Xft2 and STSF

A Side by Side Comparison

Revision A, March 27, 2003

The information and various opinions in this paper were written by a number of different engineers at Sun Microsystems, Inc.

Abstract

This document presents an overview of two different text rendering and font handling technologies for the X Window System - STSF and Xft.

The two technologies are compared and contrasted with a view to highlighting both the major differences between the two projects and the areas in which each project excels.

This study was conducted specifically to illustrate the suitability and benefits of each technology for the Sun GNOME desktop project. As a result, discussion is limited in scope to the features of each project which are required by Sun GNOME.

Software related to these two technologies, such as fontconfig and Freetype, are also analyzed.

Both STSF and Xft are equally able to utilize the font configuration and font caching features of fontconfig for GNOME.

Table of Contents

Introduction

The text rendering requirements of modern day applications differ significantly from when the X11 windowing system was designed. Advances in font technology have not been adopted by X11, leaving it far behind when compared with the typographical sophistication and APIs present in Microsoft Windows and Apple Mac OS systems.

Two new technologies have emerged that aim to fill this increasingly important void in the X11 architecture. Both systems are designed to bring high quality text output, including advanced features such as alpha-blended and anti-aliased text, to X11 users.

This paper compares and contrasts these two approaches to this difficult problem. The intent is to identify the strengths and weaknesses of both technologies, specifically with respect to the requirements of the Sun GNOME desktop system.

Fontconfig and Xft2

Background

Fontconfig and Xft were developed by Keith Packard of the XFree86 project with sponsorship by Suse, Compaq and HP. Xft was designed to both add high quality text output to X applications and toolkits, and to simplify the task of font configuration and customization.

During the second design iteration the project was split into two distinct parts, Fontconfig and Xft2. Fontconfig provides new font naming, matching and selection mechanisms as well as a new font configuration and customization implementation which is completely independent of X. Xft2 provides a convenient interface to the FreeType font rasterizer and the X RENDER extension, allowing X applications to render high quality, anti-aliased and alpha blended text.

Architecture

Fontconfig is a library which provides font configuration and matching which is entirely independent of X11. It automatically detects newly installed fonts, has an XML based configuration file, can identify a set of fonts required to cover a set of languages and does not depend on X in any way.

FreeType is a font rasterizer - it takes a font file, parses it and renders the pixel information into a buffer.

The X RENDER extension is an X extension which adds alpha-compositing, tessellation and the ability to address to sub-pixels directly to the X rendering system. The extension also allows applications to add to a server-side glyph image cache and render sequences of glyphs directly from that cache.

Xft2 is a client side API for rendering text. It uses fontconfig for font management, FreeType to rasterize the glyphs and renders the glyphs using the X RENDER extension. On servers where the RENDER extension is not available it uses the core X11 protocol drawing primitives to render glyphs.

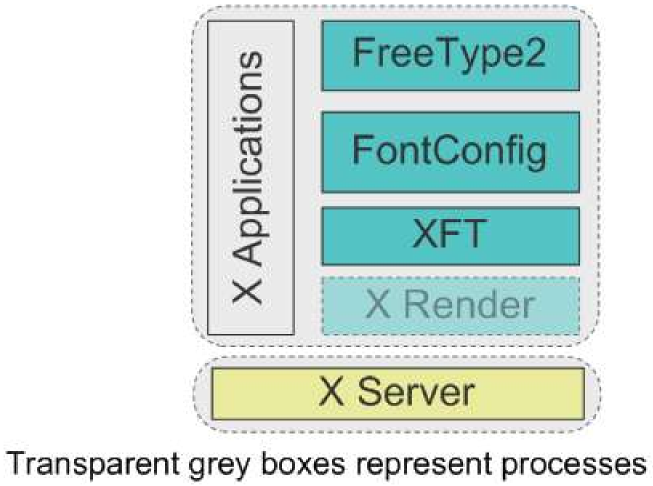

Figure 1. Xft2/Fontconfig Architecture

Key Features

Font Configuration

Fontconfig uses a sophisticated XML based configuration mechanism to allow font customization at both a user and system level. The choice of XML for the configuration file format ensures that external configuration editors are straightforward to develop.

Fontconfig automatically detects the installation of new fonts into the font directories and, unlike the core X font handling mechanism, does not require font configuration files in each font directory. The absence of these per-directory configuration files makes the system more robust by removing this common point of failure.

Fontconfig maintains an on-disk cached mapping of font name to font properties so that font files do not need to be parsed to obtain the list of properties. This minimizes the startup time of applications which use the fontconfig library. This font cache is maintained automatically by the library but can be pre-generated using the fc-cache utility program.

The configuration of the font matching process provided by Fontconfig is very flexible. The configuration consists of a list of match/edit rules that can be used to either modify a font pattern before matching the pattern against the list of available fonts or to modify the result returned from the matching process. For example, to modify the spelling of Sans Serif family in a given input pattern from sans serif to sans-serif, the following rule could be used:

```
<match target="pattern">
        <test qual="any" name="family">
                <string>sans serif</string>
        </test>
        <edit name="family" mode="assign">
                <string>sans-serif</string>
        </edit>
</match>
```

Fontconfig also provides an extensive set of APIs to allow applications to programmatically query and modify the configuration at runtime. This allows, for example, applications to add a font directory which contains fonts specific to the application.

Font Naming, Listing and Matching

Fontconfig implements a new font naming and matching mechanism intended to replace the traditional XLFD mechanism. Significant effort was invested to ensure flexibility for the application developer and user.

In order to request a font, an application presents a pattern to Fontconfig which is used to locate a matching font. A font pattern consists of a list of named attributes, each associated with a list of property values. The list of property values may be used by the application to specify a list of preferences for each attribute. For example, instead of requesting a Times New Roman font, the application may request a font using a list of font family preferences e.g. Times New Roman, Luxi Serif, Serif.

Fontconfig also has a standard textual representation of a font pattern. There is no requirement on applications to ever use this format, but is very useful for applications who wish to store font preferences in its configuration database. The format and some examples are illustrated below:

```
<families>-<point sizes>:<name1>=<values1>:<name2>=<values2>...

Times,serif:style=bold:outline=1   # A serif, bold, outline font -
                                   # preferably Times New Roman

Arial:scalable=True                # A scalable Arial font
```

Fontconfig's matching algorithm has similar semantics to the font selection algorithm in the W3C CSS (Cascading Style Sheet) specification. A font pattern is matched against all available fonts and a closeness metric is calculated for each font. The font which resembles the input pattern the most is returned.

There are some important details to note about the matching process. Because the user may specify a list of acceptable values to match against for each attribute, the application developer need no longer code various levels of fallbacks - the fallbacks may be specified as part of the pattern. Also, the font returned from the matching process is assured to actually be the font with the closest resemblance to the user's wishes as sane defaults are specified for attributes like font weight e.g. if a user specifies Demi Bold, the matching process will return Bold before returning Medium. This is a significant advance in flexibility over the XFLD matching mechanism. It also supports font versioning, this means that it returns the later version of a font multiple versions of a given font are present. From an internationalization point of view it is worth noting that the matching mechanism provides a standardized way to combine fonts for different languages to render in the same glyph string.

Fontconfig's font listing interface recognizes that font listing is an inherently different operation from font matching from an application developer's point of view. When listing the available fonts matching a pattern, the application wishes to discover the available options for a given parameter - e.g. the list of available font families. For this reason, when listing the available fonts matching a given pattern, the application may specify the list of font attributes it wishes to have returned. The key here is that only a unique set of patterns are returned, ensuring the application gets the exact amount of information it requires.

Fontconfig exposes a wealth of information on the fonts available to the application developer. Details like the precise Unicode coverage and language group of the font were found to be vital to certain application's needs only after integrating fontconfig to these applications.

In order to aid migration to the new font naming mechanism, Fontconfig implements a primitive compatibility layer with which applications can continue to use XLFDs font naming, listing and matching.

Client Side Fonts

A key design decision made during the development of Xft2 was to manage fonts exclusively within the application itself rather than relying on the X server or X font server to handle it. There were several factors in making this decision:

- Extended font file access and faster adoption of new font technologies - applications increasingly need access to better information on the fonts they are using. Detailed font metrics and information on font features are required. During the development of the RENDER extension it was decided that providing an abstraction of all available font file formats was unwise, as protocol adoption takes much longer than new font file formats development and the protocol would thus be rendered obsolete before its widespread adoption. Because applications using client side fonts no longer depend on the capabilities of the X server, the adoption of new font technologies can move at the more rapid pace of application development rather than the traditionally slow moving X server technology.
- Application specific fonts - many applications ship with their own fonts which they need to be able to use reliably. By only using fonts which are directly available to the client, the inherent difficulty with providing application specific fonts to the X server is avoided.
- Incremental rasterization - with the core X protocol, clients may only retrieve the entire set of metrics for a font rather than individual glyph information. With outline fonts this entails rasterizing every single glyph in the font as part of the application initialization. Given that Unicode fonts may potentially contain potentially hundreds of thousands of glyphs this becomes a serious performance burden, especially given that only a fraction of the information will ever actually be used. With a server-side scheme it may be possible to implement incremental rasterization, but each client would then need to incrementally request glyph information requiring a dramatic increase in the number of roundtrips to the X server. With a client side model glyphs may be rasterized incrementally without requiring those extra roundtrips.
- Ability to share fonts with the rest of the environment - some applications need to be able to guarantee that they have access to the physical font file in use. An example of an application with such a requirement is a PDF editor, as fonts may be embedded directly into PDF files.

High Quality Text Rendering

Through its use of the X RENDER extension, Xft2 allows the alpha blending of text with the destination drawable. This support for translucent text is a natural progression of the alpha compositing operators made available by the RENDER extension.

Because Xft2 itself is little more than a conduit by which glyphs rasterized by FreeType can be rendered to the X server using the RENDER extension, the text quality is almost entirely dependent on the quality of the rasterizer used by FreeType. The default rasterizer shipped with FreeType 2.1.3 supports sub-pixel rendering for LCD displays and anti-aliased glyph rasterization using 256 levels of gray.

By default, FreeType also comes with an auto-hinting module which performs as well as its TrueType hint interpreter[1] , except with those fonts that have high quality TrueType hints.

Proprietary rasterization engines, which implement higher quality rasterization than FreeType's default rasterization engine, may also be shipped along with FreeType by vendors. In order to do so, an implementation of FreeType's FT_Renderer abstraction must be developed for each such rasterization engine. However, no facility to install new rasterization plugins has yet been developed - plugins must currently be linked directly to the FreeType library at build time.

STSF

Background

STSF (Standard Type Services Framework) is a joint project of the Sun Microsystems X11 and Globalization Engineering teams along with input from engineers from IBM and HP. STSF is conceived as a neutral framework to provide state of the art font rendering, text layout and font management capabilities to the broader audience of desktop and server based UNIX users in the global market. The framework is technology and platform neutral with respect to font renderers, layout engines, operating system, and even neutral with respect to the X Window System so that it may be used not only by X applications, but also by print drivers and server based applications such as Java servlets.

STSF APIs are made available for X11 applications through the XST X11 Server extension and libXst X11 library.

STSF provides an object oriented, end to end solution for application developers to manage fonts, and render and control the layout of text. STSF incorporates many typographically sophisticated features from the most highly regarded existing APIs such as Apple Type Services for Unicode Imaging (ATSUI) and Java2D TextLayouts. As a neutral framework, STSF facilitates late binding of rendering and layout engines so that system developers can easily make value-add decisions for unmodified applications.

Complete globalization support is a major requirement. STSF supports Unicode and complex text layout for languages that require it.

Architecture

STSF's architecture consists of the following components:

STSF Font Server

> The font server is a daemon process which resides on the same host as the X server. It manages the loading of fonts, scalers and layout engines.
>
> The font server implements most of the functionality of STSF. The font server is responsible for loading fonts, rasterizing glyphs (using the scalers available to it), calculating font metrics and performing text layout.

1 There are unresolved patent infringement issues with the TrueType bytecode interpreter. This would prevent most, if not all, vendors from shipping the bytecode interpreter. See http://freetype.sourceforge.net/patents.html for more details.

STSF Scalers

A scaler is a shared library plugin which wrap a given rasterization engine. Scalers may be installed at any time and immediately used by the font server.

STSF Client Library

The STSF Client Library is a shared library that communicates with the STSF Font Server using a private protocol. The STSF client library implements STSF APIs. STSF Client Library expects its user to provide a set of callbacks wrapped in a special structure – STDevice to do actual rendering.

Any application[2] may use the library to render text. In order to do so the application implements a device abstraction which is passed to the library for all rendering operations. The abstraction implementation handles the actual rendering of the pixels to the underlying device technology.

X11 Server

The X11 server implements an special STDevice for use with the STSF Client Library. The X server also translates STSF protocol requests into calls to the STSF Client Library.

STSF X11 Extension and STSF X11 client library libXst

The STSF X11 extension (XST) is an X11- based abstraction of the STSF Client Library API. STSF objects are represented by XIDs and XST protocol requests mirror closely the STSF Client Library API.

The STSF X11 client library libXst translates X11 STSF API to XST protocol requests and communicates them to the X11 server. If the XST extension is not supported by the X server, libXst calls the ST Client Library to rasterize the text and then can send the bitmaps over the wire to the X server using the core X11 protocol.

2 In the case of X, the application referred to here is actually the X server.

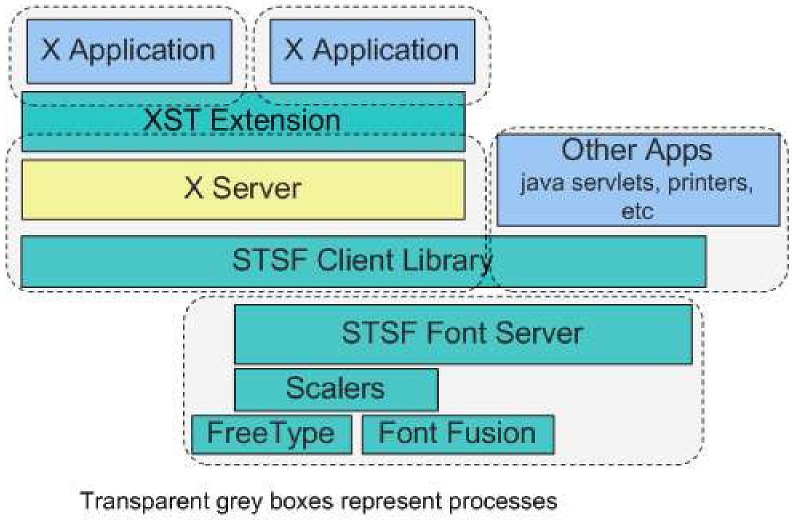

Figure 2. STSF Architecture

Key Features

Font Configuration

One of the original goals of STSF project was to simplify font configuration, selection and installation. The basic assumption for STSF font installation architecture is that modern fonts are self-contained and do not require any external configuration files to be available for applications to use.

STSF Font Server maintains a list of font directories from which it loads fonts. Newly installed fonts are automatically detected and made available to clients. Applications may set the list of font directories using the XSTTypeEnvSetFontFolders() api. The list of font directories may also be set on server startup using the STFONTPATH environmental variable.

STSF defines three types of font directories - system, local and user[3]. Applications may specify which of these directories should be used by the font server.

STSF also has a font fallback policy whereby an application can specify a list of fallback fonts to be used in the case where a specified font does not contain the required glyph. Applications can also programmatically enable, disable or force the usage of font fallback lists.

Font Listing and Matching

With STSF each font is associated with a list of names. Each of these names have associated platform, an encoding and language IDs. These font names correspond directly to the TrueType font names. Each font typically has several names associated with it e.g. full font name, font family, version, Postscript name etc.

3 Usually, the system directory is `/usr/X11/lib/fonts/`, the local directory `/usr/local/lib/fonts/` and the user directory is `~/.fonts/`.

An application can list the fonts which match against a font name. So, for example, to list all Times New Roman fonts available, the application may invoke XSTTypeEnvFindAllFonts():

```
fonts = XSTTypeEnvFindAllFonts (
                display, xst_env,
                "Times New Roman", sizeof ("Times New Roman"),
                NULL, TT_NAME_FONTFAMILY, &n_fonts);
```

Alternatively, applications may use the font family mechanism whereby all fonts are grouped by family name. Applications may enumerate the list of available font families and the list of fonts grouped under a given font family. Thus, to list all Times New Roman fonts:

```
family = XSTTypeEnvFindFontFamily (
                display, xst_env,
                "Times New Roman", sizeof ("Times New Roman"),
                NULL, NULL);

fonts = XSTFontFamilyGetFonts (display, xst_env, family, &n_fonts);
```

Matching a single font is essentially the same operation as listing the available fonts which match a given name. The XST protocol only allows listing so the first matching font is taken from the list of matching fonts. XST always sorts fonts within font families, so that the first font of a list of fonts returned by XSTFontFamilyGetFonts() is the default font of the font family.

The STSF's font selection API was designed to match the traditional font naming paradigm of GUI applications. Most typical applications present the user with a two-level menu allowing the user to choose the font family (e.g. "Times New Roman", "Verdana" etc.) and typeface variant (e.g. "Regular", "Bold" etc.) from the choices available on that system. STSF provides a simple API by which such an interface may be implemented.

STSF allows the use of fontconfig for font configuration, listing and matching. In this mode, STSF is accessing the same set of fonts as other fontconfig-aware applications.

Server Side Fonts

STSF adopts the server side font and glyph management model. The font server implements an interface by which the X server can request font information, font metrics and rasterized glyphs.

A client application may choose to load a set of fonts into the font server. These fonts are called session fonts and may be referenced by URI's[4] . In the atypical case where a font is not directly available to the font server the client transfers the entire font file to the STSF Font Server via the XST protocol extension.

Since there is only a single copy of STSF Font Server per machine, scaled glyph bitmaps and metrics data can be effectively cached and shared between different clients, including multiple X11 servers running on the same server.

In the future, it is planned for STSF font servers to be able to communicate with each

4 A client may specify a session font by a URI detailing the physical location of the font file. Currently only file: URIs are supported, but it is planned that http: URIs for web fonts will be supported in the future.

other, thus allowing the automatic synchronization of fonts.

A complete X11 client side font mode also exists. In such a mode, libXst does not communicate with the STSF Font Server running on the X11 server machine. Instead it communicates directly with STSF running on the same machine where the X11 client application is running and communicates with the X11 server using the traditional X11 protocol. The STSF team is working on a variant of the client-side mode that communicates with the X server using RENDER extension.

Some clear advantages of handling fonts on the server side include:

Only X11 Server-side text rendering can exploit features of the graphics hardware. Relevant features include caching, anti-aliasing, and alpha blending support.

The network is falling behind Moore's Law. Transmitting bitmaps across the connection between X11 clients and the X11 server is a performance bottleneck. This bottleneck will be worsened by the very success of high quality rendering and improved, high resolution displays, as these movements will increase the traffic geometrically. For example, new LCD monitors are increasing the dpi by more than four times. Consider the case of printers, already at 1200 to 2400 dpi, this causes an even larger jump in bandwidth requirements.

Rasterizing on the client side and caching on the server causes additional complexity and overhead in the client as a cache is necessary to determine if the glyph exists on the server or not.

A single large glyph cache can service many X server clients at the same time. Creating a glyph cache of this nature is a primary design goal of STSF. Other projects make best sense in single user Linux/pc-type environments, neglecting shared computing and the SunRay architecture in particular.

High Quality Text Rendering

STSF supports both anti-aliased and alpha blended text output. Text underline, strikethrough and highlight colors may also have an alpha component allowing full text translucency natively.

STSF also allows the rotation, shearing and scaling of text using an affine transformation matrix associated with each output device.

STSF's text quality also depends completely on the rasterization engine used. STSF provides a rasterization engine abstraction which allows the font server to choose between engines at runtime. New rasterization engines can be installed and immediately used. The default engine used by STSF is FreeType which, as stated above, implements a wide range of highly important features. It is expected that vendors shipping STSF may license other commercial font rasterizers and ship these with STSF.

Technology Independence

STSF is, at its core, a complete framework for the implementation of text rendering systems. This framework is entirely platform and technology independent. This was achieved by designing a set of abstractions with which the framework could be applied to different technologies. There are three core abstractions in this framework:

STDevice

> In order to render to any technology an application using the STSF Client Library must implement this interface. This is the interface by which the STSF Client

Library actually renders the text to the required technology.

The STRasterDevice class is used where the underlying technology is a discrete pixel-oriented device, e.g. a frame buffer. Conversely, the STVectorDevice class is used where actual output device has no discrete pixels, e.g. Java™ 2D Development Environment. STRasterDevice implementations must define a method which copies an array of pixels from the internal STSF buffer to the output device. STVectorDevice implementations must provide a method which renders a given vector path.

Each device is associated with an affine transformation matrix which is applied by STSF during the drawing process before the appropriate rendering function is invoked. This allows the device implementation to control the rotation, scaling or shearing of the output.

STScaler

Any font rasterization engine may be used in conjunction with STSF by implementing the STScaler abstraction. Through this interface the ST Font Server individually requests glyphs to be rasterized. Each implementation exposes to the font server which font formats and font features it supports. Each scaler is responsible for populating the font server's global glyph bitmap and metric cache.

STLayoutEngine

Different text layout engines may be used in conjunction with STSF by providing an implementation of the STLayoutEngine interface. The font server presents the layout engine with an array of characters and the engine returns a list of glyphs and their positions. Additionally, clients may choose to perform their own layout and simply provide a list of positioned glyphs to ST for it to rasterize.

Side by Side Comparison

Font Configuration and Selection

Fontconfig was designed to solve specific problems with the existing X font infrastructure. The new font naming and selection mechanism is intended to solve the inadequacies of XLFDs which in the past had led application developers to implement their own mechanisms on top of XLFDs. It also solves internationalization problems related to adequate font selection and matching for complex languages by setting meaningful language font fallbacks as oppose to defaulting to generic fonts.

Similarly, the font configuration implementation was designed so that there could be a single location for font configuration which could be shared amongst not only X applications, but also applications like printing. Both of these mechanisms have already been proven to solve real problems in serious applications e.g. replacing Mozilla's W3C CSS implementation.

STSF font configuration design is based on the concept that modern fonts are self-contained and do not require any external configuration files to be available for applications to use. Older bitmap fonts can be grouped together with scalable fonts to provide specific instances. STSF provides several methods for listing and matching fonts as well as allowing the user to obtain detailed information relating to each font.

STSF Font Server will be able to utilize fontconfig to discover its fonts. In addition to that nothing precludes STSF client application from utilizing fontconfig APIs for font discovery and matching in parallel to STSF font enumeration APIs.

Client Side and Server Side Fonts

The issue over whether a new X font technology should use server side or client side font management seems to be the fundamental issue which needs to be addressed. The reason this issue is so hotly debated is there are many pros and cons to both schemes. Some of the arguments in favor of each scheme are presented above.

One clear advantage of a server side implementation is the ability to utilize the specific hardware available. The client side implementation has no ability to interact directly with the hardware and therefore loses the ability to optimize its output.

Because of X's client/server model, one particular problem haunts any server side scheme. Experience shows that typically the X server moves at a glacial speed when it comes to adopting and deploying new technologies. Thus, large scale applications are reluctant to rely completely on these new technologies so as to not limit their potential uptake. This issue affects both technologies presented here. Xft2 has a dependency upon the server side Render extension, and STSF has a dependency on the server side Xst extension. However, both technologies have a full client side fallback in case the desired extension is not available on the particular server. Additionally, vendors choosing to include one of these technologies in their desktop can ensure it's presence in the X server they provide, limiting the fallback cases to only that of remotely displaying to systems without the extensions.

An argument has been made that a client side technology would have an advantage when it comes to including newer font technologies. However, this argument fails under closer examination as the font requires two parts, the renderer and the loader. For both Xft2 and STSF, the renderers are pluggable modules. STSF has the advantage of allowing any renderer to be chosen at runtime. Xft2 inherits the flexible mechanism of FreeType, which it is married to. FreeType has no dynamic load provision, the renderer must be compiled in, so by association, Xft2 has a compile time binding to the renderer. For the loader portion, both the STSF server and Xft2 library are separate from the X server and are therefore equally replaceable.

Technology Dependence

One of the key features of STSF is that, as a framework, it is not locked to any specific technologies. The STDevice interface allows STSF to render to any technology. The benefit of this approach is that STSF provides a consistent API no matter what the underlying technology is.

On the other hand Xft2 is designed to be a simple implementation layer by which glyphs rasterized by FreeType are rendered to an X display using RENDER and fonts are configured using Fontconfig. In order to apply this to a different rendering technology, Xft2 would be wholly replaced by another layer which would leverage Fontconfig and FreeType but could (potentially) provide an entirely different API.

The key issue here is whether or not application developers would benefit from an API which is consistent across different technologies and platforms or whether an API tailored to each specific technology would be preferred. The STSF answer is to define a rich API that allows market preference and globalization needs to be satisfied.

The STSF STScaler abstraction allows different rasterization engines to be shipped (and licensed) independently of STSF. This feature is beneficial to vendors who wish to ship higher quality proprietary engines. However, FreeType also provides a rasterization engine abstraction. It may be possible for FreeType to borrow a page from STSF and achieve a similar capability, assuming that the FreeType (and Xft2) specified APIs do not lock out any of these advanced typographical features. If this work was carried out, STSF and Xft2 could utilize this abstraction to allow the same engines to be used unmodified with both technologies.

Fontconfig was designed to be entirely independent of X. As such, any application may use Fontconfig and benefit from a shared font configuration as well as Fontconfig's font naming and selection interfaces.

Text Rendering

Both Xft2 and STSF offer similar text rendering features. Both support anti-aliased, alpha blended text and sub-pixel glyph positioning. Both projects are also limited in the quality of text output mainly by the abilities of the rasterization engine currently being used.

As Xft2 is only a simple interface by which glyphs rasterized with FreeType can be rendered to X using the RENDER extension, Xft2 does not provide a text layout interface. This task is left to higher level, and application specific modules. For example, the GNOME desktop project uses Pango which performs the required text layout using Xft2 to render the text.

STSF incorporates rendering and layout engine interfaces in one API. It also declares a lower level Glyph Vector API – a set of functions for manipulating and rendering arrays of positioned glyphs. The Glyph Vector API allows external layout engines to position glyphs and use STSF to render them in a single atomic operation.

It is easy to port all GNOME desktop applications to use the Glyph Vector API since it fits well within the GNOME/Pango model of laying out text that is nothing else but the process of turning lines of text into arrays of positioned glyphs.

Portability and Interoperability

As with any X technology that depends on a new server extension, how that technology operates when the X server doesn't actually have the extension is a very important consideration for any application developer considering using that new technology.

Xft2 depends on the X RENDER extension to render glyphs. However, if the extension is not available, Xft2 falls back to core X rendering routines at the expense of performance, but still allowing alpha blended and anti-aliased text.

In the absence of an XST-enabled X server, XST client fallback is accomplished by the Xclient directly communicating with an STSF font server and uploading scaled glyph bitmaps to the X server via the core X protocol.

Thus, both Xft2 and STSF have very similar characteristics when operating on X servers where the required X Server extension is not available.

Current Product Readiness/Deployment

Xft2 and Fontconfig are very close to being feature complete. The first stable release of fcpackage[5] was released in September 2002. Xft2 has been fully integrated into Pango and gtk+ and will be a high profile part of GNOME 2.2. Xft2 has also been integrated into KDE and Qt and work is almost complete on its integration into Mozilla. The Mozilla port, in particular, provided valuable insight into the requirements for Xft2 and Fontconfig and the availability of Unicode coverage and Language group information was added during the port. However, Xft2 depends on the RENDER extension that requires X11 Server DDX modules to provide support for 32-bit pixmaps. Many XFree86 and Sun DDX modules do not yet support this 32-bit pixmaps, meaning that RENDER and therefore Xft2 still have many obstacles to overcome.

STSF is near to being feature complete as well. The XST client library with fallbacks, XST protocol and STSF client library APIs have been fully designed. Only a couple minor functions remain. The STSF server has been completed and will be shipped in an upcoming release. STSF has been linked into Pango and is fully compatible with the requirements of the GNOME desktop. Work is underway to integrate STSF support into Mozilla and StarOffice.

Performance Evaluation

In order to properly conduct a comparison of the performance of these two technologies, accurate benchmarks are needed. Full benchmarks of STSF have not yet been created. Nevertheless, some conjectural discussion is possible.

A key feature of the STSF architecture is that its glyph cache is stored in shared memory between the STSF font server and all STSF client applications, including any X11 servers. Xft2 stores each glyph cache within the memory space of its host X11 server. Running Xft2 in multiple display environments makes each X11 server keep the entire copy of the glyph cache resulting in significant unshared memory usage, but since RENDER's glyph cache size is tunable, the amount of unshared memory used on multi-display servers can be limited. This makes it possible to tune Xft2 to use the same amount of space as STSF's larger single cache. The disadvantage of this approach is that Xft2 will have a smaller cache to store glyphs, and as many applications share the same fonts, there will be much duplication within those caches, and much less space for other glyphs. Therefore, Xft2 caching benefits applications on a single user's desktop, but does not benefit multiple users, even when they are running identical applications. RENDER's glyph cache can be tuned down to prevent a small number of users from overrunning a multiuser system, but this may cause performance degradation.

On the other hand, STSF shares its glyph cache across all clients thus benefiting both single user desktops and multiuser environments.

An Xft2 client sends every rasterized glyph data down to the X11 server. This bandwidth demand has been shown to be less than the demand previously required by X11 applications when enumerating fonts and querying font metrics. However, a direct comparison of Xft2 vs STSF bandwidth utilization has not yet been made.

When an X11 client is using a high latency connection to the X server, the number of roundtrips required for an operation is very important for application's performance. With RENDER adding glyph images to the server-side glyph cache does not require a roundtrip and, as such, is completely asynchronous.

5 Fcpackage is the name given to a tarball of Fontconfig, Xft2, Xft1 and XRender. The first stable release was version 2.0.

Given that XST extension is effectively only an X11 abstraction of the STSF Client library API, many operations require a roundtrip both from the client to the X server and from the X server to the font server. Furthermore, some simple operations require a number of roundtrips e.g. to display all the font "names" associated with a font you must first list each of the font tags available for the font and then, for each tag, individually request the string associated with the tag - each of these operations require a "double roundtrip".

Nevertheless, for rendering purposes, STSF has been streamlined to reduce the number of round trips to a minimum, and initial testing has shown that `x11perf` STSF benchmark is more than 30% faster than `x11perf` Xft2 benchmark on all tested hardware.

GNOME Integration

GNOME has its own text layout implementation, Pango, which is designed to meet GNOME's specific needs and, thus, Xft2 and STSF must provide certain capabilities through their API in order to allow Pango integration. The API must allow the rendering of individual glyphs at specific positions, and access to individual glyph extents.

Xft2 has already been proven to meet those needs and Pango integration is complete and well tested, at least on Linux. Fontconfig provides Pango with an effective mechanism to solve internationalization problem related to font coverage for complex languages.

The prototype of the STSF-GNOME/Pango integration has been implemented by utilizing STSF Glyph Vector APIs. This exercise allows close comparison between Xft2 and STSF in a Sun GNOME environment. Additionally, intense developer knowledge gained could be applied to a critical evaluation of the Pango internals and possibly a more efficient and elegant layout engine architecture

STSF/Xft Bridge

No matter how advantageous is it for application developers to switch to a newer and better API is, porting all existing code to a different API is a difficult and a time-consuming task. STSF developers implemented an Xft compatibility layer, “STSF/Xft Bridge” that provides a binary-compatible replacement for libXft.so.

All existing open-source and commercial applications that use Xft can switch to STSF by simply replacing a single shared library.

With STSF/Xft bridge applications use STSF only for rendering glyphs. All other services that applications might use, including font configuration by fontconfig and text layout by Pango, remain unchanged.

The STSF/Xft bridge is implemented on top of a light-weight glyph vector object of STSF that does not perform text layout.

Summary

At first glance, STSF and Xft2 look to be similar technologies with similar goals. Both aim to bring typographically high quality text to X11 users, leveraging relatively new font technologies and formats. However, upon examining each project's sub-goals, the differences become clearer.

Xft2/Fontconfig aims to provide sane font configuration, naming and matching, as well as a relatively low-level text rendering API. This API leaves the high level implementation details in the hands of outside developers, in particular the way it relies on the text rendering and pluggable architecture of the FreeType renderer, and leaves text layout to the layout engine in Pango, The downside is that these decisions are locked in place. Xft2 lacks mechanism and structure for adopting other renderers or layout engines. Xft2 moves away from the traditional X-server side font management in favor of a client side scheme for reasons of expected wire protocol performance and more rapid developer acceptance.

The design goals for STSF include a consistent cross-platform framework, scalability, and potential for hardware-tuned performance. The framework approach directly supports the wide variety of text renderers and layout engines from both open source and commercial vendors. STSF declares a rich, high level API, designed to encapsulate all known text rendering and layout attributes and capabilities as exercised in the leading products from Microsoft, Adobe, Bitstream, Apple, and others. STSF includes a Glyph Vector API that allows for the use of external layout engines. The scalability decisions make STSF economical across multi-display systems, especially SunRay configurations. The server-side font management permits STSF to use high speed, low overhead and low cost capabilities of graphics hardware to assist anti-aliasing and text rendering. The client-side fallback mechanism allows STSF clients to run on any X display.

Some portions of STSF's implementation are still being implemented and more thorough performance testing is needed. Preliminary testing with `x11perf` utility demonstrates STSF performance to be up to 200% faster than Xft2 running on the same hardware.

STSF has a much smaller memory footprint on multi-display systems. The API has been tested using the GNOME desktop which resulted in GNOME looking the same as with Xft2.

The design approach inherent in Xft2 is that of a modest proposal to improve text rendering in X, which will then grow organically to address more point problems in this area. The design approach in STSF more ambitiously specifies a complete text rendering and layout framework that harnesses value-added rendering engines both commercial and free, with function and performance gains that work well in the various worlds of GNOME, Linux, and Solaris, from PCs on up to E15K class Sparc servers.

References

Gelfenbain, Alexander. “Standard Type Services Framework – Unicode-based Framework for Rendering Typographically Sophisticated Text.”
Proceedings of the 22nd International Unicode Conference

Packard, Keith. “The Xft Font Library : Architecture and User Guide.”
http://keithp.com/~keithp/talks/xtc2001

Turner, David. “The Design of FreeType 2.”
http://www.freetype.org/freetype2/docs/design/index.html

Hersch, Roger D. “Font Rasterization: The State of the Art.”
http://diwww.epfl.ch/w3lsp/publications/typography/frsa.html

Taylor, Owen. “The Pango Documentation.”
http://www.pango.org

The STSF Team. “Standard Type Services Framework.”
http://stsf.sourceforge.net

Packard, Keith. “Fontconfig.Org”
http://www.fontconfig.org

Packard, Keith. “A New Rendering System For X.”
http://keithp.com/~keithp/talks/usenix2000/render.html

Packard, Keith. “Challenges of the X Rendering Extension”
http://keithp.com/~keithp/talks/renderproblems/