Technische Universität München
Fakultät für Informatik

System-Entwicklungsprojekt

RemoteFinder
Eine graphische Oberfläche für SCP

Max Berger

Aufgabenstelller: Univ-Prof. Bernd Brügge, Ph.D.
Betreuer: Dipl.-Inform. Patrick Renner

Abgabedatum: 27. Mai 2003

System-Entwicklungsprojekt

RemoteFinder
Eine graphische Oberfläche für SCP

Max Berger

31. Mai 2003

Inhaltsverzeichnis

1 Einleitung

In dieser Dokumentation soll das System-Entwicklungs-Projekt Remote-Finder vorgestellt werden. Dabei handelt es sich um Programm, das eine graphische Benutzer-Oberfläche für SCP bietet. Es wurde für das Betriebssystem Mac OS X 10.2 oder neuer entwickelt.

In Unix artigen Umgebungen hat sich die Software SSH als Quasi-Standard für sicheres anmelden auf anderen Rechnern etabliert. Diese arbeitet Zeichen-orientiert. Auf SSH setzt die Erweiterung SCP auf. SCP verwendet das SSH System zur sicheren Übertragung von Dateien.

SCP bietet aber leider wenig Funktionalität. Um Dateien mit SCP kopieren zu können, ist nämlich eine genaue Kenntnis deren Orte notwendig: SCP kann keine Inhalte von Verzeichnissen anzeigen. Also benötigt man im Normalfall immer eine zweite Verbindung, um die Dateien zu finden.

Um diese Probleme zu umgehen, wurde ein neues Protokoll aufgesetzt: SFTP. SFTP arbeitet als so genanntes SSH Subsystem. Der Name SFTP soll andeuten, dass es eine ähnliche Funktionalität wie FTP bietet, aber „Secure“ (sicher) ist. Dieser Name führt oft zu Verwechslungen mit FTPS, welches das tatsächliche FTP Protokoll ist, welches über eine SSL-Verbindung verschlüsselt wird. In Wahrheit haben aber beide Protokolle nahezu nichts gemeinsam. SFTP hat bisher keine so breite Akzeptanz gefunden. SFTP bietet eine gute Grundlage zum Kopieren von Daten, ist aber noch nicht so weit verbreitet wie SCP.

2 Bestehendes System, Ähnliche Projekte

Das mittlerweile nicht mehr ganz so neue Mac OS X setzt auch auf Unix-Derivat der sog. BSD-Familie auf, dass Darwin genannt wird. Viele der traditionellen Unix Kommandos funktionieren wie gewohnt. Unter anderem auch das SSH und das SCP Kommando. Wie unter Unix gewohnt heißt aber nicht, wie von Mac Benutzern gewohnt. Die Bedienung ist für Mac Anwender sehr umständlich: Zu erst muss eine Kommandozeile geöffnet werden. Sind die genauen Dateinamen nicht bekannt, muss erst eine SSH Terminal-Verbindung aufgebaut werden, um die Verzeichnisse zu durchsuchen. Dann kann eine zweite Verbindung aufgebaut werden, die den tatsächlichen Kopiervorgang durchführt. Entsprechen die Dateien auf dem entferntem System nicht einem bestimmten Muster, müssen mehrere Kopierverbindungen hintereinander aufgebaut werden. Diese Kommandos können nicht wie im graphischen Programmen einfach mit der Maus angewählt werden, sondern müssen mit der Tastatur in einer Konsole eingegeben werden.

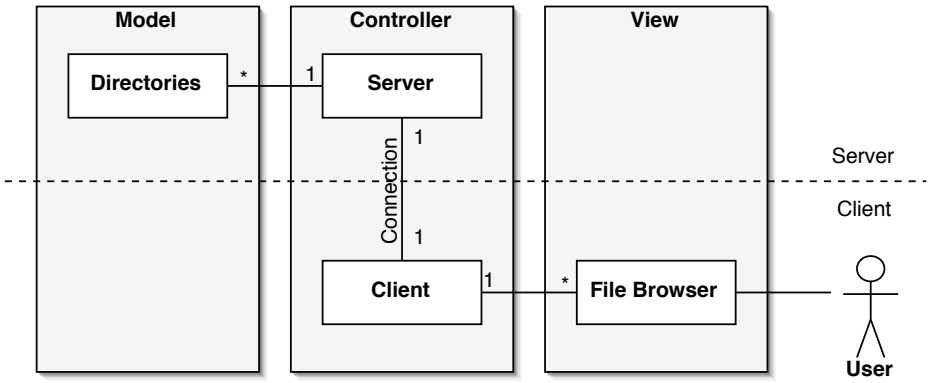

Abbildung 1: Einfache Übersicht über ein System zum Anzeigen von Dateien auf entfernten Rechnern im Model-View-Controller Muster

Was hier fehlt, ist ein Programm, das eine Graphische Benutzer-Oberfläche bietet, die diese Komplexität vor dem Benutzer versteckt, aber dennoch sicheres Kopieren bietet. Das Programm sollte zudem frei verfügbar sein.

Bis Dezember 2002 war mir kein solches Programm bekannt, weshalb ich beschloss, das RemoteFinder Projekt ins Leben zu rufen. Im Laufe der Arbeit daran, erfuhrt ich von dem Fugu Projekt.

Ich werde hier kurz das Fugu Programm in der Version 1.0 vom Mai 2003 vorstellen: Das Fugu Programm bietet eine einfache Benutzer-Oberfläche für SFTP. Es bietet zwar auch SCP Unterstützung, jedoch ist diese sehr umständlich zu bedienen, da man wie beim Kommandozeilen SCP immer wissen muss, um welche Dateinamen es sich handelt. Die SFTP Oberfläche orientiert sich an bekannten FTP Programmen. Sie bietet eine zweigeteilte Verzeichnis-Ansicht mit dem lokalen Dateisystem links und einem entfernten Dateisystem auf der rechten Seite (Siehe Abbildung 2 auf Seite 4). Fugu kann maximal eine Verbindung gleichzeitig offen halten. Will man von mehreren Rechnern Dateien kopieren, muss man die Verbindungen zwischenzeitlich schließen. Außerdem ist es nicht möglich, während eines Kopiervorgangs den Verzeichnisbaum des entfernten Rechners weiter zu durchsuchen und weitere Dateien zur Kopie vorzumerken.

Im Hintergrund setzt Fugu auf die in Mac OS X enthaltenen OpenSSH Programme. Damit erbt es all deren Stärken und Schwächen. Als Stärke sei hierbei hervorzuheben, dass Sicherheitsupdates automatisch mit dem Update des Betriebssystems eingespielt werden. Ein großer Nachteil allerdings ist, dass so ein Update auch eine neue Fugu Version erfordern kann, da sie sich in Verhalten und vor allem Ausgabe-Texten unterscheiden kann.

Fugu ist ein sehr gutes Programm. Es bietet einen wesentlichen Fortschritt

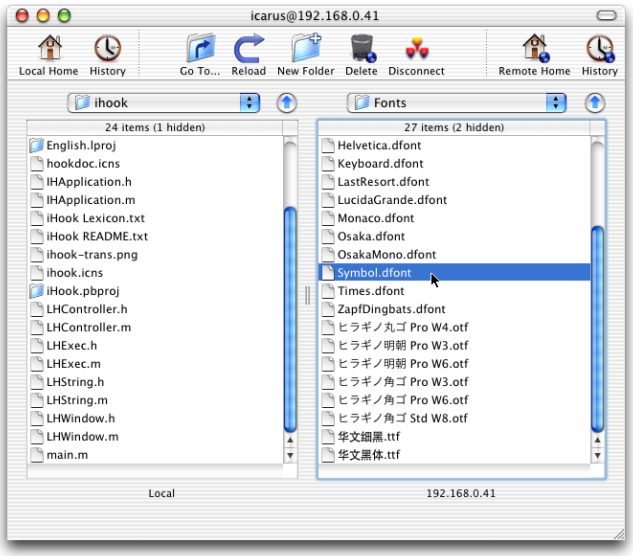

Abbildung 2: Fugu als Vertreter der Programme mit der klassischen zweigeteilten FTP-Ansicht

gegenüber den Kommandozeilen Tools. Leider bietet es noch nicht genug. Welche weiteren Dinge RemoteFinder im Einzelnen bieten soll, werde ich nun in den Anforderungen auflisten.

3 Anforderungen

Ziel des RemoteFinder Projektes war es nun, ein weiteres Programm zu schaffen, dass die Nachteile der oben genannten Programme nicht mehr besitzt. Die Grundidee ist, sich an dem in Mac OS X eingebautem Finder Programm zu orientieren. Finder ist der Standardbrowser für Dateien und Verzeichnisse in Mac OS X und entspricht in etwa dem Explorer unter Windows. Vom Finder sollte nun möglichst viel Verhalten imitiert werden. Nur so kann gewährleistet werden, dass Benutzer die Software nahezu intuitiv bedienen können.

Im Detail ergaben sich folgende funktionale Anforderungen:

- Mehrere Verbindungen gleichzeitig, mehrere Fenster pro Verbindung: Es sollte nicht notwendig sein, eine Fenster oder eine Verbindung zu schließen, um einen anderen Teil des Verzeichnisbaumes oder einen anderen entfernten Rechner anzeigen zu lassen.
- Kopiervorgänge unabhängig vom durchblättern der Verzeichnisse:

Es muss möglich sein, sich weitere Verzeichnisse anzeigen zu lassen, und weitere Kopiervorgänge zu starten, während ein Kopiervorgang läuft.

- Da OpenSSH bereits Session Management und Überprüfung von Maschinen Identitäten bietet, sollten dafür die entsprechenden Konfigurationsdateien von OpenSSH eingelesen oder sogar ausschließlich verwendet werden.
- Dateien sollten sich auf dem gleichen entfernten Rechner via Drag'n'Drop verschieben und kopieren lassen.
- Dateien sollten sich auf dem entfernten Rechner umbenennen lassen.
- Sowohl das SSH 1 als auch das SSH 2 Protokoll müssen unterstützt werden.
- Weitere Protokolle wie z.B. SFTP, FTP, WebDAV sollten unterstützt werden oder sich leicht in den bestehenden Rahmen einbauen lassen.

Des weiteren bildeten die folgenden nicht-funktionalen Anforderungen:

- Für die SSH Funktionen sollte auf eine externe Implementierung zurückgegriffen werden, die sich bereits als „sicher" bewährt hat. Dazu soll eine bestehende Bibliothek verwendet werden.
- Zu Mac OS X passende Oberfläche (Design und Handling): z.B. Drag'n'Drop vom entfernten Fenster in lokale Finder-Fenster oder auf den Desktop.
- Implementierung unter Verwendung des Cocoa-Frameworks. Cocoa stellt sicher, dass auch das Aussehen an Mac OS X angepasst ist.
- Die Anwendung soll alle notwendigen Programmteile enthalten. Sie sollte nicht auf externe Programme, wie z.b. SSH oder SCP zurückgreifen, um unabhängig von einer neueren Version des Betriebssystems zu sein.
- Auf dem entfernten System darf keine extra Software außer dem SSH Server notwendig sein (ein Posix System darf allerdings vorausgesetzt werden: /bin/sh, /bin/ls, /bin/pwd, /bin/echo).
- Die Software soll für Mac OS X 10.2 geschrieben sein.

4 System-Entscheidungen

Um das Projekt zu realisieren, müssen nun auf der Basis dieser Anforderungen weitere Entscheidungen über das System getroffen werden. Da diese nicht zu den ursprünglichen Anforderungen gehören, wurden jeweils verschiedene Möglichkeiten angedacht.

4.1 Programmiersprache

Wie schon in den Anforderungen genannt, soll das System das Cocoa Framework verwenden. Cocoa ist eine Framework, dass einen objektorientierten Zugriff für die Mac OS X Oberfläche zur Verfügung stellt.

Für Cocoa stellen sich als Programmiersprachen Java und Objective C zur Wahl. Java bietet deutliche Vorteile in der Plattformunabhängigkeit, während Objective C wesentlich performanter ist. Da das Programm für eine bestimmte Plattform geschrieben werden soll, ist die Plattformunabhängigkeit nicht mehr wichtig. Deshalb wird Objective C als Implementierungssprache gewählt.

4.2 Oberfläche

Die Oberfläche orientiert sich am Mac OS X eingebautem Finder. So ist gewährleistet, dass das Programm sich „Mac like“ bedient.

4.3 SSH-Backend

Zur Auswahl finden sich im Netz: OpenSSH (http://www.openssh.org), FreSSH (http://www.fressh.org), LSH (http://www.lysator.liu.se/ nisse/lsh/xi) und Putty (http://www.chiark.greenend.org.uk/ sgtatham/putty/).

OpenSSH: Ist klar als Unix Terminal Programm ausgelegt. Der Quellcode ist leider nicht übersichtlich genug. Kann als Startpunkt verwendet werden.

FreSSH: Ist die einzige mir bekannte nicht-Java Implementierung die als Bibliothek vorliegt. Wird leider seit Dezember 01 nicht mehr weiterentwickelt, und hat SSH 2 Unterstützung nur als „TODO“.

LSH: Steht unter der GPL (Damit musste dieses Programm dann auch zwingend unter der GPL stehen) und implementiert bisher nur das SSH 2 Protokoll.

Putty: Bietet im CVS auch einen Unix-Port, der unter MAC OS X ohne Probleme läuft. Das enthaltene Programm „plink“ ist ein sehr gutes Beispiel, wie die Putty-Funktionen zu nutzen sind. Putty ist sehr generisch über Callbacks realisiert (z.b um den Benutzer nach einem Kennwort zu fragen). Die Lizenz ist: „Tue was immer du willst damit“. Putty wird auch in den beiden Windows SCP Programmen WinSCP (http://winscp.vse.cz/eng/) und Secure iXplorer (http://www.itree.org/gpl/ixplorer.htm) als SSH Backend verwendet.

5 System-Entwurf

Dieses System lässt sich besonders einfach mit dem Modell-View-Controller Muster realisieren. Das Modell sind die entsprechenden Verbindungen, die den Datentransfer durchführen. Der Controller ist für dafür Verantwortlich, die entsprechenden Daten (z.B. Verzeichniseinträge) an entsprechende Fenster weiterzugeben und Benutzereingaben in Kommandos um zu formen und an den Server weiterzugeben. Die View ist schließlich das, was der Benutzer direkt sieht und bedienen kann. Damit ergibt sich dann das Modell in Abbildung 3 auf Seite 8. Zur Klarstellung wurden die Klassen aus dem Klassen-Entwurf bereits eingetragen:

5.1 Klassen-Design

Die in Abbildung 3 auf Seite 8 schon vorgestellten Klassen haben folgende Funktionen:

5.1.1 Modell

An einer Verbindung zu einem anderen Rechner sind Instanzen von drei Klassen beteiligt. Diese übernehmen jeweils einen unterschiedlichen Teil der tatsächlichen Verbindung.

Connection Objekte dieser Klasse halten die eigentlichen Verbindungsdaten. Sie verwalten z.B. den Rechnernamen des Servers und die dortige Benutzerkennung. Pro Verbindung existiert eine Instanz dieser Klasse. Diese Klasse ist generisch gehalten, damit der tatsächliche Verbindungstyp variieren kann.

ConnectionProxy Mit Hilfe dieser Klasse wird die Multithreadfähigkeit des Programmes realisiert. Damit das Programm nicht blockiert, wenn Netzwerk-Ereignisse anstehen oder versendet werden müssen, laufen

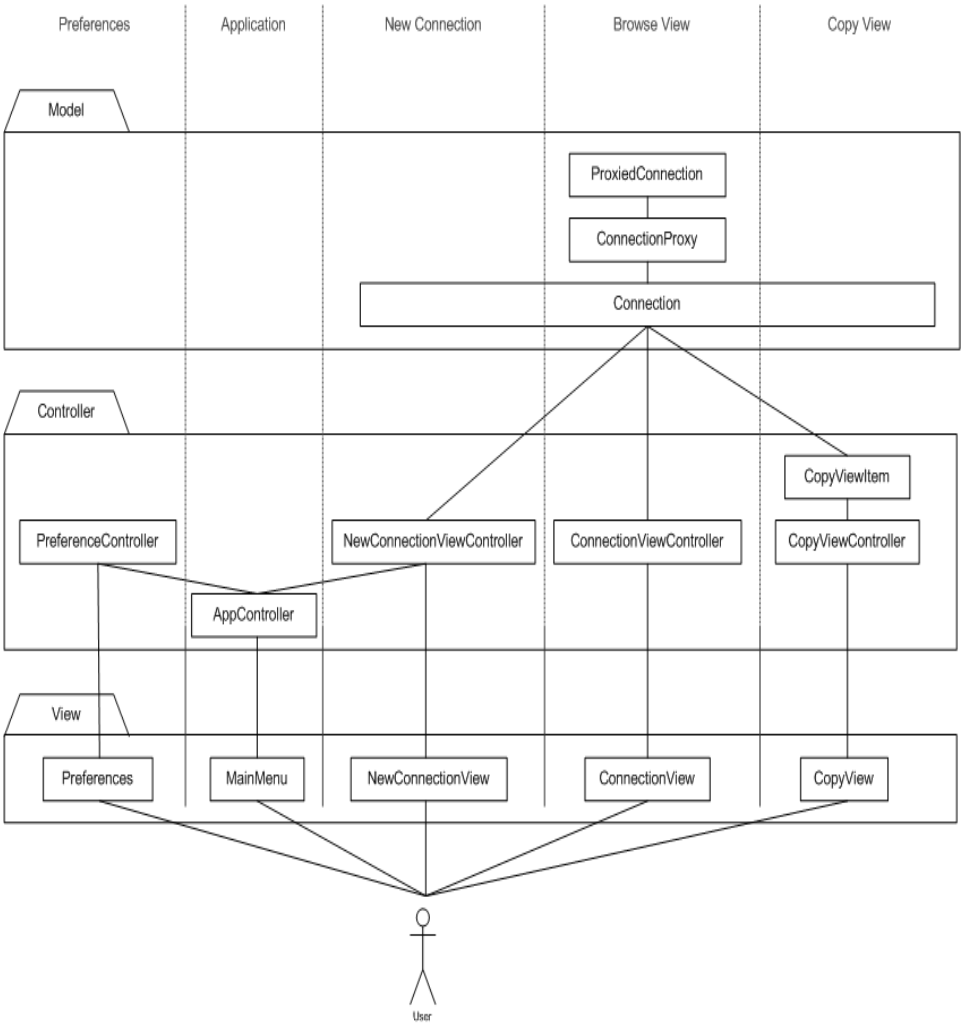

Abbildung 3: Übersicht über die zur Anzeige verwendeten Klassen im RemoteFinder

die tatsächlichen Verbindungen in einem eigenen Thread. ConnectionProxy übernimmt dabei das Interface zwischen den parallelen Threads. Es stellt durch locking-Mechanismen sicher, dass es keine Konflikte zwischen Threads gibt. Diese treten auf, wenn zwei Threads gleichzeitig auf die gleichen Variablen zugreifen wollen. Aufträge werden von einem Thread an eine Queue angehängt, und von ProxiedConnection wieder ausgelesen. Anfragetypen sind z.B. das Auflisten von Verzeichnissen oder das Kopieren von Dateien. An dieser Stelle müssten für die Zukunft noch weitere Dateimanagementfunktionen wie Umbenennen oder Verschieben hinzugefügt werden.

Pro Connection existieren bis zu 3 ConnectionProxies (und damit tatsächliche Verbindungen): ControlConnection zum Anzeigen der Verzeichnisse, und CopyToConnection und CopyFromConnection zum tatsächlichen Kopieren.

ProxiedConnection Damit das Programm verschiedenste Verbindungen handhaben kann, ist ProxiedConnection ein Interface für jede Art von Verbindung.

PLinkConnection Ist eine Beispiel-Implementation einer ProxiedConnection. Liest die Queue des ConnectionProxy und arbeitet diese über eine SSH Verbindung ab.

PLinkConnectionManager Da die Putty Implementation leider nur für single-threaded Applikationen gedacht ist, übernimmt PLinkConnectionManager die Funktion des Putty-Threads. Anstelle eines Threads für jede einzelne Verbindung übernimmt dieser Thread alle Verbindungen und leitet die entsprechenden Ereignisse wieder an die einzelnen PLinkConnection Objekte weiter

5.1.2 Controller

PreferenceController In der Applikation existiert genau ein Präferenzen Fenster. In diesem lassen sich globale Einstellungen festlegen. hier kann z.B. eingestellt werden, ob versteckte Dateien angezeigt oder ausgeblendet werden sollen.

AppController Der AppController ist die Klasse, die eine Cocoa Applikation zusammenhält. Hier finden sich globale Funktionen, wie z.B. Beenden des Programmes.

NewConnectionViewController Mit dem NewConnectionView lassen sich neue Verbindungen zu andern Rechnern aufmachen. Es existiert zwar immer nur genau ein NewConnectionView, dieser lässt sich jedoch beliebig oft wieder öffnen, um weitere Verbindungen aufzubauen.

NewConnectionView ist ebenfalls für ein Session Management verantwortlich. Sessions können hier erstellt, verändert oder gelöscht werden.

ConnectionViewController Jede Verbindung besitzt verschieden viele Verbindungsansichten. Diese entsprechen den einzelnen Finder Fenstern. Sie holen sich ihre Information von der Verbindung und können neue Kopiervorgänge in Auftrag geben. Ein ConnectionView kann Verzeichnisse entweder als Baum oder in der vertikalen Browser-Ansicht darstellen. Außerdem übernimmt es Drag'n'Drop Funktionen zum kopieren von Dateien.

CopyViewController Ein CopyView entspricht den Kopierfenstern beim Finder. Hier wird angezeigt: Welche Datei gerade kopiert wird, die wie vielte von wie viel sie ist, der Gesamt-Fortschritt, der Datei-Fortschritt, etc.

Das CopyView Fenster ist ein Container für die einzelnen CopyView-Item Objekte.

CopyViewItem Ein einzelnes CopyViewItem zeigt die aktuellen Kopierinformationen für genau eine Verbindung an.

5.1.3 View

Die Elemente des Views wurden allesamt im Interface-Builder entworfen. Da es zu jedem Fenster einen entsprechenden Kontroller gibt, werde ich hier nicht weiter auf die einzelnen Elemente eingehen.

5.2 Kontrollfluss

Die statische Spezifikation der Klassen ist noch nicht ausreichend, um das Verhalten des Programmes zu zeigen. Daher soll hier anhand von ein paar Beispielen das Zusammenspiel der verschiedenen Klassen gezeigt werden.

5.2.1 Neue Verbindung

Eine Verbindung starten zu wollen, ist noch relativ einfach. Was danach geschieht, bis die (in diesem Falle SSH-Verbindung) auch läuft, zeigt Abbildung 4 auf Seite 11. In diesem Fall handelt es sich um die allererste Verbindung so dass der PLinkConnectionManager auch noch gestartet werden muss.

Was hier sehr deutlich erkennbar ist, ist das Schichtmodell des RemoteFinders. Dies führt zu dieser Gabel artigen Struktur.

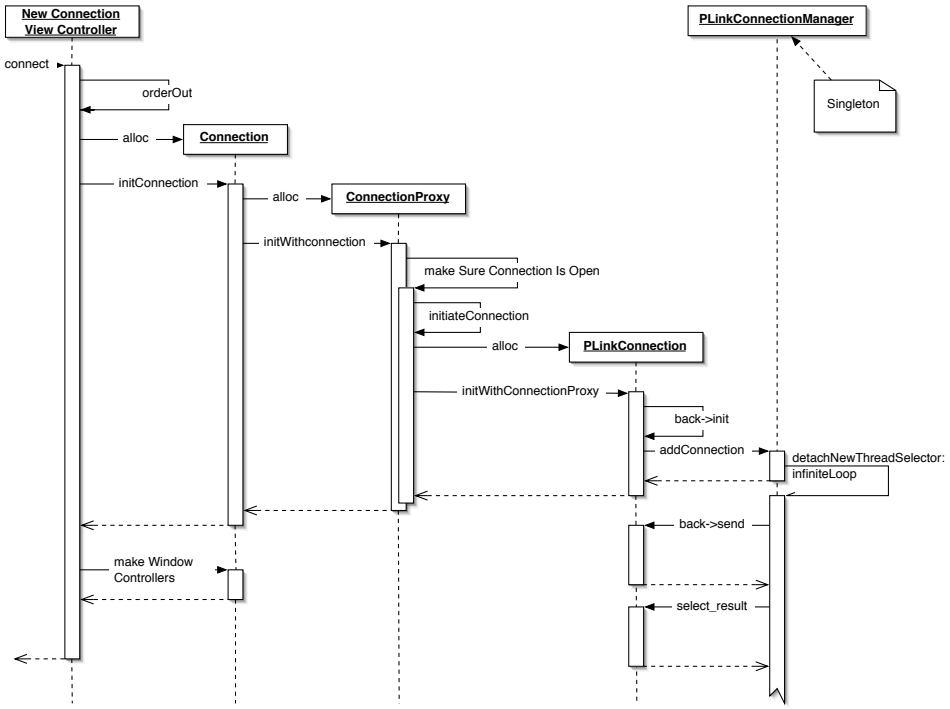

Abbildung 4: Sequenz-Diagramm des Verbindungsaufbaus in RemoteFinder

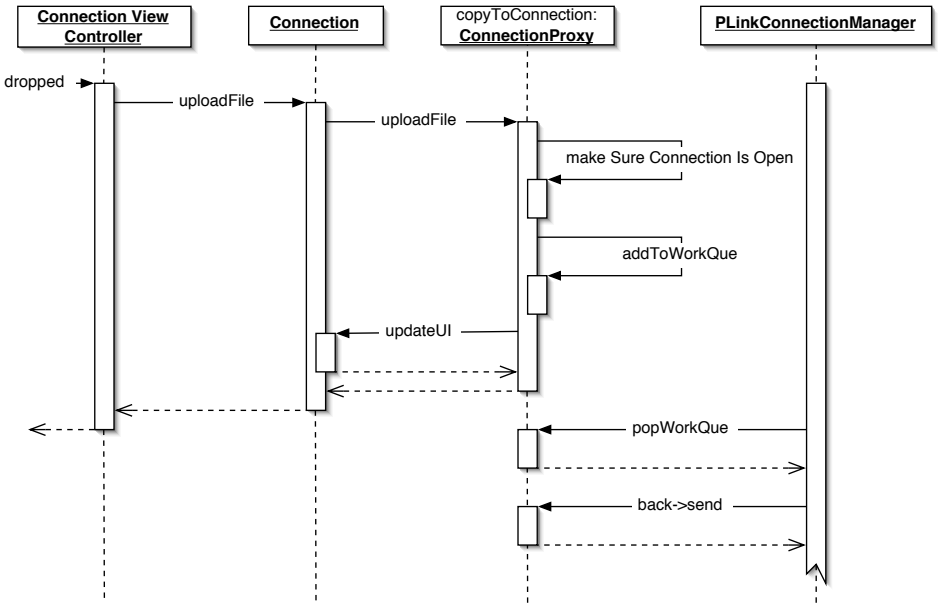

Abbildung 5: Sequenz-Diagramm des Uploads in RemoteFinder als Beispiel für alle Kommandos. Ein Kommando wird in der workQue abgelegt, und dort von der Verbindung ausgelesen. So können weitere Kommandos in die Warteschlange gelegt werden, ohne dass auf deren Abarbeitung gewartet werden muss.

5.2.2 Upload

Als Beispiel für ein Kommando, das auf dem Entfernten Rechner ausgeführt wird, wird hier der Upload gezeigt. Abbildung 5 auf Seite 12 zeigt, wie ein Kommando an die entsprechende Instanz der ConnectionProxy-Klasse weitergeleitet wird, und dann vom PLinkConnectionManager ausgelesen und verarbeitet wird.

Was danach geschieht, lässt sich am einfachsten mit einem Zustandsdiagramm wie in Abbildung 6 auf Seite 13 zeigen, da hier das Klassenzusammenspiel nicht mehr so interessant ist.

6 Implementation

Nachdem nun alle größeren Entscheidungen bereits im System-Entwurf vorweggenommen wurden, gestaltet sich die eigentliche Implementation trivial. Deshalb will ich hier nur einige interessante Aspekte hervorheben.

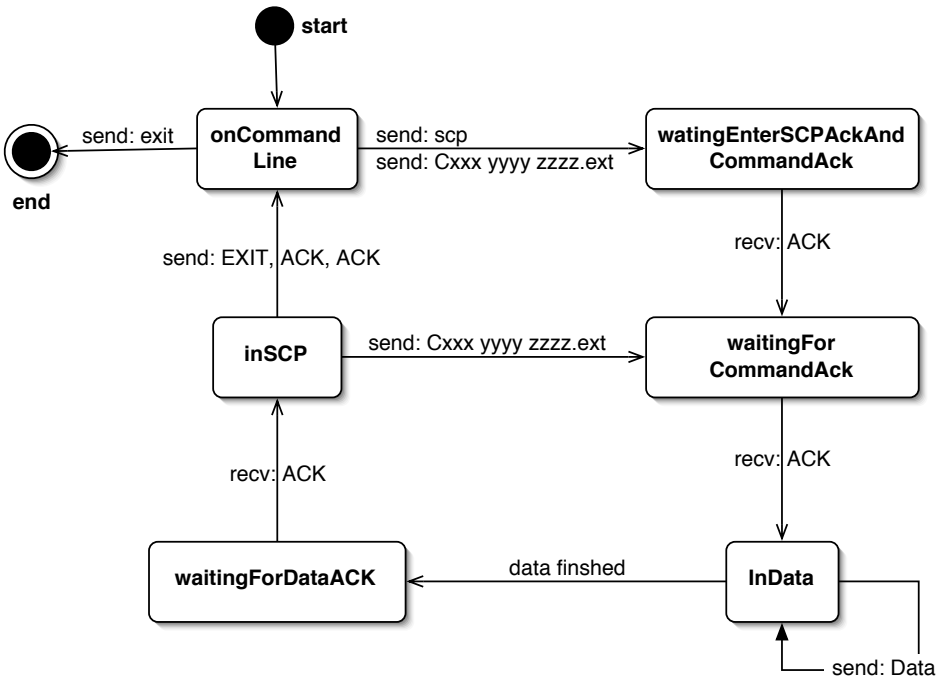

Abbildung 6: Zustands-Diagramm des SCP Protokolls im Falle eines Datei-Uploads. Ein Datei-Download funktioniert ähnlich, allerdings sind dann Sender und Empfänger vertauscht. Leider unterstützen nicht alle SCP Implementierungen die Kante von `inSCP` nach `waitingForCommandAck`, so dass zur Sicherheit immer der Umweg über die Kommandozeile gewählt wird.

6.1 Putty-Kern

Die Idee, als SSH Kernstück Putty zu verwenden, hat sich alles in allem als sehr gute Wahl herausgestellt. Viele Funktionen werden von Putty in einer sehr einfach zu implementierenden Weise bereitgestellt.

Natürlich gibt es auch wie immer eine Kehrseite der Medaille. So übernimmt der Putty Kern völlig selbstständig die Verwaltung von mehreren Verbindungen und deren Sockets. Da aber der ursprüngliche Ansatz vorsah, für jede Verbindung einen eigenen Thread zu benutzen, gab es damit massive Multithreading Probleme.

Zur Lösung wurde zum einen ein „Putty-lock“ eingeführt. Diese schützt den Putty-Kern davor, von mehreren Thread gleichzeitig aufgerufen zu werden.

Zum anderen werden nun wieder alle Putty-Netzwerk-Ereignisse in einem Thread abgearbeitet. Nur so kann gewährleistet werden, dass sich mehrere Threads nicht durch unnötiges locking mehr blockieren, als notwendig.

6.2 File Promises

Es war eine sehr gute Entscheidung, diese Projekt erst jetzt anzufangen, nachdem Mac OS 10.2 verfügbar war. Neu in dieser Version ist nämlich der Zwischenablagen-Typ „File Promise“.

Drag'n'Drop Ereignisse werden in Mac OS X über die Zwischenablage realisiert. Startet ein „Drag“-Vorgang, so werden die entsprechenden Objekte in die Zwischenablage gelegt und wieder heraus geholt, wenn der Vorgang beendet wird. In die Zwischenablage können jegliche Daten gelegt werden. Im Falle von Dateien sind dies meist Dateinamen oder gleich der ganze Datei-Inhalt. Neu an „File-Promises“ ist nun, dass die Datei noch gar nicht auf dem aktuellen System vorhanden sein muss. Es wird der Ziel-Applikation quasi versprochen, dass eine Datei erzeugt (in diesem Falle vom anderen System geholt) wird.

So ideal wie diese „File-Promises“ nun klingen, sind sie leider nicht: Da die Ziel-Applikation die Ursprungs-Applikation auch informieren muss, wo sie die Dateien erzeugen soll, werden File Promises über mehrere, nicht öffentlich spezifizierte, spezielle Typen der Zwischenablage realisiert. Die einzige Möglichkeit, so ein File-Promise zu erzeugen, besteht in Cocoa über die Funktion `dragPromisedFilesOfTypes: fromRect: source: slideBack: event:`. Diese Funktion muss aus einem `mouseDown:` oder `mouseDragged:` Eventhandler heraus aufgerufen werden.

Leider verwenden sowohl die im Programm für die Tabellen-Ansicht benützte „NSOutlineView“ als auch das für die vertikale Ansicht verantwortliche

„NSBrowserView“ eine Kurzschluss-Event-Behandlung: In der Behandlungsfunktion des MouseDown Events wird auf ein MouseUp Ereignis gewartet. Dies hat aber den Effekt, dass die Behandlungsmethoden für die MouseDragged und MouseUp Ereignisse nie aufgerufen werden. MouseDragged ist aber dass Ereignis, aus dem die `dragPromisesFiles:` Funktion aufgerufen werden muss, damit nicht bei jedem einzelnen Klick schon eine Drag'n'Drop Operation gestartet wird.

Zu mindestens für den `NSOutlineView` haben die Entwickler die Problematik des fehlenden `mouseDragged:` Ereignis erkannt. Hier gibt es eine extra Funktion `outlineView:writeItems:toPasteboard:`, die man für Drag'n'Drop Funktionalität verwenden sollte. Dies funktioniert mit regulären Zwischenablage-Typen auch wunderbar. Versucht man jedoch von dieser Funktion aus, File-Promises zu schreiben, so führt dies zu interessantem, aber leider völlig fehlerhaftem Verhalten.

Für das `NSBrowserView` steht eine derartige Funktion gar nicht zur Verfügung, so dass hier schon von vornherein auf Drag'n'Drop Funktionalität verzichtet werden muss.

Die Lösung dieses Problems ist es, eigene `mouseDown:` Funktionalität zu bieten. Dies kann in zwei Stufen realisiert werden:

Die `mouseDown:` Ereignisse werden vollständig im Programm bearbeitet. Die von den Cocoa Klassen bereitgestellte Funktionalität wird nicht mehr benutzt. Nur so kann eine vollständige Unterstützung gewährleistet werden. Allerdings müssen dann auch alle Sonderfälle, wie z.B. shift-klick berücksichtigt werden.

Da gerade die korrekte Behandlung aller Sonderfälle schon ein eigener Projekt für sich ist, habe ich mich für eine vereinfachte Methode entschieden:

- Ist das Objekt, auf das geklickt wurde, nicht selektiert, so verwende die Standard-Behandlungsmethode (mit der "Kurzschluss-Auswertung„)
- Ist das Objekt jedoch nicht markiert, dann ignoriere die Standard-Methode. So werden nachfolgende Ereignisse wie `mouseDragged:` erkannt, und dort kann das File-Promise geschrieben werden.

Diese Methode hat natürlich einige Nachteile: Ist ein Element selektiert, so kann es nicht mehr durch klick auf das selbe deselektiert werden. Um ein Objekt zu verschieben, muss es, anders als im Finder, erst selektiert werden, und kann danach verschoben werden. Dieses Verhalten ist nicht mehr ganz so intuitiv. Das letzte Problem ist schließlich, dass bei dieser Methode auch die Behandlung des Doppel-Klicks sehr schwer ist.

Auf langfristige Sicht werde ich das Programm auf die Selbstverwaltung der Selektionen umstellen. Da dies jedoch nur Bedienkomfort und keine

weitere Funktionalität bietet, wurde dass allerdings auf später verschoben.

7 Diskussion

7.1 Geschwindigkeitsmessung

Eine wichtige Eigenschaft eines Datenübertragungsprogramms ist natürlich die Performance. Kleinere Dateien spielen bei den Übertragungsgeschwindigkeiten heutzutage keine Rolle mehr, aber bei größeren Dateien (ab etwa 100 MByte) werden kleine Unterschiede plötzlich richtig groß.

Bevor man aber über Performance diskutieren kann, muss diese zu erst gemessen werden. Idealerweise kann während einer Datenübertragung zu jedem Zeitpunkt ausgerechnet werden, welches die durchschnittliche Übertragungsrate ist, welches die aktuelle Übertragungsrate ist und wie lange der Kopiervorgang noch dauert.

Dies ist allerdings ein ziemlich frommer Wunsch: Mir ist kaum ein Programm bekannt, welches Rest-Zeiten gut approximiert.

Es wurde zu erst einmal untersucht, welche Daten zugrunde liegen: Jedes mal, wenn ein Paket verschickt (upload) oder empfangen (download) wurde, werden die aktuelle Zeit t_n und die übertragenen Bytes b_n festgestellt. Mit Hilfe der Formel

$$R_n = \frac{b_n - b_{n-1}}{t_n - t_{n-1}} \tag{1}$$

lässt sich daraus die aktuelle Datenübertragungsrate R_n feststellen. Diese Messung ist jedoch Ziemlich ungenau. Die Werte schwanken sehr stark. Deshalb soll sie im Folgenden verbessert werden. Ziel ist hierbei eine vernünftige Anzeige, die die Messschwankungen ausgleicht, aber dennoch auf Änderungen der Übertragungsgeschwindigkeit reagiert.

Der erste Schritt besteht darin, nicht mehr jeden Messwert zu nehmen. Da die Daten-rate um so mehr schwankt, um so schneller die Übertragung läuft, wird nicht mehr jeder Messwert genommen, sondern nur noch dann, wenn seit dem letzten Wert mindestens 0.9 Sekunden vergangen sind. 0.9 Sekunden wurden gewählt, damit der Benutzer jede Sekunde einen neuen Wert sehen kann.

Aber auch das reicht noch nicht aus, um die Messwerte stabil zu halten. Der nächste Versuch war, über einen längeren Zeitraum T zu mitteln, Dies hat aber den Nachteil, dass die letzten Messwerte im Speicher behalten werden müssen. Dies ist zwar dank der Datenstrukturen in ObjC kein Problem, allerdings sind diese ziemlich Rechenzeitaufwendig.

Die jetzt verwendete Lösung orientiert sich an einem Gauß-Tiefpass. Die

Daten-rate wird wie in Formel (1) berechnet und als Eingangs-wert verwendet. Zusätzlich benötigen wir noch einen Dämpfungsparameter ε

$$A_n = A_{n-1} + \varepsilon(R_n - A_{n-1}) \tag{2}$$

oder in der weniger fehleranfälligen Form:

$$A_n = (1 - \varepsilon)A_{n-1} + \varepsilon R_n \tag{3}$$

Welcher Wert wird nun als ε verwendet? Einfacher ist es, sich zu überlegen, über welchen Zeitraum T gemittelt werden soll. Daraus ergibt sich der Parameter τ:

$$\tau = \frac{T}{t_n - t_{n-1}} \tag{4}$$

aus dem sich ε berechnen lässt:

$$\varepsilon = \frac{1}{\tau} \tag{5}$$

Jetzt ließen sich Formel (4) und (5) zusammenführen. Lässt man diese jedoch getrennt, so kann τ dynamisch anpasst werden. Wird das Ergebnis aus Formel (4) nur als Maximalwerte für τ verstanden, so kann τ zu Beginn der Datenübertragung klein gehalten werden, um Messfehler, die durch Übertragungspuffer entstehen, schnell auszugleichen.

Vor der Anzeige wird das Ergebnis noch auf 2 gültige Stellen gerundet, da mehr in den meisten Fällen keine Rolle spielen.

Mit all diesen Berechnungen ist die Geschwindigkeitsmessung von RemoteFinder zwar nicht perfekt, aber viel besser als die der meisten anderen Programme. Da die einzelne Berechnung nicht sehr aufwändig ist, führt die Geschwindigkeitsmessung auch zu keinen nennenswerten Performance Verlusten.

7.2 Performance

In den ersten Versionen hatte das Projekt ziemliche Performance Probleme. Das Kopieren von kleinen Dateien ging immer recht flott, sobald diese jedoch größer wurden, gab es immer mehr oder weniger schlimme Übertragungsrateneinbrüche.

Im Upload musste eine Möglichkeit gefunden werden, immer genug Daten in den Ausgangs-Puffer des Putty-Kerns zu haben. Da dieser durch die Putty-Architektur keine feste Größe hat, könnte theoretisch einfach die ganze zu übertragende Datei in den Puffer geschoben werden. Dies hätte aber zwei große Nachteile: Der Puffer liegt irgendwo im Hauptspeicher, was bei

sehr großen Dateien, wie z.B. 700 MByte CD Images, zu Problemen führt. Zum anderen wäre dann die Berechnung der Upload-Rate nicht mehr so leicht messbar, weil der Ausgangs-Puffer von Putty mitberechnet werden muss. Die Lösung, die ich eingesetzt habe, besteht in einer Feedback Schleife: Für den Upload gibt es eine feste Blockgröße. Es wird dafür gesorgt, dass die Blockgröße immer in etwa der gemessenen Rate in KByte/sek entspricht. Dies braucht zwar etwas Zeit, bis es sich eingependelt hat, bringt dann aber relativ gute Performance Werte.

Die Performance des Downloads war einfacher und schwerer zugleich. Da sich der Putty-Bern darum sorgt, immer alle Daten zu empfangen, musste nur dafür gesorgt werden, dass diese so schnell wie möglich weiterverarbeitet werden. Erste Versuche mit der ObjC Klasse NSString zeigten, dass hier noch deutliche Verbesserungen notwendig waren. Deshalb werden jetzt stattdessen in diesem Teil der Software Low-Level C Funktionen verwendet. Damit waren die Performance-Probleme fürs erste gelöst. Später zeigte sich jedoch, dass die Download-Raten exponentiell abfallen. Dieses seltsame Problem lies vermuten, dass irgendein Puffer nicht geleert wird, und bei jedem ankommendem Paket mehr Arbeit notwendig ist. Nach einigem Suchen stieß ich im Putty-Kern auf die Funktion `unthrottle`. Diese sagt dem Putty-Kern, dass die Daten weiterverarbeitet wurden, und dass der Puffer wiederverwendet werden darf. Der Einsatz der `unthrottle` Funktion hat nochmal eine wesentliche Performance-Steigerung gebracht.

Interessant ist nun der Vergleich von RemoteFinder mit einigen anderen Übertragungsmethoden. Bei einem Netzwerk bis 11 MBit zeigen sich keine wesentliche Unterschiede zwischen den verschiedenen Programmen: RemoteFinder schafft es hier, die Leitungskapazität voll auszunutzen.

Bei einer 100 MBit Leitung wird es schon interessanter. Ich habe die Performance auf einem leerem Netz zwischen einem PowerBook G4 800 MHZ unter Mac OS X (Client) und einem Athlon XP 1300 unter OpenBSD (Server) mit großen Dateien (700 MByte) getestet und verglichen.

Programm	Upload in KB/s	Download in KB/s
FTP	8800	8800
Samba	7500	7500
SCP	5600	5600
Fugu	4600	4000
RemoteFinder	3600	2800

Wie man sieht, liegt RemoteFinder im Vergleich hinten, spielt aber noch in der selben Liga mit. Dies liegt zum einen daran, dass beim RemoteFinder viele Dinge gleichzeitig zum Kopiervorgang stattfinden. Da diese aber einen wesentlichen Komfort-Vorteil bieten, sollte nicht auf diese verzichtet werden. Wie viel Performance alleine eine graphische Oberfläche kostet, kann am Vergleich Fugu/SCP gesehen werden, da Fugu intern den SCP

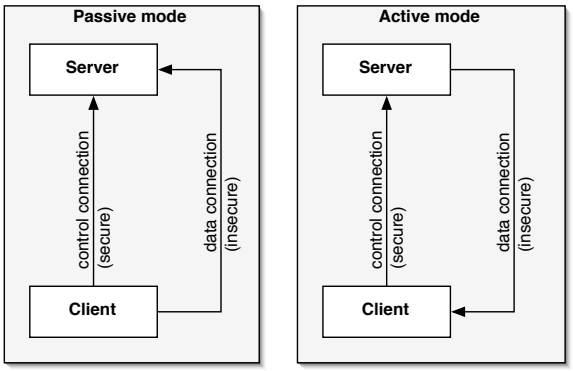

Abbildung 7: Verbindungen im `netcat` Modus. Als Vorbild dienen hier der aktive uns passive Modus von FTP. Im passiven Modus darf der Client hinter einer Firewall stehen, im aktiven Modus darf der Server hinter einer Firewall stehen.

Befehl benutzt. Das zweite große Performance Loch liegt in der Benutzung des Putty-Kerns als eigenständige Bibliothek: Wurde dieser auf die Anwendung hin optimiert, ließe sich hier noch etwas Performance gewinnen.

In Planung ist aber bereits eine weitere wesentliche Steigerung der Performance. Ein Übertragungsmodus ähnlich dem von FTP, wie in Abbildung 7 auf Seite 19 dargestellt: Die Kontrollverbindung ist verschlüsselt und sicher, aber reine Daten werden über eine zweite, unverschlüsselte Verbindung mit Hilfe des Programms `netcat` übertragen. Dies ist immer dann sinnvoll, wenn die eigentlichen Daten nicht sensitiv sind. Des weiteren funktioniert diese Verbindung nur, wenn maximal eine Firewall zwischen den beiden Rechnern steht. Zu guter Letzt muss auch noch das Programm `netcat` auf dem Server-Rechner installiert sein.

Auf den ersten Blick scheint dies alles wie eine große Menge an Einschränkungen. Allerdings liegt in diesem Modus die zu erwartende Datenrate noch über der von `Samba`. Und sollte dieser Modus nicht funktionieren, so kann ohne Probleme auf den verschlüsselten Modus zurückgeschaltet werden.

8 Fehlerbehandlung

Leider funktioniert nicht immer alles so, wie es soll. Gerade Netzwerkapplikationen müssen auf Fehler von außen reagieren. Im folgenden werden erstmal potentielle Fehler aufgelistet, und welche Reaktionen das Programm darauf ausführt.

8.1 SSH Schlüssel falsch / unbekannt

Eine wichtige Eigenschaft des SSH Protokolls ist die Überprüfung des Schlüssels des entfernten Rechners. Ist dieser unbekannt oder hat sich geändert, wird der Benutzer gefragt, wie er weiter verfahren möchte (speichern, abbrechen, . . .)

8.2 Netzwerk-Unterbrechung

Der erste Fehler, der bei einem Netzwerk-Programm einfällt, ist die fehlende / unterbrochene / . . . Netzwerkverbindung. Da das Programm mit mehreren Threads arbeitet, wird das entsprechende Kopierfenster keine neuen Daten mehr anzeigen, und Kopieraufträge zwar entgegennehmen, aber nie mehr abarbeiten. Andere Verbindungsfehler und das Programm an sich sind aber davon nicht betroffen, das Programm lässt sich also z.B. immer noch ordnungsgemäß beenden.

8.3 Fehler im laufendem Kopiervorgang

Treten im laufendem Kopiervorgang Fehler auf, so werden diese in einer Liste gesammelt. Diese Liste wird dem Benutzer vor dem entsprechendem Browser-Fenster angezeigt. Er muss die Fehlermeldungen quittieren, bevor er mit dieser Verbindung weiterarbeiten kann.

8.4 Interne Programmfehler (Segmentation Fault)

Programmierfehler die den berühmten „Segmentation Fault“ hervorrufen, treten mit ObjC wesentlich seltener auf als in anderen Programmiersprachen, da ein „Null“ Zeiger auch gültig ist. Sollte dennoch ein solcher Fehler auftreten, wird das Programm wie jedes andere abstürzen.

A Installation

Die Installation des Programms gestaltet sich dank der sog. „fat Binaries“ auf dem Client ziemlich einfach: Einfach die virtuelle Disketten-Abbildung von einem Server holen. Sie findet sich z.B. auf remotefinder.sf.net. Diese Diskettenabbildung kann durch Doppelklick eingehängt, und dann mit dem Finder geöffnet werden. Das darin enthaltene RemoteFinder Programm kann sofort verwendet werden, oder in den Programme-Ordner (Applications) kopiert werden.

Die Installation der Server-Software ist für die verschiedenen Systeme unterschiedlich. Glücklicherweise bieten die meisten Systeme bereits eine gute eingebaute SSH-Unterstützung, so dass die tatsächliche Installation meist bereits geschehen ist. Hier möchte ich auf die Entsprechende Dokumentation des Systems verweisen.

Literatur

[1] Bernd Brügge, Allen Dutoit, *Object–Oriented Software Engineering*, 1999

[2] Aaron Hillegass, *Cocoa Programming for OS X*, 2002

[3] Various, *OpenSSH manpage*, 1999