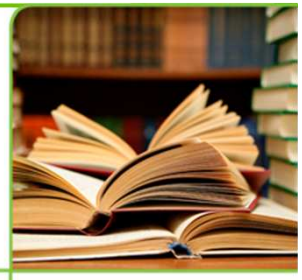
Qt in Education

Networking and Integrating the Web

digia

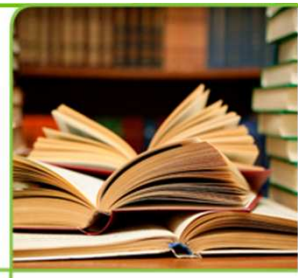
digia

Networking in Qt



- The `QtWebKit` module provides a full web renderer, JavaScript engine, and more
- `QNetworkAccessManager` provides an interface for sending requests and receiving replies over networks
- `QFtp` implements client side ftp
- `QTcpSocket` and `QTcpServer` provide TCP sockets
  - `QSslSocket` provides encrypted TCP sockets
- `QUdpSocket` provides access to UDP sockets

QtWebKit



- Based on the Open Source WebKit engine
- WebKit is basis for Apple's Safari browser, and numerous other browsers
- Apple originally based WebKit on KHTML and KJS from KDE

- KDE is built upon Qt technology

What is QtWebKit

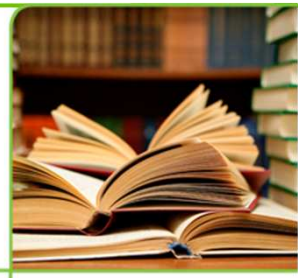
- Web rendering engine
- JavaScript engine

- Classes for integrating Qt and web contents to create hybrid applications

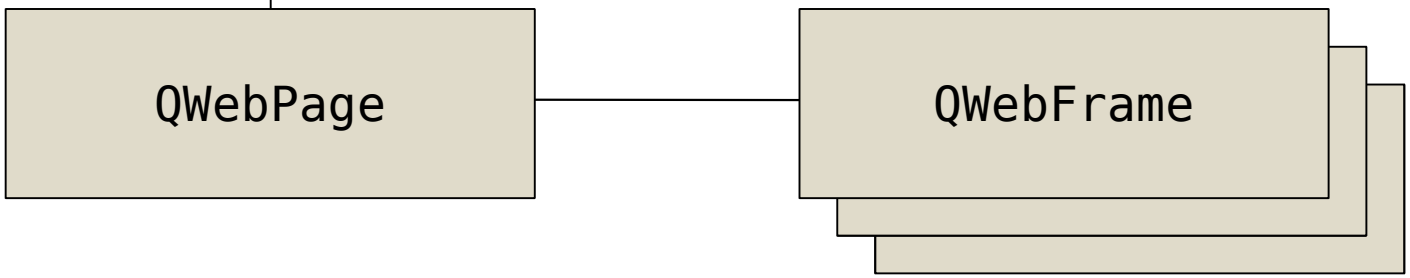
WebKit's Rendering Capabilities

- HTML 4.01, and parts of HTML 5
  - 2D canvas
  - Audio video playback
  - Off-line applications
  - Web workers, storage and SQL database
- CSS 1+2, and parts of CSS 3
  - Backgrounds and borders
  - Fonts
  - 2D and 3D transformations
  - Transitions and animations

Viewing a web page



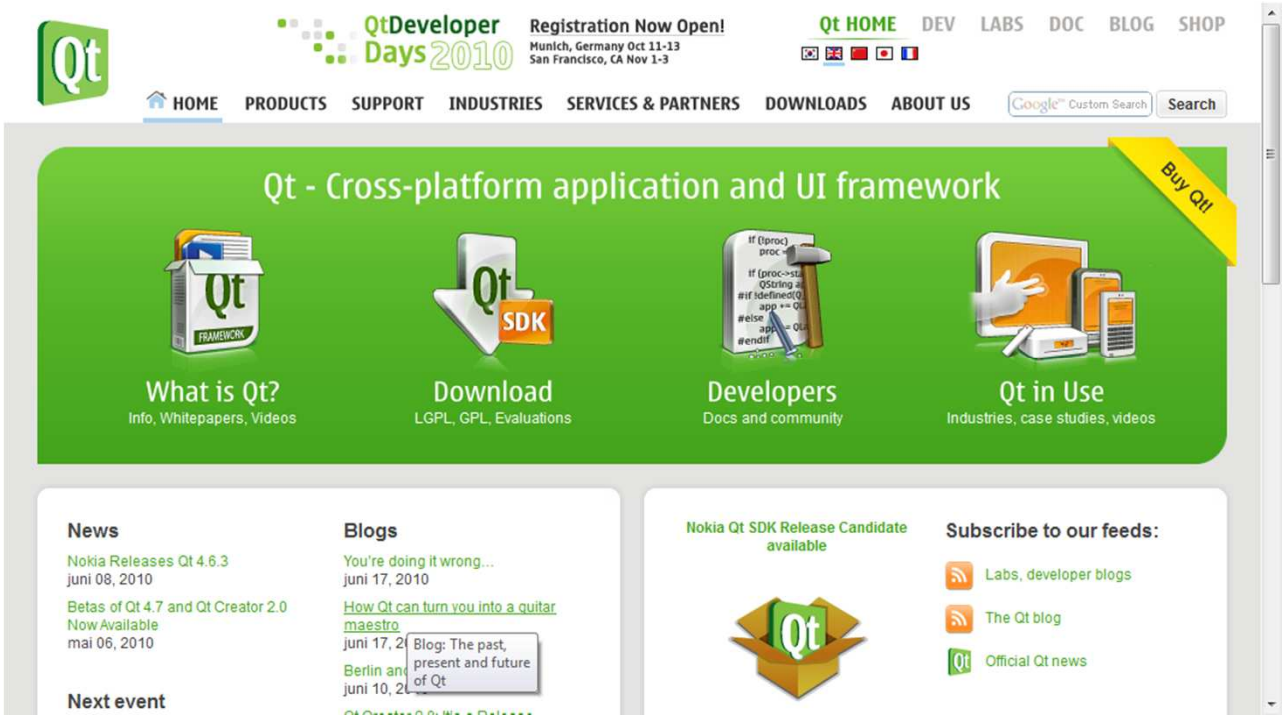
QWebView



digia

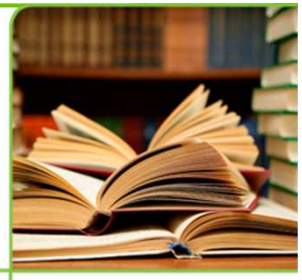
Viewing a web page

```
QWebView *view = new QWebView();
view->load(QUrl("http://qt.nokia.com"));
```



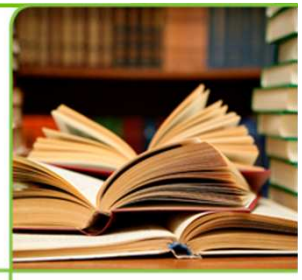
digia

Hybrid Applications



- By integrating your application with a web page, the user is given a familiar interface while you can focus on the functionality
- Integration means
  - Embedding Qt contents in web pages
  - Accessing web and JavaScript from Qt
  - Accessing Qt from JavaScript

Embedding Widgets



- It is possible to integrate QWidgets in HTML pages as plugins
- When a web page contains the `object` tag, Qt looks up the mime-type
- The mime-type is used to query the available QWebPluginFactory instances
- If the type matches, the plugin factory is requested to create a widget

Integrating a QWidget

- A detailed look at integrating the QCalendarWidget into a web page
  - The HTML code needed
  - The plugin factory
  - Handling properties
  - Enabling plugins in webkit

digia

Embedding Widgets

- Widgets are embedded through the object tag

```
<html>
  <body>
    <h1>Integrated Widget</h1>
    <p>
      <object type="application/x-qt-calendar">
        <param name="gridVisible" value="true" />
      </object>
       
      <object type="application/x-qt-calendar">
        <param name="gridVisible" value="false" />
      </object>
    </p>
</html>
```

Embedding Widgets

- A QWebPluginFactory object is used to create widgets from mime-types

```
#include <QWebPluginFactory>

class PluginFactory : public QWebPluginFactory
{
    Q_OBJECT
public:
    explicit PluginFactory(QObject *parent = 0);

    QObject *create(const QString &mimeType,
                    const QUrl &url,
                    const QStringList &argumentNames,
                    const QStringList &argumentValues) const;
    QList<Plugin> plugins() const;
};
```

Embedding Widgets

```
QObject *PluginFactory::create(const QString &mimeType,
    const QUrl &url, const QStringList &argumentNames,
    const QStringList &argumentValues) const
{
    QWidget *result = 0;

    if(mimeType == "application/x-qt-calendar")
    {
        result = new QCalendarWidget();
        for(int i=0; i<argumentNames.count(); ++i)
            result->setProperty(argumentNames[i].toLatin1().constData(),
                argumentValues[i]);
    }

    return result;
}
```

Embedding Widgets

- Before plugins can be loaded, they must be enabled through the QWebSettings object of the view

```
QWebView view;
view.settings()->setAttribute(
    QWebSettings::PluginsEnabled, true);
```

- The plugin factory must then be set in the page

```
view->page()->setPluginFactory(
    new PluginFactory(this));
```

Embedding Widgets

Integrated Widget

juni 2010

	sö	må	ti	on	to	fr	lö
22	30	31	1	2	3	4	5
23	6	7	8	9	10	11	12
24	13	14	15	16	17	18	19
25	20	21	22	23	24	25	26
26	27	28	29	30	1	2	3
27	4	5	6	7	8	9	10

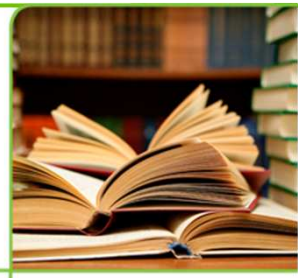
juni 2010

	sö	må	ti	on	to	fr	lö
22	30	31	1	2	3	4	5
23	6	7	8	9	10	11	12
24	13	14	15	16	17	18	19
25	20	21	22	23	24	25	26
26	27	28	29	30	1	2	3
27	4	5	6	7	8	9	10

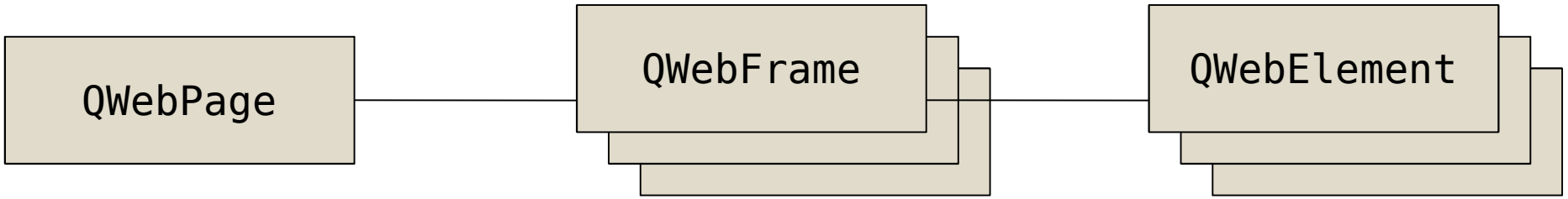
```
<html>
  <body>
    <h1>Integrated Widget</h1>
    <p>
      <object type="application/x-qt-calendar">
        <param name="gridVisible" value="true" />
      </object>
       
      <object type="application/x-qt-calendar">
        <param name="gridVisible" value="false" />
      </object>
    </p>
</html>
```

digia

Accessing DOM from Qt



- The Document Object Model is accessible through the QWebElement class



Navigating the DOM

- Each QWebFrame contains a documentElement.
- This is the root element of the frame

```
QWebView *view = ...;

QWebFrame *frame =
    view->page()->currentFrame();

QWebElement documentRoot =
    frame->documentElement();
```

Navigating the DOM

- From each QWebElement it is possible to traverse or search
  - traversing
    - firstChild – returns the first child element
    - nextSibling – returns the next sibling element
    - isNull – is true if there are no children / siblings
  - searching
    - findFirst and findAll – takes a CSS2 selector as argument, e.g. findAll(".*class tag*")

Inspecting the DOM

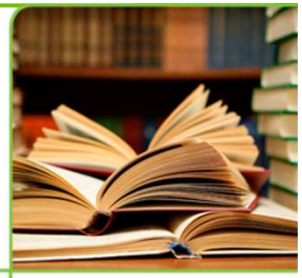
- Each QWebElement holds information about the current DOM element

```
<a href="http://qt.nokia.com/developer/qt-roadmap">Qt Road map</a>
```

- e.tagName = "A"
- e.toPlainText = "Qt Road map"
- e.classes = QStringList()
- e.attributeNames = QStringList("href")
- e.attribute("href") = "http://qt.nokia.com/developer/qt-roadmap"

Modifying the DOM

- It is possible to modify QWebElements in a multitude of ways
  - encloseWith – encloses the element in another element
  - setAttribute – sets an attribute
  - toggleClass – toggles a class
  - setPlainText / setInnerXml – replaces the contents of the element
  - setOuterXml – replaces the element and its contents



JavaScript Integration

- It is possible to integrate the Qt object model and JavaScript
  - Expose QObjects to JavaScript
  - Trigger JavaScript from Qt
- Great for mixing Qt and web contents

Exposing QObjects

- When exposing a QObject to JavaScript, properties and slots will be made available
- Call addToJavaScriptWindowObject on a QWebFrame to add an object to the frame

```
view->page()->currentFrame()->
    addToJavaScriptWindowObject("helloqt", javaScriptObject);
```

- When a new page is loaded, the object references will be cleared and the javaScriptWindowObjectCleared signal emitted
  - Add the objects from a slot connected to that signal

Accessing QObjects

```
class MyJavaScriptObject : public QObject
{
    Q_OBJECT
    Q_PROPERTY(QString text READ text WRITE setText)

public:
    explicit MyJavaScriptObject(QObject *parent = 0);

    const QString &text() const;

public slots:
    void setGreeting();
    void setText(const QString &text);
    ...
};
```

Accessing QObjects


```
<html>
  <body>
    <h1>JavaScript Integration</h1>
    <div><span id="greeting">Hello World!</span>
      <button type="button" onclick="helloqt.setGreeting();">
        Change Text!
      </button>
    </div>
    <div>
      <input type="text" id="textInput" />
      <button type="button" onclick="helloqt.setText(textInput.value);">
        Update!
      </button>
    </div>
  </body>
</html>
```

Triggering JavaScript

- JavaScript can be executed using the evaluateJavaScript method available from QWebFrame and QWebElement

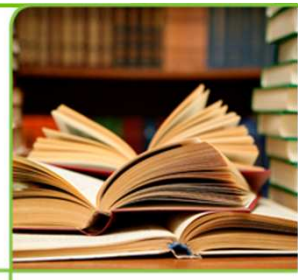
```
view->page()->currentFrame()->
    evaluateJavaScript(QString("textInput.value=\"%1\"").arg(text));
```

- The signal loadFinished is emitted from QWebPage when the page has been fully loaded. This is a good point to trigger JavaScript from

Integrating JavaScript

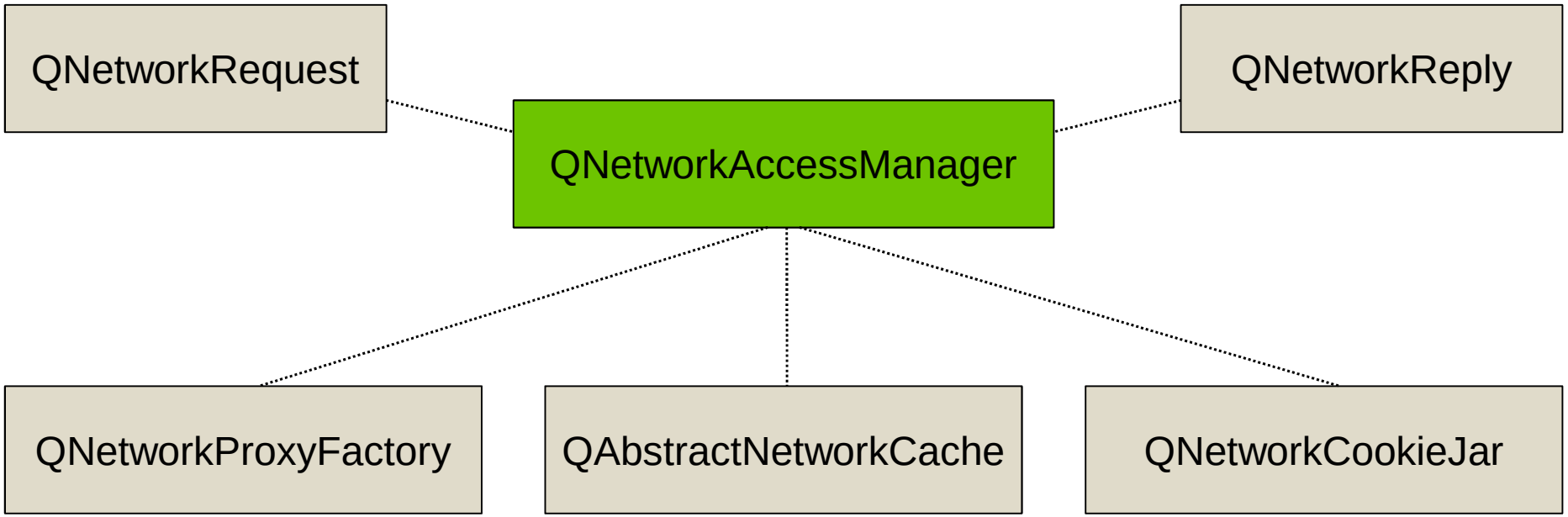
- The QWebPage class contains a number of protected methods that need to be considered when integrating JavaScript
  - javaScriptAlert
  - javaScriptConfirm
  - javaScriptConsoleMessage
  - javaScriptPrompt
  - createWindow
- Signals
  - windowCloseRequested
  - printRequested
- Slots
  - shouldInterruptJavaScript

Accessing the net



- The `QtWebKit` classes uses the `QNetworkAccessManager` to access the net
- The network access manager provides access to the web without any connections to a user interface. It can
  - handle requests and replies
  - cache web pages
  - keep track of cookies
  - use proxies
  - act as a protocol translator

Classes for Network Access



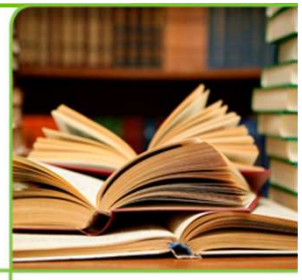
Accessing HTTP programatically

- Example – download a file via HTTP using QNetworkAccessManager

```
QNetworkAccessManager *manager = new QNetworkAccessManager(this);
connect(manager, SIGNAL(finished(QNetworkReply*)),
    this, SLOT(downloadDone(QNetworkReply*)));
manager->get(QNetworkRequest(QUrl("http://doc.qt.nokia.com/images/logo.png")));
```

```
MyClass::downloadDone(QNetworkReply *reply)
{
    QImageReader reader(reply, "png");
    mImage = reader.read());
    emit imageChanged(mImage);
    reply->deleteLater();
}
```

QNetworkAccessManager for interfacing other protocols

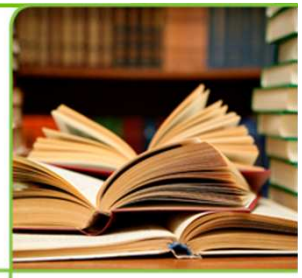


- By sub-classing the QNetworkAccessManager it is possible to provide a web-like interface to any data source
- As the QtWebKit classes use the network access manager for network access it is possible to use them with alternate data sources
- Qt Quarterly contains an example of implementing an FTP browser using this approach

http://doc.qt.nokia.com/qq/32/qq32-webkit-protocols.html

Break

digia



Protocols

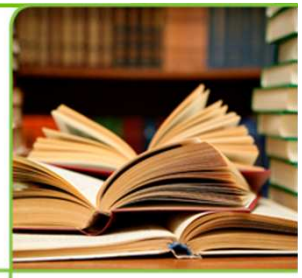
- Browsing the web uses the HTTP protocol
  - Hyper-Text Transfer Protocol
  - Sometimes encrypted as https

HTTP, FTP, SMTP, POP, IMAP, etc

TCP

IP
(IPv4 and IPv6)

- HTTP is built on top of TCP which is built on top of IP
- There are many other protocols built on top of TCP/IP, e.g. FTP, SMTP, POP, IMAP



FTP

- The QFtp class encapsulates the FTP protocol
  - File Transfer Protocol
- FTP lets you
  - log into servers
  - list files
  - move around the file system
  - upload and download files

Downloading a File

- To download the file ftp.qt.nokia.com/qt/source/README using a QFtp object, the following steps must be taken
  - connectToHost("ftp.qt.nokia.com")
  - login
  - get("/qt/source/README")
  - close

Splitting a URL

- The QUrl class can be used to split a URL into its parts


```
QUrl url("ftp://ftp.qt.nokia.com/qt/sources/README");
QString host = url.host();
QString path = url.path();
```

Downloading a File

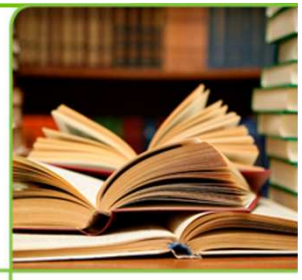
- Each command requested through QFtp is asyncronous
- When a command has finished, the commandFinished(int id, bool error) is emitted
  - id – an integer id for each command, returned when requesting the command, e.g. int QFtp::close()
  - error – is true if the command has resulted in an error

Download a File

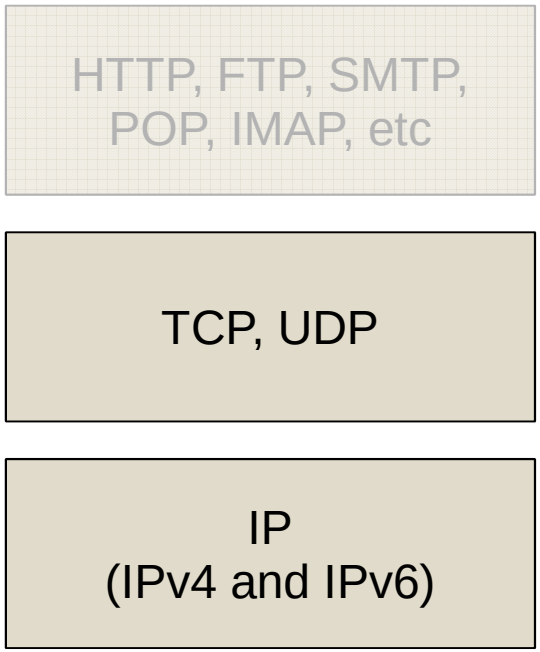
- All commands can be requested at once
- Or a simple state machine can be constructed

```
void Downloader::start()
{
    m_ftpState = Connecting;
    m_ftp->connectToHost(host);
}

void Downloader::ftpFinished(int, bool error)
{
    switch(m_ftpState)
    {
    case Connecting:
        m_ftpState = LoggingIn;
        m_ftp->login();
        break;
    case LoggingIn:
        m_ftpState = Downloading;
        m_ftp->get(file, 0, QFtp::Ascii);
        break;
    case Downloading:
        result = m_ftp->readAll();
        m_ftpState = Disconnecting;
        m_ftp->close();
        break;
    case Disconnecting:
        m_ftpState = Inactive;
        break;
    }
}
```

Accessing Socket



- HTTP, FTP, etc are all based on TCP and IP

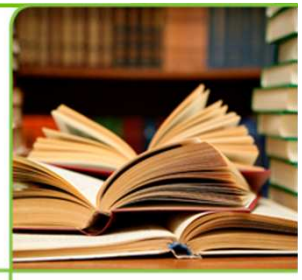


- Qt has support for accessing TCP and UDP directly at socket level

Accessing Sockets

TCP Sockets	UDP Sockets
• Guaranteed in-order delivery	Fire and forget
• Point-to-point only	Point-to-point or broadcasts
• Great when correct delivery is important	Great when time is more important than delivery

TCP Sockets



- There are two TCP classes in Qt
  - QTcpSocket – representing a socket
  - QTcpServer – representing a server, listening for incoming connections, generating QTcpSocket instances for each connection
- We will build a simple server greeting each connection with a text string

A TCP Server

- The TCP server consist of a QTcpServer that listens to port 55555
  - Generates a newConnection signal

```
Server::Server() : QObject(0)
{
    m_tcpServer = new QTcpServer(this);

    connect(m_tcpServer, SIGNAL(newConnection()),
            this, SLOT(serverConnected()));

    m_tcpServer->listen(QHostAddress::Any, 55555);
}
```

A TCP Server

- The next connection is retrieved using nextPendingConnection

```
void Server::serverConnected()
{
    QTcpSocket *connection = m_tcpServer->nextPendingConnection();
    connect(connection, SIGNAL(disconnected()),
            connection, SLOT(deleteLater()));

    QByteArray buffer;
    ...

    connection->write(buffer);
    connection->disconnectFromHost();
}
```

A TCP Server

- The reply is constructed in a buffer

```
void Server::serverConnected()
{
    ...

    QByteArray buffer;
    QDataStream out(&buffer, QIODevice::WriteOnly);
    out.setVersion(QDataStream::Qt_4_6);

    QString greeting = QString("Hello! The time is %1")
        .arg(QTime::currentTime().toString());
    out << (quint16)0;
    out << greeting;
    out.device()->seek(0);
    out << (quint16)(buffer.size() - sizeof(quint16));

    ...
}
```

When using a QDataStream with a socket it is important to handle the size of the data manually.

A TCP Client

- Use a QTcpSocket to connect to the host
  - readyRead is necessary, but there are more signals that are interesting, e.g. error

```
Client::Client() : QObject(0)
{
    m_tcpSocket = new QTcpSocket(this);

    connect(m_tcpSocket, SIGNAL(readyRead()),
            this, SLOT(readyToRead()));

    m_tcpSocket->connectToHost("localhost", 55555);
}
```

A TCP Client

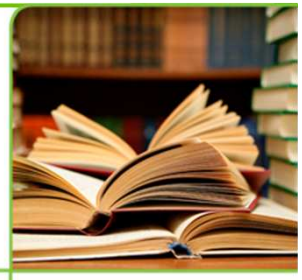
Using the QTcpSocket's buffer as our buffer

```
void Client::readyToRead()
{
    QDataStream in(m_tcpSocket);
    in.setVersion(QDataStream::Qt_4_6);

    if(m_tcpBlockSize == 0)
    {
        if(m_tcpSocket->bytesAvailable()<sizeof(quint16))
            return;

        in >> m_tcpBlockSize;
    }

    if(m_tcpSocket->bytesAvailable() < m_tcpBlockSize)
        return;

    QString greeting;
    in >> greeting;
    doSomething(greeting);
    m_tcpBlockSize = 0;
}
```

TCP Protocols

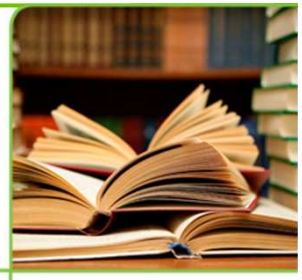
- The protocol demonstrated is very basic
  - Reply to all connections, then close
- A real world protocol would probably
  - Keep the connection open and use a set of commands for requesting and manipulating data
  - Carry some sort of versioning information
  - etc

Encrypted Sockets



- TCP/IP traffic is easy to overhear
- QSslSocket provides encrypted TCP sockets
  - Use connectToHostEncrypted
- SSL, Secure Sockets Layer, is a layer on top of TCP
  - Relies on CAs – Certificate Authorities

UDP Sockets



- The QUdpSocket provides a UDP socket
  - Usable for both clients and servers
  - User Datagram Protocol
- Datagrams are sent as one block
  - 512 bytes is ok, 8192 bytes usually work, larger might be possible
  - Can arrive or not
  - Can arrive out-of-order
  - Can arrive in duplicates