

KDE Architecture Overview

Bernd Gehrman



KDE Architecture Overview

Contents

1	Library structure	1
1.1	Libraries by name	1
1.2	Grouped classes	1
2	Graphics	6
2.1	Low-level graphics with QPainter	6
2.1.1	Rendering with QPainter	6
2.1.2	Transformations	7
2.1.3	Setting stroking attributes	8
2.1.4	Setting fill attributes	9
2.1.5	Color	10
2.1.6	Other settings	10
2.1.7	Drawing graphics primitives	11
2.1.8	Drawing pixmaps and images	11
2.1.9	Drawing text	11
2.2	Structured graphics with QCanvas	12
2.3	3D graphics with OpenGL	13
2.3.1	Low-level interface	13
2.3.2	High-level interfaces	14
3	User interface	15
3.1	The action pattern	15
3.2	Defining menus and toolbars in XML	15
3.2.1	Introduction	15
3.2.2	An example: Menu in KView	16

KDE Architecture Overview

3.2.3	An example: Toolbars in Konqueror	18
3.2.4	Dynamical menus	19
3.2.5	Context menus	19
3.3	Providing online help	20
4	Components and services	23
4.1	KDE services	23
4.1.1	What are KDE services?	23
4.1.2	Defining service types	24
4.1.3	Defining shared library services	24
4.1.4	Using shared library services	25
4.1.5	Defining DCOP services	26
4.1.6	Using DCOP services	27
4.2	MIME types	28
4.2.1	What are MIME types?	28
4.2.2	Defining MIME types	29
4.2.3	Determining the MIME type of data	29
4.2.4	Mapping a MIME type to an application or service	31
4.2.5	Miscellaneous	32
4.3	Network transparency	32
4.3.1	Introduction	32
4.3.2	Using KIO	33
4.3.3	Directory entries	35
4.3.4	Synchronous usage	36
4.3.5	Meta data	37
4.3.6	Scheduling	38
4.3.7	Defining an ioslave	39
4.3.8	Implementing an ioslave	40
4.3.9	Communicating back to the application	41
4.3.10	Interacting with the user	42
A	Licensing	43

Abstract

This documentation gives an overview of the KDE Development Platform

Chapter 1

Library structure

1.1 Libraries by name

kdecore The `kdecore` library is the basic application framework for every KDE based program. It provides access to the configuration system, command line handling, icon loading and manipulation, some special kinds inter-process communication, file handling and various other utilities.

kdeui The `kdeui` library provides many widgets and standard dialogs which Qt doesn't have or which have more features than their Qt counterparts. It also includes several widgets which are subclassed from Qt ones and are better integrated with the KDE desktop by respecting user preferences.

kio The `kio` library contains facilities for asynchronous, network transparent I/O and access to mimetype handling. It also provides the KDE file dialog and its helper classes.

kjs The `kjs` library provides an implementation of JavaScript.

khtml The `khtml` library contains the KHTML part, a HTML browsing widget, DOM API and parser, including interfaces to Java and JavaScript.

1.2 Grouped classes

Core application skeleton - classes needed by almost every application.

- **KApplication** Initializes and controls a KDE application.
- **KUniqueApplication** Makes sure only one instance of an application can run simultaneously.

KDE Architecture Overview

- **KAboutData** Holds information for the about box.
- **KCmdLineArgs** Command line argument processing.

Configuration settings - access to KDE's hierarchical configuration database, global settings and application resources.

- **KConfig** Provides access to KDE's configuration database.
- **KSimpleConfig** Access to simple, non-hierarchical configuration files.
- **KDesktopFile** Access to `.desktop` files.
- **KGlobalSettings** Convenient access to not application-specific settings.

File and URL handling - decoding of URLs, temporary files etc.

- **KURL** Represents and parses URLs.
- **KTempFile** Creates unique files for temporary data.
- **KSaveFile** Allows to save files atomically.

Interprocess communication - DCOP helper classes and subprocess invocation.

- **KProcess** Invokes and controls child processes.
- **KShellProcess** Invokes child processes via a shell.
- **PtyProcess** Communication with a child processes through a pseudo terminal.
- **KIPC** Simple IPC mechanism using X11 ClientMessages.
- **DCOPClient** DCOP messaging.
- **KDCOPPropertyProxy** A proxy class publishing Qt properties through DCOP.
- **KDCOPActionProxy** A proxy class publishing a DCOP interface for actions.

Utility classes - memory management, regular expressions, string manipulation, random numbers

- **KRegExp** POSIX regular expression matching.
- **KStringHandler** An extravagant interface for string manipulation.
- **KZoneAllocator** Efficient memory allocator for large groups of small objects.
- **KRandomSequence** Pseudo random number generator.

KDE Architecture Overview

Keyboard accelerators - classes helping to establish consistent key bindings throughout the desktop.

- **KAccel** Collection of keyboard shortcuts.
- **KStdAccel** Easy access to the common keyboard shortcut keys.
- Collection of system-wide keyboard shortcuts.

Image processing - icon loading and manipulating.

- **KIconLoader** Loads icons in a theme-conforming way.
- **KIconTheme** Helper classes for KIconLoader.
- **KPixmap** A pixmap class with extended dithering capabilities.
- **KPixmapEffect** Pixmap effects like gradients and patterns.
- **KPixmapIO** Fast QImage to QPixmap conversion.

Drag and Drop - drag objects for colors and URLs.

- **KURLDrag** A drag object for URLs.
- **KColorDrag** A drag object for colors.
- **KMultipleDrag** Allows to construct drag objects from several others.

Auto-Completion

- **KCompletion** Generic auto-completion of strings.
- **KURLCompletion** Auto-completion of URLs.
- **KShellCompletion** Auto-completion of executables.

Widgets - widget classes for list views, rules, color selection etc.

- **KListView** A variant of QListView that honors KDE's system-wide settings.
- **KListBox** A variant of QListBox that honors KDE's system-wide settings.
- **KIconView** A variant of QIconView that honors KDE's system-wide settings.
- **KLineEdit** A variant of QLineEdit with completion support.
- **KComboBox** A variant of QComboBox with completion support.
- **KFontCombo** A combo box for selecting fonts.

KDE Architecture Overview

- **KColorCombo** A combo box for selecting colors.
- **KColorButton** A button for selecting colors.
- **KURLCombo** A combo box for selecting file names and URLs.
- **KURLRequester** A line edit for selecting file names and URLs.
- **KRuler** A ruler widget.
- **KAnimWidget** animations.
- **KNumInput** A widget for inputting numbers.
- **KPasswordEdit** A widget for inputting passwords.

Dialogs - full-featured dialogs for file, color and font selection.

- **KFileDialog** A file selection dialog.
- **KColorDialog** A color selection dialog.
- **KFontDialog** A font selection dialog.
- **KIconDialog** An icon selection dialog.
- **KKeyDialog** A dialog for editing keyboard bindings.
- **KEditToolBar** A dialog for editing toolbars.
- **KTipDialog** A Tip-of-the-day dialog.
- **KAboutDialog** An about dialog.
- **KLineEditDlg** A simple dialog for entering text.
- **KURLRequesterDlg** A simple dialog for entering URLs.
- **KMessageBox** A dialog for signaling errors and warnings.
- **KPasswordDialog** A dialog for inputting passwords.

Actions and XML GUI

- **KAction** Abstraction for an action that can be plugged into menu bars and tool bars.
- **KActionCollection** A set of actions.
- **KXMLGUIClient** A GUI fragment consisting of an action collection and a DOM tree representing their location in the GUI.
- **KPartManager** Manages the activation of XMLGUI clients.

Plugins and Components

KDE Architecture Overview

- **KLibrary** Represents a dynamically loaded library.
- **KLibLoader** Shared library loading.
- **KLibFactory** Object factory in plugins.
- **KServiceType** Represents a service type.
- **KService** Represents a service.
- **KMimeType** Represents a MIME type.
- **KServiceTypeProfile** User preferences for MIME type mappings.
- **KTrader** Querying for services.

Chapter 2

Graphics

2.1 Low-level graphics with QPainter

2.1.1 Rendering with QPainter

Qt's low level imaging model is based on the capabilities provided by X11 and other windowing systems for which Qt ports exist. But it also extends these by implementing additional features such as arbitrary affine transformations for text and pixmaps.

The central graphics class for 2D painting with Qt is [QPainter](#). It can draw on a [QPaintDevice](#). There are three possible paint devices implemented: One is [QWidget](#) which represents a widget on the screen. The second is [QPrinter](#) which represents a printer and produces Postscript output. The third is the class [QPicture](#) which records paint commands and can save them on disk and play them back later. A possible storage format for paint commands is the W3C standard SVG.

So, it is possible to reuse the rendering code you use for displaying a widget for printing, with the same features supported. Of course, in practice, the code is used in a slightly different context. Drawing on a widget is almost exclusively done in the `paintEvent()` method of a widget class.

```
void FooWidget::paintEvent ()
{
    QPainter p(this);
    // Setup painter
    // Use painter
}
```

When drawing on a printer, you have to make sure to use `QPrinter::newPage()` to finish with a page and begin a new one - something that naturally is not relevant for painting widgets. Also, when printing, you may want to use the [device metrics](#) in order to compute coordinates.

2.1.2 Transformations

By default, when using QPainter, it draws in the natural coordinate system of the device used. This means, if you draw a line along the horizontal axis with a length of 10 units, it will be painted as a horizontal line on the screen with a length of 10 pixels. However, QPainter can apply arbitrary affine transformations before actually rendering shapes and curves. An affine transformation maps the x and y coordinates linearly into x' and y' according to

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} m_{11} & m_{12} & 0 \\ m_{21} & m_{22} & 0 \\ dx & dy & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

The 3x3 matrix in this equation can be set with `QPainter::setWorldMatrix()` and is of type `QWMatrix`. Normally, this is the identity matrix, i.e. m_{11} and m_{22} are one, and the other parameters are zero. There are basically three different groups of transformations:

- **Translations** These move all points of an object by a fixed amount in some direction. A translation matrix can be obtained by calling method `m.translate(dx, dy)` for a `QWMatrix`. This corresponds to the matrix

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ dx & dy & 1 \end{pmatrix}$$

- **Scaling** These stretch or shrink the coordinates of an object, making it bigger or smaller without distorting it. A scaling transformation can be applied to a `QWMatrix` by calling `m.scale(sx, sy)`. This corresponds to the matrix

$$\begin{pmatrix} sx & 0 & 0 \\ 0 & sy & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

By setting one of the parameters to a negative value, one can achieve a mirroring of the coordinate system.

- **Shearing** A distortion of the coordinate system with two parameters. A shearing transformation can be applied by calling `m.shear(sh, sv)`, corresponding to the matrix

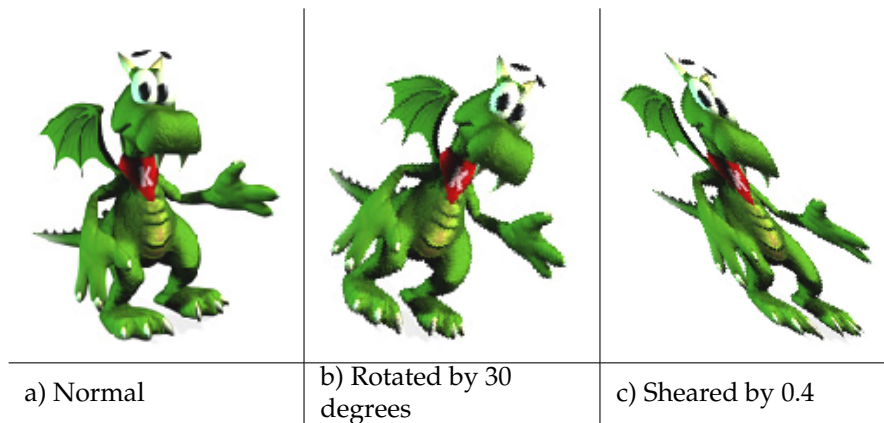
$$\begin{pmatrix} 1 & sv & 0 \\ sh & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

- **Rotating** This rotates an object. A rotation transformation can be applied by calling `m.rotate(alpha)`. Note that the angle has to be given in degrees, not as mathematical angle! The corresponding matrix is

$$\begin{pmatrix} \cos \alpha & \sin \alpha & 0 \\ -\sin \alpha & \cos \alpha & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Note that a rotation is equivalent with a combination of scaling and shearing.

Here are some pictures that show the effect of the elementary transformation to our masquot:



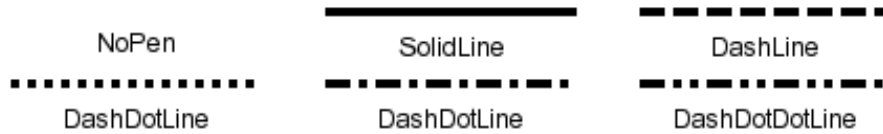
Transformations can be combined by multiplying elementary matrices. Note that matrix operations are not commutative in general, and therefore the combined effect of a concatenation depends on the order in which the matrices are multiplied.

2.1.3 Setting stroking attributes

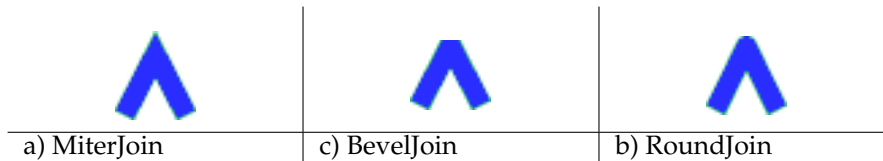
The rendering of lines, curves and outlines of polygons can be modified by setting a special pen with `QPainter::setPen()`. The argument of this function is a `QPen` object. The properties stored in it are a style, a color, a join style and a cap style.

KDE Architecture Overview

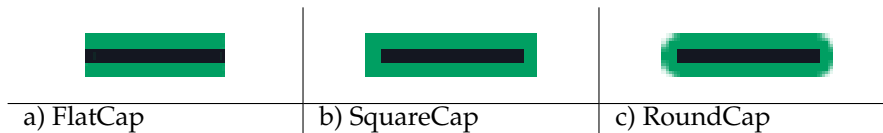
The pen style is member of the enum `Qt::PenStyle`. and can take one of the following values:



The join style is a member of the enum `Qt::PenJoinStyle`. It specifies how the junction between multiple lines which are attached to each other is drawn. It takes one of the following values:



The cap style is a member of the enum `Qt::PenCapStyle` and specifies how the end points of lines are drawn. It takes one of the values from the following table:



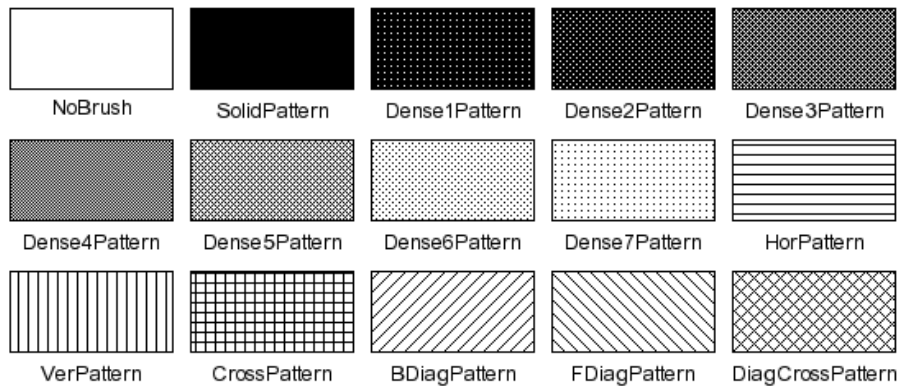
2.1.4 Setting fill attributes

The fill style of polygons, circles or rectangles can be modified by setting a special brush with `QPainter::setBrush()`. This function takes a `QBrush` object as argument. Brushes can be constructed in four different ways:

- `QBrush::QBrush()` - This creates a brush that does not fill shapes.
- `QBrush::QBrush(BrushStyle)` - This creates a black brush with one of the default patterns shown below.
- `QBrush::QBrush(const QColor &, BrushStyle)` - This creates a colored brush with one of the patterns shown below.
- `QBrush::QBrush(const QColor &, const QPixmap)` - This creates a colored brush with the custom pattern you give as second parameter.

A default brush style is from the enum `Qt::BrushStyle`. Here is a picture of all predefined patterns:

KDE Architecture Overview



A further way to customize the brush behavior is to use the function `QPainter::setBrushOrigin()`.

2.1.5 Color

Colors play a role both when stroking curves and when filling shapes. In Qt, colors are represented by the class `QColor`. Qt does not support advanced graphics features like ICC color profiles and color correction. Colors are usually constructed by specifying their red, green and blue components, as the RGB model is the way pixels are composed of on a monitor.

It is also possible to use hue, saturation and value. This HSV representation is what you use in the Gtk color dialog, e.g. in GIMP. There, the hue corresponds to the angle on the color wheel, while the saturation corresponds to the distance from the center of the circle. The value can be chosen with a separate slider.

2.1.6 Other settings

Normally, when you paint on a paint device, the pixels you draw replace those that were there previously. This means, if you paint a certain region with a red color and paint the same region with a blue color afterwards, only the blue color will be visible. Qt's imaging model does not support transparency, i.e. a way to blend the painted foreground with the background. However, there is a simple way to combine background and foreground with boolean operators. The method `QPainter::setRasterOp()` sets the used operator, which comes from the enum `RasterOp`.

The default is `CopyROP` which ignores the background. Another popular choice is `XorROP`. If you paint a black line with this operator on a colored image, then the covered area will be inverted. This effect is for example used to create the rubberband selections in image manipulation programs known as "marching ants".

2.1.7 Drawing graphics primitives

In the following we list the elementary graphics elements supported by QPainter. Most of them exist in several overloaded versions which take a different number of arguments. For example, methods that deal with rectangles usually either take a [QRect](#) as argument or a set of four integers.

- Drawing a single point - `drawPoint()`.
- Drawing lines - `drawLine()`, `drawLineSegments()` and `drawPolyLine()`.
- Drawing and filling rectangles - `drawRect()`, `drawRoundRect()`, `fillRect()` and `eraseRect()`.
- Drawing and filling circles, ellipses and parts of them - `drawEllipse()`, `drawArc()`, `drawPie` and `drawChord()`.
- Drawing and filling general polygons - `drawPolygon()`.
- Drawing bezier curves - `drawQuadBezier()` [`drawCubicBezier` in Qt 3.0].

2.1.8 Drawing pixmaps and images

Qt provides two very different classes to represent images.

[QPixmap](#) directly corresponds to the pixmap objects in X11. Pixmap are server-side objects and may - on a modern graphics card - even be stored directly in the card's memory. This makes it *very* efficient to transfer pixmaps to the screen. Pixmap also act as an off-screen equivalent of widgets - the [QPixmap](#) class is a subclass of [QPaintDevice](#), so you can draw on it with a [QPainter](#). Elementary drawing operations are usually accelerated by modern graphics. Therefore, a common usage pattern is to use pixmaps for double buffering. This means, instead of painting directly on a widget, you paint on a temporary pixmap object and use the [bitBlt](#) function to transfer the pixmap to the widget. For complex repaints, this helps to avoid flicker.

In contrast, [QImage](#) objects live on the client side. Their emphasis is on providing direct access to the pixels of the image. This makes them of use for image manipulation, and things like loading and saving to disk ([QPixmap](#)'s `load()` method takes [QImage](#) as intermediate step). On the other hand, painting an image on a widget is a relatively expensive operation, as it implies a transfer to the X server, which can take some time, especially for large images and for remote servers. Depending on the color depth, the conversion from [QImage](#) to [QPixmap](#) may also require dithering.

2.1.9 Drawing text

Text can be drawn with one of the overloaded variants of the method `QPainter::drawText()`. These draw a [QString](#) either at a given point or in a given rectangle, using the

font set by `QPainter::setFont()`. There is also a parameter which takes an ORed combination of some flags from the enums `Qt::AlignmentFlags` and `Qt::TextFlags`

Beginning with version 3.0, Qt takes care of the complete text layout even for languages written from right to left.

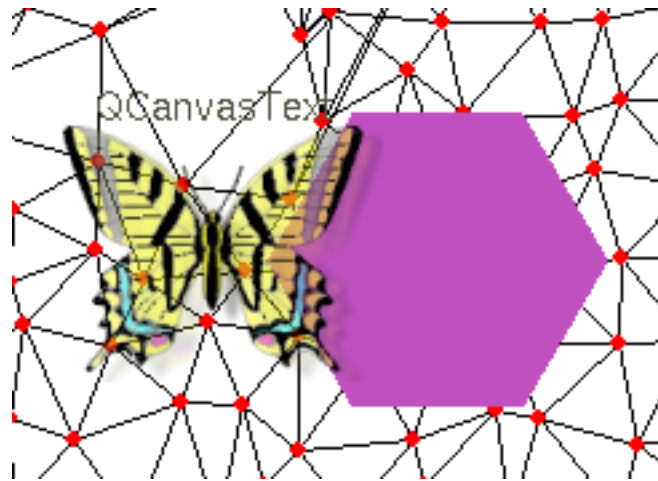
A more advanced way to display marked up text is the `QSimpleRichText` class. Objects of this class can be constructed with a piece of text using a subset of the HTML tags, which is quite rich and provides even tables. The text style can be customized by using a `QStyleSheet` (the documentation of the tags can also be found here). Once the rich text object has been constructed, it can be rendered on a widget or another paint device with the `QSimpleRichText::draw()` method.

2.2 Structured graphics with QCanvas

`QPainter` offers a powerful imaging model for painting on widgets and pixmaps. However, it can also be cumbersome to use. Each time your widget receives a paint event, it has to analyze the `QPaintEvent::region()` or `QPaintEvent::rect()` which has to be redrawn. Then it has to setup a `QPainter` and paint all objects which overlap with that region. For example, image a vector graphics program which allows to drag objects like polygons, circles and groups of them around. Each time those objects move a bit, the widget's mouse event handler triggers a paint event for the whole area covered by the objects in their old position and in their new position. Figuring out the necessary redraws and doing them in an efficient way can be difficult, and it may also conflict with the object-oriented structure of the program's source code.

As an alternative, Qt contains the class `QCanvas` in which you put graphical objects like polygons, text, pixmaps. You may also provide additional items by subclassing `QCanvasItem` or one of its more specialized subclasses. A canvas can be shown on the screen by one or more widgets of the class `QCanvasView` which you have to subclass in order to handle user interactions. Qt takes care of all repaints of objects in the view, whether they are caused by the widget being exposed, new objects being created or modified or other things. By using double buffering, this can be done in an efficient and flicker-free way.

Canvas items can overlap each other. In this case, the visible one depends on the z order which can be assigned by `QCanvasItem::setZ()`. Items can also be made visible or invisible. You can also provide a background to be drawn "behind" all items and a foreground. For associating mouse events with objects, in the canvas, there is the method `QCanvas::collisions()` which returns a list of items overlapping with a given point. Here we show a screenshot of a canvas view in action:



Here, the mesh is drawn in the background. Furthermore, there is a `QCanvasText` item and a violet `QCanvasPolygon`. The butterfly is a `QCanvasPixmap`. It has transparent areas, so you can see the underlying items through it.

A tutorial on using `QCanvas` for writing sprite-based games can be found [here](#).

2.3 3D graphics with OpenGL

2.3.1 Low-level interface

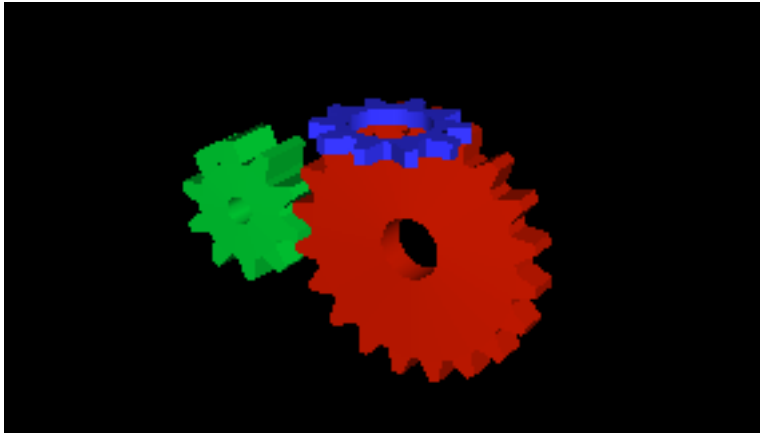
The de facto standard for rendering 3D graphics today is [OpenGL](#). Implementations of this specification come with Microsoft Windows, Mac OS X and XFree86 and often support the hardware acceleration features offered by modern graphics cards. OpenGL itself only deals with rendering on a specified area of the framebuffer through a *GL context* and does not have any interactions with the toolkit of the environment

Qt offers the widget `QGLWidget` which encapsulates a window with an associated GL context. Basically, you use it by subclassing it and reimplementing some methods.

- Instead of reimplementing `paintEvent()` and using `QPainter` to draw the widget's contents, you override `paintGL()` and use GL commands to render a scene. `QGLWidget` will take care of making its GL context the current one before `paintGL()` is called, and it will flush afterwards.
- The virtual method `initializeGL()` is called once before the first time `resizeGL()` or `paintGL()` are called. This can be used to construct display lists for objects, and make any initializations.
- Instead of reimplementing `resizeEvent()`, you override `resizeGL()`. This can be used to set the viewport appropriately.

- Instead of calling `update()` when the state of the scene has changed - for example when you animate it with a timer -, you should call `updateGL()`. This will trigger a repaint.

In general, `QGLWidget` behaves just like any other widget, i.e. for example you can process mouse events as usual, resize the widget and combine it with others in a layout.



Qt contains some examples of `QGLWidget` usage in its `demo` example. A collection of tutorials can be found [here](#), and more information and a reference of OpenGL is available on the [OpenGL homepage](#).

2.3.2 High-level interfaces

OpenGL is a relatively low-level interface for drawing 3D graphics. In the same way `QCanvas` gives the programmer a higher-level interface which details with objects and their properties, there are also high-level interfaces for 3D graphics. One of the most popular is Open Inventor. Originally a technology developed by SGI, there is today also the open source implementation [Coin](#), complemented by a toolkit binding to Qt called `SoQt`.

The basic concept of Open Inventor is that of a *scene*. A scene can be loaded from disk and saved in a special format closely related to [VRML](#). A scene consists of a collection of objects called *nodes*. Inventor already provides a rich collection of reusable nodes, such as cubes, cylinders and meshes, furthermore light sources, materials, cameras etc. Nodes are represented by C++ classes and can be combined and subclassed.

An introduction to Inventor can be found [here](#) (in general, you can substitute all mentions of `SoXt` by `SoQt` in this article).

Chapter 3

User interface

3.1 The action pattern

3.2 Defining menus and toolbars in XML

3.2.1 Introduction

While the [action pattern](#) allows to encapsulate actions triggered by the user in an object which can be "plugged" somewhere in the menu bars or toolbars, it does not by itself solve the problem of constructing the menus themselves. In particular, you have to build all popup menus in C++ code and explicitly insert the actions in a certain order, under consideration of the style guide for standard actions. This makes it pretty difficult to allow the user to customize the menus or change shortcuts to fit his needs, without changing the source code.

This problem is solved by a set of classes called `XMLGUI`. Basically, this separates actions (coded in C++) from their appearance in menu bars and tool bars (coded in XML). Without modifying any source code, menus can be simply customized by adjusting an XML file. Furthermore, it helps to make sure that standard actions (such as `File → Open` or `Help → About`) appear in the locations suggested by the style guide. `XMLGUI` is especially important for modular programs, where the items appearing in the menu bar may come from many different plugins or parts.

KDE's class for toplevel windows, `KMainWindow`, inherits `KXMLGUIClient` and therefore supports `XMLGUI` out of the box. All actions created within it must have the client's `actionCollection()` as parent. A call to `createGUI()` will then build the whole set of menu and tool bars defined the applications XML file (conventionally with the suffix `ui.rc`).

3.2.2 An example: Menu in KView

In the following, we take KDE's image view KView as example. It has a `ui.rc` file named `kvviewui.rc` which is installed with the `Makefile.am` snippet

```
rcdir = $(kde_datadir)/kview
rc_DATA = kvviewui.rc
```

Here is an excerpt from the `kvviewui.rc` file. For simplicity, we show only the definition of the View menu.

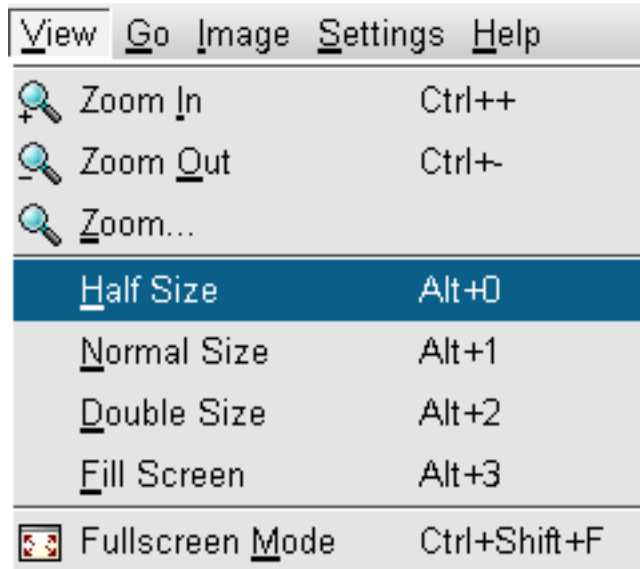
```
<!DOCTYPE kpartgui>
<kpartgui name="kview">
  <MenuBar>
    <Menu name="view" >
      <Action name="zoom50" />
      <Action name="zoom100" />
      <Action name="zoom200" />
      <Action name="zoomMaxpect" />
      <Separator/>
      <Action name="fullscreen" />
    </Menu>
  </MenuBar>
</kpartgui>
```

The corresponding part of the setup in C++ is:

```
KStdAction::zoomIn    ( this, SLOT(slotZoomIn()), ←
    actionCollection() );
KStdAction::zoomOut  ( this, SLOT(slotZoomOut()), ←
    actionCollection() );
KStdAction::zoom     ( this, SLOT(slotZoom()), ←
    actionCollection() );
new KAction          ( i18n("&Half size"), ALT+Key_0,
    this, SLOT(slotHalfSize()),
    actionCollection(), "zoom50" );
new KAction          ( i18n("&Normal size"), ALT+Key_1,
    this, SLOT(slotDoubleSize()),
    actionCollection(), "zoom100" );
new KAction          ( i18n("&Double size"), ALT+Key_2,
    this, SLOT(slotDoubleSize()),
    actionCollection(), "zoom200" );
new KAction          ( i18n("&Fill Screen"), ALT+Key_3,
    this, SLOT(slotFillScreen()),
    actionCollection(), "zoomMaxpect" ) ←
    ;
new KAction          ( i18n("Fullscreen &Mode"), CTRL+ ←
    SHIFT+Key_F,
    this, SLOT(slotFullScreen()),
    actionCollection(), "fullscreen" );
```

KDE Architecture Overview

The View menu resulting from this GUI definition looks like in this screenshot:



The XML file begins with a document type declaration. The DTD for kpartgui can be found in the kdelibs sources in `kdeui/kpartgui.dtd`. The outermost element of the file contains the instance name of the application as attribute. It can also contain a version number in the form "version=2". This is useful when you release new versions of an application with a changed menu structure, e.g. with more features. If you bump up the version number of the `ui.rc` file, KDE makes sure that any customized version of the file is discarded and the new file is used instead.

The next line, `<MenuBar>`, contains a declaration of a menu bar. You can also insert any number of `<ToolBar>` declarations in order to create some tool bars. The menu contains a submenu with the name "view". This name is already predefined, and thus you see a translated version of the word "View" in the screenshot. If you declare your own submenus, you have to add the title explicitly. For example, KView has a submenu with the title "Image" which is declared as follows:

```
<Menu name="image" >
  <text>&Image</text>
  ...
</Menu>
```

In KDE's automake framework, such titles are automatically extracted and put into the application's `.po` file, so it is considered by translators. Note that you have to write the accelerator marker "&" in the form XML compliant form "&";

Let us come back to the example. KView's View menu contains a couple of custom actions: `zoom50`, `zoom100`, `zoom200`, `zoomMaxpect` and `fullscreen`, declared

with a `<Action>` element. The separator in the screenshots corresponds to the `<Separator>` element.

You will note that some menu items do not have a corresponding element in the XML file. These are *standard actions*. Standard actions are created by the class `KStdAction`. When you create such actions in your application (such as in the C++ example above), they will automatically be inserted in a prescribed position, and possibly with an icon and a shortcut key. You can look up these locations in the file `kdeui/ui_standards.rc` in the `kdelibs` sources.

3.2.3 An example: Toolbars in Konqueror

For the discussion of toolbars, we switch to Konqueror's GUI definition. This excerpt defines the location bar, which contains the input field for URLs.

```
<ToolBar name="locationToolBar" fullWidth="true" newline="↵
true" >
  <text>Location Toolbar</text>
  <Action name="clear_location" />
  <Action name="location_label" />
  <Action name="toolbar_url_combo" />
  <Action name="go_url" />
</ToolBar>
```

The first thing we notice is that there are a lot more attributes than for menu bars. These include:

- `fullWidth`: Tells XMLGUI that the toolbar has the same width as the toplevel window. If this is "false", the toolbar only takes as much space as necessary, and further toolbars are put in the same row.
- `newline`: This is related to the option above. If `newline` is "true", the toolbar starts a new row. Otherwise it may be put in the row together with the previous toolbar.
- `noEdit`: Normally toolbars can be customized by the user, e.g. in Settings → Configure Toolbars in Konqueror. Setting this option to "true" marks this toolbar as not editable. This is important for toolbars which are filled with items at runtime, e.g. Konqueror's bookmark toolbar.
- `iconText`: Tells XMLGUI to show the text of the action next to the icon. Normally, the text is only shown as a tooltip when the mouse cursor remains over the icon for a while. Possible values for this attribute are "icononly" (shows only the icon), "textonly" (shows only the text), "icontextright" (shows the text on the right side of the icon) and "icontextbottom" (shows the text beneath the icon).
- `hidden`: If this is "true", the toolbar is not visible initially and must be activated by some menu item.

- `position`: The default for this attribute is "top", meaning that the toolbar is positioned under the menu bar. For programs with many tools, such as graphics programs, it may be interesting to replace this with "left", "right" or "bottom".

3.2.4 Dynamical menus

Obviously, an XML can only contain a static description of a user interface. Often, there are menus which change at runtime. For example, Konqueror's Location menu contains a set of items Open with Foo with the applications able to load a file with a given MIME type. Each time the document shown changes, the list of menu items is updated. XMLGUI is prepared to handle such cases with the notion of *action lists*. An action list is declared as one item in the XML file, but consists of several actions which are plugged into the menu at runtime. The above example is implemented with the following declaration in Konqueror's XML file:

```
<Menu name="file">
  <text>&amp;Location</text>
  ...
  <ActionList name="openwith">
  ...
</Menu>
```

The function `KXMLGUIClient::plugActionList()` is then used to add actions to be displayed, whereas the function `KXMLGUIClient::unplugActionList(-)` removes all plugged actions. The routine responsible for updating looks as follows:

```
void MainWindow::updateOpenWithActions ()
{
    unplugActionList ("openwith");
    openWithActions.clear ();
    for ( /* iterate over the relevant services */ ) {
        KAction *action = new KAction ( ... );
        openWithActions.append (action);
    }
    plugActionList ("openwith", openWithActions);
}
```

Note that in contrast to the static actions, the ones created here are *not* constructed with the action collection as parent, and you are responsible for deleting them for yourself. The simplest way to achieve this is by using `openWithActions.setAutoDelete(true)` in the above example.

3.2.5 Context menus

The examples above only contained cases where a main window's menu bar and toolbars were created. In the cases, the processes of constructing these

containers is completely hidden from you behind the `createGUI()` call (except if you have custom containers). However, there are cases, where you want to construct other containers and populate them with GUI definitions from the XML file. One such example are context menus. In order to get a pointer to a context menu, you have to ask the client's factory for it:

```
void MainWindow::popupRequested()
{
    QWidget *w = factory()->container("context_popup", this);
    QPopupMenu *popup = static_cast<QPopupMenu *>(w);
    popup->exec(QCursor::pos());
}
```

The method `KXMLGUIFactory::container()` used above looks whether it finds a container in the XML file with the given name. Thus, a possible definition could look as follows:

```
...
<Menu name="context_popup">
  <Action name="file_add"/>
  <Action name="file_remove"/>
</Menu>
...
```

3.3 Providing online help

Making a program easy and intuitive to use involves a wide range of facilities which are usually called online help. Online help has several, partially conflicting goals: on the one, it should give the user answers to the question "How can I do a certain task?", on the other hand it should help the user exploring the application and finding features he doesn't yet know about. It is important to recognize that this can only be achieved by offering several levels of help:

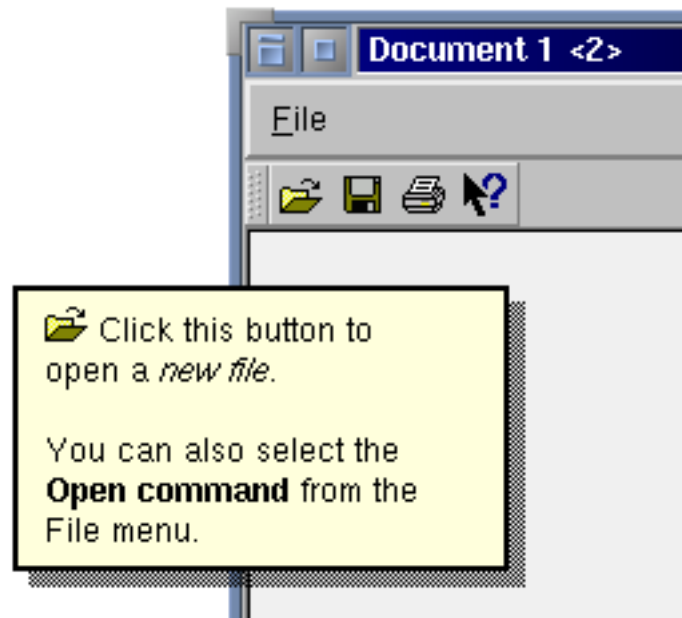
- Tooltips are tiny labels that pop up over user interface elements when the mouse remains there longer. They are especially important for tool-bars, where icons are not always sufficient to explain the purpose of a button.
- "What's this?" help is usually a longer and richer explanation of a widget or a menu item. It is also more clunky to use: In dialogs, it can be invoked in two ways: either by pressing Shift-F1 or by clicking on the question mark in the title bar (where the support of the latter depends on the window manager). The mouse pointer then turns into an arrow with a question mark, and the help window appears when a user interface element has been clicked. "What's this?" help for menu items is usually activated by a button in the toolbar which contains an arrow and a question mark.

KDE Architecture Overview

- The problem with this approach is that the user can't see whether a widget provides help or not. When the user activates the question mark button and doesn't get any help window when clicking on a user interface element, he will get frustrated very quickly.

The advantage of "What's this?" help windows as provided by Qt and KDE is that they can contain [rich text](#), i.e. they may contain different fonts, bold and italic text and even images and tables.

An example of "What's this?" help:



- Finally, every program should have a manual. A manual is normally viewed in KHelpCenter by activating the Help menu. That means, a complete additional application pops up and diverts the user from his work. Consequently, consulting the manual should only be necessary if other facilities like tooltips and what's this help are not sufficient. Of course, a manual has the advantage that it does not explain single, isolated aspects of the user interface. Instead, it can explain aspects of the application in a greater context. Manuals for KDE are written using the [DocBook](#) markup language.

From the programmer's point of view, Qt provides an easy to use API for on-line help. To assign a tooltip to widget, use the [QToolTip](#) class.

```
QToolTip::add(w, i18n("This widget does something."))
```

If the menu bars and tool bars are created using the [action pattern](#), the string used as tooltip is derived from the first argument of the [KAction](#) constructor:

KDE Architecture Overview

```
action = new KAction(i18n("&Delete"), "editdelete",  
                    SHIFT+Key_Delete, actionCollection(), " ←  
                    del")
```

Here it is also possible to assign a text which is shown in the status bar when the respective menu item is highlighted:

```
action->setStatusText(i18n("Deletes the marked file"))
```

The API for "What's this?" help is very similar. In dialogs, use the following code:

```
QWhatsThis::add(w, i18n("<qt>This demonstrates <b>Qt</b>'s"  
                        " rich text engine.<ul>"  
                        "<li>Foo</li>"  
                        "<li>Bar</li>"  
                        "</ul></qt>"))
```

For menu items, use

```
action->setWhatsThis(i18n("Deletes the marked file"))
```

The invocation of KHelpCenter is encapsulated in the [KApplication](#) class. In order to show the manual of your application, just use

```
kapp->invokeHelp()
```

This displays the first page with the table of contents. When you want to display only a certain section of the manual, you can give an additional argument to `invokeHelp()` determining the anchor which the browser jumps to.

Chapter 4

Components and services

4.1 KDE services

4.1.1 What are KDE services?

The notion of a *service* is a central concept in KDE's modular architecture. There is no strict technical implementation connected with this term - services can be plugins in the form of shared libraries, or they can be programs controlled via [DCOP](#). By claiming to be of a certain *service type*, a service promises to implement certain APIs or features. In C++ terms, one can think of a service type as an abstract class, and a service as an implementation of that interface.

The advantage of this separation is clear: An application utilizing a service type does not have to know about possible implementations of it. It just uses the APIs associated with the service type. In this way, the used service can be changed without affecting the application. Also, the user can configure which services he prefers for certain features.

Some examples:

- The HTML rendering engine used in Konqueror is an embedable component that implements the service types `KParts/ReadOnlyPart` and `Browser/View`.
- In KDevelop HEAD, most functionality is packaged in plugins with the service type `KDevelop/Part`. At startup, all services with this type are loaded, such that you can extend the IDE in a very flexible way.
- In the icon view, Konqueror displays - if enabled - thumbnail pictures of images, HTML pages, PDF and text files. This ability can be extended. If you want it to display preview pictures of your own data files with some MIME type, you can implement a service with service type `ThumbCreator`.

Obviously, a service is not only characterized by the service types it implements, but also by some *properties*. For example, a `ThumbCreator` does not only claim to implement the C++ class with the type `ThumbCreator`, it also has a list of MIME types it is responsible for. Similarly, `KDevelop` parts have the programming language they support as a property. When an application requests a service type, it can also list constraints on the properties of the service. In the above example, when `KDevelop` loads the plugins for a Java project, it asks only for the plugins which have Java as the programming language property. For this purpose, KDE contains a full-blown CORBA-like *trader* with a complex query language.

4.1.2 Defining service types

New service types are added by installing a description of them into the directory `KDEDIR/share/servicetypes`. In an automake framework, this can be done with this `Makefile.am` snippet:

```
kde_servicetypesdir_DATA = kdeveloppart.desktop
EXTRA_DIST = $(kde_servicetypesdir_DATA)
```

The definition `kdeveloppart.desktop` of a `KDevelop` part looks as follows:

```
[Desktop Entry]
Type=ServiceType
X-KDE-ServiceType=KDevelop/Part
Name=KDevelop Part

[PropertyDef::X-KDevelop-Scope]
Type=QString

[PropertyDef::X-KDevelop-ProgrammingLanguages]
Type=QStringList

[PropertyDef::X-KDevelop-Args]
Type=QString
```

In addition to the usual entries, this example demonstrates how you declare that a service has some properties. Each property definition corresponds to a group `[PropertyDef::name]` in the configuration file. In this group, the `Type` entry declares the type of the property. Possible types are everything that can be stored in a `QVariant`.

4.1.3 Defining shared library services

Service definitions are stored in the directory `KDEDIR/share/services`:

```
kde_servicesdir_DATA = kdevdoxygen.desktop
EXTRA_DIST = $(kde_servicesdir_DATA)
```

KDE Architecture Overview

The content of the following example file `kdevdoxygen.desktop` defines the `KDevDoxygen` plugin with the service type `KDevelop/Part`:

```
[Desktop Entry]
Type=Service
Comment=Doxygen
Name=KDevDoxygen
ServiceTypes=KDevelop/Part
X-KDE-Library=libkdevdoxygen
X-KDevelop-ProgrammingLanguages=C,C++,Java
X-KDevelop-Scope=Project
```

In addition to the usual declarations, an important entry is `X-KDE-Library`. This contains the name of the `libtool` library (without the `.la` extension). It also fixes (with the prefix `init_` prepended) the name of the exported symbol in the library which returns an object factory. For the above example, the library must contain the following function:

```
extern "C" {
    void *init_libkdevdoxygen()
    {
        return new DoxygenFactory;
    }
};
```

The type of the factory class `DoxygenFactory` depends on the specific service the service implements. In our example of a `KDevelop` plugin, the factory must be a `KDevFactory` (which inherits `KLibFactory`). More common examples are `KParts::Factory` which is supposed to produce `KParts::ReadOnlyPart` objects or in most cases the generic `KLibFactory`.

4.1.4 Using shared library services

In order to use a shared library service in an application, you need to obtain a `KService` object representing it. This is discussed in the [section about MIME types](#) (and in a section about the trader to be written :-)

With the `KService` object at hand, you can very simply load the library and get a pointer to its factory object:

```
KService *service = ...
QString libName = QFile::encodeName(service->library());
KLibFactory *factory = KLibLoader::self()->factory(libName);
if (!factory) {
    QString name = service->name();
    QString errorMessage = KLibLoader::self()-> ←
        lastErrorMessage();
    KMessageBox::error(0, i18n("There was an error loading ←
        service %1.\n"
```

KDE Architecture Overview

```
        "The diagnostics from libtool ←  
        is:\n%2") ←  
        .arg(name).arg(errorMessage); ←  
    } ←
```

From this point, the further proceeding depends again on the service type. For generic plugins, you create objects with the method `KLibFactory::create()`. For KParts, you must cast the factory pointer to the more specific `KParts::Factory` and use its `create()` method:

```
if (factory->inherits("KParts::Factory")) { ←  
    KParts::Factory *partFactory = static_cast<KParts:: ←  
        Factory*>(factory); ←  
    QObject *obj = partFactory->createPart(parentWidget, ←  
        widgetName, ←  
        parent, name, " ←  
        KParts:: ←  
        ReadOnlyPart") ←  
        ; ←  
    ... ←  
} else { ←  
    cout << "Service does not implement the right factory" << ←  
        endl; ←  
} ←
```

4.1.5 Defining DCOP services

A DCOP service is usually implemented as a program that is started up when it is needed. It then goes into a loop and listens for DCOP connections. The program may be an interactive one, but it may also run completely or for a part of its lifetime as a daemon in the background without the user noticing it. An example for such a daemon is `kio_uiserver`, which implements user interaction such as progress dialog for the KIO library. The advantage of such a centralized daemon in this context is that e.g. the download progress for several different files can be shown in one window, even if those downloads were initiated from different applications.

A DCOP service is defined differently from a shared library service. Of course, it doesn't specify a library, but instead an executable. Also, DCOP services do not specify a `ServiceType` line, because usually they are started by their name. As additional properties, it contains two lines:

`X-DCOP-ServiceType` specifies the way the service is started. The value `Unique` says that the service must not be started more than once. This means, if you try to start this service (e.g. via `KApplication::startServiceByName()`), KDE looks whether it is already registered with DCOP and uses the running service. If it is not registered yet, KDE will start it up and wait until is registered. Thus, you can immediately send DCOP calls to the service. In such a case, the service should be implemented as a `KUniqueApplication`.

KDE Architecture Overview

The value `Multi` for `X-DCOP-ServiceType` says that multiple instances of the service can coexist, so every attempt to start the service will create another process. As a last possibility the value `None` can be used. In this case, a start of the service will not wait until it is registered with DCOP.

`X-KDE-StartupNotify` should normally be set to `false`. Otherwise, when the program is started, the task bar will show a startup notification, or, depending on the user's settings, the cursor will be changed.

Here is the definition of `kio_uiserver`:

```
[Desktop Entry]
Type=Service
Name=kio_uiserver
Exec=kio_uiserver
X-DCOP-ServiceType=Unique
X-KDE-StartupNotify=false
```

4.1.6 Using DCOP services

A DCOP service is started with one of several methods in the `KApplication` class:

```
DCOPClient *client = kapp->dcopClient();
client->attach();
if (!client->isApplicationRegistered("kio_uiserver")) {
    QString error;
    if (KApplication::startServiceByName("kio_uiserver", ←
        QStringList(), &error))
        cout << "Starting kioserver failed with message " << ←
            error << endl;
}
...
QByteArray data, replyData;
QCString replyType;
QDataStream arg(data, IO_WriteOnly);
arg << true;
if (!client->call("kio_uiserver", "UIServer", "setListMode( ←
    bool)",
                data, replyType, replyData))
    cout << "Call to kio_uiserver failed" << endl;
...
```

Note that the example of a DCOP call given here uses explicit marshalling of arguments. Often you will want to use a stub generated by `dcopidl2cpp` instead, because it is much simpler and less error prone.

In the example given here, the service was started "by name", i.e. the first argument to `KApplication::startServiceByName()` is the name appearing

in the `Name` line of the desktop file. An alternative is to use `KApplication::startServiceByDesktopName()`, which takes the file name of its desktop file as argument, i.e. in this case `"kio_uiserver.desktop"`.

All these calls take a list of URLs as a second argument, which is given to the service on the command line. The third argument is a pointer to a `QString`. If starting the service fails, this argument is set to a translated error message.

4.2 MIME types

4.2.1 What are MIME types?

MIME types are used to describe the content type of files or data chunks. Originally they were introduced in order to allow sending around image or sound files etc. by e-mail (MIME stands for "Multipurpose Internet Mail Extensions"). Later this system was also used by web browsers to determine how to present data sent by a web server to the user. For example, an HTML page has a MIME type `"text/html"`, a postscript file `"application/postscript"`. In KDE, this concept is used at a variety of places:

- In Konqueror's icon view, files are represented by icons. Each MIME type has a certain associated icon shown here.
- When you click onto a file icon or a file name in Konqueror, either the file is shown in an embedded view, or an application associated with the file type is opened.
- When you drag and drop some data from one application to another (or within the same application), the drop target may choose to accept only certain data types. Furthermore, it will handle image data different from textual data.
- Clipboard data has a MIME type. Traditionally, X programs only handle pixmaps or texts, but with Qt, there are no restrictions on the data type.

From the above examples, it is clear that MIME handling is a complex issue. First, it is necessary to establish a mapping from file names to MIME types. KDE goes one step further in allowing even file contents to be mapped to MIME types, for cases in which the file name is not available. Second, it is necessary to map MIME types to applications or libraries which can view or edit a file with a certain type, or create a thumbnail picture for it.

There is a variety of APIs to figure out the MIME type of data or files. In general, there is a certain speed/reliability trade-off you have to make. You can find out the type of a file by examining only its file name (i.e. in most cases the file name extension). For example, a file `foo.jpg` is normally `"image/jpeg"`. In cases where the extension is stripped off this is not safe, and you actually have to look at the contents of the file. This is of course slower, in particular for files that have to be downloaded via HTTP first. The content-based method is based

on the file `KDEDIR/share/mimelnk/magic` and therefore difficult to extend. But in general, MIME type information can easily be made available to the system by installing a `.desktop` file, and it is efficiently and conveniently available through the KDE libraries.

4.2.2 Defining MIME types

Let us define a type `"application/x-foo"` for our new `foobar` program. To this end, you have to write a file `foo.desktop` and install it into `KDEDIR/share/mimelnk/application`. (This is the usual location, which may differ between distributions). This can be done by adding this to the `Makefile.am`:

```
mimedir = $(kde_mimedir)/application
mime_DATA = foo.desktop
EXTRA_DIST = $(mime_DATA)
```

The file `foo.desktop` should look as follows:

```
[Desktop Entry]
Type=MimeType
MimeType=application/x-foo
Icon=fooicon
Patterns=*.foo;
DefaultApp=foobar
Comment=Foo Data File
Comment[de]=Foo Datei
```

The "Comment" entry is supposed to be translated. Since the `.desktop` file specifies an icon, you should also install an icon `fooicon.png`, which represents the file e.g. in Konqueror.

In the KDE libraries, such a type definition is mapped to an instance of the class [KMimeType](#). Use this like in the following example:

```
KMimeType::Ptr type = KMimeType::mimeType("application/x-foo ←
");
cout << "Type:      " << type->name() < endl;
cout << "Icon:      " << type->icon() < endl;
cout << "Comment:  " << type->icon() < endl;
QStringList patterns = type->patterns();
QStringList::ConstIterator it;
for (it = patterns.begin(); it != patterns.end(); ++it)
    cout << "Pattern: " << (*it) << endl;
```

4.2.3 Determining the MIME type of data

The fast method for determining the type of a file is `KMimeType::findByURL()`. This looks for the URL string and in most cases determines the type from the

KDE Architecture Overview

extension. For certain protocols (e.g. http, man, info), this mechanism is not used. For example, CGI scripts on web servers written in Perl often have the extension .pl, which would indicate a "text/x-perl" type. However, the file delivered by the server is the output of this script, which is normally HTML. For such a case, `KMimeType::findByURL()` returns the MIME type "application/octet-stream" (available through `KMimeType::defaultMimeType()`), which indicates a failure to find out the type.

```
KMimeType::Ptr type = KMimeType::findByURL("/home/bernd/ ↵
    foobar.jpg");
if (type->name() == KMimeType::defaultMimeType())
    cout << "Could not find out type" << endl;
else
    cout << "Type: " << type->name() << endl;
```

(this method has some more arguments, but these are undocumented, so simply forget about them.)

You may want to find out a MIME from the contents of file instead of the file name. This is more reliable, but also slower, as it requires reading a part of the file. This is done with the [KMimeMagic](#) class, which has different error handling:

```
KMimeMagicResult *result = KMimeMagic::self()->findFileType ↵
    ("/home/bernd/foobar.jpg");
if (!result || !result->isValid())
    cout << "Could not find out type" << endl;
else
    cout << "Type: " << result->mimeType() << endl;
```

As a variant of this function, you can also determine the type of a memory chunk. This is e.g. used in Kate in order to find out the highlighting mode:

```
QByteArray array;
...
KMimeMagicResult *result = KMimeMagic::self()->findBufferType ↵
    (array);
if (!result || !result->isValid())
    cout << "Could not find out type" << endl;
else
    cout << "Type: " << result->mimeType() << endl;
```

Of course, even `KMimeMagic` is only able to determine a file type from the contents of a local file. For remote files, there is a further possibility:

```
KURL url("http://developer.kde.org/favicon.ico");
QString type = KIO::NetAccess::mimetype(url);
if (type == KMimeType::defaultMimeType())
    cout << "Could not find out type" << endl;
else
    cout << "Type: " << type << endl;
```

KDE Architecture Overview

This starts a KIO job to download a part of the file and check this. Note that this function is perhaps quite slow and blocks the program. Normally you will only want to use this if `KMimeType::findByURL()` has returned `"application/octet-stream"`.

On the other hand, if you do not want to block your application, you can also explicitly start the KIO job and connect to some of its signals:

```
void FooClass::findType ()
{
    KURL url("http://developer.kde.org/favicon.ico");
    KIO::MimeTypeJob *job = KIO::mimetype(url);
    connect( job, SIGNAL(result(KIO::Job*)),
            this, SLOT(mimeResult(KIO::Job*)) );
}

void FooClass::mimeResult (KIO::Job *job)
{
    if (job->error())
        job->showErrorDialog();
    else
        cout << "MIME type: " << ((KIO::MimeTypeJob *)job)-> ←
            mimetype() << endl;
}
```

4.2.4 Mapping a MIME type to an application or service

When an application is installed, it installs a `.desktop` file which contains a list of MIME types this application can load. Similarly, components like KParts make this information available by their service `.desktop` files. So in general, there are several programs and components which can process a given MIME type. You can obtain such a list from the class `KServiceTypeProfile`:

```
KService::OfferList offers = KServiceTypeProfile::offers(" ←
    text/html", "Application");
KService::OfferList::ConstIterator it;
for (it = offers.begin(); it != offers.end(); ++it) {
    KService::Ptr service = (*it);
    cout << "Name: " << service->name() << endl;
}
```

The return value of this function is a list of service offers. A `KServiceOffer` object packages a `KService::Ptr` together with a preference number. The list returned by `KServiceTypeProfile::offers()` is ordered by the user's preference. The user can change this by calling `"keditfiletype text/html"` or choosing Edit File Type on Konqueror's context menu on a HTML file.

In the above example, an offer list of the applications supporting `text/html` was requested. This will - among others - contain HTML editors like Quanta

Plus. You can also replace the second argument "Application" by "KPart-s::ReadOnlyPart". In that case, you get a list of embedable components for presenting HTML content, for example KHTML.

In most cases, you are not interested in the list of all service offers for a combination of MIME type and service type. There is a convenience function which gives you only the service offer with the highest preference:

```
KService::Ptr offer = KServiceTypeProfile::preferredService(" ←  
    text/html", "Application");  
if (offer)  
    cout << "Name: " << service->name() << endl;  
else  
    cout << "No appropriate service found" << endl;
```

For even more complex queries, there is a full-blown CORBA-like [trader](#).

In order to run an application service with some URLs, use [KRun](#):

```
KURL::List urlList;  
urlList << "http://www.ietf.org/rfc/rfc1341.txt?number=1341";  
urlList << "http://www.ietf.org/rfc/rfc2046.txt?number=2046";  
KRun::run(offer.service(), urlList);
```

4.2.5 Miscellaneous

In this section, we want to list some APIs which are loosely related to the previous discussion.

Getting an icon for a URL. This looks for the type of the URL and returns the associated icon.

```
KURL url("ftp://ftp.kde.org/pub/incoming/wibble.c");  
QString icon = KMimeType::iconForURL(url);
```

Running a URL. This looks for the type of the URL and starts the user's preferred program associated with this type.

```
KURL url("http://dot.kde.org");  
new KRun(url);
```

4.3 Network transparency

4.3.1 Introduction

In the age of the world wide web, it is of essential importance that desktop applications can access resources over the internet: they should be able to download files from a web server, write files to an ftp server or read mails from a

KDE Architecture Overview

web server. Often, the ability to access files regardless of their location is called *network transparency*.

In the past, different approaches to this goals were implemented. The old NFS file system is an attempt to implement network transparency on the level of the POSIX API. While this approach works quite well in local, closely coupled networks, it does not scale for resources to which access is unreliable and possibly slow. Here, *asynchronicity* is important. While you are waiting for your web browser to download a page, the user interface should not block. Also, the page rendering should not begin when the page is completely available, but should updated regularly as data comes in.

In the KDE libraries, network transparency is implemented in the KIO API. The central concept of this architecture is an IO *job*. A job may copy, or delete files or similar things. Once a job is started, it works in the background and does not block the application. Any communication from the job back to the application - like delivering data or progress information - is done integrated with the Qt event loop.

Background operation is achieved by starting *ioslaves* to perform certain tasks. ioslaves are started as separate processes and are communicated with through UNIX domain sockets. In this way, no multi-threading is necessary and unstable slaves can not crash the application that uses them.

File locations are expressed by the widely used URLs. But in KDE, URLs do not only expand the range of addressable files beyond the local file system. It also goes in the opposite direction - e.g. you can browse into tar archives. This is achieved by nesting URLs. For example, a file in a tar archive on a http server could have the URL

```
http://www-com.physik.hu-berlin.de/~bernd/article.tgz#tar:/ ←  
paper.tex
```

4.3.2 Using KIO

In most cases, jobs are created by calling functions in the KIO namespace. These functions take one or two URLs as arguments, and possible other necessary parameters. When the job is finished, it emits the signal `result(KIO::Job*)`. After this signal has been emitted, the job deletes itself. Thus, a typical use case will look like this:

```
void FooClass::makeDirectory()  
{  
    SimpleJob *job = KIO::mkdir(KURL("file:/home/bernd/kiodir ←  
    "));  
    connect( job, SIGNAL(result(KIO::Job*)),  
            this, SLOT(mkdirResult(KIO::Job*)) );  
}  
  
void FooClass::mkdirResult(KIO::Job *job)
```

KDE Architecture Overview

```
{
    if (job->error())
        job->showErrorDialog();
    else
        cout << "mkdir went fine" << endl;
}
```

Depending on the type of the job, you may connect also to other signals.

Here is an overview over the possible functions:

KIO::mkdir(const KURL &url, int permission) Creates a directory, optionally with certain permissions.

KIO::rmdir(const KURL &url) Removes a directory.

KIO::chmod(const KURL &url, int permissions) Changes the permissions of a file.

KIO::rename(const KURL &src, const KURL &dest, bool overwrite) Renames a file.

KIO::symlink(const QString &target, const KURL &dest, bool overwrite, bool showProgressInfo) Creates a symbolic link.

KIO::stat(const KURL &url, bool showProgressInfo) Finds out certain information about the file, such as size, modification time and permissions. The information can be obtained from `KIO::StatJob::statResult()` after the job has finished.

KIO::get(const KURL &url, bool reload, bool showProgressInfo) Transfers data from a URL.

KIO::put(const KURL &url, int permissions, bool overwrite, bool resume, bool showProgressInfo) Transfers data to a URL.

KIO::http_post(const KURL &url, const QByteArray &data, bool showProgressInfo) Posts data. Special for HTTP.

KIO::mimetype(const KURL &url, bool showProgressInfo) Tries to find the MIME type of the URL. The type can be obtained from `KIO::MimetypeJob::mimetype()` after the job has finished.

KIO::file_copy(const KURL &src, const KURL &dest, int permissions, bool overwrite, bool resume, bool showProgressInfo) Copies a single file.

KIO::file_move(const KURL &src, const KURL &dest, int permissions, bool overwrite, bool resume, bool showProgressInfo) Renames or moves a single file.

KIO::file_delete(const KURL &url, bool showProgressInfo) Deletes a single file.

KIO::listDir(const KURL &url, bool showProgressInfo) Lists the contents of a directory. Each time some new entries are known, the signal `KIO::ListJob::entries()` is emitted.

KIO::listRecursive(const KURL &url, bool showProgressInfo) Similar to the listDir() function, but this one is recursive.

KIO::copy(const KURL &src, const KURL &dest, bool showProgressInfo) Copies a file or directory. Directories are copied recursively.

KIO::move(const KURL &src, const KURL &dest, bool showProgressInfo) Moves or renames a file or directory.

KIO::del(const KURL &src, bool shred, bool showProgressInfo) Deletes a file or directory.

4.3.3 Directory entries

Both the KIO::stat() and KIO::listDir() jobs return their results as a type UDSEntry, UDSEntryList resp. The latter is defined as QList<UDSEntry>. The acronym UDS stands for "Universal directory service". The principle behind it is that the a directory entry only carries the information which an ioslave can provide, not more. For example, the http slave does not provide any information about access permissions or file owners. Instead, a UDSEntry is a list of UDSAtoms. Each atom provides a specific piece of information. It consists of a type stored in m_uds and either an integer value in m_long or a string value in m_str, depending on the type.

The following types are currently defined:

- UDS_SIZE (integer) - Size of the file.
- UDS_USER (string) - User owning the file.
- UDS_GROUP (string) - Group owning the file.
- UDS_NAME (string) - File name.
- UDS_ACCESS (integer) - Permission rights of the file, as e.g. stored by the libc function stat() in the st_mode field.
- UDS_FILE_TYPE (integer) - The file type, as e.g. stored by stat() in the st_mode field. Therefore you can use the usual libc macros like S_ISDIR to test this value. Note that the data provided by ioslaves corresponds to stat(), not lstat(), i.e. in case of symbolic links, the file type here is the type of the file pointed to by the link, not the link itself.
- UDS_LINK_DEST (string) - In case of a symbolic link, the name of the file pointed to.
- UDS_MODIFICATION_TIME (integer) - The time (as in the type time_t) when the file was last modified, as e.g. stored by stat() in the st_mtime field.
- UDS_ACCESS_TIME (integer) - The time when the file was last accessed, as e.g. stored by stat() in the st_atime field.

- `UDS_CREATION_TIME` (integer) - The time when the file was created, as e.g. stored by `stat()` in the `st_ctime` field.
- `UDS_URL` (string) - Provides a URL of a file, if it is not simply the concatenation of directory URL and file name.
- `UDS_MIME_TYPE` (string) - MIME type of the file
- `UDS_GUESSED_MIME_TYPE` (string) - MIME type of the file as guessed by the slave. The difference to the previous type is that the one provided here should not be taken as reliable (because determining it in a reliable way would be too expensive). For example, the `KRun` class explicitly checks the MIME type if it does not have reliable information.

Although the way of storing information about files in a `UDSEntry` is flexible and practical from the ioslave point of view, it is a mess to use for the application programmer. For example, in order to find out the MIME type of the file, you have to iterate over all atoms and test whether `m_uds` is `UDS_MIME_TYPE`. Fortunately, there is an API which is a lot easier to use: the class `KFileItem`.

4.3.4 Synchronous usage

Often, the asynchronous API of KIO is too complex to use and therefore implementing full asynchronicity is not a priority. For example, in a program that can only handle one document file at a time, there is little that can be done while the program is downloading a file anyway. For these simple cases, there is a much simpler API in the form of a set of static functions in `KIO::NetAccess`. For example, in order to copy a file, use

```
KURL source, target;
source = ...;
target = ...
KIO::NetAccess::copy(source, target);
```

The function will return after the complete copying process has finished. Still, this method provides a progress dialog, and it makes sure that the application processes repaint events.

A particularly interesting combination of functions is `download()` in combination with `removeTempFile()`. The former downloads a file from given URL and stores it in a temporary file with a unique name. The name is stored in the second argument. *If* the URL is local, the file is not downloaded, and instead the second argument is set to the local file name. The function `removeTempFile()` deletes the file given by its argument if the file is the result of a former download. If that is not the case, it does nothing. Thus, a very easy to use way of loading files regardless of their location is the following code snippet:

```
KURL url;
url = ...;
QString tempFile;
```

KDE Architecture Overview

```
if (KIO::NetAccess::download(url, tempFile) {
    // load the file with the name tempFile
    KIO::NetAccess::removeTempFile(tempFile);
}
```

4.3.5 Meta data

As can be seen above, the interface to IO jobs is quite abstract and does not consider any exchange of information between application and IO slave that is protocol specific. This is not always appropriate. For example, you may give certain parameters to the HTTP slave to control its caching behavior or send a bunch of cookies with the request. For this need, the concept of meta data has been introduced. When a job is created, you can configure it by adding meta data to it. Each item of meta data consists of a key/value pair. For example, in order to prevent the HTTP slave from loading a web page from its cache, you can use:

```
void FooClass::reloadPage()
{
    KURL url("http://www.kdevelop.org/index.html");
    KIO::TransferJob *job = KIO::get(url, true, false);
    job->addMetaData("cache", "reload");
    ...
}
```

The same technique is used in the other direction, i.e. for communication from the slave to the application. The method `Job::queryMetaData()` asks for the value of the certain key delivered by the slave. For the HTTP slave, one such example is the key "modified", which contains a (stringified representation of) the date when the web page was last modified. An example how you can use this is the following:

```
void FooClass::printModifiedDate()
{
    KURL url("http://developer.kde.org/documentation/kde2arch ←
/index.html");
    KIO::TransferJob *job = KIO::get(url, true, false);
    connect( job, SIGNAL(result(KIO::Job*)),
            this, SLOT(transferResult(KIO::Job*)) );
}

void FooClass::transferResult(KIO::Job *job)
{
    QString mimetype;
    if (job->error())
        job->showErrorDialog();
    else {
        KIO::TransferJob *transferJob = (KIO::TransferJob*) ←
        job;
    }
}
```

KDE Architecture Overview

```
QString modified = transferJob->queryMetaData(" ←  
    modified");  
cout << "Last modified: " << modified << endl;  
}
```

4.3.6 Scheduling

When using the KIO API, you usually do not have to cope with the details of starting IO slaves and communicating with them. The normal use case is to start a job and with some parameters and handle the signals the jobs emits.

Behind the curtains, the scenario is a lot more complicated. When you create a job, it is put in a queue. When the application goes back to the event loop, KIO allocates slave processes for the jobs in the queue. For the first jobs started, this is trivial: an IO slave for the appropriate protocol is started. However, after the job (like a download from an http server) has finished, it is not immediately killed. Instead, it is put in a pool of idle slaves and killed after a certain time of inactivity (current 3 minutes). If a new request for the same protocol and host arrives, the slave is reused. The obvious advantage is that for a series of jobs for the same host, the cost for creating new processes and possibly going through an authentication handshake is saved.

Of course, reusing is only possible when the existing slave has already finished its previous job. when a new request arrives while an existing slave process is still running, a new process must be started and used. In the API usage in the examples above, there are no limitation for creating new slave processes: if you start a consecutive series of downloads for 20 different files, then KIO will start 20 slave processes. This scheme of assigning slaves to jobs is called *direct*. It not always the most appropriate scheme, as it may need much memory and put a high load on both the client and server machines.

So there is a different way. You can *schedule* jobs. If you do this, only a limited number (currently 3) of slave processes for a protocol will be created. If you create more jobs than that, they are put in a queue and are processed when a slave process becomes idle. This is done as follows:

```
KURL url("http://developer.kde.org/documentation/kde2arch/ ←  
    index.html");  
KIO::TransferJob *job = KIO::get(url, true, false);  
KIO::Scheduler::scheduleJob(job);
```

A third possibility is *connection oriented*. For example, for the IMAP slave, it does not make any sense to start multiple processes for the same server. Only one IMAP connection at a time should be enforced. In this case, the application must explicitly deal with the notion of a slave. It has to deallocate a slave for a certain connection and then assign all jobs which should go through the same connection to the same slave. This can again be easily achieved by using the KIO::Scheduler:

KDE Architecture Overview

```
KURL baseUrl("imap://bernd@albert.physik.hu-berlin.de");
KIO::Slave *slave = KIO::Scheduler::getConnectedSlave(baseUrl ←
);

KIO::TransferJob *job1 = KIO::get(KURL(baseUrl, "/INBOX;UID ←
=79374"));
KIO::Scheduler::assignJobToSlave(slave, job1);

KIO::TransferJob *job2 = KIO::get(KURL(baseUrl, "/INBOX;UID ←
=86793"));
KIO::Scheduler::assignJobToSlave(slave, job2);

...

KIO::Scheduler::disconnectSlave(slave);
```

You may only disconnect the slave after all jobs assigned to it are guaranteed to be finished.

4.3.7 Defining an ioslave

In the following we discuss how you can add a new ioslave to the system. In analogy to services, new ioslaves are advertised to the system by installing a little configuration file. The following Makefile.am snippet installs the ftp protocol:

```
protocoldir = $(kde_servicesdir)
protocol_DATA = ftp.protocol
EXTRA_DIST = $(mime_DATA)
```

The contents of the file ftp.protocol is as follows:

```
[Protocol]
exec=kio_ftp
protocol=ftp
input=none
output=filesystem
listing=Name,Type,Size,Date,Access,Owner,Group,Link,
reading=true
writing=true
mkdir=true
deleting=true
Icon=ftp
```

The "protocol" entry defines for which protocol this slave is responsible. "-exec" is (in contrast what you would expect naively) the name of the library that implements the slave. When the slave is supposed to start, the "**kdeinit**"

executable is started which in turn loads this library into its address space. So in practice, you can think of the running slave as a separate process although it is implemented as library. The advantage of this mechanism is that it saves a lot of memory and reduces the time needed by the runtime linker.

The "input" and "output" lines are not used currently.

The remaining lines in the `.protocol` file define which abilities the slave has. In general, the features a slave must implement are much simpler than the features the KIO API provides for the application. The reason for this is that complex jobs are scheduled to a couple of subjobs. For example, in order to list a directory recursively, one job will be started for the toplevel directory. Then for each subdirectory reported back, new subjobs are started. A scheduler in KIO makes sure that not too many jobs are active at the same time. Similarly, in order to copy a file within a protocol that does not support copying directly (like the `ftp:` protocol), KIO can read the source file and then write the data to the destination file. For this to work, the `.protocol` must advertise the actions its slave supports.

Since slaves are loaded as shared libraries, but constitute standalone programs, their code framework looks a bit different from normal shared library plugins. The function which is called to start the slave is called `kdmain()`. This function does some initializations and then goes into an event loop and waits for requests by the application using it. This looks as follows:

```
extern "C" { int kdmain(int argc, char **argv); }

int kdmain(int argc, char **argv)
{
    KLocale::setMainCatalogue("kdelibs");
    KInstance instance("kio_ftp");
    (void) KGlobal::locale();

    if (argc != 4) {
        fprintf(stderr, "Usage: kio_ftp protocol "
                    "domain-socket1 domain-socket2\n");
        exit(-1);
    }

    FtpSlave slave(argv[2], argv[3]);
    slave.dispatchLoop();
    return 0;
}
```

4.3.8 Implementing an ioslave

Slaves are implemented as subclasses of `KIO::SlaveBase` (`FtpSlave` in the above example). Thus, the actions listed in the `.protocol` correspond to certain virtual functions in `KIO::SlaveBase` the slave implementation must reimplement. Here is a list of possible actions and the corresponding virtual functions:

reading - Reads data from a URL void get(const KURL &url)

writing - Writes data to a URL and create the file if it does not exist yet. void put(const KURL &url, int permissions, bool overwrite, bool resume)

moving - Renames a file. void rename(const KURL &src, const KURL &dest, bool overwrite)

deleting - Deletes a file or directory. void del(const KURL &url, bool isFile)

listing - Lists the contents of a directory. void listDir(const KURL &url)

makedir - Creates a directory. void mkdir(const KURL &url, int permissions)

Additionally, there are reimplementable functions not listed in the `.protocol` file. For these operations, KIO automatically determines whether they are supported or not (i.e. the default implementation returns an error).

Delivers information about a file, similar to the C function stat(). void stat(const KURL &url)

Changes the access permissions of a file. void chmod(const KURL &url, int permissions)

Determines the MIME type of a file. void mimetype(const KURL &url)

Copies a file. copy(const KURL &url, const KURL &dest, int permissions, bool overwrite)

Creates a symbolic link. void symlink(const QString &target, const KURL &dest, bool overwrite)

All these implementation should end with one of two calls: If the operation was successful, they should call `finished()`. If an error has occurred, `error()` should be called with an error code as first argument and a string in the second. Possible error codes are listed as enum `KIO::Error`. The second argument is usually the URL in question. It is used e.g. in `KIO::Job::showErrorDialog()` in order to parameterize the human-readable error message.

For slaves that correspond to network protocols, it might be interesting to reimplement the method `SlaveBase::setHost()`. This is called to tell the slave process about the host and port, and the user name and password to log in. In general, meta data set by the application can be queried by `SlaveBase::metaData()`. You can check for the existence of meta data of a certain key with `SlaveBase::hasMetaData()`.

4.3.9 Communicating back to the application

Various actions implemented in a slave need some way to communicate data back to the application using the slave process:

- `get()` sends blocks of data. This is done with `data()`, which takes a `QByteArray` as argument. Of course, you do not need to send all data at once. If you send a large file, call `data()` with smaller data blocks, so the application can process them. Call `finished()` when the transfer is finished.
- `listDir()` reports information about the entries of a directory. For this purpose, call `listEntries()` with a `KIO::UDSEntryList` as argument. Analogously to `data()`, you can call this several times. When you are finished, call `listEntry()` with the second argument set to true. You may also call `totalSize()` to report the total number of directory entries, if known.
- `stat()` reports information about a file like size, MIME type, etc. Such information is packaged in a `KIO::UDSEntry`, which will be discussed below. Use `statEntry()` to send such an item to the application.
- `mimetype()` calls `mimeType()` with a string argument.
- `get()` and `copy()` may want to provide progress information. This is done with the methods `totalSize()`, `processedSize()`, `speed()`. The total size and processed size are reported as bytes, the speed as bytes per second.
- You can send arbitrary key/value pairs of meta data with `setMetaData()`.

4.3.10 Interacting with the user

Sometimes a slave has to interact with the user. Examples include informational messages, authentication dialogs and confirmation dialogs when a file is about to be overwritten.

- `infoMessage()` - This is for informational feedback, such as the message "Retrieving data from <host>" from the http slave, which is often displayed in the status bar of the program. On the application side, this method corresponds to the signal `KIO::Job::infoMessage()`.
- `warning()` - Displays a warning in a message box with `KMessageBox::information()`. If a message box is still open from a former call of `warning()` from the same slave process, nothing happens.
- `messageBox()` - This is richer than the previous method. It allows to open a message box with text and caption and some buttons. See the enum `SlaveBase::MessageBoxType` for reference.
- `openPassDlg()` - Opens a dialog for the input of user name and password.

Appendix A

Licensing

This documentation is licensed under the terms of the [GNU Free Documentation License](#).

This program is licensed under the terms of the [GNU General Public License](#).