GENIVI Browser

Proof of Concept

Prep.	Marcel Schuette, Pelagicore	Revision	0.4	Date	March 12, 2014	No. of pages	14
Qual.	Marcel Schuette	Language	EN	Location	Munich, Germany		
File Name	browser			Status	Draft		

Contents

1 Introduction and project background

The GENIVI Networking Expert Group (NW-EG) would like to exercise a D-Bus-based software component to test the usability and quality of a web browser that uses the D-Bus interface. Therefore the expert group instructed a browser Proof-of-Concept (PoC) to be executed to evaluate existing APIs and concepts.

2 Scope of this document

This document should give an overview about the architecture of the GENIVI Browser PoC and details about the various components and their implementation. With that document, it should be possible to understand the source code more easily and get a better understanding of the used concept. Nevertheless the document doesn't replace reading the source code.

3 Requirements

The NW-EG defined a set of D-Bus APIs between the browser application and the HMI. These D-Bus APIs were provided as XML files. In addition, a header with definitions of specific types and structures were provided prior to project start.

The NW-EG defined the following requirements:

3.1 General Requirements

ID	Requirement	Description
SW-BRW-POC-001	Environment	The proof of concept must run under Ubuntu Linux 12.04 in a Virtual Box virtual machine installed on a desktop PC.
SW-BRW-POC-002	Environment	The implementation of the browser PoC will be done using the Qt 5 package which is under the LGPL v2.1 license.
SW-BRW-POC-003	Documentation	Documentation describing how to install the software and run a set of acceptance tests shall be provided

3.2 Architecture

ID	Requirement	Description
SW-BRW-POC-004	Separation of Browser HMI and Browser-Core	The Browser shall be separated in HMI and Browser Core. A provided Qt-based test-HMI will control the Browser Core.
SW-BRW-POC-005	Test-HMI as Separate Process	The provided test-HMI application will run as an independent process.
SW-BRW-POC-006	Qt Webkit 1 implementation	The Browser PoC application must use the Qt5 Webkit1 implementation.
SW-BRW-POC-007	QtWebview	The Browser POC must use QGraphicsWebView instead of QML webview.

3.3 Browser Interfaces

ID	Requirement	Description
SW-BRW-POC-008	D-Bus API	The Browser PoC shall implement the specified APIs as handed over in xml-form (see interfaces section)
SW-BRW-POC-009	Qt Webkit Bridge	The Qt Webkit bridge should be useable in the Browser POC
SW-BRW-POC-010	Netscape Plugin API (NPAPI)	The Browser POC should support the Netscape Plugin API (NPAPI)
SW-BRW-POC-011	Configuration	The browser POC should implement a basic configuration mechanism for the browser

3.4 Interfaces

The following D-Bus interfaces files were provided as input to the project:

1) IBrowser.xml
2) IWebPageWindow.xml
3) IUserInput.xml
4) IBookmarkManager.xml
5) IUserInput.xml
6) INetworkManager.xml
7) IErrorLogger.xml
8) IWebPageWindow.xml
9) ICacheManager.xml

These interfaces represent a subset of the GENIVI APIs defined by NW-EG and contains only methods and signals, which has to be supported by the PoC. The XML files can be found in the repository (`http://git.projects.genivi.org/browser-poc.git/`) in the folder `/common`. For detailed information about the methods and signals for D-Bus interfaces were defined, you can have a look at the files in the repository.

4 Overview

Following the requirements, the GENIVI Browser PoC is implemented with Qt 5 and Qt Webkit 1. Although existing projects like the snowshoe browser were taken into account, a 'from scratch' approach was chosen. The specific requirements of the project made a re-use of an existing project not appropriate, as a reduction of efforts was not expected.

The GENIVI Browser PoC consists of following components, which are also represented in separate folders in the repository:

- A browser application
- A test user-interface application (testUI)
- A demo user-interface application (demoUI)
- A set of automated tests (`browser/unit-tests/*`)
- A folder with common components

According to the defined architecture the browser application, which is responsible for web page rendering and bookmark management, is separated from the HMI (represented by the demoUI application and the testUI application). For more detailed information about the architecture defined by NW-EG, you can have a look at the group's wiki page (`https://collab.genivi.org/wiki/display/genivi/Networking+Expert+Group`).

For instructions, how to build all applications or only a single application, refer to the file `BUILDINSTRUCTIONS`, or `Build.instructions.Ubuntu.txt` in the repository.

5 Common part

here is a common part in the project, which contains files, which are shared by all applications. This common part is represented in the repository in the folder `/common`:

- The XML files describing the interfaces
- A class defining a bookmark object (bookmark.h|cpp)
- A class defining the D-Bus interfaces on client side (demoUI and test application) (browserd-bus.h|cpp)
- A header file defining common types and structures (browserdefs.h)

The XML files are used to automatically generate the interface classes for client and server. Via the generated interfaces of these classes, the clients can call remote objects in the server via D-Bus.

6 Browser Application

6.1 Introduction

The Browser application is the core part of the PoC. It is responsible for rendering and displaying a webpage with the QML webview element. It also implements the defined server side interfaces for `IBrowser`, `IUserInput`, `IWebPageWindow`, `ICacheManager`, `IErrorLogger`, `INetworkManager` and `IBookmarkManager`. The bookmark manager includes also logic for persistent bookmark storage in the system.

6.2 Usage

The browser application always needs to be started before the user-interface application (demoUI or testUI) is started, because it creates the D-Bus connection. As the window will be created by the user-interface application, no window is shown at application start. You can add an instance id as parameter to the application start, e.g. `./browser 23`. The instance id will be added to D-Bus service name, e.g. `genivi.poc.browser23`. If no parameter is given, a default instance id 1 is used.

6.3 Structure

The source code of the browser application can to be found in the `/browser` folder in the repository:

*.cpp, *.h	source and header files
browser.pro	Qt project file

6.4 Implementation

The browser application is implemented using Qt Quick 1 and Qt Webkit 1. Qt Quick 1 needed to be used, because of the requirement to use Qt Webkit 1.

There is one class available for each defined interface group (XML file) implementing the functions for the defined interfaces on server side. These functions interact with the interfaces provided by the QML file.

As a central class the browserhelper class creates the connection to D-Bus on the session bus and registers a service name on the D-Bus server. The default service name for the GENIVI browser PoC is `genivi.poc.browser + instance id`, e.g. `genivi.poc.browser23`. The default instance id is 1, if no parameter is given, or the parameter given at application start.

The class also creates all interface objects and D-Bus interface adaptors, registers needed types with the D-Bus system and registers the browser and bookmark interface class with the D-Bus connection under an object path (/Browser/IBrowser and /Browser/IBookmarkManager). The interface for webpagewindow and userinput will be registered, when a new page is actually created.

bookmarkmanager.h|cpp implements the IBookmarkManager interfaces and manages persistent storage of bookmarks.

userinput.h|cpp implements the IUserInput interfaces.

browser.h|cpp implements the IBrowser interfaces, creates and set up a declarative view with the main QML file (the webview) and registers webpagewindow and userinput objects under a unique object path (/Browser/IWebPageWindow + window handle resp. /Browser/IWebPageWindow + window handle/IUserInput). This is needed to control different webpages or tabs (created by the createPageWindow interface) with the testUI application. That means e.g. routing a reload command to the right webpage window.

browserconfig.h|cpp handles persistant storage of configuration values and provides a singleton for any part of the browser to access and set these values.

browserpage.h|cpp is a subclass of QWebPage allowing BrowserView to intercept dialog boxes and prompts.

cachemanager.h|cpp handles caching policies using `QNetworkConnectionManager`.

errorlogger.h|cpp handles error logging to an in-memory storage, and allows D-Bus clients to query the error log.

7 Test Application

7.1 Introduction

With the test user-interface application, you are able to test all implemented interfaces with all defined parameters. For this purpose, not the UI design of the application was the key requirement, but the possibility to test the functionality. This test application was also used to do manual tests, which resulted in a test report (`https://collab.genivi.org/wiki/display/genivi/Browser+Proof-Of-Concept+-+Web+content#BrowserProof-Of-Concept-Webcontent-Testreport`).

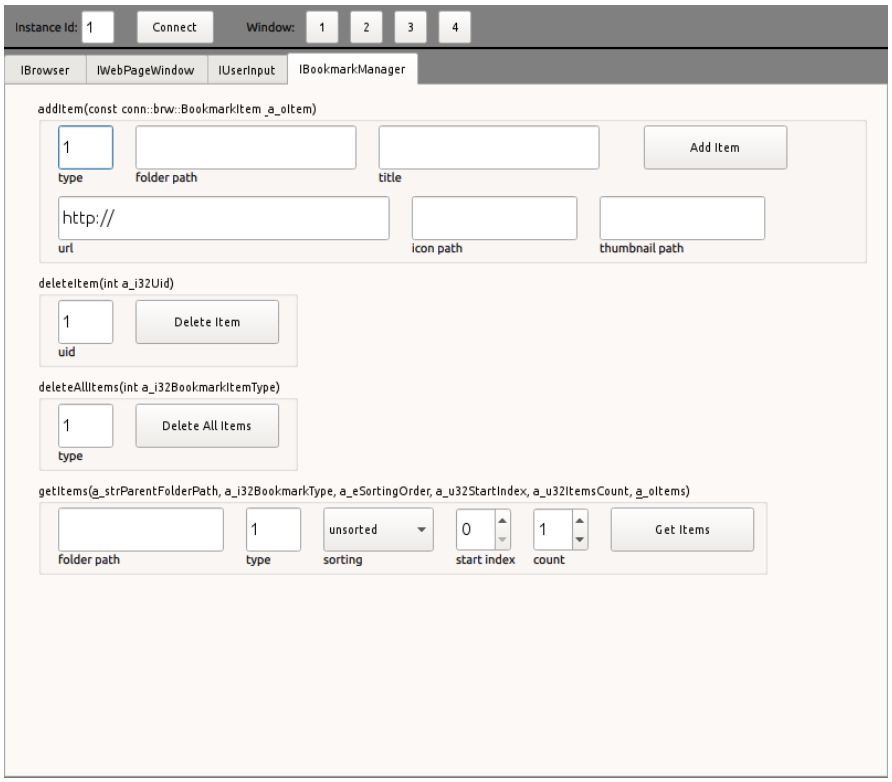

Figure 1: The picture above shows the testUI application with the bookmark manager tab visible.

7.2 Usage

Before the testUI application is started, the browser application must be started. At startup only the bar on the top is visible, not the tabs or buttons. First the testUI must connect to the D-Bus service. Therefore add the correct instance id and press the connect button. The instance id is defined with the browser application start, either 1 as default or the number given as parameter at browser start.

The testUI application shows tabs representing the four interface groups (`IBookmarkManager`, `IBrowser`, `IUserInput`, `IWebPageWindow`, `ICacheManager` and `IErrorLogger`).

To navigate between the groups, just press on the tab header.

On each page (or tab) each defined interface with its parameter is visually grouped within a small frame. Each group consists of a button and optional input fields, spin boxes or combo boxes, depending on the definition of the parameters. By pressing the button of a group, the client calls a D-Bus interface with given parameters, which calls a remote object on the server side.

To be able to handle more than one opened window (e.g. redirect a press on the reload button to the right window) with the testUI application, four buttons (numbered from 1 to 4) were added to the bar on the top. For simplicity reasons the buttons are limited to four, means only four windows can be managed with the testUI application. The numbers on the buttons refer to the sequence the windows were opened. E.g., if you want to reload the third opened window, first press button 3 and then the reload button on the IWebPageWindow tab.

7.3 Structure

The source code of the test user-interface application can to be found in the `/testapp` folder in the repository:

main.cpp	mainsource file
testapp.pro	Qt project file
`QML/testapp/*`	QML files for user interface
`images/*`	Images used in implementation (icons taken from KDE oxygen theme 4.10.3)

7.4 Implementation

The testUI application supports all interfaces described in the XML files.

The testUI application is implemented as a Qt Quick 2 application. The user-interface is described with QML using Qt Quick Controls (reusable UI controls provided by Qt 5.1). Besides the main.QML file, which describes the main view with all tabs, each interface group is described in a separate QML file. The backend logic of the testUI application is written in C++. The main task of the C++ part is to set up the view for QML, load the QML file and register custom C++ type in the QML system (bookmark and browserdbus classes). The browserdbus class represents the D-Bus interface for the client. It creates the D-Bus channel connections, registers custom types with the QtDBustype system and defines and calls all D-Bus interfaces and handles the return values. The output of all return values, resulting from D-Bus remote object calls, is logged onto the console.

8 Demo User-Interface

8.1 Introduction

The idea of the demo user-interface application is to have a user-interface, which looks more like a possible real browser user-interface (other than the testUI application) and can be used to demonstrate the GENIVI Browser PoC at shows and events. The demoUI application doesn't support the full set of defined interfaces, but only a subset needed to demonstrate the used features. The chosen features were selected to be the main features known from common web browsers.

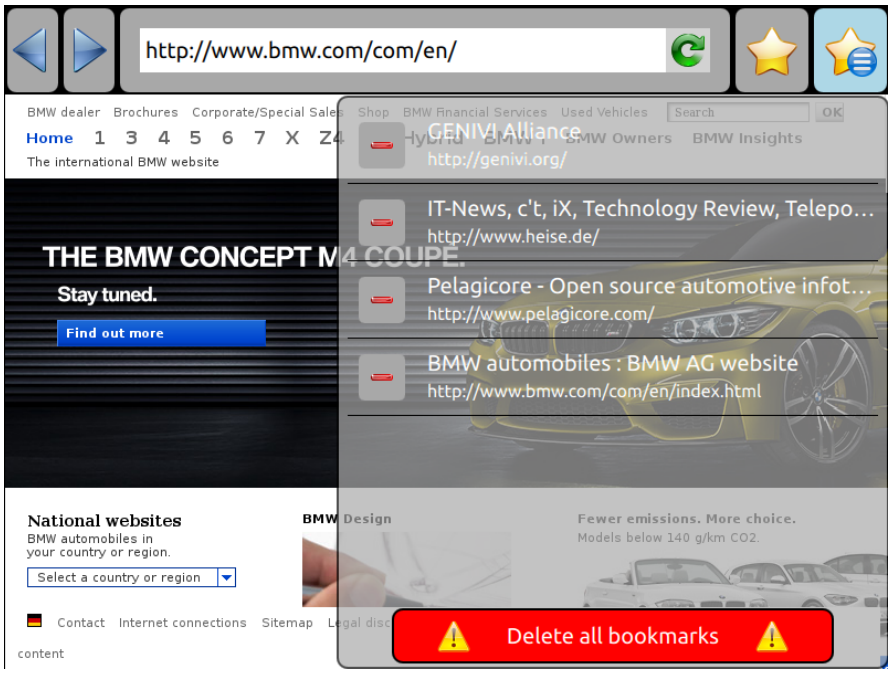

Figure 2: The picture above shows the demo user-interface application with open bookmarks pane. The website shown in the background is part of the browser application.

The demoUI supports the following features:

- loading a web page defined by a URL
- bookmark management (add, select, delete and delete all bookmarks)
- navigation back and forward in the browser history
- reload and stop loading a web page
- display a progress bar representing the loading progress of a web page (displayed below the URL in the input field)
- keypad navigation on web page with keyboard navigation keys

8.2 Usage

Before the demoUI application is started, the browser application must be started. With the start of the demoUI application, also the browser application becomes visible. The two applications (browser and demoUI) are arranged in a seamless way, to represent the look of only one application.

The icons in the demoUI are hopefully self-explaining and known from other web browser user-interfaces. To load an URL type the URL in the input field on the top and press the enter key on an attached keyboard. The two top-right icons represent 'add bookmark to bookmark list' and 'show bookmark list' functionality. If the bookmark pane is open, you can select bookmarks to be loaded in the webview, delete individual bookmarks by pressing the red 'minus' icon at the front of a bookmark or delete all bookmarks with the red button on the bottom of the pane. For keypad navigation on webpages, the up-, down-, right- or left-key (line mode) or the space bar (page down) and tab key (page up) (page mode) can be pressed. It's important that the demoUI application has the focus to receive the key presses. To activate a link on a web page or move the web page, the browser application window can be directly accessed. These actions are not handled over a D-Bus interface.

8.3 Structure

The source code of the demo user-interface application can to be found in the `/demoui` folder in the repository:

main.cpp	mainsource file
demoui.pro	Qt project file
`QML/demoui/*`	QML files for user interface
`qtquick2application/*`	Classes for displaying a QtQuick UI
`images/*`	Images used in implementation (icons taken from KDE oxygen theme 4.10.3)

8.4 Implementation

The demoUI supports the following D-Bus interfaces:

- IBrowser
  - createPageWindow (with fixed parameters)
- IWebPageWindow
  - load
  - reload
  - stop
  - back
  - forward
  - scroll (line or page scrolling)

  - onLoadStarted
  - onLoadProgress
  - onLoadFinished

- IBookmarkManager
  - addItem (with url and title, all other parameter fix)
  - getItems (with fixed parameters)
  - deleteItem
  - deleteAllItems (with fixed parameters)

The demoUI is implemented as Qt Quick 2 application. The user-interface is described with QML. The backend logic of the demoUI is done in C++. Main task of the C++ part is to set up the view for QML, load the QML file, register custom C++ type in the QML system (bookmark and browserdbus classes). The browserdbus class represents the D-Bus interface for the client. It creates the D-Bus channel connections, registers custom types with the QtDBustype system and defines and calls all D-Bus interfaces and handles the return values. The output of all return values, resulting from D-Bus remote object calls, is logged onto the console.

If the bookmark pane is open, the demoUI application (geometry) is resized. The height of the application is changed. This resize is needed, to show the pane, which has the height of the complete browser, and to allow (if closed) getting access to the browser window (e.g. pressing on links on the browser window). Otherwise the above lying application with the focus (even if parts of the application are transparent) would get all mouse and key events and it wouldn't be possible to click on links on a web page.