Portierung von Qt auf DROPS

Großer Beleg

Carsten Weinhold

Bearbeiter:	Carsten Weinhold cw183155@inf.tu-dresden.de
Matr.-Nr.:	2766369
in Zusammenarbeit mit:	Josef Spillner js177634@inf.tu-dresden.de
Betreuer:	Jork Löser jork@os.inf.tu-dresden.de

Inhaltsverzeichnis

1 Motivation - Warum Qt?

Qt [4] ist ein in C++ geschriebenes Toolkit, welches von der norwegischen Firma Trolltech [3] entwickelt und vertrieben wird. Auf den ersten Blick scheint es sich dabei um ein Produkt unter vielen zu handeln, mit dem Softwareentwickler ihre Anwendungen mit einer grafischen Benutzeroberfläche versehen können. Im Bereich der freien Software gibt es viele Frameworks und Klassenbibliotheken, die ähnliches ermöglichen, beispielsweise *GTK+* [7], auf dem der Unix-Desktop GNOME [8] basiert. Kommerzielle Produkte dieser Kategorie existieren natürlich ebenfalls, oft sind sie Teil der Entwicklungsumgebungen der Betriebssysteme und werden vom jeweiligen Hersteller zur Verfügung gestellt. Das *Cocoa*-Framework für die Benutzeroberfläche von Mac OS X sei hier als Beispiel angeführt.

Die Besonderheit von Qt ist aber seine Plattformunabhängigkeit. Das Toolkit selbst ist für alle wichtigen Betriebssysteme verfügbar und mit Qt/Embedded kommen auch PDAs oder Mobiltelefone als Zielplattform in Frage. Hervorzuheben ist dabei, dass Qt nicht nur eine grafische Benutzerschnittstelle zur Verfügung stellt, sondern auch Abstraktionen für einen Großteil der vom jeweiligen Betriebssystem bereitgestellten Dienste anbietet. Dadurch wird es möglich, mit ein und demselben Quellcode Anwendungen für jedes System zu entwickeln, auf dem Qt verfügbar ist.

Die Entwicklung von Software für eine Vielzahl von Plattformen geht oft mit einer Steigerung der Codekomplexität und des für die Entwicklung notwendigen Aufwands einher. Dies ist auch ein häufig vorgebrachtes Argument, wenn es darum geht zu rechtfertigen, warum ein vom Nutzer gewünschtes Programm nicht für die bevorzugte Rechnerumgebung erhältlich ist. Trolltech hat an dieser Stelle das Kunststück vollbracht, den Entwicklungsaufwand im Vergleich zu einigen nativen Toolkits sogar zu reduzieren, indem eine breite Palette von mächtigen und dennoch einfach zu nutzenden Programmierschnittstellen angeboten wird. Neben der grafischen Benutzeroberfläche, dem wichtigsten Teil von Qt, gibt es auch plattformübergreifend einheitliche APIs für den Zugriff auf Dateien und Netzwerk. Multithreading ist ebenfalls kein Problem, auch hierfür gibt es Klassen.

Qt erleichtert es aber auch, komplexere Funktionen zu verwenden. Angefangen bei der Fähigkeit, alle von Qt unterstützten Grafikformate in eigenen Anwendungen unkompliziert nutzen zu können bis hin zur Einbindung von verschiedensten SQL-Datenbanksystemen.

Durch den großen Funktionsumfang bei zugleich einfacher Anwendbarkeit ist es mit verhältnismäßig wenig Aufwand möglich, auch komplexere Anwendungen zu programmieren. Dabei kann sich der Softwareentwickler einiger leistungsstarker Werkzeuge bedienen. So ist mit dem *Qt Designer* (siehe auch Abbildung 1 auf Seite 5) ein grafischer Editor zum Erstellen von Dialogen und anderen Elementen der grafischen Benutzeroberfläche vorhanden. Er leistet auch Unterstützung bei der Einbindung von Datenbanken. Ein anderes Werkzeug, der *Qt Linguist*, erleichtert die Lokalisierung von Anwendungen.

Dies alles macht Qt auch für das an der Technischen Universität Dresden [1] entwickelte Echtzeitbetriebssystem DROPS [12] interessant. Mit einer Verfügbarkeit von Qt für DROPS ergäbe sich die interessante Möglichkeit, bereits existierende Qt-Anwendungen auch unter DROPS einsetzen zu können. Umgekehrt wäre es möglich, ein Programm für DROPS zu entwickeln und es dann beispielsweise auch unter Linux ausführen zu lassen. Sei es, weil das Programm für beide Systeme interessant ist, oder weil das Testen unter Linux einfacher und schneller möglich

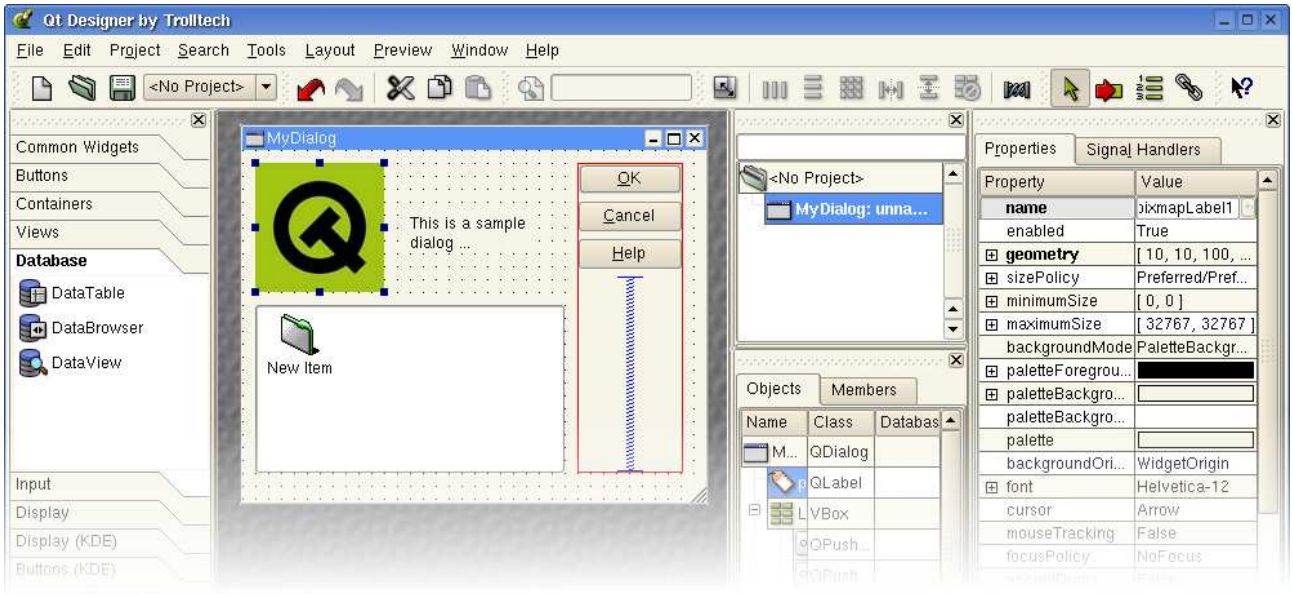

Abbildung 1: Mit dem *Qt Designer* können auf sehr einfache Art und Weise grafische Benutzeroberflächen gestaltet werden.

ist (immerhin entfällt dort die Notwendigkeit, den Rechner neu zu starten). Denkbar wäre zum Beispiel, dass man die grafische Benutzeroberfläche eines Programms ganz bequem im *Qt Designer* unter Linux gestaltet und testet. Der von den Werkzeugen automatisch generierte Quellcode muss dann nur noch in die DROPS-Anwendung eingebunden werden. Dabei ist es natürlich auch möglich, nur Teile der von Qt gebotenen Funktionalität zu verwenden, etwa die grafische Benutzeroberfläche. Der restliche Teil des Programms macht Gebrauch von unter DROPS zur Verfügung stehenden Diensten. Prinzipiell können damit auch echtzeitkritische Aufgaben bearbeitet werden. Die Konfiguration einer Anwendung würde dann beispielsweise über die Qt-basierte Benutzeroberfläche erfolgen, während in einem separaten Fenster ein Video im Echtzeitmodus abgespielt wird.

Mit seinem großen Funktionsumfang und den zur Verfügung stehenden Werkzeugen eröffnet ein 'Qt für DROPS' also einige sehr interessante Möglichkeiten zur effizienten Softwareentwicklung. Darin eingeschlossen ist die Möglichkeit zur Wiederverwendung von bereits bestehendem Code und gegebenenfalls auch von kompletten Anwendungen.

Die hier aufgeführten Argumente, insbesondere zur Plattformunabhängigkeit, können auch den Einsatz von Java anregen. Und durch die Verfügbarkeit der freien Java-Implementierung *Kaffe* [19, 20] unter DROPS ist diese Möglichkeit ja auch gegeben. Beim Vorstellen verwandter Arbeiten in Kapitel 2 wird daher – ergänzend zur Motivation in diesem ersten Teil des Belegs – unter anderem diskutiert, welchen Gewinn die Verwendung von Qt gegenüber Java ermöglicht.

Kapitel 3 beleuchtet dann einige größtenteils technische Aspekte von Qt und betrachtet Grundlagen, die im darauf folgenden Kapitel relevant sind. Dort wird die eigentliche Portierung beschrieben und es wird erklärt, warum sich Kapitel 5 mit Entwurf und Implementierung eines

Servers für Unix Domain Sockets befasst, was ja zunächst einmal nicht mit Qt in Verbindung zu stehen scheint.

Kapitel 6 verdeutlicht an einem Beispiel die doch recht komplizierten inneren Abläufe beim Betrieb von Qt-Anwendungen. Abschließend findet in den beiden letzten Kapiteln eine Leistungsbewertung statt und die erreichten Ergebnisse werden zusammengefasst.

2 Einführung

2.1 Abgrenzung der Belegarbeit

Qt ist mit ca. 380.000 Zeilen Quellcode sehr umfangreich, wenngleich der Teil, welcher bei einer Portierung angepasst oder neu implementiert werden muss, natürlich eine deutlich geringere Größe hat. Jedoch ist dieser Teil noch immer hinreichend groß, deshalb wird die Portierung von zwei Studenten vorangetrieben. Josef Spillner führt einen Teil der Portierungsarbeiten im Rahmen eines Komplexpraktikums durch. Ich bearbeite diese Aufgabe als Großen Beleg.

Nach gemeinsamen Anstrengungen die Bibliothek unter DROPS zu übersetzen, wurden die einzelnen Aufgaben verteilt. Die Ein- und Ausgabe fällt in Josef Spillners Bereich und umfasst die Implementierung von Treibern für die Eingabe via Maus und Tastatur sowie für die Bildschirmdarstellung. Außerdem untersucht er die Möglichkeiten zur einfachen Audioausgabe.

Meine Aufgabe ist es, die Betriebssystemabstraktionen zur Nutzung von Dateien und Netzwerk zu portieren. Die Klassen für Multithreading und die dafür nötigen Synchronisationsmechanismen zählen ebenfalls zu meinem Aufgabenbereich. Außerdem obliegt mir die Portierung der Kernkomponenten des *Qt Window System*, welches ein wesentlicher Bestandteil des hier als Grundlage verwendeten Qt/Embedded ist.

Dieser Beleg befasst sich nur mit den von mir durchgeführten Arbeiten an der Portierung von Qt auf DROPS. Im Rahmen von Erläuterungen und Argumentationen werde ich aber immer wieder auch auf die Ein- und Ausgabe, insbesondere die Grafikdarstellung und die zugrundeliegende Problematik des Framebuffers, zurückkommen. Diese Bereiche sind zwar nicht weiter Gegenstand dieser Arbeit, jedoch für einige Betrachtungen relevant.

2.2 Verwandte Arbeiten und Stellung von Qt in DROPS

Wie bereits im Kapitel 1 angedeutet, findet Echtzeitfähigkeit im Rahmen dieser Belegarbeit keine Berücksichtigung. Für diesen Zweck existiert mit *DOpE* [10] bereits eine andere grafische Benutzeroberfläche für DROPS. Vielmehr soll Qt eine Ergänzung zu *DOpE* darstellen und als Basis für die nicht zeitkritische Teile von Anwendungen unter DROPS dienen.

Neben der reinen Bereitstellung einer modernen Benutzeroberfläche ist es auch ein wichtiges Ziel der Qt-Portierung, Kompatibilität zu anderen Betriebssystemen herzustellen. Es soll möglich sein, für andere Systeme geschriebene Anwendungen mit minimalem Aufwand auch unter DROPS einzusetzen. Mit dieser Zielstellung befassen sich noch drei weitere Projekte innerhalb der Betriebssystem-Gruppe der Technischen Universität Dresden. Sie sollen hier kurz beleuchtet werden:

L4VFS* und *dietlibc Das erste Projekt wird maßgeblich von Martin Pohlack vorangetrieben und strebt die Nachbildung einer POSIX-Umgebung unter DROPS mit Hilfe der *dietlibc* und *L4VFS* [12] an. Es stellt eine wichtige Grundlage bei der Portierung von Qt auf DROPS dar, denn hier wird ein Teil der von Qt benötigten Betriebssystemschnittstellen unter DROPS bereitgestellt.

SDL – Simple DirectMedia Layer Bei *SDL* [14] handelt es sich ebenfalls um ein Toolkit, das plattformunabhängige Softwareentwicklung ermöglicht. Es offeriert Abstraktionen für die Grafik- und Audiofunktionalität der jeweiligen Betriebssysteme mit einer einheitlichen Schnittstelle auf allen Plattformen. In der DROPS-Portierung werden auch Multithreading und Dateizugriff unterstützt.

SDL ist hauptsächlich für die Entwicklung von Multimedia-Anwendungen gedacht, mit besonderen Augenmerk auf Spiele. Die angebotenen APIs haben daher auch ein recht niedriges Abstraktionsniveau und orientieren sich nahe an der Hardware. *SDL* bietet im Gegensatz zu Qt kein eigenes Framework für grafische Benutzeroberflächen an. Folglich besteht auch keine echte Konkurrenz zwischen beiden Lösungen, sondern vielmehr ergänzen sie sich.

***Kaffe* – Java für DROPS** Das dritte Projekt befasst sich damit, Java unter DROPS verfügbar zu machen. Dazu hat Alexander Böttcher im Rahmen seines Großen Belegs die freie Java-Implementierung *Kaffe* [19, 20] auf DROPS portiert. Java und die dafür verfügbaren Klassenbibliotheken und Frameworks wurden bekanntlich explizit für die plattformunabhängige Softwareentwicklung entworfen, welche ja auch einen der wesentlichen Einsatzbereiche von Qt darstellt. In so mancher Hinsicht verfolgen Java und Qt also die gleichen Ziele, dennoch ist das eine dem anderen gegenüber natürlich nicht als redundant anzusehen. Tatsächlich werden von Alexander Böttcher sogar Anstrengungen unternommen, die grafische Benutzeroberfläche für Java-Programme unter DROPS funktionsfähig zu machen. Die Basis dafür ist das hier diskutierte Qt für DROPS.

Dadurch, dass Qt auf der Programmiersprache C++ basiert, ergeben sich allerdings einige Vorteile gegenüber dem Einsatz von Java. Zum einen haben Programme, die in C oder C++ geschrieben wurden, in vielen Fällen deutliche Performancevorteile. Der für Java-Anwendungen erzeugte Bytecode wird heute zwar meist nicht mehr zur Laufzeit interpretiert, sondern es kommen optimierende Just-in-Time-Compiler zum Einsatz, welche recht effizienten Maschinencode erzeugen können. *Kaffe* für DROPS unterstützt im Moment aber nur den Interpretermodus. Somit sind Java-Programme unter DROPS bei der Ausführungsgeschwindigkeit den in Maschinencode übersetzten Qt-Anwendungen unterlegen.

Weiterhin ist die Nutzung der unter DROPS nativ zur Verfügung stehenden Dienste durch C- oder C++-Programme – also auch solche, die Qt nutzen – einfacher und direkter möglich. Bei Java-Anwendungen muss dafür auf das *Java Native Interface* zurückgegriffen werden. Gemäß [20] gibt es dabei aber noch einige Schwierigkeiten.

2.3 Systemvoraussetzungen für Qt unter DROPS

Die wichtigste Komponente von Qt ist die grafische Benutzeroberfläche, die von der Hardware und dem Betriebssystem natürlich die Fähigkeit zur Grafikdarstellung erfordert. Es sei vorweggenommen, dass diese Voraussetzung unter DROPS auf verschiedene Art und Weise erfüllt werden kann. Die Arbeitsfläche mit den Qt-Anwendungen kann sowohl durch direkte Verwendung des Hardware-Framebuffers dargestellt werden, als auch über die Framebuffer-Abstraktionen von *con* oder *DOpE*. Momentan wird der Betrieb mit *con* von Qt für DROPS am besten unterstützt, was aber nur temporär der Fall sein sollte.

Für die Übersetzung der meisten Qt-Programme werden außerdem zusätzliche Werkzeuge aus der Qt-Distribution benötigt. Diese Werkzeuge können nicht ohne weiteres aus dem DROPS-Quellcodebaum heraus gebaut werden, weil sie ihrerseits die Linux-Version der Qt-

Bibliothek erfordern. Die für DROPS kompilierte Bibliothek kann zu diesem Zweck leider nicht verwendet werden. Die benötigten Werkzeuge sind aber in praktisch allen Linux-Distributionen bereits vorhanden. In Anhang B findet sich eine Liste der zusätzlich benötigten Software und deren Bezugsquellen. Eine Auflistung der weiteren Systemvoraussetzungen ist dort ebenfalls zu finden.

3 Grundlagen

3.1 Qt im Überblick

Das in Norwegen ansässige Unternehmen Trolltech [3] hat mit Qt ein Toolkit geschaffen, welches die plattformunabhängige Softwareentwicklung in einfacher Weise ermöglichen soll. Es existieren Portierungen für Microsoft Windows, Mac OS X und diverse Unix-Varianten. Außerdem gibt es mit Qt/Embedded eine spezielle Variante für Linux, die an Geräte mit eingeschränkten Ressourcen, wie zum Beispiel PDAs, angepasst ist.

Qt ist ein kommerzielles Produkt, steht aber für Unix/X11, Mac OS X und in der Variante Qt/Embedded auch unter den Bedingungen der GNU General Public License [5] zur Verfügung. Seit einiger Zeit ist Qt auch für Microsoft Windows in einer kostenlosen Version für den nichtkommerziellen Einsatz erhältlich.

Qt stellt eine moderne grafische Benutzeroberfläche zur Verfügung und ermöglicht damit das Erstellen von grafischen Anwendungen für alle durch Qt unterstützten Plattformen. Dabei ist im Idealfall lediglich eine Neukompilierung notwendig, um eine Anwendung unter einem von Qt unterstützten Betriebssystem wie Windows oder Unix ablaufen zu lassen. Voraussetzung dafür ist, dass ausschließlich Programmierschnittstellen verwendet werden, die auf allen gewünschten Zielplattformen zur Verfügung stehen. Das sind einerseits alle von Qt selbst angebotenen Schnittstellen, sowie auch solche, die durch andere Standards vorgegeben werden. Die POSIX-Spezifikation ist ein Beispiel dafür, die entsprechenden Schnittstellen sind unter den verschiedenen Unix-Varianten und zum Teil auch unter Microsoft Windows vorhanden.

Da bereits der Zugriff auf Dateien und Verzeichnisse die Voraussetzung nach identischen Schnittstellen auf allen Systemen nicht immer erfüllt, sind neben der Bereitstellung der reinen Benutzeroberfläche noch andere Abstraktionen notwendig. Qt stellt für alle unterstützten Plattformen einheitliche Schnittstellen für derart essenzielle und auch einige weniger oft benötigte Dienste bereit. Das sind zum einen Möglichkeiten für den bereits erwähnten Zugriff auf Dateien und Verzeichnisse, zum anderen auch Netzwerkfunktionen und eine Abstraktion für parallele Programmierung mittels Threads. Weiterhin macht das Toolkit die Einbindung von Datenbanken und die Ausgabe auf Druckern möglich. Eine einfachen Ansprüchen genügende Audioausgabe beherrscht Qt ebenfalls.

Qt erhebt somit den Anspruch, weitreichende Plattformunabhängigkeit für Entwickler und Nutzer von Anwendungen zu ermöglichen. Eine breite Akzeptanz sowohl im kommerziellen Bereich als auch in der Welt freier Software belegt, dass Qt diesem Anspruch auch gerecht wird. Zum Beispiel basiert der freie Unix-Desktop KDE [6] auf Qt, ebenso wie auch der Internetbrowser Opera [9] des gleichnamigen Softwarehauses.

3.2 Qt/Embedded

Qt/Embedded weist gegenüber Qt für Unix/X11, Mac OS X oder Microsoft Windows einige Besonderheiten auf. Während diese drei Betriebssysteme bereits umfangreiche Möglichkeiten für das Fenstermanagement bereitstellen und Qt darauf basierend eine einheitliche Programmierschnittstelle anbietet, verfügt Qt/Embedded über ein eigenes Fenstersystem. Dies ist dem Umstand geschuldet, dass die entsprechende Zielhardware in den meisten Fällen nur über eingeschränkte Ressourcen verfügt. Ein so komplexes System wie beispielsweise das X Window

System auf einem kleinen Gerät, etwa einem PDA oder Mobiltelefon, zu betreiben ist weder sinnvoll noch mit vertretbarem Aufwand bezüglich der Hardware möglich. Viele der vom X Window System angebotenen Funktionen werden ohnehin nicht benötigt, die Netzwerktransparenz zum Beispiel. Daher steht mit dem *Qt Window System* für Qt/Embedded eine Ressourcen schonende Alternative zur Verfügung.

Weiterhin ist es möglich, nicht benötigte Funktionalität aus dem Toolkit zu entfernen. So ist in heute gängigen eingebetteten Systemen die Unterstützung von externen Datenbanksystemen selten notwendig. Und auch bezüglich der verwendeten Datenformate (zum Beispiel für Grafiken und Fonts) besteht Einsparungspotenzial. Damit ist abhängig vom Anwendungsfall eine erhebliche Reduzierung des Speicherverbrauchs von Anwendungen möglich. Dabei entsteht, von der entfernten Funktionalität abgesehen, für den Programmierer keine Einschränkung oder Veränderung der Qt-Schnittstellen. Auf Quellcodeebene bleibt die Kompatibilität innerhalb dieses Rahmens erhalten.

3.3 Architektur von Qt

Allgemein betrachtet handelt es sich bei Qt um eine Bibliothek, die statisch oder dynamisch in die Anwendung eingebunden wird. Sie bildet eine Schnittstelle zwischen dem plattformunabhängigen Programmcode der Anwendung und dem spezifischen Betriebssystem eines Rechners. Der Großteil der Bibliothek ist dabei selbst plattformunabhängig programmiert und greift je nach verwendetem Betriebssystem auf für die jeweilige Umgebung spezifische Backends zurück.

Voneinander unabhängige Dienste des Betriebssystems, wie Ein-/Ausgabe bei Dateien, Netzwerkzugriff oder Threading behalten ihren unabhängigen Charakter auch bei Sicht auf die entsprechenden C++-Klassen in Qt. Bestehende Abhängigkeiten bleiben erwartungsgemäß erhalten, zum Beispiel bedingt die Datenbankanbindung die Netzwerkunterstützung. Die einzelnen Komponenten besitzen somit keine engere Bindung als notwendig, womit Qt insgesamt als modular angesehen werden kann. Dadurch ist es möglich, isolierte Teile der Bibliothek zu deaktivieren, wie in Abschnitt 3.2 bereits angedeutet.

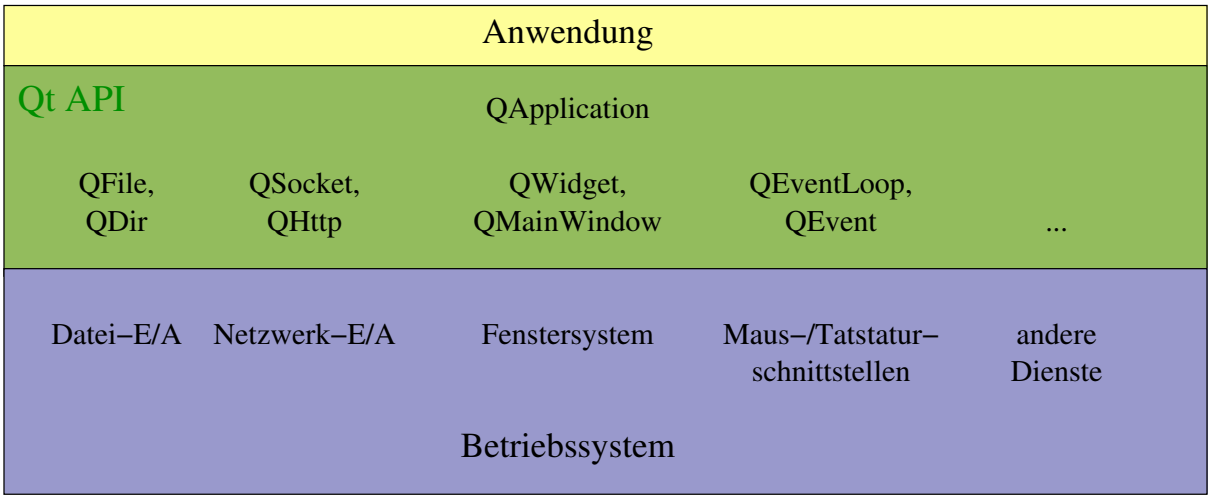

Abbildung 2: Schematische Darstellung der grundlegenden Architektur von Qt

Abbildung 2 vermittelt einen Eindruck von der Architektur von Qt-basierten Anwendungen. Die Klasse `QApplication` repräsentiert eine laufende Qt-Anwendung auf einem Rechner. Sie ist für die Initialisierung des Programms zuständig und führt bei dessen Beendigung Aufräumarbeiten durch.

`QWidget` und `QMainWindow` sind Teil des Frameworks für grafische Benutzeroberflächen. Wie praktisch alle modernen Fenstersysteme ist Qt ereignisorientiert. Wenn eine Anwendung keine Aktivität von sich aus entfaltet (zum Beispiel Berechnungen durchführen), wird sie nur dann aktiv, wenn etwas von außen auf sie einwirkt. Das können Ereignisse sein, die von den Eingabegeräten wie der Maus verursacht werden, oder solche, die von anderen Anwendungen ausgehen. Folglich stellt die für die Ereignisbehandlung zuständige Klasse `QEventLoop` das Kernstück eines Qt-Programms dar. Alle ein Ereignis betreffenden Informationen werden in von `QEvent` erbenden Klassen gekapselt und in entsprechenden Objekten an andere Objekte – die jeweiligen Empfänger – verschickt.

Beispielsweise wird auf diesem Weg die Information, dass mit der Maus an eine bestimmte Stelle geklickt wurde, an das Widget-Objekt weitergeleitet, das sich unter dem Mauszeiger auf dem Bildschirm befindet.

Klassen wie `QFile` oder `QSocket` stellen im Wesentlichen nur eine andere, eben plattformunabhängige Schnittstelle zum Betriebssystem zur Verfügung. Sie gehören zu dem Teil von Qt, welcher als Klassenbibliothek angesehen werden kann. Die Klassen sind unabhängig voneinander und auch weitgehend ohne die grafische Benutzeroberfläche verwendbar, wenngleich sich bei Qt 3.3 hier noch alles in einer einzigen statisch oder dynamisch gelinkten Bibliotheksdatei befindet. In der kommenden Version 4.0 von Qt wird es von Seiten der Bibliothek erstmals eine strikte Trennung von Benutzeroberfläche und der Klassenbibliothek geben. Damit wird Qt zum Beispiel auch für die Entwicklung plattformunabhängiger Server-Dienste interessant.

Für die Klassen zur Nutzung von Threads und den dafür notwendigen Synchronisationsmechanismen gelten die gleichen Aussagen wie für die Datei- und Netzwerkklassen.

In den Abschnitten 4.2, 4.3 und 4.4 wird näher auf diese Subsysteme mit ihren jeweiligen Klassen eingegangen.

3.4 Besonderheiten der Architektur von Qt/Embedded

Wie bereits in Abschnitt 3.2 angedeutet, verfügt Qt/Embedded über eine für geringen Ressourcenverbrauch optimierte Fensterverwaltung: das *Qt Window System (QWS)*. Die zentrale Komponente des QWS stellt der QWS-Server dar. Dabei handelt es sich um einen Qt/Embedded-Prozess, welcher gegenüber gewöhnlichen Qt/Embedded-Anwendungen einige zusätzliche Aufgaben übernimmt. Das ist zum einen das Fenstermanagement und zum anderen werden in diesem Prozess die Eingabegeräte, also Maus und Tastatur, abgefragt. Alle anderen Prozesse sind Clients des QWS-Servers, die über Botschaften mit dem Server-Prozess kommunizieren. Dabei werden nur Eingabeereignisse sowie Statusänderungen von Anwendungsfenstern übertragen.

Abbildung 3 zeigt schematisch die wichtigsten Unterschiede von Qt/Embedded zu anderen Qt-Varianten, hier Qt/X11. Aus diesem Schema vielleicht nicht direkt abzulesen ist noch eine weitere Besonderheit, die durch das QWS bedingt wird. Qt/Embedded verwaltet die Arbeitsfläche komplett selbst, inklusive der nicht von Fenstern bedeckten Bereiche. Dabei wird also

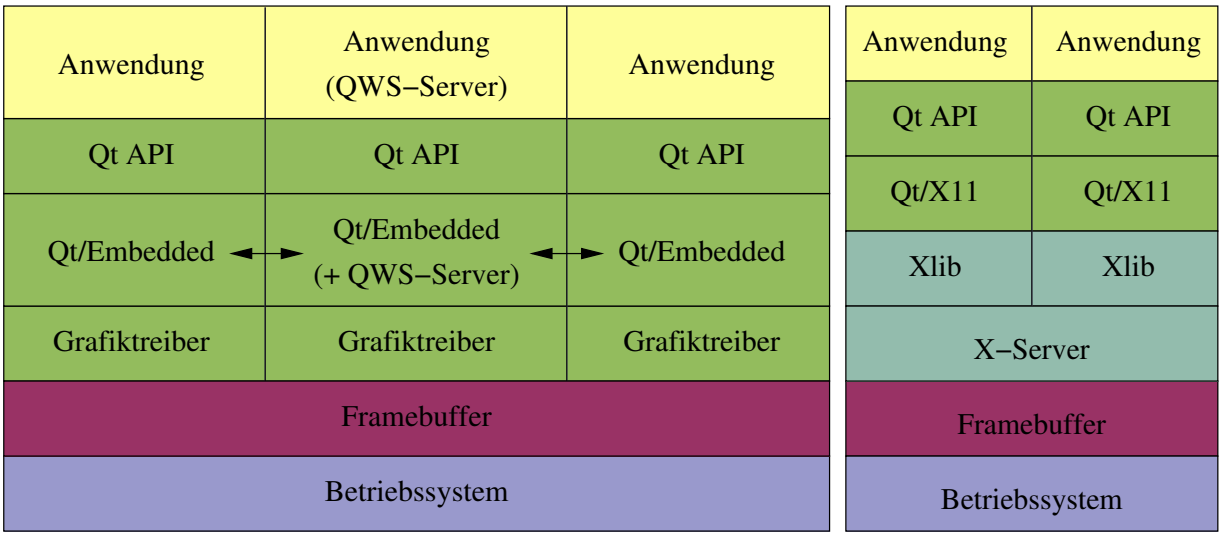

Abbildung 3: Architektur von Qt im Detail: links für Qt/Embedded, rechts für Qt/X11

der gesamte zur Anzeige verwendete Grafikspeicher, der Framebuffer, durch Qt/Embedded kontrolliert.

Während in klassischen Fenstersystemen wie dem X Window System durch die Anwendungen kein direkter Zugriff auf den Framebuffer erfolgt, ist dies beim QWS explizit vorgesehen. Jeder Prozess, auch ein QWS-Client, besitzt jeweils eine eigene Instanz des Grafiktreibers, mit dessen Hilfe er in den für alle zugreifbaren Framebuffer zeichnen kann. Das ermöglicht hohe Performance, leider aber zu Lasten der Sicherheit, da Anwendungen auch auf die grafische Repräsentation der Fenster von anderen gerade ablaufenden Programmen Zugriff haben. Das QWS verwendet an dieser Stelle gemeinsamen Speicher, um sowohl den Framebuffer als auch eine Speicherregion mit Informationen über die Sichtbarkeit der Fenster für alle Prozesse zugänglich zu machen. Andere notwendige Kommunikation zwischen QWS-Server und seinen Clients erfolgt bei Qt/Embedded über Unix Domain Sockets. In Abschnitt 4.5 wird unter dem Gesichtspunkt der Portierung detailierter darauf eingegangen und Kapitel 6 verdeutlicht die Abläufe beim Nachrichtenaustausch dann an einem konkreten Beispiel.

3.5 Softwareentwicklung mit Qt

Im Verlauf dieses Kapitels wurde bisher ein grober Überblick zum Funktionsumfang und der Architektur von Qt gegeben. Interessant ist nun natürlich die Frage, wie man mit Qt Software entwickelt. Insbesondere das von Trolltech eingeführte Konzept der *Signals* und *Slots* bedarf einiger Erläuterungen. Dabei handelt es sich um eine präprozessorbasierte Spracherweiterung für C++, mit der auf einfache und flexible Weise Rückrufe (callbacks) realisiert werden können. Ein umfangreiches Tutorial würde hier allerdings den Rahmen sprengen, zumal Trolltech die entsprechende Dokumentation frei zur Verfügung stellt. Daher soll an dieser Stelle der Hinweis auf Anhang A genügen. Dort finden sich Verweise auf die API-Dokumentation von Qt sowie eine Reihe von Tutorials zur Einführung in die Programmierung mit Qt und die verfügbaren Werkzeuge. Die zu Grunde liegenden Konzepte werden dort leicht verständlich erklärt.

3.6 Eignung für die Portierung auf DROPS

Um zu entscheiden, welche Qt-Variante sich am besten auf DROPS portieren lässt, müssen die verfügbaren Qt-Varianten genauer untersucht werden.

Das hier nicht näher betrachtete Qt für Microsoft Windows ist kaum geeignet, da der plattformspezifische Teil natürlich extensiven Gebrauch von den Windows-APIs macht. Die grafische Benutzeroberfläche und praktisch alle weitere Funktionalität wie beispielsweise Threading müssten komplett neu geschrieben werden. Eine ähnliche Argumentation kann bezüglich Qt für Mac OS X geführt werden.

Damit kommen noch Qt für Unix/X11 und Qt/Embedded in Frage, welche beide für Unix-artige Betriebssysteme bereitstehen. Das ist, wie in Kapitel 4 weiter ausgeführt, sehr nützlich, da ein Teil der Unix-Schnittstellen bereits unter DROPS nachgebildet wurde und somit der Portierungsaufwand geringer ausfällt.

Ein weiteres Entscheidungskriterium stellen natürlich auch die Lizensierungsbedingungen dar. Sowohl Qt/X11 als auch Qt/Embedded sind unter den Bedingungen der GNU General Public License (GPL) [5] verfügbar. Damit sind alle Modifikationen, die bei einer Portierung auf DROPS notwendig sind, ebenfalls wieder der GPL unterworfen. Dies ist akzeptabel, da DROPS selbst dieser Lizenz unterliegt.

Alle genannten Varianten von Qt werden vom Hersteller Trolltech aktiv weiterentwickelt und sie sind auf Anwendungsebene quellcodekompatibel. Letztlich ist es daher ausreichend, an dieser Stelle die jeweiligen Eigenarten von nunmehr nur noch Qt/X11 und Qt/Embedded zu betrachten.

Bei Verwendung von Qt/X11 als Basis sind sehr umfangreiche Anpassungen am Framework für die grafische Benutzeroberfläche von Qt nötig. Die X11-spezifischen Codepassagen haben in der Qt-Version 3.3 einen Umfang von ca. 25.000 LOC (lines of code), leider können diese für DROPS nicht wiederverwendet werden. Stattdessen wäre es erforderlich, die plattformabhängigen Teile der betroffenen Klassen für beispielsweise *DOpE* [10] neu zu implementieren. Allerdings wird mit Qt 4.0 ein vollkommen neuer Unterbau für die grafischen Benutzeroberfläche eingeführt, daher wären für diese Version einmal mehr Anpassungen nötig.

Auf der anderen Seite bringt Qt/Embedded mit dem QWS ein eigenes Fenstersystem mit allen wünschenswerten Eigenschaften wie beispielsweise *Drag'n'Drop* mit. Eine Umsetzung des QWS auf DROPS erfordert hier lediglich das Implementieren von Treibern für die Grafikausgabe sowie die Maus- und Tastaturunterstützung. Der Kernteil des QWS kann auch unter DROPS weiterverwendet werden, in Abschnitt 4.5 wird dies ausführlich diskutiert.

Die benötigten Treiber sind nicht sehr umfangreich, es ist daher auch die Unterstützung mehrerer DROPS-seitiger Backends möglich. Zum Beispiel kann man für den gemeinsamen Framebuffer sowohl *DOpE* als auch *con* [12] oder direkt den Grafikkartenspeicher verwenden.

Andere Teile von Qt, die einer Anpassung an DROPS bedürfen, sind bei Qt/Embedded und Qt für Unix/X11 identisch. Sie spielen bei dieser Analyse hier also nur in der Hinsicht eine Rolle, dass sie natürlich ebenfalls einen signifikanten Anteil des Portierungsaufwand insgesamt ausmachen.

Nach Abwägung aller Argumente wurde hier letztendlich Qt/Embedded als Grundlage für die Portierung ausgewählt. Die Aufwandsreduzierung wird dabei mit gewissen Sicherheitsdefi-

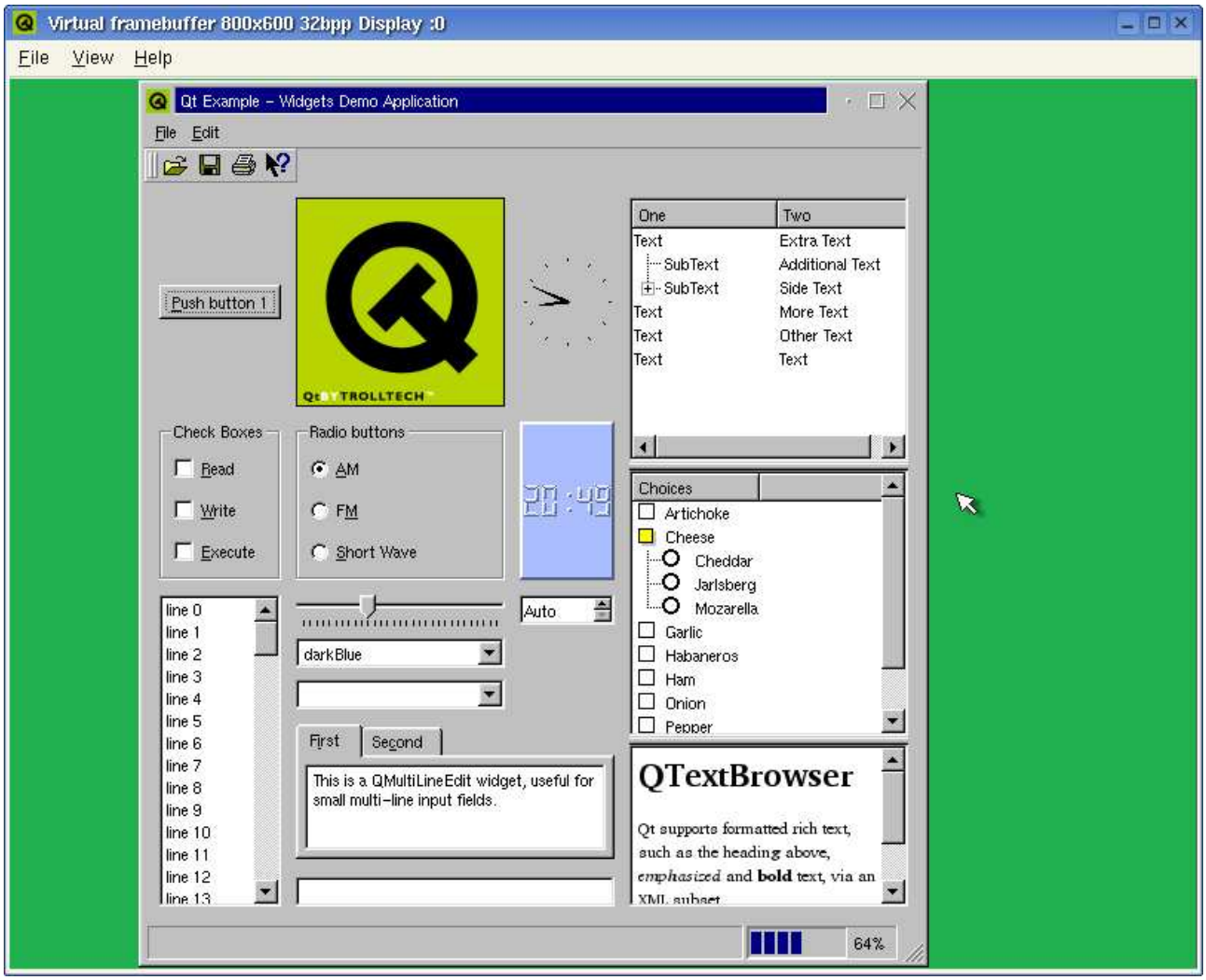

Abbildung 4: Beispielprogramm *widgets* für Qt/Embedded: Das Programm läuft als Qt/Embedded-Anwendung unter einem normalen Linux. Die Grafikausgabe erfolgt über den *Qt/Embedded Virtual Framebuffer*.

ziten erkauft, denn beim QWS von Qt/Embedded haben ja alle Prozesse Zugriff auf den gemeinsamen Framebuffer. Dieser Nachteil kann aber in einer späteren Arbeit beseitigt werden, indem dann eine Anpassung für die direkte Nutzung von, zum Beispiel, *DOpE* durch einzelne Qt-Anwendungen durchgeführt wird. Dabei sollte dann natürlich Qt in der Version 4.0 verwendet werden, denn gerade die dort vorgenommenen Veränderungen an der Basis der grafischen Benutzeroberfläche machen das zur Zeit verfügbare Qt 3.3 für diesen Ansatz wenig attraktiv. Die mit dieser Belegarbeit geschaffenen Grundlagen bei den übrigen Teilen von Qt können dann voraussichtlich in großen Teilen weiterverwendet werden.

4 Portierung

4.1 Vorbetrachtung

Qt ist von sich aus bereits sehr portabel, wie die Verfügbarkeit für eine Vielzahl von Betriebssystemen zeigt. Durch die Trennung von plattformunabhängigem und systemspezifischem Code beschränken sich Portierungsarbeiten auf relativ wenige und lokal konzentrierte Teile des Quellcodes. Diese müssen offensichtlich entweder für DROPS modifiziert oder komplett neu implementiert werden. Wie dabei vorgegangen werden kann, soll im Folgenden geklärt werden.

Die Implementierung von Klassen, die direkt mit dem Betriebssystem interagieren, liegt meist in drei Teilen vor. Die Programmierschnittstelle ist dabei in einer C++-Headerdatei definiert und zwei weitere C++-Dateien enthalten je den systemunabhängigen und den systemspezifischen Teil der Implementierung. Letzterer ist somit der für die Portierung relevante, dort müssen die Anpassungen für DROPS vorgenommen werden.

Ganz offensichtlich gilt es zu vermeiden, dass sich das anwendungsseitige API durch die Portierung auf DROPS verändert. Denn sollte dies geschehen, würde unmittelbar einer der größten Vorteile von Qt verloren gehen, nämlich die Quellcodekompatibilität mit den übrigen Qt-Varianten. Die Forderung nach Wahrung von Kompatibilität kann aber auch dann gestellt werden, wenn die in Qt/Embedded bereits implementierten Schnittstellen zum Betriebssystem Gegenstand der Betrachtung sind. Die systemspezifischen Teile von Qt sind zwar deutlich weniger umfangreich als die plattformunabhängigen, dennoch ist der Aufwand einer Neuimplementation oder Anpassung für DROPS natürlich nicht zu unterschätzen. Es stellt sich also die Frage, ob man den bewährten, für Linux/Unix geschriebenen Code aus Qt/Embedded nicht vielleicht auch unter DROPS wiederverwenden kann. An vielen Stellen greift Qt/Embedded auf POSIX-Schnittstellen zu, die unter DROPS zum Teil schon nachgebildet wurden. Diese können also offensichtlich genutzt werden, so dass die Portierung an diesen speziellen Punkten mit einer Neukompilierung unter DROPS bereits erledigt ist.

Dies ist aber nicht überall möglich. Wenn die geforderten APIs unter DROPS nicht verfügbar sind, muss also entschieden werden, ob man die fehlende Funktionalität inklusive der geforderten Schnittstelle für DROPS implementiert, oder stattdessen Qt an entsprechender Stelle modifiziert.

4.2 Ein-/Ausgabe bei Dateien

Für den Zugriff auf Dateien stehen in DROPS zwei Abstraktionen zur Verfügung, nämlich die File Provider-Schnittstelle *fprov* und *L4VFS* zusammen mit der *dietlibc* [13, 12].

Mit der *fprov*-Schnittstelle wird der Zugriff auf Dateien in Form von Dataspaces möglich, die Dateiinhalte werden also in den Adressraum eines Prozesses eingeblendet. Verzeichnisse und Unterverzeichnisse im traditionellen Sinne existieren nicht, jedoch kann eine hierarchische Struktur über die Namensvergabe realisiert werden.

L4VFS stellt einen hierarchischen Namensraum zur Verfügung, der den aus Unix bekannten Verzeichnisbäumen mit eingehängten Dateisystemen nachempfunden ist. Der Zugriff auf Dateien erfolgt mit den Funktionen `open()`, `read()`, `write()` und `close()`, welche von der *dietlibc* angeboten werden. Dabei handelt es sich um die unter Unix gebräuchliche POSIX-Schnittstelle für Dateioperationen. Es steht hier allerdings kein monolithischer Unix-Kernel im Hintergrund, sondern die *dietlibc* spricht stattdessen eine Reihe von *L4VFS*-Servern an.

In Qt/Embedded dienen die Klassen `QFile`, `QFileInfo` und `Qdir` der dateiorientierten Ein- und Ausgabe. *L4VFS* und *dietlibc* stellen praktischerweise genau die Schnittstelle zur Verfügung, die von der bereits vorhandenen Linux/Unix-Implementierung der genannten Klassen benötigt wird. Damit ist es nicht notwendig, diese Klassen für DROPS anzupassen.

Bei der schrittweisen Durchführung der Portierung hat sich aber doch herausgestellt, dass die eine oder andere benötigte Funktion in *L4VFS* und der *dietlibc* gar nicht oder nicht in ausreichendem Maße implementiert wurde. Offensichtlich macht es wenig Sinn, die fehlende Funktionalität direkt in die Qt-Klassen einzubauen. Wesentlich sinnvoller ist es, *L4VFS* und *dietlibc* dahingehend zu ergänzen. Zum einen profitieren dann auch andere Projekte davon, zum anderen können bestimmte Funktionen nur dann implementiert werden, wenn die zu Grunde liegenden *L4VFS*-Server die notwendige Funktionsbasis überhaupt erst einmal bereitstellen. Ein Beispiel dafür ist die Funktion `stat()`, mit der Informationen über eine Datei erfragt werden können. Hier kann nur der zuständige *L4VFS*-Server diese Informationen liefern.

Im Detail wurden die Funktionen `stat()` und `access()` hier für diesen Beleg ergänzt bzw. neu implementiert.

4.3 Ein-/Ausgabe über Netzwerk

Für die Nutzung von Netzwerkfunktionalität bietet Qt die Klassen `QSocketDevice`, `QSocket`, und Protokollklassen wie `QHttp` an. Wesentlich sind die beiden erstgenannten Klassen, diese verwenden in der Implementierung aus Qt/Embedded die BSD-Socketschnittstelle. Diese wird unter DROPS von der *dietlibc* angeboten. Mit *FLIPS* [12] existiert auch ein *L4VFS*-Server, der einen IP-Stack für die Verwendung von Sockets im Internet-Namensraum bereitstellt. Außerdem werden von Qt/Embedded Unix Domain Sockets benötigt, siehe dazu auch Abschnitt 4.5. Ein entsprechender Server für DROPS ist hier ebenfalls notwendig.

Die Klassen `QSocketDevice` und `QSocket` wurden sowohl mit dem Server für Unix Domain Sockets als auch mit dem IP-Stack *FLIPS* getestet. Änderungen am Netzwerkcode für Qt/Embedded sind nicht notwendig, da die benötigte BSD-Socketschnittstelle dank *dietlibc* und *L4VFS* ja vorhanden ist. Allerdings musste die Funktion `fcntl()` dort nachgerüstet werden, damit die Sockets auch im nicht blockierenden Betrieb verwendet werden können. Ebenso wurde die Funktion `getsockopt()` hinzugefügt.

4.4 Parallele Threads und Synchronisation

Die auf den unterstützten Betriebssystemen vorhandenen Möglichkeiten zur Nutzung von parallelen Threads und deren Synchronisation bekommen mit Qt ebenfalls eine plattformübergreifend einheitliche Schnittstelle. Die Klassen `QThread` und `QThreadStorage` kapseln Threads und Thread-lokale Datenspeicherung. Für die Synchronisation finden die Klassen `QMutex`, `QSemaphore` und `QWaitCondidition` Verwendung.

Die genannten Klassen nutzen im verwendeten Qt/Embedded die unter Unix gebräuchliche PThread-Schnittstelle. Diese Thread-Abstraktion ist für DROPS nicht verfügbar, daher muss ein Ersatz gefunden werden. Die Implementierung von PThreads für DROPS ist durchaus wünschenswert, für den Rahmen dieses Belegs aber zu umfangreich, wenn eine vollständige Abbildung erfolgen soll. Daher ist die direkte Anpassung der betroffenen Klassen hier eine

praktikable Lösung. Die komplexesten Eigenschaften der PThread-Schnittstelle werden von Qt ohnehin nicht genutzt und die übrigen Operationen können auf den L4Env-Paketen *thread* und *semaphore* aufsetzen. Der dabei neu geschriebene Quellcode lässt sich auch in kommenden Versionen von Qt mit großer Wahrscheinlichkeit weiterverwenden. So wird in Qt 4.0 die Programmierschnittstelle nicht wesentlich verändert, es kommt lediglich die Klasse `QSpinLock` hinzu.

Es existiert noch eine weitere Klasse, `QMutexPool`, welche aber nur intern verwendet wird. Diese muss nicht explizit portiert werden, da sie intern lediglich `QMutex` benutzt. Für `QSpinLock` und die übrigen, bereits in der aktuellen Qt-Version vorhandenen Klassen stellt die Portierung keine wesentliche Schwierigkeit dar. Bei den in der Version 3.3 vorhandenen Klassen ist die Anpassung an DROPS mit diesem Großen Beleg vollständig erfolgt.

4.5 Kernfunktionalität von Qt/Embedded: Das Qt Window System

Die Kernkomponente von Qt-Anwendungen ist die Event Loop, implementiert in der Klasse `QEventLoop`. Diese Klasse erfordert in der Qt/Embedded-Variante ebenfalls ein Unix-Betriebssystem. Teil der Event Loop ist auch Code zur Unterstützung der Klasse `QTimer`. Für den Betrieb mehrerer Qt/Embedded-Anwendungen nebeneinander sind außerdem die Klassen `QWSServer` und `QWSDisplay` von entscheidender Bedeutung.

Um zu entscheiden, wie eine Portierung bezüglich dieser Klassen am besten durchzuführen ist, muss deren Zusammenspiel genau betrachtet werden. Mit Hilfe der Funktion `select()` wird innerhalb von `QEventLoop` die Nutzung von mehreren Sockets gleichzeitig ermöglicht, ohne dass die gesamte Anwendung dabei an einem einzelnen Socket blockiert. Außerdem wird das Erreichen von Timeouts überwacht, die über die Klasse `QTimer` gesetzt werden. `QWSServer` und `QWSDisplay` implementieren die notwendige Kommunikation zwischen dem QWS-Server-Prozess und den Clients. Dabei wird pro Client ein Unix Domain Socket für das Versenden und Empfangen von QWS-Botschaften verwendet.

Abbildung 5 stellt diesen Sachverhalt schematisch dar, welcher in Kapitel 6 durch ein entsprechendes Beispiel auch noch einmal detaillierter betrachtet wird. Die Dateideskriptoren der vom QWS benutzten Sockets werden in der Event Loop durch die `select()`-Funktion überwacht. Die von `QSocket` erbende Klasse `QWSSocket` kapselt diese Sockets. Die Funktionalität von `select()` wird durch die `QSocketNotifier` abstrahiert und außerhalb von `QEventLoop` durch `QWSServer` und `QWSDisplay` genutzt. Die genannten Klassen sind in hohem Maße voneinander abhängig und müssen hier in ihrer Gesamtheit betrachtet werden.

Als Problem stellte sich bei der Arbeit an diesem Beleg heraus, dass es bisher keine Nachbildung von Unix Domain Sockets unter DROPS gibt. Um dieses Problem zu lösen, sind zwei Vorgehensweisen möglich:

1. Modifikation von Qt/Embedded zur Nutzung von L4-IPC anstelle von Unix Domain Sockets.

2. Implementieren eines Servers, der Unix Domain Sockets bereitstellt.

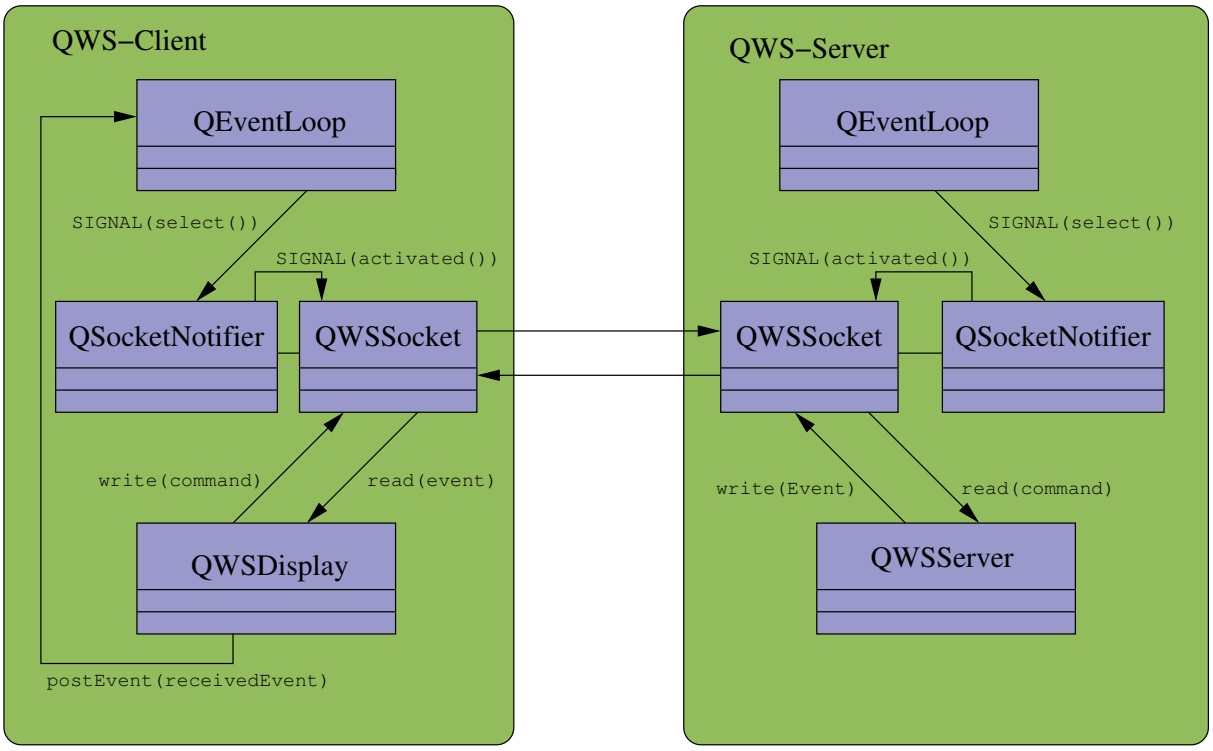

Abbildung 5: Vereinfachte Darstellung der Kommunikation zwischen QWS-Server und QWS-Client: die serverinterne Klasse *QWSClient* wurde weggelassen, die Aufrufpfade der Methoden sind auf das wesentliche verkürzt durch Pfeile dargestellt.

Die erste Möglichkeit hat den Vorteil, dass hier der Aufwand zum Implementieren eines zusätzlichen Servers entfällt. `QWSServer` und die Gegenstelle `QWSDisplay` benötigen lediglich das Senden und Empfangen von Nachrichten, ferner muss der entsprechende L4-IPC-Kanal geeignet in die Event Loop integriert werden.

Auf Grund der Architektur von Qt/Embedded finden Senden und Empfangen von QWS-Nachrichten in dem Thread statt, in dem die Event Loop aktiv ist. Sowohl der QWS-Server-Prozess als auch ein Client können Initiator einer Sendeoperation sein kann und eine solche kann zu einem beliebigen Zeitpunkt beginnen. Dabei müssen Situationen vermieden werden, in denen der Sender blockiert. Das ist immer dann der Fall, wenn der Empfänger nicht empfangsbereit in der Event Loop wartet. Es ist davon auszugehen, dass diese Situation regelmäßig eintritt. Um gute Performance zu erhalten müssen folglich in jedem Prozess dedizierte Threads ablaufen, die IPC-Botschaften für das QWS ohne Verzögerung empfangen und zunächst puffern. Bei Verwendung der L4v2-Kernschnittstelle ist es außerdem erforderlich, dass eine eventuelle Antwort aus genau diesem Thread zurückgeschickt wird.

In der vorgegebenen Implementierung von `QEventLoop` in Qt/Embedded werden `QWSServer` und `QWSDisplay` indirekt aus der Event Loop heraus aktiv, wenn über die Socketverbindung QWS-Botschaften empfangen werden. Bei Umstellung auf L4-IPC müssen Änderungen an der Event Loop vorgenommen werden, um die Botschaften auf andere Weise für `QWSServer` und `QWSDisplay` verfügbar zu machen. Dies könnte vermieden werden, indem die Methoden beider Klassen von den QWS-Empfangs-Threads aufgerufen werden. Dafür sind zusätzliche Locks

einzuführen, um den notwendigen wechselseitigen Ausschluss zu gewährleisten. Positive oder negative Folgen für die Performance sind hier im Vorfeld schwer abzuschätzen.

Die Umstellung von QWS auf L4-IPC kommt offensichtlich in weiten Teilen einer nachträglichen Parallelisierung einer Architektur gleich, welche für die Verwendung eines einzelnen Threads entworfen und implementiert wurde. Mit Blick auf die Version 4.0 von Qt ist ebenfalls fraglich, welcher Aufwand nötig sein wird, um die durchgeführten Änderungen in die neue Version zu integrieren.

Bei der zweiten Möglichkeit für die Portierung der Qt/Embedded-Kernkomponenten, dem Bereitstellen eines Servers für Unix Domain Sockets, sind die Änderungen an Qt/Embedded selbst minimal. Sie beschränken sich auf die Funktionalität zum Aufwecken des Threads, welcher die Event Loop ausführt. Neben weiteren Dateideskriptoren wird dort im zentralen `select()`-Aufruf in `QEventLoop` auch das Leseende einer Unix-Pipe überwacht. Durch Schreiben eines Dummy-Wertes in das Schreibende dieser Pipe können andere Threads den in `select()` blockierten Thread aufwecken. Unter DROPS wird der dafür notwendige `pipe()`-Systemaufruf von der *dietlibc* aber nicht unterstützt. Allerdings lässt sich der gleiche Effekt erzielen, indem mit der Funktion `socketpair()` zwei miteinander verbundene Sockets erzeugt werden. Der übrige auf Unix Domain Sockets basierende Code kann unverändert übernommen werden, wenn unter DROPS eine entsprechende Socket-Implementierung verfügbar gemacht wird.

`QWSSocket`, `QWSDisplay` und `QEventLoop` für die Nutzung von L4-IPC anzupassen stellt bei näherer Betrachtung eine Insellösung dar. Der Grad der Wiederverwendbarkeit ist bereits beim Übergang auf Qt 4.0 nicht abschätzbar. Dagegen ist die Implementierung eines L4Env-Servers für die vom vorhandenen Code benötigten Sockets möglicherweise aufwändiger. Der Server kann aber auch für andere Zwecke weitergenutzt werden, selbst wenn zukünftige Qt-Versionen andere Ansprüche stellen sollten. Zumindest für Qt/Embedded 4.0 ist dies aber nicht der Fall.

Die Bereitstellung von Unix Domain Sockets ist hier somit die bessere Wahl. Für diesen Beleg wurde daher ein *L4VFS*-Server implementiert, der die benötigte Funktionalität zur Verfügung stellt. Kapitel 5 befasst sich mit diesem Server.

4.6 Interne Hilfsklassen des Qt Window Systems

Neben `QWSServer`, `QWSDisplay` und dem von `QSocket` erbenden `QWSSocket` verwendet das Qt Window System intern noch weitere Klassen. Für die Portierung zu betrachten sind dabei `QLock`, `QSharedMemory` und `QWSRegionManager`.

4.6.1 QSharedMemory, QWSRegionManager

Aus Performancegründen werden im QWS einige Datenstrukturen des QWS-Server-Prozesses über gemeinsamen Speicher auch den Clients zugänglich gemacht. Dabei handelt es sich um Datenstrukturen, in denen Sichtbarkeit und Position der Fenster der Client-Prozesse vermerkt sind. Außerdem gibt es noch einen Speicherbereich, in dem Informationen zum gemeinsam genutzten Framebuffer abgelegt sind.

Dieser gemeinsame Speicher wird im vorliegenden Qt/Embedded in den Klassen `QSharedMemory` und `QWSRegionManager` über System V-Shared-Memory bereitgestellt.

`QWSRegionManager` enthält dabei redundanten Code, welcher ebenfalls in `QSharedMemory` vorhanden ist. Diese Klasse kann jedoch mit geringen Modifikationen auch `QSharedMemory` nutzen, daher kann sich die Portierung auf die letztgenannte beschränken.

Da die System V-IPC-Schnittstellen unter DROPS nicht verfügbar sind, muss an dieser Stelle ein Ersatz entwickelt werden. System V-IPC unter DROPS nachzubilden ist nicht sinnvoll, da selbst unter Unix die Zahl der Programme, die diesen Standard nutzen, geringer wird.

Die unter L4Env gebräuchlichen Dataspaces sind aber als Grundlage für `QSharedMemory` hervorragend geeignet. Um auf die jeweils gewünschte Region gemeinsamen Speichers zugreifen zu können, müssen der QWS-Server und die Clients diesen über einen eindeutigen Namen identifizieren können. Die übliche Vorgehensweise in DROPS ist dabei, dass ein Server einen neuen Dataspace anlegt und dessen Dataspace-ID dann per IPC vom Client erfragt wird. Qt/Embedded verwendet wie bei System V-IPC vorgesehen eindeutige IDs, die hier mit der Funktion `ftok()` aus dem Dateinamen des vom QWS verwendeten Sockets und einem weiteren Wert gebildet werden. Der Name des Sockets adressiert dabei eine Menge von Qt/Embedded-Prozessen, die alle denselben QWS-Server verwenden, der zweite Teil des Namens bestimmt den gemeinsamen Speicherbereich.

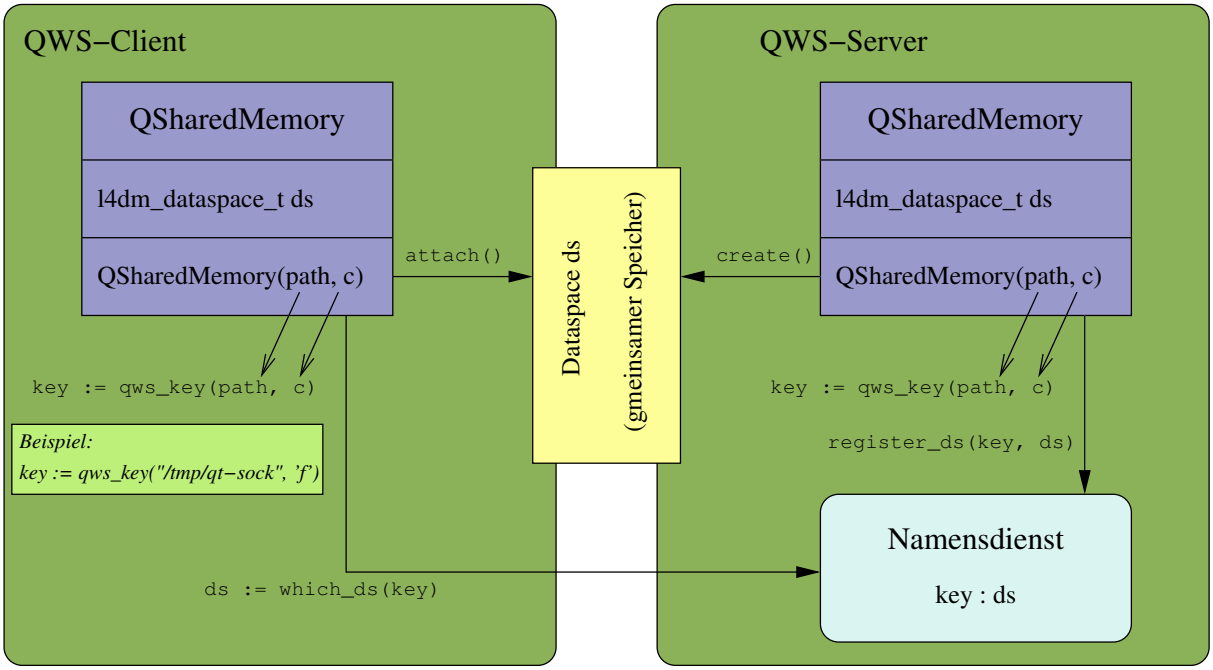

Abbildung 6: Portierung der System V-IPC basierten Klasse *QSharedMemory* auf DROPS mittels Dataspaces.

`QSharedMemory`-Objekte werden nur an drei Stellen im Qt/Embedded-Quellcode erzeugt und ausschließlich intern verwendet. Dennoch ist es sinnvoll, das von Qt/Embedded vorgegebene Namensschema beizubehalten und die Portierung an dieser Stelle komplett in der Klasse `QSharedMemory` zu kapseln.

Erforderlich ist für die Clients nun nach wie vor, die Dataspace-ID vom QWS-Server zu erfragen. Dies macht die Implementierung eines lokalen Namensdienstes erforderlich, der die an

die Klasse `QSharedMemory` übergebenen Namen auf eine Dataspace-ID abbildet. Abbildung 6 veranschaulicht die hier diskutierte Vorgehensweise.

Im Rahmen dieses Belegs wurde die die Klasse `QSharedMemory` neu implementiert, so dass die beschriebenen Anforderungen erfüllt werden. Dabei wird der benötigte Namensdienst durch einen zusätzlichen Thread im QWS-Server bereitgestellt, der auf Anfrage die Dataspace-ID liefert.

4.6.2 QLock

Der Zugriff auf den Framebuffer und den gemeinsamen Speicher durch mehrere Prozesse erfordert eine Synchronisierung, welche durch die Klasse `QLock` erfolgt. Die Klasse bietet ein Lese-/Schreib-Lock an und bedient sich in Qt/Embedded dafür eines System V-Semaphors.

Eine Möglichkeit zur prozessübergreifenden Synchronisierung dieser Art existiert für DROPS zur Zeit nicht, dafür implementieren die einzelnen Anwendungen eigene, auf die jeweiligen Anforderungen zugeschnittene Verfahren unter Nutzung von L4-IPC. Insbesondere gibt es keine System V-Semaphore unter DROPS. Daher ist eine Neuimplementierung von `QLock` nötig. Bezüglich der Portierung auf DROPS kann hier wie bei der zuvor diskutierten Klasse `QSharedMemory` vorgegangen und damit die äußere Schnittstelle unverändert übernommen werden. Das verwendete Namensschema für die Adressierung des gewünschten Locks ist dasselbe wie bei `QSharedMemory`.

Da ein durch `QLock` gekapseltes Lock von mehreren Qt/Embedded-Prozessen genutzt wird, ist natürlich Interprozesskommunikation zur Synchronisierung notwendig. Für den Fall, dass das Lock bereits durch einen anderen Thread – eines anderen Prozesses – belegt ist, muss die Ausführung des neuen Bewerbers zunächst suspendiert und nach Freigabe des Locks fortgesetzt werden. Dazu ist eine zentrale Instanz notwendig, die die Warteschlange der suspendierten Threads verwaltet und diese gegebenenfalls wieder aufweckt.

Die Möglichkeit, das bereits existierende L4Env-Paket *semaphore* als Grundlage zu benutzen, erscheint auf den ersten Blick interessant, stellt sich bei genauerer Betrachtung allerdings als schlecht realisierbar heraus. Zum einen wird vorausgesetzt, dass alle Bewerber um einen Semaphor Threads des eigenen Prozesses sind. Zum anderen wird von der Implementierung der Zugriff auf gemeinsamen Speicher gefordert, der bei getrennten Prozessen nicht ohne weiteres vorhanden ist. Das *semaphore*-Paket hätte also erhebliche Anpassungen erfordert, die außerdem die Komplexität noch weiter in die Höhe getrieben hätten.

Deshalb findet eine andere Lösung Verwendung. Analog zur prozesslokalen Implementierung für Semaphore im *semaphore*-Paket wird dazu ein dedizierter Thread zur Koordination eingesetzt, der hier aber auch Anfragen von Threads anderer Prozesse akzeptiert und diesen exklusiven oder gemeinsamen Zugriff auf die vom Lock geschützten Ressourcen gewährt. Es wird also ein Lese-/Schreib-Lock realisiert. Wegen der getrennten Adressräume müssen bei dieser Architektur grundsätzlich alle Operationen, also jedes Sperren und Freigeben des Locks, über den erwähnten Koordinations-Thread erfolgen. Das heißt, dass auch beim Anfordern eines nicht belegten Locks Kommunikation mit besagtem Thread erforderlich ist.

Im Gegensatz zu `QSharedMemory` kann die von `QLock` bereitgestellte Funktionalität zur Synchronisierung mehrerer Prozesse prinzipiell auch für andere Projekte in der DROPS-Umgebung

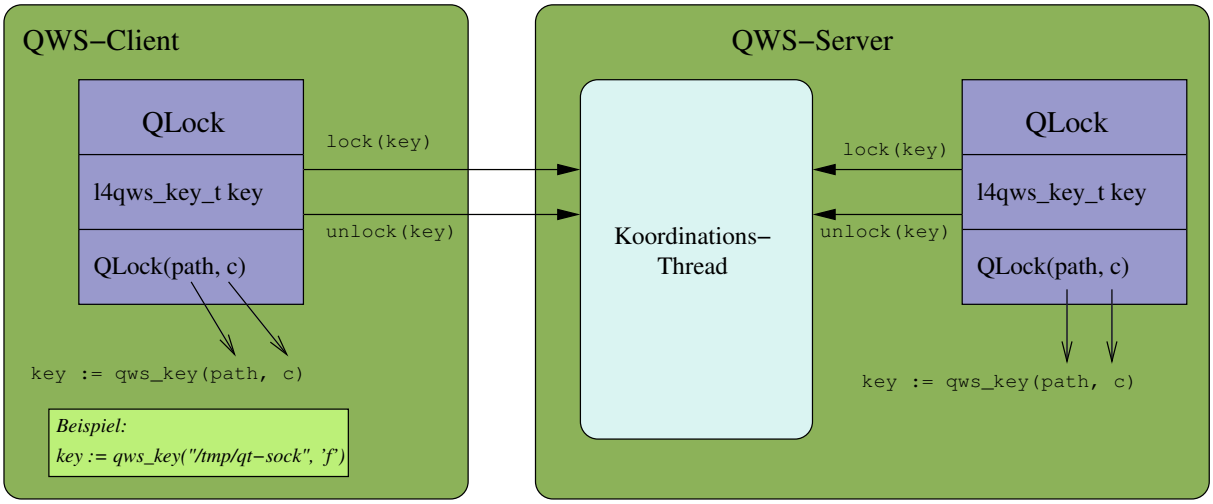

Abbildung 7: Portierung der System V-IPC basierten Klasse *QLock* auf DROPS.

nützlich sein. Eine Berücksichtigung von verschiedenen Thread-Prioritäten findet in der Klasse `QLock` aber nicht statt, was für die allgemeine Verwendung in DROPS wünschenswert wäre. Deshalb wird die Synchronisation nicht durch einen unabhängigen Server-Prozess unter DROPS realisiert. Stattdessen kommt die in Abbildung 7 dargestellte Architektur zum Einsatz, bei welcher der diskutierte Koordinations-Thread im ohnehin schon mit Koordinierungsaufgaben betrauten QWS-Server abläuft. Dies bedeutet praktischerweise einen Performancevorteil bei Benutzung des Locks durch den QWS-Server selbst, weil dann zwei Adressraumumschaltungen entfallen können.

In Abschnitt 7.4 wird die nach dem hier vorgestellten Entwurf vorgenommene Implementierung der Klasse `QLock` hinsichtlich ihrer Performance genauer untersucht und bewertet.

5 L4VFS-Server für Unix Domain Sockets

5.1 Anforderungen an die Implementierung des Socket-Servers

Nachdem in Abschnitt 4.5 zunächst nur auf die Notwendigkeit der Bereitstellung von Unix Domain Sockets unter DROPS eingegangen wurde, soll nun geklärt werden wie dies am zweckmäßigsten erfolgen kann und welche Anforderungen dabei zu erfüllen sind. Dies wird im Kontext der Portierung von Qt/Embedded auf DROPS betrachtet. Dazu wird analysiert, welche Funktionalität der Unix Domain Sockets von Qt, im besonderen also dem *Qt Window System*, benötigt wird.

Unix Domain Sockets können in zwei verschiedenen Betriebsarten verwendet werden. Zum einen verbindungsorientiert und zum anderen verbindungslos, indem einfach nur Datenpakete verschickt werden, ohne das vorher eine Verbindung aufgebaut werden müsste. Das QWS verwendet Unix Domain Sockets in der verbindungsorientierten Betriebsart, das heißt die Funktionen `bind()`, `listen()`, `accept()` und `connect()` werden genutzt, um die Nachrichtenkanäle zwischen QWS-Server und QWS-Client einzurichten. Im Folgenden werden die Botschaften dann mit den Funktionen `send()` und `recv()` bzw. `read()` und `write()` ausgetauscht. Es ist ebenfalls notwendig, dass der Socket-Server auch nicht-blockierenden Zugriff auf die Sockets erlaubt, eine Leseoperation sollte also beispielsweise nicht blockieren, wenn keine Daten von der Gegenseite gelesen werden können. Damit einhergehend kommt auch die Funktion `select()` zum Einsatz, wie in Abschnitt 4.5 bereits ausgeführt.

Damit ist der von Qt/Embedded benötigte Funktionsumfang bereits vollständig erfasst. Die für Unix Domain Sockets noch verbleibende paketorientierte und verbindungslose Betriebsart wird nicht verwendet. Dies wird auch von der anwendungsseitigen Programmierschnittstelle von Qt nicht gefordert, da Unix Domain Sockets dort anders als Sockets im Internet-Namensraum – IP-Sockets – nicht angeboten werden. Damit ist Unterstützung für verbindungslosen Betrieb im Socket-Server auch nicht zwingend erforderlich.

5.2 Integration des Socket-Servers

Die *dietlibc* und *L4VFS* haben sich bereits in Kapitel 4 als eine ideale Grundlage für die Portierung von Qt auf DROPS herausgestellt. Es liegt also nahe, diese Komponenten von DROPS daraufhin zu untersuchen, wie sie sich als Basis für einen Socket-Server eignen.

Es existieren bereits zwei andere Socket-Server, welche durch *L4VFS* und die *dietlibc* nutzbar sind. Das ist zum einen der IP-Stack *FLIPS*, zum anderen *pf_key_v2*, ein Server, der Version 2 des PF_KEY Management API implementiert [21, 22]. Beide Server implementieren dieselbe generische *L4VFS*-Schnittstelle, welche alle für die Nutzung von Sockets notwendigen Funktionen bereitstellt. Das sind also zum Beispiel `socket()`, `connect()`, `accept()` sowie `send()` und `recv()`. Die *dietlibc* bildet die von Unix gewohnte BSD-Socket-Schnittstelle dann auf die Funktionalität der *L4VFS*-Server ab. Dazu wird ausgehend vom lokalen Dateideskriptor die Thread-ID des jeweiligen *L4VFS*-Servers ermittelt und schließlich der IDL-Stub der gewünschten Socket-Funktion für diesen Server aufgerufen.

Generell erfolgt die Einbindung von Servern in *L4VFS* immer auf diese Weise, in Abbildung 8 ist dies in einem Schema dargestellt.

Um Unix Domain Sockets unter DROPS zur Verfügung zu stellen, ist es somit lediglich notwendig, einen *L4VFS*-Server zu implementieren, der die bereits definierte Socket-Schnittstelle

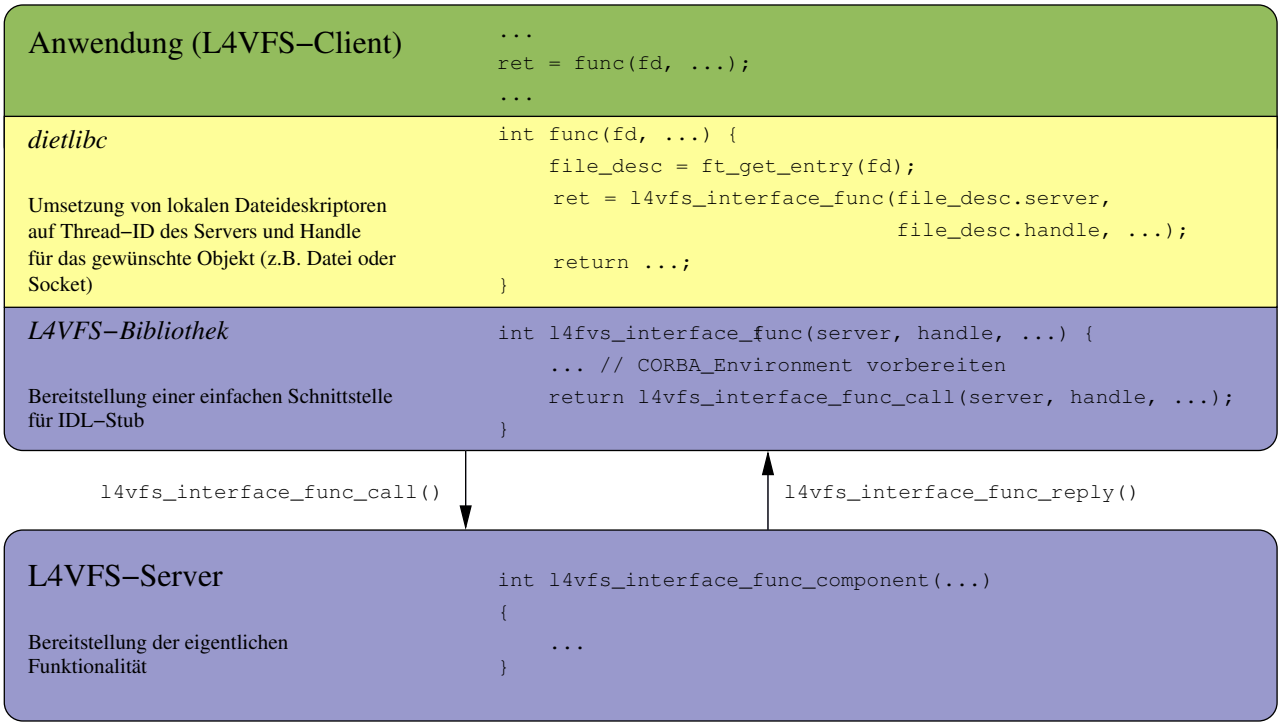

Abbildung 8: Architektur von *L4VFS* am Beispiel der Funktion *func()* für die IDL-Schnittstelle *interface*.

anbietet. Die gesamte notwendige Infrastruktur zur Nutzung mit der *dietlibc* ist bereits vorhanden. Darin eingeschlossen ist auch die Verfügbarkeit der `select()`-Funktion, welche ja ebenfalls benötigt wird.

Da Qt an sich – mit Blick auf kommende Versionen – nur so wenig wie möglich verändert werden soll, ist eine Unterstützung von Sockets auf Ebene der C-Bibliothek notwendig. Dies ist bei dem hier diskutierten Ansatz gegeben. Andere Methoden zur Einbindung eines Socket-Servers erscheinen nicht sinnvoll, würde dies doch die Entwicklung bzw. Modifikation einer zum vorhandenen – und gewollten – BSD-Socket-API redundanten Schnittstelle bedeuten. Auch der Verzicht auf die *dietlibc* verbietet sich wegen ihrer Nützlichkeit für andere Dienste, zum Beispiel beim Dateizugriff.

5.3 Entwurf des Socket-Servers

5.3.1 Grundlegende Entwurfsentscheidungen

Nach den allgemeinen Betrachtungen in den vorangegangenen Abschnitten 5.1 und 5.2 wird nun konkreter diskutiert, wie die Entwicklung des Socket-Server erfolgen soll. An dieser Stelle wird es Zeit, einen Namen für den neuen Server einzuführen. Unix Domain Sockets werden gelegentlich auch als 'lokale Sockets' oder 'Sockets im lokalen Namensraum' bezeichnet. Ausgehend davon wurde für den hier diskutierten Server der Name *local_socks* gewählt, welcher im Folgenden verwendet wird.

Die beiden bereits existierenden Socket-Server *FLIPS* und *pf_key_v2* basieren in großen Teilen auf Linux-Kernelcode. Das heißt, die originalen C-Dateien des Linux-Kernel können zusammen mit einer Linux-Emulationsumgebung, dem *DROPS Device Driver Environment*, auch unter DROPS genutzt werden.

Um Unix Domain Sockets unter DROPS bereitzustellen, besteht nun die Möglichkeit ähnlich wie bei den beiden zuvor genannten Socket-Servern vorzugehen und die Linux-Implementierung unter DROPS nutzbar zu machen. Oder man führt an dieser Stelle eine eigene Implementierung ein.

Für diesen Beleg wurde die zweite Möglichkeit gewählt. Dieser Entscheidung liegt zu Grunde, dass ein Herauslösen des Linux-Codes die Auflösung einer großen Menge von Abhängigkeiten zu anderen Teilen des Linux-Kernel bedeutet hätte. Dies wiederum hätte neben der Einarbeitung in die Interna von Qt/Embedded eine ebensolche bezüglich des Linux-Kerns erforderlich gemacht. Der Aufwand einer Neuimplementierung ist dagegen geringer und vor allem besser abschätzbar. Da die Unix Domain Sockets schließlich frühzeitig für die Portierung benötigt wurden, ist somit der Entschluss einer den Bedürfnissen von Qt/Embedded entsprechenden Neuimplementation gefasst worden.

Eine Nutzung des Linux-Kernelcodes hätte, mit welchem Aufwand auch immer, die vollständige Bereitstellung der von Linux her gewohnten Funktionalität bezüglich Unix Domain Sockets ermöglicht. Es ist daher wünschenswert, dass eine neue Implementierung in *local_socks* auch bei Nutzung des Vorteils der Einfachheit keine Schranken für die zukünftige Erweiterbarkeit oder bei der zu erreichenden Performance schafft. Da zunächst nur die verbindungsorientierte Nutzung von Sockets ermöglicht werden soll, muss ein Nachrüsten der Unterstützung für verbindungslose Socket-Kommunikation ohne großen Aufwand möglich sein.

Bei näherer Betrachtung sind diese beiden Betriebsarten aber gar nicht so verschieden. In beiden Fällen ist eine bidirektionale, gepufferte Kommunikation zu gewährleisten, die sich in zwei Phasen unterteilen lässt, wenn man das finale Schließen des Kommunikationskanals einmal außen vor lässt. Zuerst muss eine Verbindung zwischen den beiden beteiligten Kommunikationspartnern hergestellt werden und dann kann in der zweiten Phase der Austausch von Nachrichten erfolgen. Im Falle der verbindungsorientierten Nutzung bleibt der Kanal offen und kann für das Senden und Empfangen weiterer Botschaften benutzt werden, im anderen Fall wird er wieder geschlossen. Das Senden und Empfangen abgeschlossener Datenpakete für die verbindungslos/paketorientierte Betriebsart lässt sich natürlich auch auf stromorientierten Nachrichtenaustausch abbilden, indem der Server die Menge der übermittelten Daten bei Versand und Empfang überprüft. Dabei darf eine solches Paket nur dann in einen Puffer kopiert werden, wenn es komplett hineinpasst.

Die Notwendigkeit von Puffern ergibt sich aus der Tatsache, dass ein Kommunikationspartner zumindest bei der stromorientierten Verbindung die Daten in größeren oder kleineren Einheiten schicken oder empfangen kann, als von der Gegenstelle erwartet wird. Außerdem kann dadurch natürlich auch ein Blockieren eines Kommunikationspartners verhindert werden, wenn dieser beispielsweise gerade in einen Socket schreibt, auf der anderen Seite jedoch in diesem Moment nicht gelesen wird. Da die Kommunikation bidirektional ist, bietet sich die Verwendung von jeweils einem Puffer für jede Richtung an.

Damit sind einige wichtige Rahmenbedingungen für den inneren Aufbau von *local_socks* betrachtet, welche nun die konkrete Architektur bestimmen.

5.3.2 Funktionelles Design

Durch *L4VFS* und natürlich auch die unter DROPS zu nutzenden L4-IPC-Möglichkeiten ergeben sich weitere Aspekte der Architektur von *local_socks*. Für jede der nach außen hin angebotenen Socket-Funktionen, also zum Beispiel `send()`, gibt es auf Server-Seite einen Funktions-Stub, der bei jeder Anfrage eines Clients aufgerufen wird. Dabei ist zu beachten, dass beispielsweise im Falle von `send()` und `write()` fast die gleiche Operation auszuführen ist. Wie in der Softwaretechnik allgemein üblich, ist es hier also sinnvoll, die äußere Schnittstelle und die eigentliche Funktionalität zu trennen. Daher wird im Funktion-Stub lediglich eine eventuell notwendige Überprüfung und Anpassung von Parametern durchgeführt und dann die eigentliche Operation von einer oder mehreren separaten Funktionen ausgeführt. Es wird also, um beim Beispiel zu bleiben, nur eine Funktion zum Senden implementiert: `send_internal()`. Daraus ergibt sich die Möglichkeit, diese Funktion später auch für die verbindungslose Kommunikation, also zum Beispiel bei der Realisierung von `sendto()`, zu verwenden. Dieser Ansatz erleichtert somit die Erweiterbarkeit und hilft, die Codebasis klein und frei von Redundanz zu halten.

Um dem Umstand gerecht zu werden, dass die Kommunikationspartner einer Socket-Verbindung immer unterschiedliche Threads meist innerhalb verschiedener Prozesse sind, ist *local_socks* von vornherein so entworfen worden, dass einzelne Client-Anfragen auch innerhalb des Servers in mehreren Threads abgearbeitet werden. Dabei wird jeder Client durch einen eigenen Server-Thread bedient. Dies hat den unmittelbaren Vorteil, dass sich die Implementierung wesentlich vereinfacht, da nicht auf umständliche Art und Weise die Zustände der für die Clients ausgeführten Operationen verwaltet werden müssen. Stattdessen sind die dafür notwendigen Verwaltungsinformationen weitestgehend lokal für einzelne Threads, beziehungsweise durch den eigenständigen Kontrollfluss innerhalb dieser ohnehin unnötig.

Ein weiterer, sehr wertvoller Gewinn durch die Parallelität innerhalb des Servers ist natürlich, dass sich die Clients nur begrenzt gegenseitig beeinflussen können. Bei einer lang andauernden Operation, wie zum Beispiel dem Senden einer langen Botschaft, können andere Clients durch separate Server-Threads parallel bedient werden.

5.3.3 Datenstrukturen

Es stellt sich schließlich auch die Frage, wie die einen Socket betreffenden Verwaltungs- und Zustandsinformationen gespeichert werden sollen und ob außer den Informationen zu den Sockets selbst noch andere Datenstrukturen benötigt werden. Beim Studium der reichlich vorhandenen Dokumentation zur BSD-Socket-Schnittstelle [16, 15] ergibt sich, dass es kaum erforderlich ist, globale Daten vorzuhalten. Stattdessen beziehen sich fast alle Verwaltungsinformationen immer auf einen konkreten Socket, das heißt auf die Repräsentation eines Kommunikationspartners innerhalb des Servers. Einzige Ausnahme stellt hier die Tabelle für die Abbildung von Adressen im lokalen Namensraum auf die daran gebundenen Sockets dar. Um effizient ermitteln zu können, zu welchem Socket beim Aufruf von `connect()` eine Verbindung hergestellt werden soll, ist eine eigens dafür vorgesehene Datenstruktur sinnvoll. Die Alternative bestünde darin, die Liste aller geöffneten Sockets nach dem einen zu durchsuchen, der zuvor mit `bind()` an diese Adresse gebunden wurde.

Es bietet sich also an, alle einen geöffneten Socket betreffenden Informationen zusammenzufassen, die ganz einfach in einer gewöhnlichen C-Struktur gespeichert werden. Beinahe alle Socket-Operationen referenzieren den entsprechenden Socket über einen Dateideskriptor, wel-

cher dann für *L4VFS* auf ein für den jeweiligen Server eindeutiges Objekt-Handle abgebildet wird. Dieses Handle findet daher naheliegenderweise auch Verwendung zum Adressieren der für jeden Socket spezifischen Verwaltungsinformationen innerhalb von *local_socks*.

5.4 Implementierung von *local_socks*

Die Implementierung von *local_socks* umfasst ohne den für die IDL-Anbindung notwendigen Code ca. 1200 LOC (lines of code). Deshalb können an dieser Stelle natürlich nicht alle Details diskutiert werden. Stattdessen wird exemplarisch auf die Implementation der beiden wichtigsten Funktionen, `send()` und `recv()`, sowie die beteiligten Datenstrukturen eingegangen.

Wie bereits mehrfach erwähnt, findet das Übermitteln von Botschaften immer zwischen genau zwei miteinander verbundenen Sockets statt. Daher ist es notwendig, die jeweilige Gegenseite in der C-Struktur für den Socket zu vermerken, sofern dieser denn überhaupt schon mit einem zweiten Socket verbunden wurde. Für die Repräsentation eines jeden Sockets gibt es eine Instanz der C-Struktur `socket_desc_t`, die auszugsweise wie folgt deklariert ist:

```
typedef struct socket_desc {
  ...
  int state:14;              /* aktueller Zustand des Sockets */
  int flags:16;              /* Flags, z.B. für bockierenden/nicht
                                blockierenden Zugriff */
  ...
  struct socket_desc *peer;  /* Socket auf der Gegenseite des Kanals */
  l4semaphore_t       lock;  /* Lock für exklusiven Zugriff */
  buffer_t            buf;   /* Schreibpuffer */
  ...
} socket_desc_t;
```

Wie im Kommentar angegeben ist `peer` darin ein Zeiger auf eine weitere Struktur vom Typ `socket_desc_t`, welche den Socket auf der anderen Seite einer Verbindung repräsentiert. Um zu verstehen, wie das Senden und Empfangen von Nachrichten abläuft, muss nun noch kurz die Bedeutung des `buf`-Feldes vom Typ `buffer_t` erläutert werden:

```
typedef struct buffer {
  char *bytes;
  int   num_bytes, r_start, w_start; /* Schreib-/Lesezeiger und
                                        Datenmenge */
  ...
  l4semaphore_t read_sem;   /* Semaphore, um Lese- und Schreib-Threads */
  l4semaphore_t write_sem;  /* bei leerem bzw. vollem Puffer zu
                               blockieren und ggf. wieder aufzuwecken */
} buffer_t;
```

Für jeden Socket gibt es genau einen Puffer, der per Konvention immer bei Sendeoperationen gefüllt wird. Da ein sende- bzw. empfangsbereiter Socket immer einen Partner hat, ist dessen Schreibpuffer auf indirektem Wege über `peer` ebenfalls erreichbar. Weil in den einer

`socket_desc_t`-Struktur zugeordneten Puffer immer nur geschrieben wird, ergibt sich, dass bei einer Empfangsoperation jeweils vom über `peer` referenzierten Puffer gelesen wird. Damit existiert wie in Abschnitt 5.1 gefordert jeweils ein Puffer sowohl zum Senden als auch für den Empfang.

Die Puffer sind in *local_socks* als Ringpuffer implementiert. Dadurch werden unnötige Kopiervorgänge beim Ablegen und Entnehmen von Daten aus dem Puffer vermieden. Wenn in einen leeren Puffer also 100 Bytes geschrieben wurden, anschließend aber nur 50 Bytes wieder gelesen werden, befinden sich die verbliebenen Daten an Position 50 im Puffer, somit also mitten im Datenbereich. Über getrennte Zeiger für Lese- und Schreiboperationen ist diese kleine Schwierigkeit aber auch bei den folgenden Lese- und Schreibzugriffen auf einen Puffer leicht zu beherrschen. Abbildung 9 verdeutlicht die Funktionsweise noch einmal.

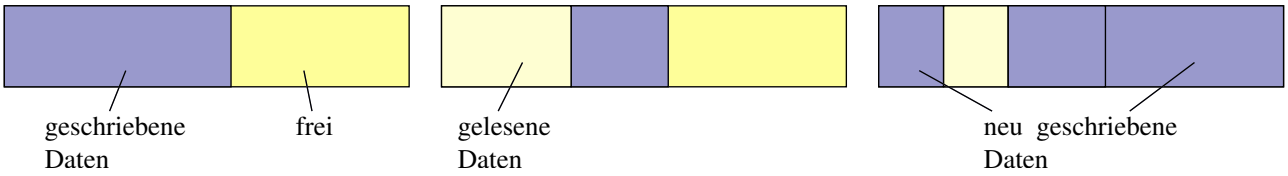

Abbildung 9: Ringpuffer: In den anfänglich leeren Puffer wird geschrieben, ein Teil davon gelesen und anschließend werden weitere Daten geschrieben. Bei Erreichen des Pufferendes wird der nach dem Lesen freigewordene Platz am Anfang genutzt.

In den internen Funktionen `send_internal()` und `recv_internal()` ist die Logik für das Senden und Empfangen von Botschaften über *local_socks* implementiert. `send_internal()` schreibt die vom Client übermittelten Daten in den Schreibpuffer des entsprechenden Sockets. Sollten nicht alle Daten im Puffer Platz finden, so blockiert der Server-Thread, in dem `send_internal()` ausgeführt wird, am Semaphor `write_sem` der zugehörigen `buffer_t`-Struktur. Analog dazu entnimmt `recv_internal()` die vom Client gewünschte Menge an Daten aus dem Schreibpuffer des Sockets, der das andere Ende des Kanals darstellt. Diese Funktion blockiert am Semaphor `read_sem`, wenn nicht genügend Daten im Puffer sind. Beide Funktionen wecken gegebenenfalls die bei der entgegengesetzten Operation blockierten Threads, wenn neue Daten gelesen werden können beziehungsweise wenn wieder Platz im Puffer ist.

In *local_socks* ist vorgesehen, dass alle potenziell blockierenden Operationen (wie etwa `send()`, `recv()` oder `accept()`) in separaten Threads innerhalb des Servers ausgeführt werden. Damit soll erreicht werden, dass auch innerhalb eines Client-Prozesses alle Threads parallel auf (natürlich verschiedene) Sockets zugreifen können, ohne sich zu beeinflussen. Leider kann *L4VFS* dies auf Grund seiner Architektur nicht gewährleisten, da zum einen jeder Socket auf Client-Seite genau einem Server-Thread zugeordnet ist und zum anderen jeder Client-Prozess überhaupt nur diesen einen Server-Thread kennt. Die Lösung dieses Problems besteht darin, das vom IDL-Compiler DICE angebotene Funktionsattribut `allow_reply_only` zu verwenden. Der Thread, welcher die Hauptschleife des Servers für den jeweiligen Client ausführt, nimmt nur noch die Anfrage für alle möglicherweise blockierenden Funktionen entgegen. Die Ausführung obliegt einem separaten Arbeits-Thread. Der ursprüngliche Server-Thread kann dann weitere Anfragen entgegennehmen, auch wenn der Arbeits-Thread zu dieser Zeit blockiert sein sollte.

Leider ist die Verteilung von Aufträgen auf andere Threads recht teuer. Hinzu kommt, dass die Antwort für einen Client vom jeweiligen Arbeits-Thread nach Ende der Ausführung zunächst an den Thread geschickt werden muss, der die Anfrage entgegengenommen hat. Dies ist notwendig, weil einige L4-Kernschnittstellen fordern, dass genau der Server-Thread dem Client antwortet, welcher anfänglich kontaktiert wurde. Die dadurch entstandenen IPC-Kosten stellen einen großen Anteil der Gesamtkosten für die Ausführung einer Operation dar.

Um höhere Performance zu erhalten, kann man dieses Verhalten für die geschwindigkeitskritischen Operationen `send()` und `recv()` auch abstellen. Dies ist sinnvoll, wenn innerhalb der Clients jeweils nur ein Thread auf Sockets zugreift. Sollen in diesem Zusammenhang mehrere Threads erlaubt werden, muss von der Client-Anwendung blockierungsfreies Senden und Empfangen, gegebenenfalls im Zusammenspiel mit `select()`, gefordert werden.

In Abschnitt 7.3 wird die Performance von *local_socks* besonders unter diesen Gesichtspunkten untersucht.

6 Innere Abläufe bei Qt/Embedded-Anwendungen an einem Beispiel

Beim Betrieb mehrerer Qt/Embedded-Programme gleichzeitig muss einer der beteiligten Prozesse zusätzlich die Rolle des QWS-Servers übernehmen. Um die inneren Abläufe in den einzelnen Prozessen und die Kommunikation zwischen Client und Server etwas besser zu verstehen, soll hier ein Beispiel betrachtet werden. In Abschnitt 4.5 wurde bereits in Grundzügen deutlich, dass dabei unter anderem Botschaften über Unix Domain Sockets ausgetauscht werden. Dies soll nun genauer diskutiert werden, indem exemplarisch erläutert wird, wie ein Client beim QWS-Server ein Fenster 'anmeldet'.

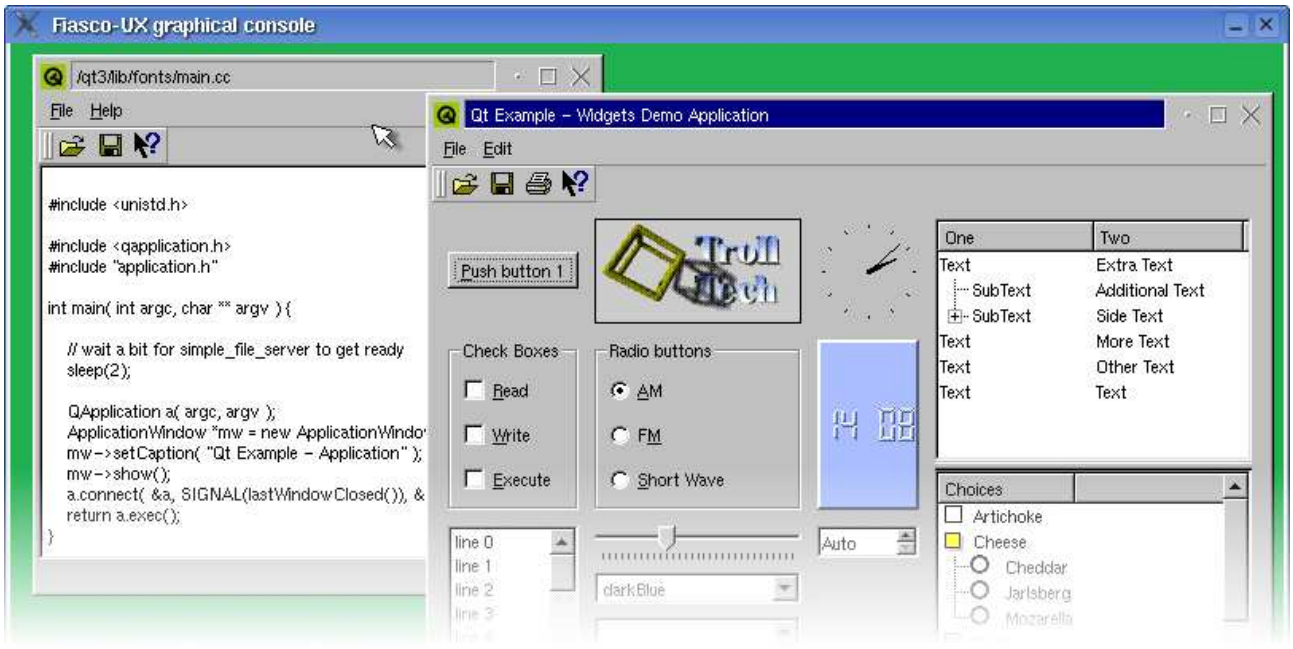

Abbildung 10: Die beiden Qt/Embedded-Anwendungen laufen gemeinsam auf einer Arbeitsfläche, dabei übernimmt ein Prozess zusätzlich die Rolle des QWS-Server.

Hintergrund dieses Vorganges ist, dass der QWS-Server für die Verwaltung aller Fenster auf der Arbeitsfläche zuständig ist. Er muss dafür sorgen, dass die Clients erfahren, welche Teile eines Fensters verdeckt sind und welche nicht. Dies können die Clients nicht dezentral unter sich ausmachen, da sie zunächst lediglich über Informationen bezüglich ihrer eigenen Fenster verfügen. Folglich muss der QWS-Server von jedem Client darüber informiert werden, wo dessen Fenster sich befinden und welche Größe und Form diese haben. Im Gegenzug macht der QWS-Server das Wissen darüber, welche Teile eines Fensters durch andere verdeckt sind, allen Clients über gemeinsamen Speicher zugänglich. Dieser Aspekt soll hier aber nicht betrachtet werden.

Für die Kommunikation zwischen den Clients und dem Server existieren zwei Typen von Botschaften:

Kommandos: Diese werden vom Client zum Server gesendet, um ihn über Zustandsänderungen an den Fenstern des Clients zu informieren, sowie für einige andere Operationen, die vom Client ausgehen.

Ereignisse: Eine Ereignisbotschaft wird vom Server gesendet, um den Client über Maus- und Tastaturereignisse u. ä. zu benachrichtigen. Ereignisse können auch als Antwort auf ein Kommando gesendet werden.

Hier werden im Folgenden nur Kommandobotschaften betrachtet, bei Ereignissen sind die Abläufe aber recht ähnlich.

Wenn ein Client ein neues Fenster erzeugt, muss er sich vom Server zunächst eine eindeutige Fenster-ID zuweisen lassen. Fenster-IDs werden zur Referenzierung der Fenster beim Server verwendet. Damit kann er dem Server anschließend in einer zweiten Kommandonachricht die Geometrie für das zu erzeugende Fenster mitteilen. Wir untersuchen nun, wie diese erste Botschaft vom Client zum Server gelangt.

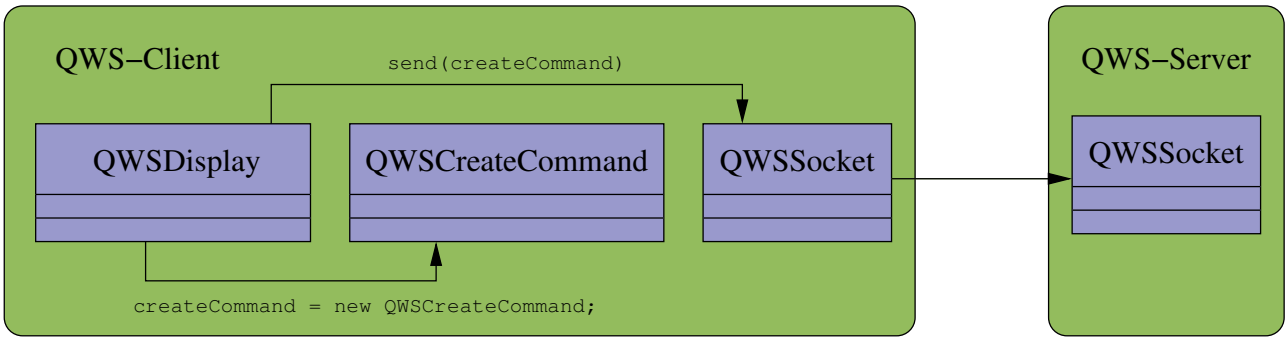

Abbildung 11: Schematische Darstellung zum Senden einer QWS-Kommandobotschaft

Wie in Abbildung 11 zu sehen, muss der Client zunächst ein Objekt für das Kommando erzeugen, welches dann in den Socket geschrieben wird. Dies geschieht innerhalb der Klasse `QWSDisplay`. Der Socket, gekapselt durch eine Instanz der von `QSocket` erbenden Klasse `QWSSocket`, stellt dabei die Verbindung zum QWS-Server dar, die beim Start des Prozesses hergestellt wurde. Im Anschluss an das Senden wartet der Client auf die Antwort des Servers, in welcher die neue Fenster-ID mitgeteilt wird.

Im QWS-Server-Prozess kehrt nun zunächst der `select()`-Aufruf in der Event Loop zurück, ausgelöst durch das Senden des Kommandos durch den Client. Aus `QEventLoop` heraus wird der zum Socket der Client-/Server-Verbindung gehörende `QSocketNotifier` aktiviert. Die Aufgabe der Klasse `QSocketNotifier` besteht darin, für den jeweils zugeordneten Dateideskriptor (in den meisten Fällen ein Socket) bei Empfangs- oder Sendebereitschaft einen Rückruf (callback) durchzuführen. Die Klasse bietet also eine plattformunabhängige Abstraktion der Funktion `select()`. Der genaue Ablauf der Aktivierung eines Socket Notifiers stellt sich wie folgt dar:

1. Rückkehr der Funktion `select()`, welche aus `QEventLoop` heraus aufgerufen wurde.

2. Entfernen der `QSockNot`-Referenz aus der Liste der zu überwachenden Socket Notifier, deren Dateideskriptor von `select()` als bereit gemeldet wurde.

   (Die Klasse `QSockNot` ist eine interne Hilfklasse von `QEventLoop`, sie enthält neben anderen Verwaltungsinformationen auch die Referenz auf das betreffende `QSocketNotifier`-Objekt.)

3. Erzeugung eines Ereignisses vom Typ `QEvent::SockAct` und lokale Zustellung dieses Ergeignisses an den Empfänger, das entsprechende `QSocketNotifier`-Objekt.

4. Emittieren des *Qt-Signals* `activated()` durch das `QSocketNotifier`-Objekt. Damit wird der Rückruf an das Objekt ausgeführt, welches die ursprüngliche Operation auf dem Dateideskriptor durchführen wollte.

Im vorliegenden Fall – dem Empfangen einer Kommandobotschaft von einem Client – wird also die Lesebereitschaft am Socket für die entsprechende Client-/Server-Verbindung signalisiert. Versand und Empfang der Botschaft sind damit abgeschlossen, sobald die Nachricht vom Socket gelesen wurde.

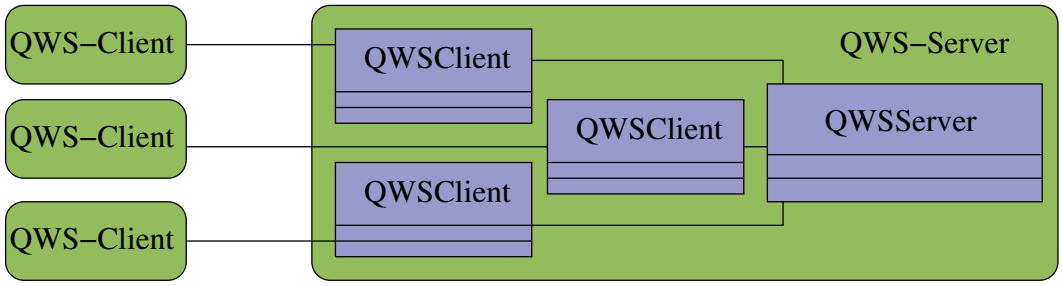

Abbildung 12: Die serverinterne Hilfsklasse *QWSClient* kapselt die Kommunikation des QWS-Servers mit den Clients.

Letzteres geschieht, wenn die Verarbeitung des Kommandos im QWS-Server erfolgt. Verantwortlich dafür sind die Klassen `QWSServer` und `QWSClient`. Wie in Abbildung 12 dargestellt, hält das `QWSServer`-Objekt für jede Verbindung zu einem Client ein Objekt der Klasse `QWSClient`. Dort werden die Client-spezifischen Daten zur Socket-Verbindung gekapselt. Eine jede empfangene Kommandobotschaft wird in `QWSServer` im Kontext des jeweiligen `QWSClient`-Objekts bearbeitet, dazu wird beim Empfang die Methode `doClient()` des `QWSServer`-Objekts aufgerufen.

Bei dem hier betrachteten Beispiel erfolgt die Ausführung des Kommandos, nach Ermitteln des Kommandotyps in `doClient()`, indem `QWSServer::invokeCreate()` aufgerufen wird. Diese Methode tut nun nichts weiter, als eine neue Fenster-ID zu generieren und diese an den Client zurückzusenden. Wie bei der Vorstellung der verschiedenen Typen von QWS-Botschaften auf Seite 32 bereits für den allgemeinen Fall angedeutet, wird dazu eine QWS-Ereignisbotschaft an den Client zurückgesendet.

Das Übermitteln von Ereignissen läuft mit vertauschten Rollen ab. Der QWS-Server sendet, während der Client eine Botschaft empfängt ist. Am prinzipiellen Ablauf ändert sich nichts, nur erfolgt die Aufbereitung der Ereignisse auf Client-Seite durch die Klasse `QWSDisplay`.

7 Leistungsbewertung

7.1 Gegenstand der Leistungsbewertung

Betrachtet man die Portierung von Qt/Embedded auf DROPS, wird schnell klar, dass für die Leistungsbewertung vor allem die Stellen interessant sind, an denen die Anpassungen für DROPS vorgenommen wurden. Dabei handelt es sich im Wesentlichen um das Qt Window System QWS, die neu implementierten Thread-Klassen und natürlich den Socket-Server *local_socks*. Andere Bereiche wurden entweder direkt übernommen oder unter DROPS waren entsprechende Schnittstellen – insbesondere die *dietlibc* – bereits vorhanden.

Der Socket-Server *local_socks* und die für das QWS neu implementierte Klasse `QLock` sollen hier unter dem Aspekt der Geschwindigkeit näher untersucht und bewertet werden. Diese Komponenten werden von Qt/Embedded intensiv genutzt. Neben den entsprechenden Einzeluntersuchungen wird daher auch die damit erreichbare Gesamtperformance der grafischen Benutzeroberfläche betrachtet.

7.2 Messmethoden und Testhardware

Um die Geschwindigkeit einer bestimmten Operation zu messen, wird praktischerweise die für deren Ausführung benötigte Anzahl von Prozessor-Taktzyklen ermittelt, gegebenenfalls auch für einzelne Teiloperationen. Um Messwertschwankungen auszugleichen werden mehrere Messungen vorgenommen und das arithmetische Mittel gebildet. Eventuelle Ausreißer, oftmals die ersten Messungen, werden dabei nicht mit einbezogen.

Der Testrechner für die Messungen verfügt über einen AMD Athlon Prozessor mit 700 MHz auf einem Mainboard mit *AMD Irongate*-Chipsatz (AMD751). Er ist mit 512 Megabyte Arbeitsspeicher ausgestattet, zwei PC100-Speichermodule mit einer Kapazität von je 256 Megabyte. Die Speicherbandbreite ohne Verwendung des Caches beträgt bei dieser Konfiguration 293 Megabyte pro Sekunde[1]. Als Grafikhardware dient eine AGP-Grafikkarte mit *nVidia GeForce2*-Chipsatz.

Die Messungen finden zum einen natürlich unter DROPS statt. Zum anderen ist aber auch interessant, wie die für DROPS neu implementierten Komponenten von Qt sich im Vergleich zur bereits vorhandenen Linux-Implementierung verhalten. Daher wird jeweils auch eine unter Linux 2.4.28 vorgenommene Messung desselben Szenarios gegenüber gestellt. Die Messwerte für Linux aus Abschnitt 7.3.1 wurden noch unter Linux 2.4.27 ermittelt.

7.3 Socket-Server *local_socks*

local_socks lässt sich zur Übersetzungszeit so konfigurieren, dass potenziell blockierende Operationen in separaten Arbeits-Threads ausgeführt werden. Alternativ dazu kann man auch auf diese Möglichkeit verzichten, jede Client-Anfrage wird dann direkt von dem Thread bearbeitet, der sie auch entgegengenommen hat. Diese Implementierungsdetails wurden in Abschnitt 5.4 bereits diskutiert.

[1] Ermittelt mit *memtest86+* Version 1.11

Für die Leistungsbewertung von *local_socks* sind also zwei Fälle zu betrachten:

1. Separate Arbeits-Threads für alle potenziell blockierenden Operationen (zum Beispiel `send()`).

2. Ein Thread pro Client, der alle Operationen sequenziell ausführt.

7.3.1 Geschwindigkeit einzelner Operationen im Detail

Es genügt hier, die Messungen auf eine vom Socket-Server angebotene Funktion zu beschränken, um einen Eindruck von der generellen Performance aller ähnlichen Funktionen zu bekommen. Insbesondere wird ersichtlich, wie sich die Gesamtausführungszeit im Detail auf die einzelnen Teiloperationen verteilt.

Für die Performance-Messungen wurde ein einfaches Testszenario entwickelt, in dem eine Socketverbindung zwischen zwei Prozessen bestand. Einer der Prozesse schickt zehn Nachrichten mit einer Länge von 32 Byte über den Socket. Diese werden von der Gegenseite mit einiger Verzögerung gelesen, so alle Nachrichten zunächst gepuffert werden. Dabei blockiert `send()` nicht, da im Puffer ausreichend freier Speicherplatz vorhanden ist.

Diese Situation ist auch repräsentativ für das Senden von Nachrichten zwischen Qt/Embedded-Prozessen, denn zum Zeitpunkt des Abschickens einer Botschaft wartet der Empfänger entweder mittels der Funktion `select()` in der Event Loop oder es werden gerade andere Operationen ausgeführt, so dass ein sofortiger Empfang nicht möglich ist. In beiden Fällen wird die entsprechende Empfangsoperation erst nach der Pufferung der Nachricht ausgeführt. Dabei ist die Länge der verschickten Nachrichten fast immer unter 100 Byte.

Die Testprogramme für das beschriebene Szenario waren bis auf minimale plattformspezifische Unterschiede für DROPS und Linux identisch. Gemessen wurde hier die Zeit, die zwischen Aufruf und Rückkehr der Funktion `send()` im Senderprozess vergeht, die folgenden Ergebnisse wurden ermittelt:

Messung	Gesamt	IPC I	Koord. I	Ausführung	Koord. II	IPC II
Linux	1638 2.3 μs	-	-	-	-	-
local_socks (ohne Arbeits-Threads)	7133 10.2 μs	3480 5.0 μs	-	1016 1.5 μs	-	2629 3.8 μs
local_socks (mit Arbeits-Threads)	14545 20.1 μs	3942 5.6 μs	1349 + 1340 1.9 + 1.9 μs	858 1.2 μs	3617 5.2 μs	3496 5.0 μs

Die Messung unter Linux dient als Referenz für die Gesamtausführungszeit, das heißt die Zeit zwischen dem Aufruf der Funktion `send()` und deren Rückkehr ins Client-Programm. Es war durchaus zu erwarten dass Linux hier schneller ist als die Implementierung mit dem *L4VFS*-Server unter DROPS, immerhin erfordert der Systemaufruf unter Linux neben der eigentlichen Bearbeitung nur einen Kerneintritt und die Rückkehr in den Nutzermodus. Offenbar findet keine Adressraumumschaltung zum zweiten Client, dem Empfänger, statt.

Die schnellste Variante unter DROPS ist erwartungsgemäß diejenige, bei der *local_socks* die Anfrage nicht in einem separaten Arbeits-Thread bearbeitet. Gemessen wurde hier neben der Gesamtdauer für einen `send()`-Aufruf auch die Zeit, die für die Ausführung der eigentlichen

Operation innerhalb von *local_socks* benötigt wird. Ebenfalls ermittelt wurde die Zeitdauer für das Verschicken der Anfrage an den Server und das Senden der Antwort zurück zum Client.

Gemessen am Gesamtaufwand ist die Bearbeitungszeit der eigentlichen Operation mit 1016 Taktzyklen recht gering. Durch die beiden IPC-Phasen *IPC I* und *IPC II* geht gegenüber Linux letztlich die meiste Zeit verloren. Hier wird die Anfrage des Clients an *local_socks* geschickt und dann natürlich dessen Antwort in umgekehrter Richtung zurückgesendet.

Leider sind diese IPC-Operationen und die dabei auftretenden Adressraumumschaltungen bei der von *L4VFS* vorgegebenen Architektur mit separatem Server-Prozess notwendig. Die hier erreichte Performance kann daher als nahe am Optimum liegend betrachtet werden.

Die Ergebnisse bei Nutzung von separaten Arbeits-Threads in *local_socks* sind etwas schlechter. Aus der Tabelle ist schnell ersichtlich, wo hier die zusätzliche Bearbeitungszeit zu suchen ist, nämlich in den *Koordinationsphasen I* und *II*. In der ersten Phase wurde gemessen, wie lange es dauert, um einen bereits wartenden Arbeits-Thread über die zu erledigende Aufgabe zu instruieren. Dazu wird zunächst eine Datenstruktur mit allen den Auftrag betreffenden Informationen angelegt, ein untätiger Arbeits-Thread gesucht und dieser dann über eine IPC-Operation aufgeweckt. Dabei sind für die IPC-Operation selbst durchschnittlich ca. 1300 Taktzyklen aufzuwenden.

Die zweite Phase dauert etwas länger, obwohl hier nur eine einzige IPC-Operation stattfindet. Dabei wird der Server-Thread, welcher den Auftrag ursprünglich vom Client entgegen genommen hat, darüber informiert, dass er nun eine Antwort an den Client zurückschicken kann. Offenbar ist dieser Thread noch nicht wieder empfangsbereit, wenn der Arbeits-Thread den Abschluss der Operation melden will. Dadurch sind zusätzliche Thread-Umschaltungen und damit etwas mehr Zeit erforderlich.

Interessanterweise geschieht die Ausführung der eigentlichen Operation bei der Variante mit separaten Arbeits-Threads in kürzerer Zeit. Hier werden durchschnittlich nur 858 Takte benötigt, während es im ersten Fall noch 1016 Taktzyklen waren. Dies kann durch Cache- oder TLB-Misses erklärt werden, die bei der ersten Variante zusätzlich zwischen den entsprechenden Messpunkten auftreten. Offenbar finden einige dieser Misses bei der zweiten Variante schon statt, wenn der Arbeits-Threads aufgeweckt wird.

Vermutlich könnte die Performance bei Verwendung von separaten Arbeits-Threads noch gesteigert werden. Dazu müsste der Server-Thread, der die Client-Anfrage entgegennimmt, atomar mit dem Aufwecken des Arbeits-Threads wieder empfangsbereit werden. Leider wird dies vom IDL-Compiler *DICE* in dieser Form nicht unterstützt, eine solche Optimierung ist nur für den Fall von Client-Anfragen vorgesehen.

Mit den nun vorhandenen Möglichkeiten zur Nutzung von *Local IPC* [23] wären eventuell weitere Verbesserungen möglich, entsprechende Messungen konnten aber nicht mehr vorgenommen werden.

7.3.2 Datendurchsatz

Für Qt/Embedded ist der Datendurchsatz beim Übermitteln von Nachrichten über Sockets nicht so entscheidend, da dort wie in Abschnitt 7.3.1 ausgeführt, eher kurze Nachrichten verschickt werden. Das Datenaufkommen ist dabei nicht sehr hoch.

Für andere Zwecke ist der mit *local_socks* erreichbare Durchsatz aber durchaus relevant. Ein typischer Fall dafür wären zwei Prozesse, von denen einer einen größeren Datenstrom an den anderen sendet. Dafür wurde ein Beispielszenario untersucht, in dem 256 Kilobyte Daten in mehreren Blöcken in einen Socket geschrieben und auf der anderen Seite wieder gelesen werden. Die Messungen wurden für verschiedene Blockgrößen von einem Byte bis zu 16 Kilobyte ausgeführt.

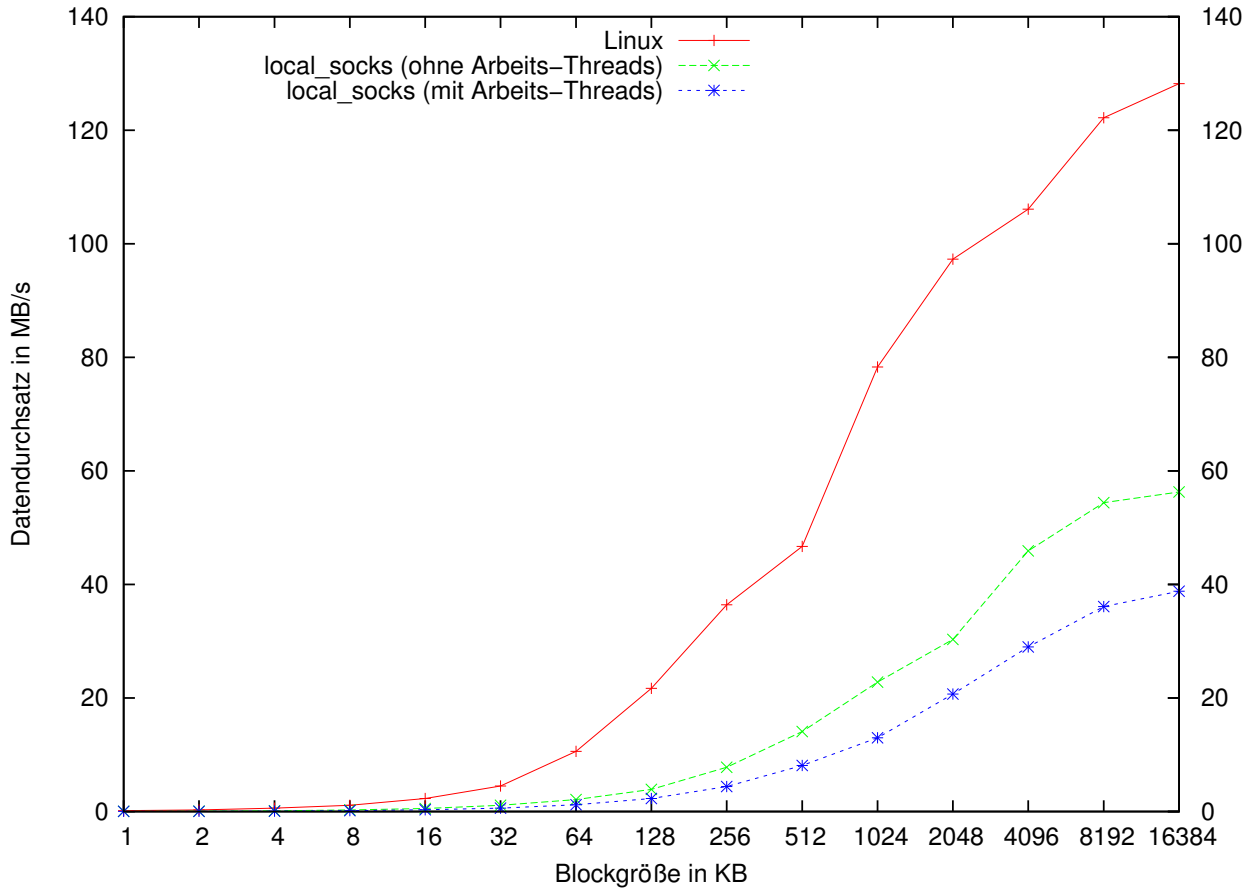

Abbildung 13: Datendurchsatz beim Übertragen von 256 KB Daten über eine Socketverbindung in Abhängigkeit der Blockgröße. Gemessen für Linux und *local_socks* unter DROPS (jeweils mit und ohne separate Arbeits-Threads).

Die dabei ermittelten Durchsatzraten für Linux und DROPS sind in Abbildung 13 zu sehen, jeweils in Abhängigkeit von der Blockgröße. Die Anzahl gesendeter und gelesener Bytes pro Block war auf Sender- und Empfängerseite jeweils gleich.

Die Implementierung für Unix Domain Sockets des Linux-Kernel erreicht die höchste Transferrate. Bei kleinen Blockgrößen ist sie im Rahmen der Erwartungen ca. vier- bis fünfmal schneller als *local_socks* ohne Verwendung separater Arbeits-Threads. Die meiste Zeit wird dort während der IPC-Phasen benötigt. Jedoch fällt *local_socks* mit nur gut halb so hohem Durchsatz bei sehr großen Blöcken recht weit ab. Der von der Nachrichtengröße unabhängige Zeitaufwand für die während der IPC-Operationen auftretenden Thread- und Adressraumumschaltungen

zwischen den Clients und dem Socket-Server macht hier nur noch einen geringen Anteil aus. Dieser Anteil ist deutlich unter 50 Prozent, für die gegenüber Linux halbierte Durchsatzleistung gibt es also auch noch andere Ursachen. Diese sind auch schnell gefunden, wenn man die Abläufe beim Übermitteln einer Nachricht über den *L4VFS*-Server genauer betrachtet. Dieser Vorgang lässt sich grob in die folgenden vier Phasen unterteilen:

1. Kopieren des Datenblockes aus dem Adressraum des sendenden Clients in den Adressraum des Socket-Servers.
2. Kopieren der empfangenen Daten aus dem IPC-Empfangspuffer in den Sendepuffer des entsprechenden Sockets innerhalb des Socket-Servers.
3. Kopieren der Daten aus dem Puffer des Sockets in den IPC-Sendepuffer innerhalb des Socket-Servers.
4. Kopieren des Datenblockes aus dem Adressraum des Socket-Servers in den Adressraum des Clients, der die Empfangsoperation ausführt.

Unter Linux sind die Schritte 1 und 2 sowie 3 und 4 jeweils zusammengefasst. Anstelle von je zwei Kopiervorgängen genügt dort ein einziger. Mit *local_socks* unter DROPS ist also auf Grund der drei beteiligten Prozesse mit ihren getrennten Adressräumen die doppelte Datenmenge zu transportieren. Für die Variante mit separaten Arbeits-Threads müssen die Daten innerhalb des Servers sogar noch ein fünftes Mal kopiert werden, weil der IPC-Empfangspuffer bei späteren Client-Anfragen während der asynchronen Bearbeitung im Arbeits-Thread überschrieben werden kann.

Die zu Grunde liegende Architektur von *L4VFS* lässt hier kaum Spielraum für Optimierungen. Für wesentliche Verbesserungen müsste daher ein gänzlich anderer Ansatz gewählt werden. Dazu könnte man beispielsweise in den jeweiligen Client-Prozessen für jeden Socket einen eigenen Empfangs-Thread einsetzen, der ohne Umweg über den Socket-Server Nachrichten empfängt und puffert. Der Socket-Server wäre dann nur noch für den Verbindungsaufbau notwendig.

Wie in Abschnitt 5.2 bereits diskutiert, hätte eine derartige Lösung einen Verzicht auf das für andere Zwecke benötigte *L4VFS* bedeutet, so dass der Performancenachteil hier in Kauf genommen werden muss. Für den Einsatz mit Qt/Embedded ist dieser Nachteil aber nur wenig relevant.

7.4 Synchronisation zwischen mehreren Prozessen

Zur Synchronisation des Zugriffs auf den gemeinsamen Framebuffer findet die zum Qt Window System gehörende Klasse `QLock` Verwendung, welche ein prozessübergreifendes Lock implementiert. Beim Zeichnen von komplexen Benutzeroberflächen mit vielen Widgets wird dieses Lock sehr häufig angefordert und wieder freigegeben, in einigen Fällen finden mehrere Tausend dieser Anfragen pro Sekunde statt. Das Anfordern und Freigeben des Locks sollte daher so schnell wie möglich erfolgen.

In Abschnitt 4.6.2 wurde erläutert, wie die DROPS-Portierung von `QLock` erfolgt ist. Für die Leistungsbewertung ist nun zu ermitteln, wie lange die Ausführung der Operationen `lock()`

und `unlock()` dauert. Die entsprechenden Zeiten wurden dazu während des Ablaufs einer typischen Qt/Embedded-Anwendung gemessen. Das Testprogramm wurde nicht durch Zeichenoperationen anderer Qt/Embedded-Prozesse beeinflusst. Damit ist der Fall ausgeklammert, in dem `lock()` wegen eines bereits belegten Locks blockiert.

	Linux (QWS-Server)	DROPS (QWS-Server)	DROPS (QWS-Client)
lock()	784 (506) 1.1 μs	2097 (1237) 3.0 μs	3108 (2159) 4.4 μs
unlock()	588 (506) 0.8 μs	1449 (1157) 2.1 μs	2844 (2069) 4.1 μs

Der Tabelle kann man die durchschnittliche Zeitdauer für die Ausführung der untersuchten Operationen entnehmen. Zusätzlich sind in Klammern die kürzesten gemessenen Zeiten angegeben. Für QWS-Clients wurden unter Linux im Rahmen der Messgenauigkeit nahezu die gleichen Zeiten ermittelt wie für den QWS-Server, der Übersicht halber werden hier daher nur letztere betrachtet.

Insgesamt sind die ermittelten Ausführungszeiten sehr breit gestreut, Durchschnittswerte allein sind hier nur bedingt geeignet, um das Zeitverhalten vollständig zu beschreiben. Die Diagramme in den Abbildungen 14 und 15 zeigen daher zusätzlich die Verteilungen der gemessenen Zeiten. Offensichtlich ist die Cache-Ausnutzung nicht sehr gut, in Anbetracht der sehr komplexen Zeichenoperationen zwischen den einzelnen `lock()`-/`unlock()`-Aufrufen ist dies aber nicht verwunderlich.

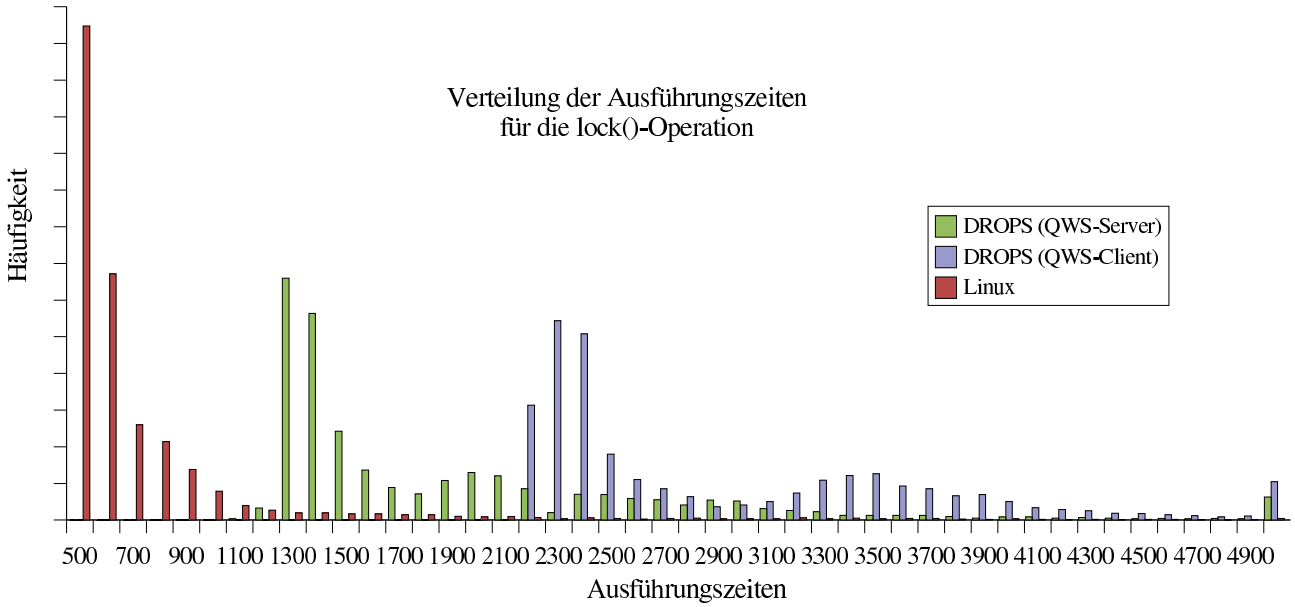

Abbildung 14: Verteilung der gemessenen Ausführungszeiten für das Anfordern des von *QLock* implementierten Locks.

Die Implementierung unter Linux ist hier am schnellsten. Der dort verwendete Systemaufruf `semop()` braucht im Schnitt nur 588 beziehungsweise 784 Taktzyklen. Für die neu implementierte Lösung unter DROPS sind zwei Fälle zu betrachten. Wenn das Lock vom QWS-Server

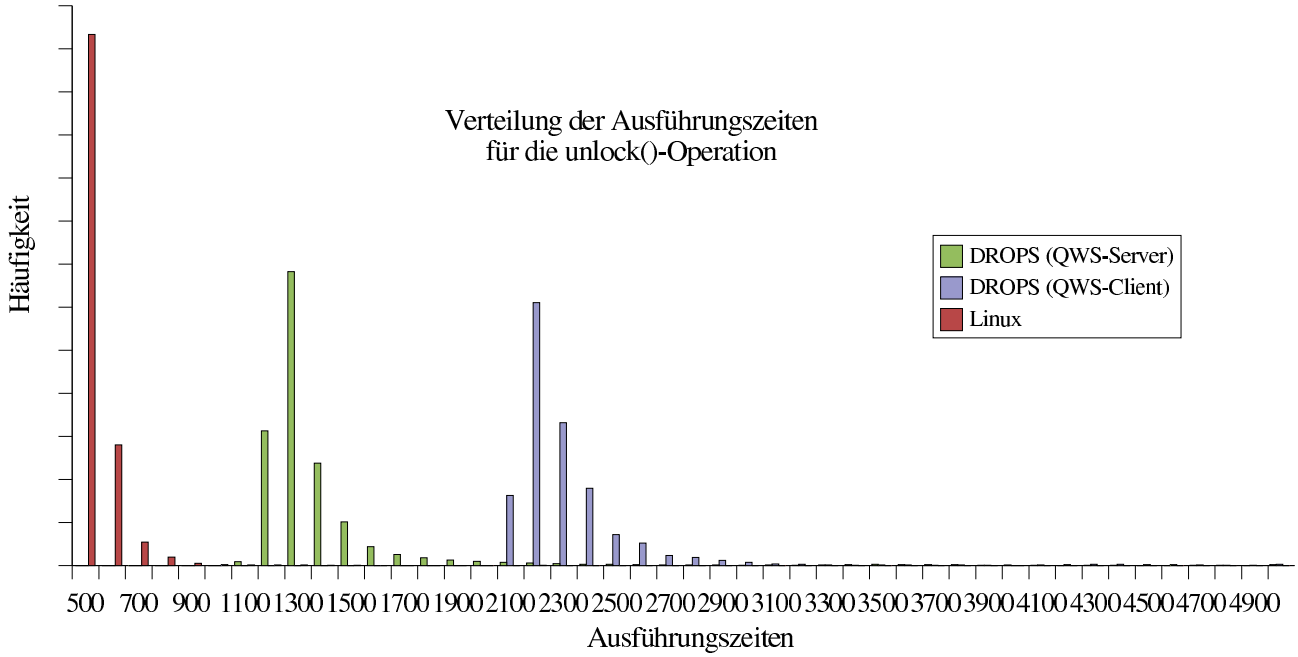

Abbildung 15: Verteilung der gemessenen Ausführungszeiten für das Freigeben des von *QLock* implementierten Locks.

benutzt wird, findet keine Adressraumumschaltung statt, da der Koordinations-Thread im selben Adressraum arbeitet, wie in Abschnitt 4.6.2 erläutert. Im Falle der QWS-Clients hingegen muss bei jeder `lock()`- oder `unlock()`-Operation in den Adressraum des QWS-Servers und wieder zurück geschaltet werden. Damit erklären sich die bei den QWS-Clients höheren Ausführungszeiten.

Die unter DROPS ermittelten Werte sind im Falle des QWS-Servers noch akzeptabel. Für die QWS-Clients schon weniger. Die Ursache dafür liegt bei den durchzuführenden IPC-Operation und den eventuellen Umschaltungen des Adressraumes, die für jedes Anfordern und Freigeben des Locks anfallen. Wenn für jede dieser Operationen immer erst der Koordinations-Threads via IPC kontaktiert werden muss, kann die Performance nicht wesentlich verbessert werden. Es ist allerdings nicht grundsätzlich unmöglich, gemeinsamen Speicher im Zusammenspiel mit atomaren Operationen wie *Compare-and-Swap* einzusetzen. Damit könnten die Fälle, in denen das Lock frei ist und kein anderer Thread suspendiert oder aufgeweckt werden muss, in der Größenordnung von 100 Takten oder weniger behandelt werden. Dies wäre sogar schneller als die Linux-Implementierung von `QLock`. Eine derartige Lösung wurde aber nicht weiter untersucht beziehungsweise implementiert.

7.5 Grafische Benutzeroberfläche

Die in den vorangegangenen Abschnitten durchgeführten Untersuchungen lassen keinen direkten Schluss auf die letztlich erreichbare Geschwindigkeit der grafischen Benutzeroberfläche von Qt/Embedded unter DROPS zu. Schließlich wurden dort nicht sämtliche Randbedingungen für den Einsatz unter DROPS berücksichtigt. Daher soll abschließend die resultierende Gesamt-

performance betrachtet werden. Für diesen Zweck kommt das in der Qt-Distribution enthaltene Beispielprogramm *widgets* zum Einsatz, bei dem die zum Neuzeichnen des Hauptfensters und aller darin enthaltenen Bedienelemente benötigte Zeit gemessen wird. Abbildung 18 auf Seite 44 zeigt dieses Programm.

Qt/Embedded unterstützt unter DROPS keine hardwarebeschleunigte Grafikausgabe, alle Zeichenoperationen werden daher in Software ausgeführt. Abhängig vom verwendeten Grafiktreiber zeichnet Qt/Embedded dabei entweder direkt in den Speicher der Grafikkarte, oder in einen sich im Hauptspeicher befindenden Framebuffer. Im letzteren Fall muss noch ein abschließendes Kopieren der Bilddaten in den Grafikkartenspeicher erfolgen, um die gezeichneten Bildinhalte tatsächlich am Monitor sichtbar zu machen.

Als Referenz für die unter DROPS erreichte Geschwindigkeit dient dasselbe Testprogramm unter Linux. Zur Grafikdarstellung findet dort der für Testzwecke entwickelte *Qt/Embedded Virtual Framebuffer* Verwendung. Dabei handelt es sich um eine X-Window-Anwendung, die über gemeinsamen Speicher den Framebuffer für Qt/Embedded bereitstellt und diesen in einem X11-Fenster zur Anzeige bringt. Weiterhin enthält Qt/Embedded den *LinuxFB*-Grafiktreiber, bei dessen Verwendung direkt in den Grafikspeicher geschrieben wird. Bei diesen Konfigurationen werden unter Linux ebenfalls die in Software implementierten Grafikroutinen von Qt/Embedded benutzt.

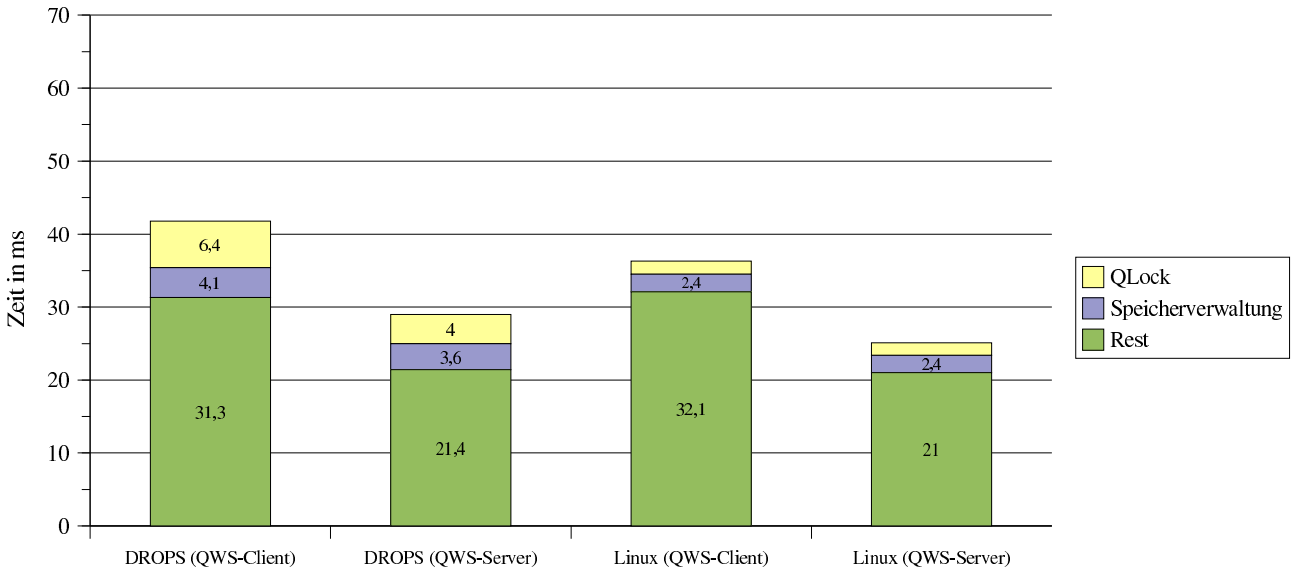

Abbildung 16: Performanceanalyse der grafischen Benutzeroberfläche: QWS-Server und QWS-Client unter DROPS und Linux.

In Abbildung 16 sind die ermittelten Messergebnisse visualisiert. Unter Linux und unter DROPS wurde jeweils die Zeit für den Aufbau des Hauptfensters der Testanwendung gemessen. Als QWS-Server lief das Programm allein ohne Clients. Im Fall des QWS-Clients wurde notwendigerweise noch ein weiterer Qt/Embedded-Prozess ausgeführt, welcher als QWS-Server diente, ansonsten aber untätig war. Die Messungen beziehen sich hier auf das reine Zeichnen durch Qt/Embedded. Der Framebuffer, in den die Daten geschrieben wurden, befand sich dabei im Hauptspeicher.

Insgesamt unterscheiden sich die Messergebnisse unter Linux und unter DROPS nur wenig. Erwartungsgemäß hat die langsamere `QLock`-Implementierung unter DROPS natürlich einen

negativen Einfluss auf die Ausführungszeiten. Als die Messungen vorgenommen wurden, erwies sich aber auch die Freispeicherverwaltung als signifikante Einflussgröße für die Gesamtperformance, da Qt extensiven Gebrauch von den C++-Operatoren `new` und `delete` macht. Die dynamische Verwaltung von Speicherbereichen sorgte anfänglich für einen extremen Performancenachteil von Qt unter DROPS gegenüber der Linux-Variante. Während der Messungen wurden daher verschiedene Implementierungen für die Speicherverwaltung im Rahmen dieses Testszenarios untersucht. Unter DROPS kamen dafür das *simple_mem*-Backend und der *buddy_slab*-Allokator der *dietlibc* zum Einsatz. Unter Linux wurden `new` und `delete` auf die *GLIBC*-Implementationen von `malloc()` und `free()` abgebildet.

	Linux (GLIBC)	DROPS (buddy_slab)	DROPS (simple_mem)
Zeit (Mio. Taktzyklen)	1.7	2.5	19.0
Faktor	1	1.5	11.2

Die Tabelle zeigt die gemessenen Zeiten für die jeweils ca. 8900 Aufrufe von `new` und `delete`, während die Testanwendung das Hauptfenster aufbaute. Die Unterschiede zwischen den beiden Implementierungen unter DROPS sind hier enorm. Für die in Abbildung 16 gemessenen Zeiten kam daher das wesentlich schnellere *buddy_slab*-Backend zum Einsatz. Damit erreicht die DROPS-Portierung von Qt/Embedded nun eine um ca. 13 Prozent geringere Geschwindigkeit als die Linux-Variante. Dies ist akzeptabel.

Es hat sich außerdem gezeigt, dass es generell von entscheidender Bedeutung für die Performance ist, ob Qt als statische oder dynamische Bibliothek in Anwendungen eingebunden wird. Im letzteren Fall wird Qt mit der Compiler-Option *-fPIC* übersetzt, um positionsunabhängigen Maschinencode zu erzeugen. Unter Linux verlängert sich die Gesamtausführungszeit für das hier diskutierte Beispielszenario im Falle des QWS-Servers von 25.1 auf 34.9 Millisekunden. Dies ist eine Verschlechterung um 39 Prozent gegenüber der statisch gebundenen Version der Qt-Bibliothek. Für die eventuelle zukünftige Nutzung einer dynamisch einbindbaren Bibliothek unter DROPS kann von einem ähnlichen Effekt bezüglich der Leistung ausgegangen werden.

Aus Gründen der Vergleichbarkeit wurde bisher nur die Ablage der Bilddaten im Hauptspeicher betrachtet. Noch nicht beantwortet wurde die Frage, wieviel Zeit es kostet, das Bild tatsächlich auf dem Monitor anzuzeigen. Dazu muss natürlich auf den Speicher der Grafikkarte zugegriffen werden. Für die entsprechenden Messungen wurde unter Linux diesmal der *LinuxFB*-Treiber von Qt/Embedded eingesetzt. Unter DROPS kam ein Äquivalent zum Einsatz, mit dem Qt/Embedded ebenfalls direkt in den Grafikkartenspeicher zeichnet.

Die entsprechenden Messergebnisse sind Abbildung 17 zu entnehmen. Im Laufe der Messungen stellte sich heraus, dass Qt/Embedded die Zugriffe auf den Speicher der Grafikhardware unter Linux schneller ausführen konnte. Die Ursache dafür liegt unter anderem in einer sehr nützlichen Funktion des verwendeten Athlon-Prozessors begründet: den *Memory Type Range Registers*. Unter Linux wurden diese so gesetzt, dass der Prozessor die transparent über den PCI/AGP-Bus ausgeführten Schreibzugriffe in den Grafikkartenspeicher optimieren konnte. Dies ist unter DROPS jedoch nicht der Fall.

Die Linux-Messungen wurden daher ohne diesen Vorteil wiederholt, um den Einfluss dieser speziellen Prozessorfunktion einschätzen zu können. Es zeigte sich, dass der Verzicht auf die geeignete Programmierung dieser Register unter Linux eine um ca. 19 Prozent schlechtere Gesamtperformance mit sich bringt. Damit lässt sich der Leistungsunterschied zwischen DROPS und Linux aber nicht vollständig erklären. Diese Diskrepanz konnte leider nicht bis ins letzte

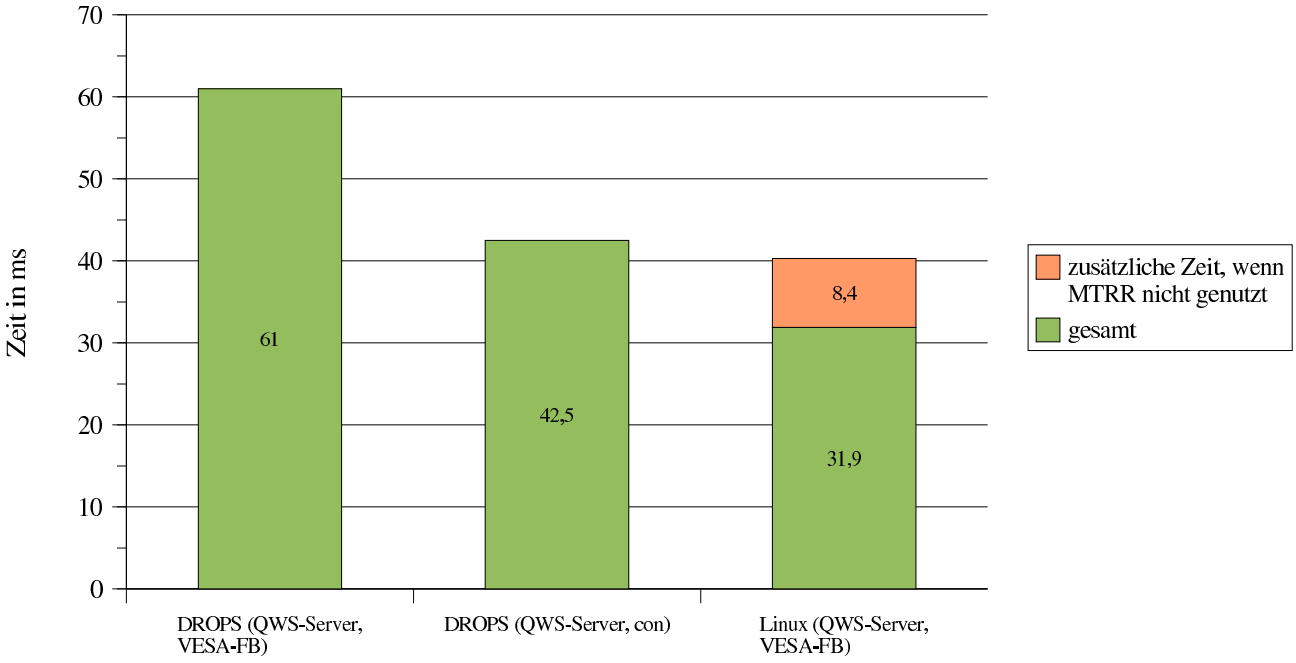

Abbildung 17: Performanceanalyse der grafischen Benutzeroberfläche: Kompletter Zeichenvorgang inklusive des Schreibens in den Hardware-Bildpuffer der Grafikkarte.

Detail untersucht werden, so dass die Ursache für die Leistungsdifferenz hier nicht ganz klar ist.

Unter DROPS wurde zusätzlich der Treiber für die grafische Konsole *con* getestet. Dieser Treiber für Qt/Embedded ist in keiner Weise optimiert, er veranlasst einfach nur in regelmäßigen Abständen, dass der komplette Inhalt des Framebuffers durch den *con*-Server in den Speicher der Grafikkarte kopiert wird. Dies geschieht auch dann, wenn keine Änderungen am Bildschirminhalt erfolgt sind. Die für diesen Treiber gemessenen Zeiten sind ebenfalls dem Diagramm in Abbildung 17 zu entnehmen. Durch die asynchrone Aktualisierung des Framebuffers werden die Ausführungszeiten hier stärker beeinflusst, es kommt zu größeren Schwankungen. Eine Veränderung der Aktualisierungsrate – während der Messungen betrug diese 25 Hz – hätte hier natürlich indirekt Auswirkungen auf die Geschwindigkeit der Zeichenoperationen. Der hier ermittelte durchschnittliche Zeitbedarf ist aber erfreulicherweise geringer als beim direkten Zugriff auf den Grafikkartenspeicher, allerdings ist die Bildwiederholrate durch den *con*-Treiber dabei wie erwähnt auf 25 Hz beschränkt.

Bei den für Qt/Embedded entwickelten Grafiktreibern ist also noch Raum für Optimierungen vorhanden. Der *con*-Treiber sowie der hier nicht betrachtete Grafiktreiber für *DOpE* ließen sich zum Beispiel dahingehend erweitern, dass nur die tatsächlich geänderten Bereiche des Bildes aktualisiert werden. Insgesamt ist die grafische Benutzeroberfläche von Qt/Embedded unter DROPS aber bereits jetzt hinreichend schnell für den praktischen Einsatz.

8 Schlussfolgerungen

8.1 Was wurde erreicht?

Mit Qt/Embedded steht nun eine der Qt-Varianten auch für DROPS zur Verfügung. DROPS-Anwendungen können ihre grafischen Benutzeroberflächen auf dem reichhaltigen Angebot an vorgefertigten Bedienelementen aufbauend anbieten. Dabei ist man nicht gezwungen, alle Details zur Anordnung von Schaltflächen selbst im Quellcode zu formulieren, sondern man kann dabei mit Hilfe eines mächtigen Werkzeuges die Bedienoberfläche in intuitiver Art und Weise grafisch gestalten, der *Qt Designer* macht dies möglich.

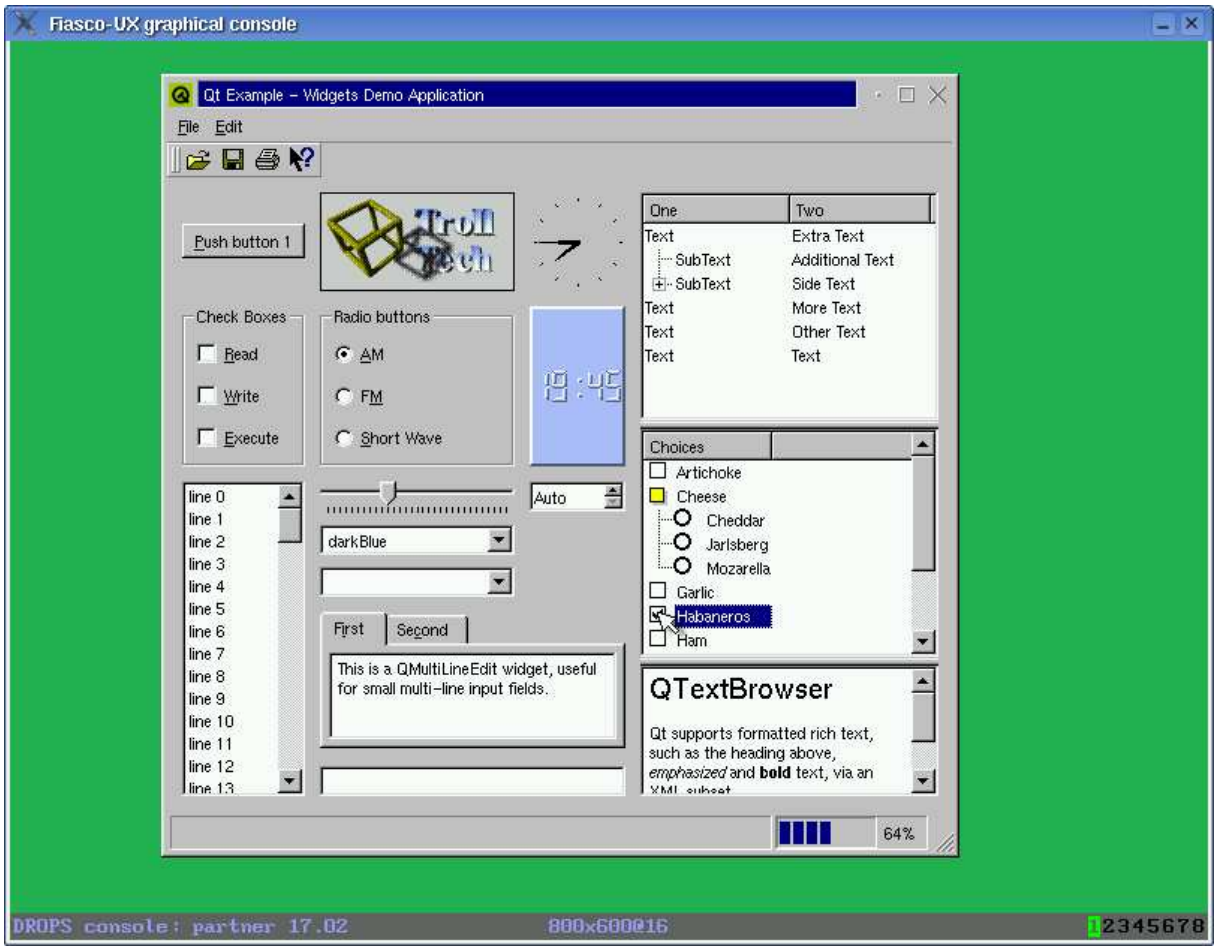

Abbildung 18: Das Beispielprogramm *widgets* für Qt/Embedded, diesmal unter DROPS mit der grafischen Konsole *con*.

Für den Einsatz in DROPS können bei Bedarf neue Bedienelemente programmiert werden. Ebenso kann man Teile der Benutzeroberflächen von Software wiederverwenden, deren Wurzeln bei anderen Betriebssystemen liegen. Die Voraussetzung dafür ist lediglich, dass sie plattformunabhängig mit Qt entwickelt wurden.

Wie im ersten Kapitel zur Motivation bereits verdeutlicht, ist das Qt-Toolkit aber mehr als nur eine Sammlung von Schaltflächen und Eingabefeldern. Vielmehr stellt es eine umfassende

Lösung zur plattformunabhängigen Softwareentwicklung dar, deren Leistungsumfang nun in weiten Teilen auch für DROPS erschlossen wurde. Beim jetzigen Stand ist ein großer Teil der mit Qt gelieferten Beispielprogramme ohne Modifikation unter DROPS einsetzbar. Andere benötigen kleinere Anpassungen, meist weil bestimmte Funktionen in der DROPS-Version der Qt-Bibliothek deaktiviert sind. Die Unterstützung für Drucker sei als Beispiel angeführt. Die wichtigsten Funktionen aber sind unter DROPS nutzbar, etwa die Thread-Unterstützung und der Zugriff auf Netzwerk und Dateisysteme.

Damit ist die Portierung von bestehenden Qt-Anwendungen auf DROPS möglich, gegebenenfalls mit einigen Modifikationen. Der Entwicklung von neuen DROPS-Programmen mit Qt steht natürlich nichts im Wege. Das Toolkit ist dabei modular genug, um auch die Nutzung einzelner Teile zu erlauben. Diese können auch im Zusammenspiel mit nativen DROPS-Komponenten eingesetzt werden, zum Beispiel eine Qt-basierte Benutzeroberfläche als Ergänzung für einen echtzeitfähigen Anwendungskern.

8.2 Welche Probleme sind noch offen?

Trotz aller Fortschritte gibt es aber noch Stellen, die weiterer Arbeit bedürfen. Der große Umfang der Aufgabe und von Qt selbst tritt hier zu Tage. Und einige von Qt angebotenen Funktionen sind in nächster Zukunft vielleicht gar nicht ohne weiteres für Qt unter DROPS umzusetzen. Unter anderem handelt es sich dabei um die folgenden Punkte:

OpenGL-Unterstützung: Qt bietet auch Klassen an, um OpenGL [17] zusammen mit der grafischen Benutzeroberfläche zu verwenden. Die nötige Hardwareunterstützung fehlt leider in DROPS, eine reine Softwarelösung wäre hier aber denkbar. Zum Beispiel durch Portieren des Software-OpenGL-Renderers von Mesa [18] auf DROPS.

Ausgabe auf Druckern: Auch hier fehlen ein paar Grundlagen, vor allem eine generische Schnittstelle und Treiber für eine Vielzahl von Druckertypen.

Starten von Prozessen und Nachladen von Plugins: Qt verfügt auch über Klassen, um zur Laufzeit Plugins in Form von dynamischen Bibliotheken nachzuladen. Ferner können über die Klasse `QProcess` neue Prozesse gestartet werden. Die L4Env-Pakete *loader* und *exec* erscheinen als Basis dafür geeignet, aus Gründen des Aufwands und weil kaum benötigt wurde diese Funktionalität aber nicht portiert.

Die genannten Funktionen stellen einen Teil des Qt-APIs dar, sind aber glücklicherweise nicht wirklich essenziell. In vielen Fällen werden sie überhaupt nicht benötigt. Diese geringfügigen Einschränkungen im Funktionsumfang können daher gegenwärtig toleriert werden.

Einige weitere kleine Probleme sind noch zu lösen. So wird unter DROPS im Moment nur eine statische Version der Qt-Bibliothek erzeugt, die dann direkt in die Binärdatei einer jeden Qt-Anwendung eingebunden wird. Damit wird letztendlich für jeden Prozess eine Kopie der Bibliothek im Arbeitsspeicher gehalten. Die in Abbildung 18 auf Seite 44 zu sehende DROPS-Version der Beispielanwendung *widgets* bringt es auf ca. fünf Megabyte ausführbaren Code. Die Einsparung durch das dynamische Einbinden und gemeinsame Nutzen der Bibliothek durch alle Qt-Prozesse wäre hier enorm. Dieser Punkt wird zusammen mit einigen anderen Modifikationen am Build-System von Qt für DROPS bearbeitet. Außerdem finden außerhalb des von diesem

Beleg umfassten Aufgabengebiets noch Verbesserungen an den Eingabe- und Grafiktreibern für das Qt Window System statt.

8.3 Ausblick

In Abschnitt 3.6 wurde ausführlich erläutert, warum Qt/Embedded in der Version 3.3 als Grundlage für die Portierung von Qt auf DROPS ausgewählt wurde. Das wesentliche Argument war dabei die bevorstehende Freigabe der Version 4.0 von Qt, welche eine neue Architektur für die Anbindung der grafischen Benutzeroberfläche an die darunter liegende Hardware und das Betriebssystem mitbringt. Unter anderem deshalb wurde von einer Anpassung von Qt an *DOpE* zum jetzigen Zeitpunkt abgesehen.

In einer weiteren Arbeit sollte diese neue Architektur genauer untersucht und die Frage geklärt werden, wie man eine engere Einbindung von Qt-Anwendungen in die von *DOpE* bereitgestellte Arbeitsfläche erreichen kann. Zwar ist es jetzt schon möglich, neben *DOpE*-Anwendungen auch Qt-Programme zu benutzen. Diese laufen dann aber auf ihrer eigenen vom QWS-Server verwalteten Arbeitsfläche ab, die ihrerseits wiederum in einem Fenster unter *DOpE* darstellt wird. Diese Einschränkung ließe sich alternativ auch für Qt/Embedded beseitigen, indem Treiber für den *Overlay Window Manager* [24] breitgestellt werden.

Um die durch Qt/Embedded bedingten Sicherheitsdefizite in Folge des gemeinsamen Framebuffer-Zugriffs zu eliminieren, ist aber dennoch eine direkte Anpassung von Qt 4.0 an *DOpE* erforderlich. In diesem Fall würde der letztlich für Qt/Embedded entwickelte *L4VFS*-Server für Unix Domain Sockets dann nicht mehr benötigt, er kann jedoch auch für andere Zwecke genutzt werden. Die übrigen Grundlagen, die mit diesem Beleg geschaffen wurden, dürften sich aber weiterhin als wertvoll für eine Portierung von Qt 4.0 auf DROPS erweisen.

8.4 Zusammenfassung

Ziel dieses Großen Belegs war es, das von dem norwegischen Unternehmen Trolltech entwickelte Toolkit Qt auf das Echtzeitbetriebssystem DROPS zu portieren. Die Motivation dafür bestand zum einen darin, eine leistungsfähige und komfortable grafische Benutzeroberfläche auf DROPS verfügbar zu machen. Damit sollten die Ansprüche von Anwendungen erfüllt werden, die keine Echtzeitanforderungen an ihre Bedienoberfläche stellen.

Zum anderen war es auch ein Ziel dieser Arbeit, die Wiederverwendung von bestehendem Programmcode in größerem Maße als bisher zu ermöglichen. Die Kompatibilität von DROPS zu anderen Betriebssystemen sollte auf eine neue Ebene gehoben werden, um letztlich auch komplette Anwendungen von anderen Systemen auf DROPS portieren zu können.

Die gesteckten Ziele wurden im Wesentlichen erreicht. Das Framework für grafische Benutzeroberflächen ist unter DROPS vollständig funktionsfähig, ebenso auch die Thread-Unterstützung und die APIs für Datei- und Netzwerkzugriff. Es können sowohl die grafische Konsole *con* als auch *DOpE* zur Darstellung der Arbeitsfläche mit den Qt/Embedded-Anwendungen verwendet werden. Der direkte Zugriff auf die Grafikhardware ist ebenfalls möglich. Die Performance der DROPS-Portierung von Qt insgesamt ist dabei auch für komplexere Bedienoberflächen ausreichend. Allerdings gibt es noch Raum für Optimierungen, dies wurde in Kapitel 7 ausführlich diskutiert.

Wie in Abschnitt 8.2 angesprochen, werden einige Funktionen von Qt unter DROPS aber noch nicht unterstützt. Dabei handelt es sich um die Druckeranbindung, OpenGL und die Möglichkeit zum Starten anderer Prozesse sowie das Nachladen von dynamischen Bibliotheken. Die genannte Funktionalität wird aber nur von wenigen Anwendungen benötigt, so dass die Auswirkungen hier sehr gering sind und die Quellcodekompatibilität weitestgehend erhalten bleibt.

Diese Kompatibilität auf Quellcodeebene – einer der wesentlichen Vorteile von Qt – hat auch Einfluss auf die Entwicklung von Anwendungen mit grafischer Benutzeroberfläche unter DROPS. Mit dem *Qt Designer* und der nun gegebenen Möglichkeit, auch unter Linux sofort zu testen, sind Entwurf und Implementierung der Schnittstelle zum Benutzer viel einfacher geworden. In kürzester Zeit kann eine Bedienoberfläche erstellt werden, die sehr hohen Ansprüchen bezüglich Funktionalität und Komfort genügt.

A Dokumentation und Tutorials

Auf den Internetseiten von Trolltech ist umfangreiche Dokumentation zu den Qt-Programmierschnittstellen und Werkzeugen vorhanden. Außerdem finden sich dort zwei Tutorials für den Einstieg in die Qt-Programmierung.

- API-Dokumention der Qt-Bibliothek:

  Alle Versionen: http://doc.trolltech.com/
  Version 3.3: http://doc.trolltech.com/3.3/index.html

- Allgemeine Informationen und Tutorials:

  Version 3.3: http://doc.trolltech.com/3.3/how-to-learn-qt.html
  http://doc.trolltech.com/3.3/tutorial.html
  http://doc.trolltech.com/3.3/tutorial2.html

- Handbücher für alle Werkzeuge zur Softwareentwicklung mit Qt:

  Version 3.3: http://doc.trolltech.com/3.3/tools-list.html

B Hard- und Softwareumgebung

Ziel dieser Belegarbeit ist die Portierung von Qt auf DROPS. Dabei ist die aktuelle Version 3.3 von Qt/Embedded die Grundlage.

Verwendete Qt-Version:

- Qt/Embedded 3.3 im aktuellen Minor-Release 3.3.4.

Qt/Embedded für DROPS stellt die folgenden Anforderungen an die Systemumgebung:

- Betriebssystemkern
  - Fiasco oder Fiasco-UX (mit VESA-Framebuffer)
- L4Env-Dienste und -komponenten:
  - con
  - dietlibc
  - dm_phys
  - dope
  - l4vfs (name_server, simple_file_server, fstab)
  - local_socks
  - log
  - names
  - rtc
  - semaphore
  - thread
- Hardware:
  - x86-kompatibler Rechner
  - 128 MB Arbeitspeicher

Die folgenden Werkzeuge werden zusätzlich zu den Systemanforderungen von DROPS benötigt. Sie müssen auf dem Computer installiert werden, auf dem DROPS kompiliert wird.

- moc - Meta Object Compiler für Qt
- uic - User Interface Compiler für Qt
- Qt Designer (optional)
- Qt Linguist (optional)

Literatur

[1] Technische Universität Dresden
http://www.tu-dresden.de/

[2] Fakultät Informatik, Technische Universität Dresden
http://www.inf.tu-dresden.de/

[3] Trolltech
http://www.trolltech.com/

[4] Qt Toolkit
http://www.trolltech.com/products/qt/

[5] GNU General Public License, Version 2
http://www.fsf.org/licenses/gpl.html

[6] The K Desktop Environment
http://www.kde.org/

[7] GTK+ - The Gimp Toolkit
http://www.gtk.org/

[8] GNOME: The Free Software Desktop Project
http://www.gnome.org/

[9] Opera Web Browser
http://www.opera.com/

[10] DOpE
http://os.inf.tu-dresden.de/dope/

[11] DICE
http://os.inf.tu-dresden.de/DICE/

[12] DROPS - The Dresden Realtime OPerating System Project
http://os.inf.tu-dresden.de/drops/

[13] L4Env
http://os.inf.tu-dresden.de/l4env/

[14] SDL - Simple DirectMedia Layer
http://www.libsdl.org/

[15] Networking Services (XNS), Issue 5.2
http://www.opengroup.org/, http://www.opengroup.org/pubs/catalog/c808.htm

[16] The GNU C Library Reference Manual
http://www.gnu.org/software/libc/manual/

[17] OpenGL - The Industry Standard for High Performance Graphics
http://www.opengl.org/

[18] The Mesa 3D Graphics Library
http://www.mesa3d.org/

[19] Kaffe.org
http://www.kaffe.org/

[20] Port of the Java Virtual Machine Kaffe to DROPS by using L4Env
http://os.inf.tu-dresden.de/papers_ps/boettcher-beleg.pdf

[21] RFC 2367 - PF_KEY Key Management API, Version 2
http://www.faqs.org/rfcs/rfc2367.html

[22] IPSec-Infrastruktur für Mikro-SINA
http://os.inf.tu-dresden.de/papers_ps/syckor-diplom.pdf

[23] Implementierung von Local-IPC auf L4/Fiasco
http://os.inf.tu-dresden.de/papers_ps/reusner-beleg.pdf

[24] Technical Report about Overlay Window Management
http://os.inf.tu-dresden.de/~nf2/files/DOpE/documents/TUD-FI04-02-april2004.pdf