As indicated earlier, in Section 3.2.1, the font server supports modules. When I started with programming, I planned to only use *Freetype2*. At that point of time, this was the most convincing alternative to me. The point is that *Freetype2* has already a module structure implemented, which supports new, self-created modules. That is why the plan was to use it as a fixed integrated part of the font server.

But, while implementing the internal of the fixed version it started to get an unclean look to me. It is because such a font server would depend on *Freetype2*. That means, if somebody wants to use another font renderer (e.g,. because of better performance) the internal code has to be altered in order to support the new font library. That is why, later, I changed the internal structure of the server for module support.

The task of a module is to serve font specific functionality. This means, all tasks that deal with information of font files are handed over to these modules. All other tasks should be performed in a global, generic manner. For instance, the list of available fonts is global, making font selection easier, which is a global task indeed.

In my implementation the modules are initialized at start-up of the font server. In fact, there is only one module up to now. Namely the *Freetype2* module. But there are other modules possible. One may wish to have a thin, minimal renderer that just supports bitmap font files and thus maybe getting a higher performance in comparison to *Freetype2*.

The interface provides the following functions:

- module initialization (`init`)
- set property attributes (`config_font`)
- render string as glyph bitmaps (`render`)
- get kerning information for a given character pair (`get_kerning`)
- get a font descriptor (`get_descr`)

All these functions need information out of the font file. Because the *Freetype2* module is a wrapper for the library's functions, it implies that the functions behind the interface will just make library calls, as the library will handle the font files and the faces.

The `init` function initializes the module. For example, in the *Freetype2* module an instance of the Freetype library (used for memory management) is created.

The functions `render` and `get_kerning` hand over their parameters to the Freetype library.

`config_font` uses Freetype to create a face (described in Section 2.4) of the font with the given parameters of size, rotation or resolution. The face that has been created will be saved as a generic face into the associated `prop_entry` structure. The face represents a loaded and configured font, which will be used in the rendering process.

Finally, `get_descr` is used to get general information about a certain font managed by a specific module. This function is basically used at the font server initialization. At that point, the available fonts are inserted into the font-list and therefore font descriptors with basic information about the fonts are needed (i. e., family or style). See Section 3.2.4 for further information about the concept.

4.5 Serial vs. Parallel

By reason that the server has to handle more than one client simultaneously, we have to think about parallelization of the server internals. The problem appeared to me during the process of implementation. What if two or more clients asking for rendered glyphs at the same time?

In the sequential case, which means that there is only one thread, the first client who makes a request will be served first (i.e., a FIFO[2] queue). That means that a client with an time-intensive call (e. g., hundreds of glyphs at once) can block a server and thus perhaps other clients with "short" calls. What is even worse, a client with such a short call may be a process with a higher priority than the other one, think of real-time applications, for instance.

However, I have to admit that the current implementation of the font server uses this naive way. It is because the other option I had in mind at the begin of planning was a multi-thread variant, which has also a tremendous drawback: a big critical section.

In the case of parallel threads, every session (i. e., every client) will use an extra thread. Now, more clients can request services of the server and will be served in parallel order. Therewith, a client won't have to wait for long, time consuming client calls. But there is a rub in it. Think of two client sessions making requests in parallel. Mostly, both of them will use client specific data structures (e. g., own shared memory, own font properties, etc.) and thus, not violating any critical sections. One critical section is the session start. While doing this, it is possible that another client does the same at this time and thus they are both in the client management structures. However, this critical section is avoidable through the use of semaphores.

Now, let us have a look at the *Freetype2* library, which is like a monolith within the font server. If we have only one instance of the library, then this instance, which has to be used for all render requests, resembles a big critical section, because the FreeType2 library is not thread-safe. That is why, all library calls has to be made in sequential order. Moreover, it is the place of the server, where the time-intensive computations are done. In other words, if we are limited to these sequential (expensive) calls into the library, the parallelization is virtually neutralized.

Another option we can think about, is the use of extra *Freetype2* instances for each client (see Section 2.4. Such a library instance - `FT_Library` - is responsible for the memory management. In the freetype library, there are no global variables used. That means, that we can use extra `FT_Library` instances in the freetype module for each client. Obviously, it is the same with other non-freetype font modules. In future work, it has to be analyzed how much it will cost, especially focusing memory.

[2] FIFO - First In First Out

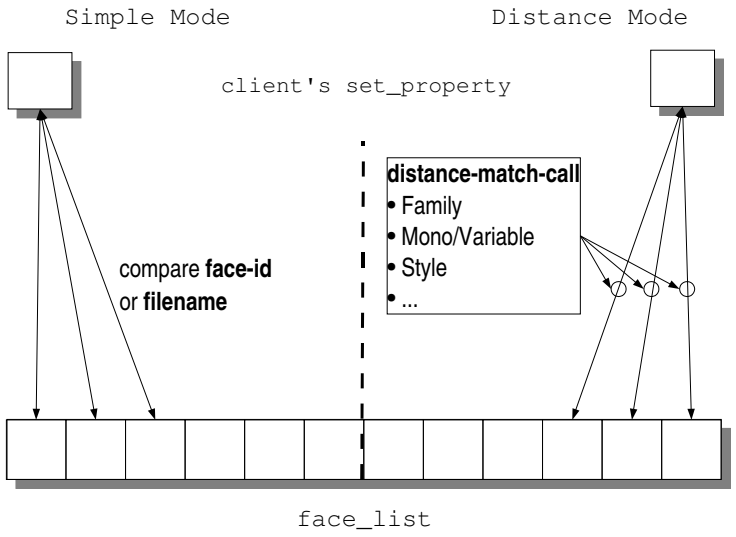

Figure 4.2: Font selection modes

4.5.1 Load Fonts

In Linux you can easily initialize a font server by defining a directory of the file-system, be it *ext3*, for example. In the L4 environment, it is desirable to have a somehow similar way for retrieving font files. One way to achieve that is a virtual file-system. Therefore, I decided to use the *simple file server* of the *L4 Virtual File System* to load font files. At startup, the simple file server can be supplied with data. If *Fiasco UX* is used, then the files can be loaded directly from the Linux file system by defining these files together with the simple file server call within the Fiasco UX startup script. Otherwise, if L4 is started directly, boot modules can be defined and thus will be loaded into the file server.

A second way to get the font file data into the font server is to define the font files as boot modules of the font server, which can be made in the start-up script. Currently, the font server checks both places for font files.

The loaded font files are read and each of the contained faces is inserted into a `face_list`. The `family_list` holds references to the corresponding faces of a family and is also updated. The data structures are needed for the selection of fonts.

4.6 Font Selection

When the user configures a font, he uses the function `set_propid()` (see in Section 3.2.5). He provides the server with the attributes the font should have.

Internally, the server has two modes of font selection, depending on what the client specifies:

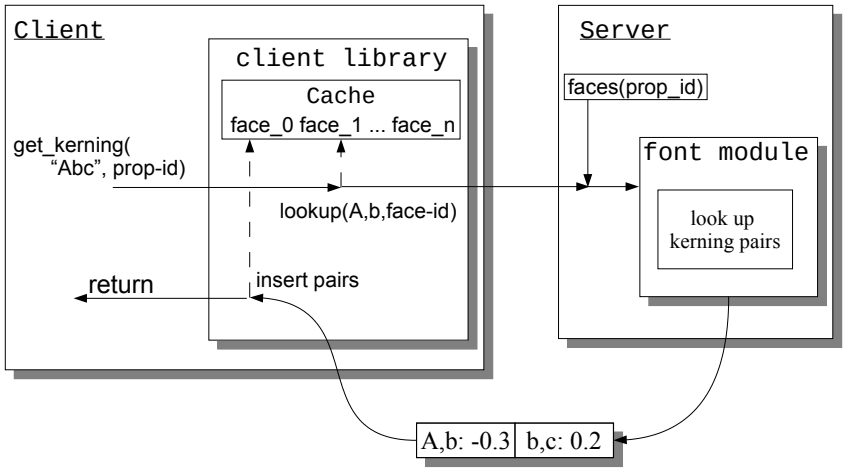

Figure 4.3: Principle of `get_kerning`

Simple Selection If the client specifies either the *filename* or the *face-id* that is returned included in the face descriptors that are returned by a `get_fontlist()`-call, the server uses the simple selection mode. Look at figure 4.2. In the simple mode the face-id or the filename of the client's `set_property` is used to access the `face_list`.

In that case, no check for them most-matching face has to be done, since there is definite selection. If the client specifies the face-id, the server can access the corresponding face by using that id as the index of the server-internal face_list. Otherwise, if the font-filename is given, the server iterates over the face-list, in order to find the face.

The simple selection may be used, if the client does the font selection, as described in Section 6.2.1.

Distance Selection If the other case, the client makes a fuzzier selection. For example he just defines the style and size of the font he wants. Which font shall be returned in that case? A mono-spaced font? Anti-aliased or monochrome bitmaps? For that, preferences have to be set, so, the selector can choose a best-fitting font.

As you can see in Figure 4.2, there is a list of attributes that have to be matched between the requested attribute-pattern and each available font face. To do the matching, the server uses a distance function that compares the attributes ordered in a certain priority. For example, the font family is more important, than a style match.

The choice between both modes is made implicitly and depends on what the client defines. If the face-id or the filename is specified the other attributes are ignored and the *Simple Selection* is preferred.

4.7 Kerning

Kerning has a special position in the meta-information of glyph (see in Section 2.2.3). This is because it doesn't depend on one glyph, but on a pair and thus, cannot be attached to the rest of meta-data that is transferred together with the glyph bitmaps, after a render call. To solve that task, an extra interface method `get_kerning()` is provided. This function accepts strings and returns a collection of glyph kerning-pairs for that string.

A client may cache rendered glyph bitmaps of a certain font face. By that, he can construct strings out of these bitmaps without calling the server, everytime he needs kerning information. As already indicated in Section 3.2.3, the *Library Layers Approach* uses more than one library layer. The kerning can be included in a higher layer of the client library, or, on the other hand, can be completely left to the client.

5 Evaluation

The section gives an evaluation of the *L4_Font_Server*. However, since up to now, there don't exist more sophisticated tools, but only example tools, the font server will have to prove its real usability in future practice (See in Section 6.2).

5.1 Performance

To evaluate the performance, we have to look, where the time-critical and memory intensive components of the *L4_Font_Server* are:

- The font handler/renderer (e.g., FreeType2)
- The communication costs between client and server
- The font selection costs
- The server-internal data structures

In the server-internal font face selection (see Section 4.6) with distance matching, the server additionally has to do the font selection by matching the faces in the server's face-list with the attributes requested by the client. This matching costs time, in particular with the one-threaded approach, currently implemented.

If the internal simple mode is used, that means the client set a face by specifying the face-id, the server can directly access the font face descriptor and by that, avoiding the costs of font-server-internal selection. Of course, the clients may select a font by an own algorithm (see 6.2.1).

The costs of client-server communication are especially high when returning rendered glyphs back to the client. Since the render call uses the shared dataspace, the most-intensive costs come along with the copies into the shared memory and maybe out of the buffer, if the client wants to store the bitmaps, instead of directly using them.

Moreover, we have to consider the computation time of the font rendering and handling libraries. Depending on the complexity of a font face (bitmap fonts, scalable, complex outlines, etc.), the rendering time varies.

The communication and rendering costs can be reduced by caching by the client, or in a higher layer of the client library (see Section 3.2.3) Thus, if bitmaps or other information (e.g., kerning pairs) are needed repeatedly then the client, at first, can look up within its cache.

To have a look at the memory aspects, we have to analyze the data structures of the server. For every client, the server holds:

- The client's thread id

- The list of configured font properties
- The dataspace object and the address of the shared dataspace

Thus, the amount of memory needed for a client depends on the count of configured font properties. There is a maximum of definable properties defined. If the client reaches this maximum has to unregister an older property in order to register a new one.

5.2 Security

To look at the security of the font server, we have to analyze the interaction with clients. What are the policies in dealing with that, which communication channels are available and where possible attack points are.

The font server uses an open policy, which means that every client will be able to register himself at the server and render strings, regardless of any rights. To get a useful rights handling, the font server has to be embedded into a system-wide rights management. But this is an aspect that has to be considered in future work.

Given that policy, it has to be checked, if a malicious client may

- block the server,
- overload (denial of service)
- or exploit covert channels to get unauthorized information.

Block the server The current version implements one-threaded serving. With that, an attacker may get to the idea to *block* the server with a long, time-consuming render request, for example with a long string and a font property with a big point size, which leads to a big bitmap size to render. However, the font server can't be blocked by such a request, because, at first, the server only renders to a maximum length of strings (e. g., 512) and only renders glyphs until the shared memory buffer is filled. After that, a client has to redo the render-call, if the string hasn't been finished. Thus, a single request can't block the server for a significant time.

Overload But now, think of a bad client that does repeated calls, no matter what kind of function, and thus *overloads* the server. Unfortunately, such a client is not really distinguishable from a good-natured client to the server, which just has to do a high count of server calls (e. g., a WSISWYG text editor). The result will be a server overload. It is hard to differentiate between both a normal client and one with bad intentions with just the render information. A possible but impractical method would be to analyze the semantics of the calls and look, whether the content of the request makes sense or not. The problem that has to be analyzed is to differentiate between *usual* and *suspicious* semantics.

Covert channels *Covert channels* can occur, when a client gets information about the server internals or other client, although neither that was intended, nor the client has been authorized for that. For instance, imagine a global collection that holds all the font properties. When

clients set properties, the font server will supply them with property id's that will be allocated in increasing order. By that, a client may register font id's and deregister them. As a consequence, the client can influence the property that another client will receive, if he does a *set_propid_-*-call. I have to admit that it is slightly hypothetical, but it is a way to pass information via the font server indeed.

To avoid this, every client has its own structure with own property id's - the cannot transfer information via property id's.

Another point of probable communication channels would be the option to enable clients to load fonts by specifying references or transmitting buffers, as explained in Section 3.2.4. With such an option clients may communicate via the font server by loading fonts that are artificially constructed for communication. Clients could use the filenames or internal attribute fields for encoding information. This resembles a direct communication channel and that is the reason, why this approach will not be implemented.

The implementation use only one thread for handling requests of all clients. That means, that a client with a time-intensive call can block other clients. Such clients have to wait for the blocking request. This wait time can be observed and there is a timing channel ([VV90]). However, to avoid this in the future, we have to look at a multi-threaded alternative, discussed in Section 4.5.

6 Conclusion and Outlook

6.1 Conclusion

The basic goal of the *L4 font server* implementation was to provide a secure font management service. Therefore fonts has to be loaded and handled. Since rendering is a non-trivial task, the server was planned to use an existing library to solve that problem. Furthermore, due to the higher count of fonts that will be loaded by such a server, an efficient way to do the font selection and configuration had to be found.

Internationalization was another aim of the design. That means, that the server can cope with different character sets of as much languages as possible. Programs may require 1-bit monochrome glyph bitmaps or gray-scaled (8-bit) bitmaps. Further color modifications like color gradients or textures are left to the user, although it is possible to include extended renderers.

Up to now, the server can render bitmaps to corresponding letters in a selected and configured font. For that, the client starts a session and calls a function to set the properties. By that, a font will be selected by one of two modes, depending on what the client defines. If he directly requests a font with a certain filename or face-id, the assigned font face will be used. If he left these fields undefined, in the configuration structure, then the other fields that he has specified will be used by a distance function to find the most-matching font.

To help the client with font selection, the server may deliver a list of font descriptors of available fonts after a `get_fontlist`-call by the client. Because kerning is related to a pair of glyphs, there is an extra call to request these information for a string. By that, a program can request kerning information for strings of glyphs that are already cached in the client's memory. On the client side, the kerning information will be stored in a client-library-internal cache.

The render call will render a requested string, according to the given property-id. Rendered glyph, consisting of meta-information and the bitmap, will be returned via shared memory. This shared dataspace will be filled with glyphs until it is full. If not all characters has been rendered, the client has to repeat the call with the proper string offset.

The performance and security of the font server has to be proven in a further, in particular long-term tests. By that, drawbacks may become clear, that has not occurred, until now.

6.2 Future Work

Further development will affect both the server's internal structure (e. g., new services and functionality) and the client side. Of course, more sophisticated clients have to be written to use the server, because there are only example application available, up to now. I will describe what I have in mind for those two branches of development.

6.2.1 Internal Challenges

1. Add Modules

   The point that the *L4 Font Server* can handle different font modules implies that there exist modules. For the moment, there is only the freetype module. Other modules are imaginable, which, for example, support other fonts or do a special rendering.

   Such a module could render glyphs in colors or with other fillings. Nonetheless, such changes may imply some slight changes to the server's interface. But in most cases, the current interface will meet the demands of special modules.

   Moreover, a module may use another module, for example as the renderer and the module itself does only post-rendering work, like applying bitmaps the glyphs.

2. Font Selection

   The font selection may be extended, for example to support the XLFD, which I explained in Section 2.5.1. A user may define a XLFD-string and use it for configuring a font property structure . Since XLFD is just another way for describing fonts and there already exists an internal representation, XLFD-strings may be handled by additional client library functions, which parse or generate XLFD's.

   Furthermore, in the future, we have to add more *intelligence* to the selection process. The font selection's principle, with it's attribute-distance based matching, is already oriented on Fontconfig (see in Section 2.5.4). But, the distance function can be made better. For example, if we group *sans* and *sans-serif* font families , like in Fontconfig.

   Last but not least, with the possibility to supply the client with face descriptors (See Section 3.2.5) by the *get flist* call, it is a logical consequence to enable the client to do the font selection. A client may request the whole descriptor list of available fonts, select a font face and then request the corresponding face by transferring the face-id with the `set propid`-call to the server (See Section 4.6). We may include the client-side font selection in an additional client-library layer, as described in Section 3.2.3.

   Besides the point, that a potential time-consuming task is passed to the client side, this approach has the advantage that clients can implement differing selection policies, if they want. Such a policy may aim to get a matching font in a short time, another one may be to get the most matching font. Of course, for the client-side selection, we have to provide the client with enough information to solve this task.

3. Multi-Threaded

   As already mentioned in Section 4.5, an alternative structure intends the usage of a multi-threaded font server. To keep the amount of critical sections low in such a structure, each registered client needs an own memory instance of the particular font module renderers (e. g., an instance of the FreeType library).

   Such a multi-threaded variant would provide a fair client-serving. A long time-consuming client would not block high-priority clients, which is an important aspect in a real-time environment indeed.

6.2.2 Client-Side Challenges

On this side we can differ between enhancements within the client library of font server and the client programs that use the library.

1. Client Library Extensions

   As described in Section 3.2.3 it is aimed to integrate a glyph bitmap caching into an extra layer of the client library. The goal is to provide client with a method to improve rendering performance through the caching. By that, not every specific client has to implement the caching.

   The most probable first client will be *DOpE*, which uses an own caching functionality for glyphs. It will render all the glyphs, it will need, at initialization and will keep them in a buffer. Because of that, the glyph cache implementation doesn't have the highest priority.

2. Client programs

   Of course, usable client programs have to be written. Think of WYSIWYG tools, like a text-writer. The first and most important client will be *DOpE*. As it is the central window manager of the L4 environment, most *DOpE* clients will use the font server only through the window manager's widgets.

Bibliography

[Bre97] BREWER, KEVIN J.: *ISO 8859*. http://www.bbsinc.com/iso8859.html, 1997. 22

[Chr03] CHROBOCZEK, JULIUSZ: *Fonts in X11R6.8.2*. http://xorg.freedesktop.org/X11R6.8.2/doc/fonts.html, 03 2003. 11

[Fon05] *Fonts (Wikipedia)*. http://en.wikipedia.org/wiki/Fonts, 2005. 3

[FTy05a] *The Freetype Project*. http://www.freetype.org, 2005. 4

[FTy05b] *Freetype Documentation*. http://freetype.sourceforge.net/freetype2/documentation.html, 2005. 10

[Kuh99] KUHN, MARKUS: *UTF-8 and Unicode FAQ for Unix/Linux*. http://www.cl.cam.ac.uk/~mgk25/unicode.html, 06 1999. 10

[L4e01] *The L4 Environment*. http://os.inf.tu-dresden.de/l4env, 2001. 1

[L4e03] *L4Env - An Environment for L4 Applications*. http://os.inf.tu-dresden.de/l4env/doc/l4env-concept/l4env.pdf, 2003. 25

[Moh02] MOHR, JAMES: *The X Windowing System*. http://www.linux-tutorial.info/modules.php?name=Tutorial&pageid=104, 2002. 11

[MST05] *Microsoft Typography*. http://www.microsoft.com/typography/, 2005. 4

[Pac05] PACKARD, KEITH: *Fontconfig user manual*. http://http://www.fontconfig.org, 2005. 12

[VV90] VAN VLECK, TOM: *Timing Channels*. http://www.multicians.org/timing-chn.html, 1990. 37