WPUT

Ein vollautomatischer Konsolen-FTP-Client

Besondere Lernleistung im Fach Informatik

von
Hagen Fritsch

Ernst-Friedrich-Oberschule
Berlin, 2005

Inhaltsverzeichnis

Teil I - Allgemeines

Teil II – Die Theorie

Teil III – Implementierung

Teil IV – Auswertung

1. Einleitung

1.1. Motivation

Das Konzept von Linux und Open-Source-Software war seit jeher faszinierend. Als Benutzer von Linux ist man immer mit Software konfrontiert, die sich in der Entwicklung befindet. Da man immer wieder auf Bereiche stößt, für die es einfach noch keine auf den Zweck zugeschnittenen Programme gibt, liegt es für erfahrene Benutzer nahe, selbst ein Programm zu schreiben oder sich an der Entwicklung bestehender Projekte zu beteiligen.

Des Weiteren kann jedermann Linux und die dazugehörige Software kostenfrei nutzen. Allerdings fühlt man sich nach einer Weile auch in gewisser Hinsicht verpflichtet, etwas zu bezahlen bzw. der Open Source-Gemeinde etwas zurückzugeben.

Unter Linux gibt es ein sehr bekanntes Konsolenprogramm, das sich `wget` nennt. Dieses Programm ist in der Lage einzelne Dateien oder auch ganze Verzeichnisse von einer Internetseite oder einem FTP-Server zu laden. Nun ist diese Funktionalität allerdings auf den Download beschränkt, also lag es eines Tages nahe, ein entsprechendes Gegenstück – `wput` – zu schreiben, da es zu dem Zeitpunkt neben dem nicht sehr flexiblen `ftp` Programm keine mir bekannten Alternativen gab, die den Zweck eines entsprechenden `wput` erfüllen würden.

Natürlich gibt es bereits eine ganze Reihe von FTP-Programmen, allerdings basieren die meisten auf graphischen Oberflächen und sind deshalb als Konsolenprogramme, die insbesondere für automatisierte Vorgänge (wie z.B. in Shellscripts) eingesetzt werden, nicht geeignet und in Umgebungen ohne graphische Oberfläche (hauptsächlich also in Serversystemen) ohnehin nicht anwendbar.

1.2. Vorüberlegungen

Bevor man sich daran machen kann das Programm selbst zu schreiben, gibt es einige grundsätzliche Dinge, die man besser gleich am Anfang klärt.

Lizenz

Die Entscheidung für eine Lizenz fiel nicht schwer. Wput sollte ein Open-Source-Projekt werden und folglich stand von Anfang an fest, dass Wput unter der GPL Lizenz für jedermann frei nutzbar und veränderbar ist, sofern der Quellcode wieder offen gelegt wird.

Programmiersprache

Wput sollte auch nicht – wie diverse andere Projekte meinerseits – nur schnell dahin geschrieben werden, sondern seinen Zweck möglichst optimal erfüllen und nebenbei einen offiziellen Charakter erhalten, was die Wahl der Programmiersprache stark beeinflusste. Eine Scriptsprache wurde ausgeschlossen, da das Programm möglichst unabhängig von der äußeren Umgebung operieren sollte. Da die meisten größeren Programme auch heute noch in C geschrieben werden, erlitt Wput nun ebenfalls dieses Schicksal.

Betriebssystem

Als vorrangiges Zielbetriebssystem wurde natürlich Linux angesetzt, aber als Konsolenprogramm soll Wput keine großen Ansprüche an das Betriebssystem stellen und somit weitestgehend plattformunabhängig sein.

1.3. Zweck / Aufgabendefinition

Wput ist ein nicht interaktiver FTP-Client, der ausschließlich auf der Konsole operiert. FTP Programme mit graphischen Benutzeroberflächen gibt es genug, aber kleine, robuste Konsolenprogramme waren Mangelware. Wput unterstützt ausschließlich das Hochladen von einer oder mehreren Dateien zu einem oder mehreren FTP-Servern. Dabei hält es sich weitestgehend an die durch das RFC vorgeschriebenen Protokollstandards und unterstützt alle für wichtig erachteten Funktionen des Protokolls. Insbesondere können Uploads halbfertiger Dateien fortgesetzt werden, bestehende Dateien übersprungen werden, nicht existierende Verzeichnisse angelegt werden und die Änderungszeitpunkte und Größen von Dateien verglichen werden, um nur veränderte Dateien hochzuladen. Datenverbindungen können über *Port* oder *Passive Mode* FTP (siehe 3.5. Das FTP-Protokoll / Datenverbindungen) erstellt werden und bei Bedarf über Proxyserver laufen. Der Übertragungsmodus der Datei (Ascii / Binary) kann über die Konsole spezifiziert werden oder wird automatisch anhand der Dateiendung festgestellt. Als besonderes Feature unterstützt Wput die TLS-Verschlüsselung und gehört somit zu den wenigen FTP-Clients, deren Datenverkehr für dritte nicht lesbar ist.

1.4. Abgrenzung der Erklärungen in diesem Dokument

Dieses Dokument soll den Aufbau, die Funktionsweise und Hintergründe von Wput darstellen. Gewisse Grundlagen der Implementierung von C-Funktionen und andere für die spezielle Funktion von Wput nicht essentielle Grundlagen würden den Rahmen dieses Dokuments sprengen und werden deshalb nicht erläutert bzw. nur grob angedeutet.

Im Implementierungsteil ist es aufgrund des großen Umfangs von Wput nicht möglich die Implementation aller Funktionen darzustellen. Hierbei werden lediglich an ausgewählten einzelnen Beispielen Einblicke in die Arbeit verschafft. Der Quellcode der Software selbst wird nicht angefügt, da man mit einer gedruckten Version sowieso recht wenig anfangen kann. Allerdings ist der Quellcode online unter http://itooktheredpill.dyndns.org/wput/lxr/source/src/ abruf- und browsebar[1]. Dieses Dokument bezieht sich auf die Version 0.5.9.

Ein gewisses Grundwissen über das Internet, die Programmiersprache C und Programmierung allgemein, wird vorausgesetzt.

1 Die Webseite bietet nicht nur einen Einblick in den Quellcode, sondern ermöglicht auch eine leichte Navigation in diesem. So kann nach bestimmten Funktionen gesucht und die Stellen der Definition und die der Verwendung dieser angezeigt werden. Des Weiteren sind Funktionsaufrufe, Datentypen sowie Datenstrukturen und viele Konstanten mit Links hinterlegt, die ebenfalls zu einer Zusammenfassung von deren Definition und Verwendung führen.
Als besonderes Schmankerl ist es möglich, zwischen den einzelnen Versionen von Wput zu wechseln und, sofern die Dateien nicht umbenannt wurden, auch die Unterschiede hervorzuheben, wodurch die Entwicklungsgeschichte der Software leichter nachvollziehbar ist.

1.5. Terminologie

Die in diesem Dokument verwendeten Begriffe entstammen dem in der Netzgemeinde üblichen Sprachgebrauch. Die übliche Sprache im Internet ist Englisch, weswegen viele Bezeichnungen, die mit dem Thema zu tun haben, englischstammig sind. Sie wurden teilweise übernommen, teilweise auch übersetzt.

Die Begriffe Server und Client beziehen sich immer (außer im Zusammenhang mit Proxies) auf das FTP-Protokoll. Dabei wird der Begriff Server teilweise für die FTP-Server-Software verwendet, teilweise aber auch für den Rechner auf dem dieses Computerprogramm läuft. Selbiges gilt für den Begriff Client, der einerseits den Computer, auf dem die Clientanwendung läuft, andererseits die Clientanwendung selbst (also z.B. Wput), bezeichnet. Die Begriffe sind dabei weitestgehend synonym zu betrachten: Wenn z.B. der Client etwas an den Server sendet, so ist es einerseits der physikalische Rechner, der die Daten verschickt, gleichzeitig aber die Clientanwendung, die den Clientrechner dazu bewegt, diese Daten zu verschicken.

Der Begriff Verbindung bezeichnet in diesem Dokument immer eine TCP-Datenverbindung.

Wird von einer Bibliotheksfunktion des Betriebssystems gesprochen, so wird diese meist als `funktion (3)` bezeichnet, wobei `3` die Sektion der Manualpage beschreibt, in der die Dokumentation zu dieser Funktion zu finden ist. Wird von `funktion()` gesprochen, so bezieht sich das auf eine Funktion, die von Wput implementiert wird. Diese ist dann online über die Identifier Suche des LXR-Systems[1] einzusehen.

1 http://itooktheredpill.dyndns.org/wput/lxr/ident?v=devel

2. Wput – Grundsätzlicher Aufbau

2.1. Phasenmodell

Im Wesentlichen besteht ein Durchlauf von Wput aus drei Phasen:

- In der ersten Phase versucht Wput sich seiner Umgebung bewusst zu werden, d.h. es prüft das Vorhandensein von Konfigurationsdateien und liest Proxyeinstellungen aus den Umgebungsvariablen heraus.
- Anschließend wird der über die Kommandozeilenparameter spezifizierte Arbeitsauftrag eingelesen. Dieser kann aus einer Anzahl von *flags*, URLs und Dateinamen bestehen. Flags wirken immer global, beziehen sich also auf den gesamten Arbeitsauftrag. URLs und Dateinamen bilden jeweils einen Teil des Arbeitsauftrags.
  Nachdem alle URLs und Dateinamen eingelesen wurden, werden aus ihnen die einzelnen *fsessions* gebaut. Eine *fsession* enthält alle für den Transfer einer Datei notwendigen Informationen (Siehe 4. Strukturprinzipien / fsessions).
- Anschließend ist Wput soweit, dass alle Informationen gesammelt wurden und mit der eigentlichen Übermittlung begonnen werden kann. Dazu wird für jede *fsession* der Übermittlungsprozess gestartet, in dem die Verbindung zum FTP-Server hergestellt wird, die Vorbereitungen für den Datentransfer getroffen werden und die Datei letztlich übertragen wird.

Die tatsächliche Datenübertragung macht dabei den geringsten Teil von Wput aus. Der Kern steckt im Wesentlichen in der Aufbereitung der zu verarbeitenden Daten und der Vorbereitung der Datenübermittlung.

2.2. Dateienstruktur

Die Programmteile von Wput wurden weitestgehend nach Funktionsbereichen in verschiedene Dateien geordnet, deren Namen größtenteils selbsterklärend sind.

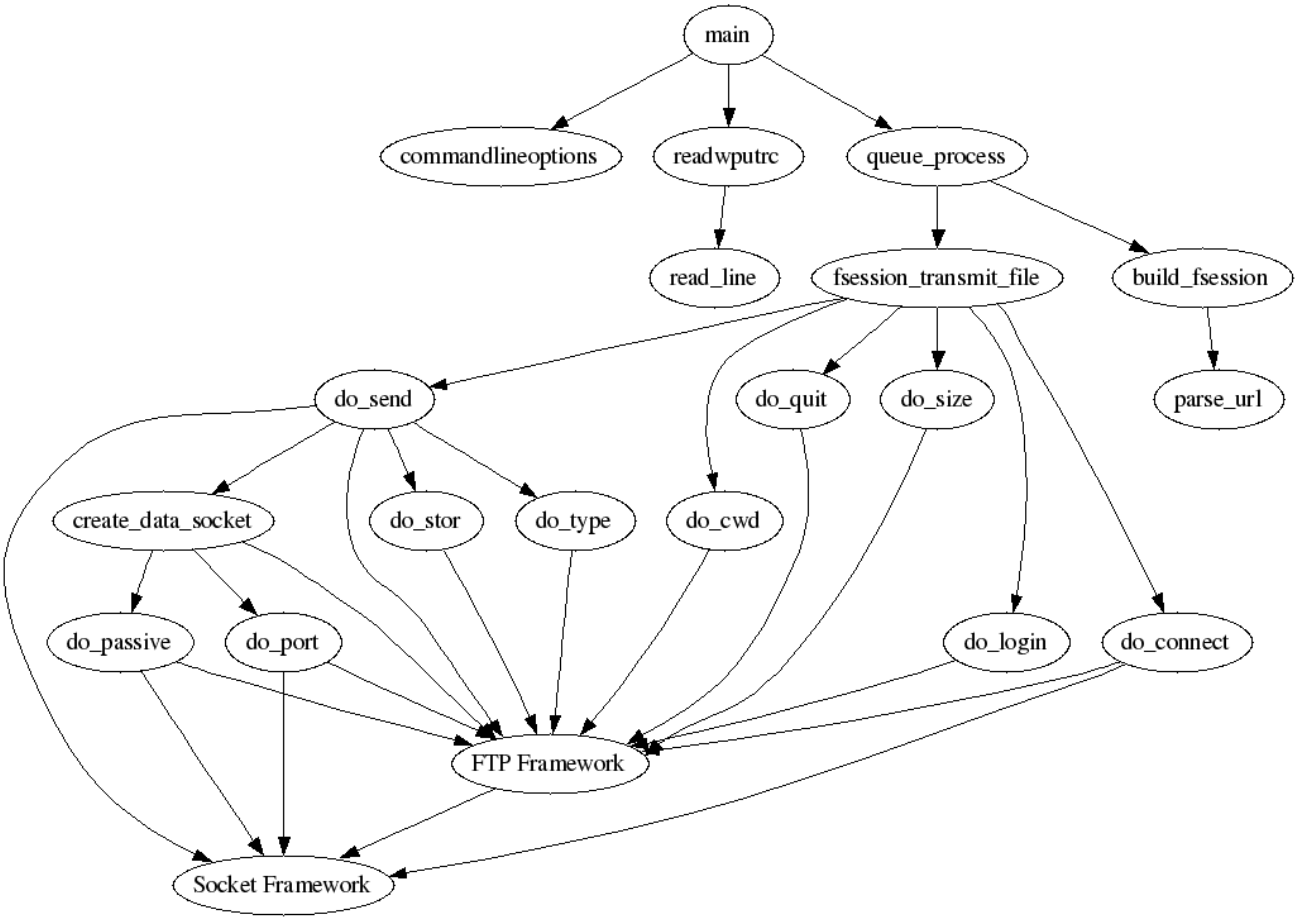
Datei	***Wichtigste Funktionen***
wput.h	Definition von Datenstrukturen, Konstanten und Makros
constants.h	Definition einiger globaler Konstanten
windows.h	Definitionen, die Unix-Systemfunktionen durch Windows-Äquivalenzen ersetzen
wput.c	Hauptprogramm, Einlesen der Konfigurationsdateien und Parsing der Kommandozeilenparameter
queue.c	Verwaltung, Erstellung von *fsessions* Bearbeitung diverser verketteter Listen
socketlib.c	Bibliotheksfunktionen für die Arbeit mit Sockets und Proxies
ftplib.c	Bibliotheksfunktionen für das FTP-Protokoll
ftp.c	Steuerung des FTP-Ablaufs und Datenübertragung, URL-Parsing

Datei	***Wichtigste Funktionen***
utils.c	Sammlung von Hilfsfunktionen (Parsing, Stringfunktionen)
progress.c	Funktionen zur Zeitmessung und zur Anzeige des Fortschrittsbalkens

2.3. Code Hierarchie

Auf unterster Ebene werden eine Reihe von Funktionen implementiert, die die Datenübertragung sicherstellen (*Socket Framework*). Da die Arbeit mit den direkten Systemfunktionen eine ziemliche Qual ist, wurde das *Framework* geschaffen, um einen möglichst einfachen Umgang zu ermöglichen. Ist die grundsätzliche Kommunikation gewährleistet, wird auf der Ebene des FTP-Protokolls ein allgemeine API zur Verfügung gestellt (*FTP Framework*). Diese ist in der Lage ein Kommando zu versenden und ankommende Datenpakete zu empfangen, zusammengehörige zusammenzufassen und vorzubearbeiten, sodass die einzelnen Funktionen mit minimalem Aufwand auf diese Daten zurückgreifen können. Auf der nächsten Ebene wird eine Reihe von Funktionen definiert, die über die verschiedenen für das FTP-Protokoll spezifizierten Kommandos bestimmte Aktionen ausführen. Diese beiden Bibliotheken sind weitgehend vom Rest von Wput unabhängig. Auf der nächsten Ebene steht die zentrale Übermittlungsfunktion `fsession_transmit_file`, die für eine *fsession* je nach Bedarf und Ergebnis einzelne FTP-Aktionen in einer sinnvollen Reihenfolge ausgeführt. Die nächst höhere Ebene ist die globale Steuerung, die aus den Eingabedaten (Konfigurationsdateien, Kommandozeilenparameter) die globalen Optionen einliest und aus den URLs und Dateien einzelne *fsessions* erstellt.

Nicht alles lässt sich direkt einer Ebene zuordnen. Überschneidungen sind unumgänglich.



Detaillierter Aufbau von Wput (stark vereinfacht)

Diese Strukturdarlegungen sollen nur grob den Aufbau Wputs skizzieren, um einen kleinen Überblick über die Konzeption zu erlangen. Ein sehr viel komplexeres (und nahezu vollständiges Funktionsdiagramm) ist im Anhang zu finden.

Im folgenden Kapitel wird das FTP-Protokoll detailliert erläutert werden und später dessen Implementierung und Integration in Wput aufgezeigt.

3. Das FTP-Protokoll

3.1. Allgemeines

Das *File Transfer Protocol* (engl. “Datenübertragungsprotokoll”, kurz FTP) ist ein aus dem Jahre 1985 stammendes Netzwerkprotokoll zur Dateiübertragung, das trotz gewisser Mängel heute noch das für diesen Zweck am meisten genutzte Protkoll ist.

Das FTP-Protokoll ist ein sogenanntes *human-readable* - also für Menschen lesbares - Client-Server Protokoll, das theoretisch mit einem einfachen *telnet* Programm bedienbar wäre.
Als Client sendet man ein Kommando an den FTP-Server und erhält eine Antwort. Diese besteht einerseits aus einem dreistelligen Zifferncode, der für die einfache Interpretation durch Programme gedacht ist und andererseits aus einem beschreibenden Text, der für den menschlichen Benutzer gedacht ist. Das sieht dann z.B. so aus:

```
[ 1] Server: 220 ProFTPD 1.2.10 Server (FoolBird FTPd)
[ 2] Client: USER donald
[ 3] Server: 331 Password required for donald.
[ 4] Client: PASS entenbrust
[ 5] Server: 230 User ftp logged in.
[ 6] Client: PWD
[ 7] Server: 257 "/" is current directory.
[ 8] Client: CWD /upload
[ 9] Server: 250 CWD command successful.
[10] Client: PASV
[11] Server: 227 Entering Passive Mode (192,168,0,1,162,207).
[12] Client: STOR example.txt
[13] Server: 150 Opening BINARY mode data connection for example.txt
[14] Server: 226 Transfer complete.
[15] Client: QUIT
[16] Server: 221 Goodbye.
```

Diese Kommunikation findet auf der Kontrollverbindung statt, die nach der Spezifikation auf Port 21 abrufbereit ist. Der grundsätzliche Ablauf einer solchen *Session* besteht aus dem Login-Prozess (2-5), dem Wechseln des Verzeichnisses sowie einiger anderer “Vorbereitungen” (7-8), dem Erstellen einer Datenverbindung (10-11), der Übertragung der Daten (12-14) und dem Beenden der Verbindung (15-16).

In diesem Kapitel werden die wichtigsten bzw. die für Wput relevanten FTP-Befehle erläutert, um einen Einblick in die Funktionsweise von Dateiübertragungen zu geben.

3.2. Die Fehlercodes

Der eben dargestellte Ablauf erfolgt leider nicht immer so reibungslos wie in dem Beispiel, denn es können eine Reihe von Fehlern, wie z.B. ein falsches Kennwort, die Nichtexistenz eines Verzeichnisses, fehlende Zugriffsrechte, Probleme mit der Datenverbindung etc., auftreten.
Der dreistellige Fehlercode gibt dabei darüber Auskunft, ob ein Befehl erfolgreich war oder ob es Probleme gab. Diese Fehlercodes lassen sich in Klassen einteilen, die auf der ersten Ziffer basieren.

100er	Vorläufige Meldungen, die keine Fehler sind, aber implizieren, dass noch eine weitere Meldung kommt, sobald der Vorgang abgeschlossen ist. Das gilt z.B. für Dateiübertragungen. Die Verbindung wird eingeleitet und durch eine 100er Meldung bestätigt (13). Wenn die Datei übertragen wurde, folgt ohne weitere Interaktion des Clients die Folgenachricht, die in der Regel den erfolgreichen Erhalt der Datei bestätigt (14).
200er	Erfolgsmeldungen, die bestätigen, dass der Befehl wie gewünscht ausgeführt wurde und kein Fehler aufgetreten ist.
300er	Zwischenzeitliche Erfolgsmeldungen, die bestätigen, dass soweit alles in Ordnung ist, aber weitere Informationen benötigt werden, um den Vorgang abzuschließen. Diese Kategorie trifft z.B. auf den Login-Prozess zu, bei dem der Client über den USER-Befehl den Benutzernamen sendet, der Server ihn aber darauf hinweist, dass auch noch ein Kennwort benötigt wird, welches dann nachgeliefert wird.
400er	Vorübergehende Fehlermeldungen, die besagen, dass der Befehl nicht ausgeführt wurde, es sich aber um einen temporären Fehler handeln könnte, der u.U. beim nächsten Versuch nicht mehr existiert. Diese Meldungen können in verschiedenen Situationen auftreten z.B. wenn nicht mehr genug Platz auf der Festplatte ist oder die Datei, auf die zugegriffen werden soll, gerade von einem anderen Programm verwendet wird.
500er	Permanente Fehlermeldungen, die auf Syntax- oder Protokollfehler hinweisen oder andere permanente Probleme bezeichnen, wie z.B. das Fehlen von Zugriffsrechten auf eine Datei oder das Überlaufen des Quota-Limits. Syntaxfehler sind z.B. für den Server unbekannte Kommandos oder Kommandos ohne Argument, die eigentlich ein Argument erwarten würden oder ähnliches. Protokollfehler sind dagegen vom Client nicht eingehaltene Abfolgen der Kommandos. So darf der Client erst das Verzeichnis wechseln oder Dateien hochladen, wenn er sich angemeldet hat.

Im RFC[1] sind für jedes Kommando die Fehlercodes festgelegt, die der Client zu erwarten hat. Allerdings kommt es schon mal vor, dass der eine oder andere Server sich nicht exakt an die Vorgaben hält, weswegen eine Grobbestimmung der Fehler anhand der Klassen im Allgemeinen ein wenig flexibler ist.

3.3. Der Anmeldevorgang

Nachdem der Server mit seiner ersten Meldung (1) seine Anwesenheit und seine grundsätzliche Bereitschaft zur Kommunikation signalisiert hat, ist es Aufgabe des Clients, sich anzumelden. Entweder hat der Benutzer einen Account auf dem Server und benutzt also dessen Benutzername und Kennwort oder er loggt sich als anonymer Benutzer ein, der dann i.d.R. eingeschränkte Zugriffsrechte hat und mitunter nicht einmal Dateien hochladen kann. Der anonyme Benutzer heißt immer `anonymous`.

1 RFCs (Requests for Comments) sind definierte Standards, die z.B. Kommunikationsabläufe wie das FTP-Protokoll genau spezifizieren. Das FTP-Protokoll ist im RFC 959 festgehalten.

Über den `USER`-Befehl wird der Benutzername gesendet. In wenigen Fällen ist das schon genug und wird durch die positive Antwort `User logged in.` bestätigt. Meistens wird man allerdings freundlich dazu aufgefordert, doch sein Kennwort zu übermitteln (`User name okay, need password.`). Dieses wird über den `PASS`-Befehl gesendet und dann entweder vom Server bestätigt oder mit einem `Login incorrect.` abgewiesen.

Neben der `USER`/`PASS` Methode gibt es noch den Account (`ACCT`) Befehl, der allerdings heutzutage kaum mehr Verwendung findet und daher auch von Wput nicht implementiert wird.

3.4. Die Vorbereitung

Nach der Anmeldung stehen dem Benutzer prinzipiell eine Menge Möglichkeiten offen. Essentiell wichtig ist nur das Wechseln des Verzeichnisses, wobei nicht einmal das heutzutage zwingend notwendig ist. Theoretisch könnte der Client gleich mit der Datenübertragung anfangen, was aber meist nicht im Interesse des Benutzers ist.

Es gilt also das Verzeichnis zu wechseln, zu prüfen, ob eine Datei bereits vorhanden ist, dabei eventuell Dateigröße oder -datum zu vergleichen, den Transfermodus einzustellen etc.

Verzeichniswechsel

Das Wechseln des Verzeichnisses geschieht über das Change Working Directory (`CWD`) Kommando, dem nur der Name des Zielverzeichnisses übergeben wird. Das können auch relative Angaben sein, sodass der Befehl ziemlich genau dem `cd` Befehl entspricht, der von Konsolenshells schon bekannt sein dürfte.

Ursprünglich war einmal angedacht, dass für jeden Bestandteil des Pfades ein extra `CWD`-Befehl ausgeführt wird. Für das Verzeichnis `/pub/doc/` sähe das dann etwa so aus:

```
CWD pub
CWD doc
```

Das macht zwar durchaus Sinn, ist aber langsam (siehe 3.7. Kritik am FTP-Protokoll) und kaum ein Client geht so vor, weswegen dabei meistens das gesamte Zielverzeichnis übergeben wird:

```
CWD pub/doc
```

Existiert ein Verzeichnis noch nicht, so bleibt zumindest im Anwendungsgebiet von Wput die Möglichkeit das Verzeichnis einfach zu erstellen, was analog zum `CWD`-Kommando mit dem Make Directory (`MKD`) Kommando möglich ist:

```
CWD pub
MKD doc
CWD doc
```

Datum / Uhrzeit und Dateigröße

Für bestimmte Features von FTP-Clients wie *Resuming*, also das Wiederaufnehmen abgebrochener Datentransfers, oder *Timestamping*, also den Vergleich der Dateidaten, ist es unabdingbar, einiges über die Datei herauszufinden, wobei das ursprüngliche FTP-Protokoll dabei nur eine recht unangenehme Möglichkeit bietet, diese Informationen zu erhalten. Bei dieser wird vom Client über das `LIST`-Kommando ein Verzeichnislisting angefordert, das je nach Betriebssystem ein etwas

anderes Format haben kann, in dem die benötigten Informationen stehen.

Da man eingesehen hat, dass diese Variante doch etwas unpraktisch ist, wurden einige Erweiterungen[1] für das FTP-Protokoll vorgeschlagen, die heutzutage recht verbreitet umgesetzt werden und den Clients die Arbeit erheblich erleichtern. So kann über das `SIZE`-Kommando die Größe einer Datei abgefragt werden und über das Modification Time (`MDTM`) Kommando der Zeitpunkt der letzten Änderung erhalten werden:

```
SIZE example.txt
213 302
MDTM example.txt
213 20041026153642
```

Somit ist nun bekannt, dass die Datei 302 Bytes groß ist. Die `20041026153642` entspricht dem Format `JJJJMMTTSSMMSS` also Jahr, Monat, Tag, Stunde, Minute, Sekunde. `Example.txt` wurde also am 26. Oktober 2004 um 15:36:42 Uhr das letzte Mal geändert, wobei die Zeitangabe UTC ist, also GMT+0 entspricht, was auch notwendig ist, denn im Internet weiß man eigentlich nur selten wirklich, in welcher Zeitzone ein Rechner steht und folglich wäre ein Vergleich der Zeiten ohne dem Wissen um die Zeitzone meist überflüssig.

Übertragungsmodus

Im Wesentlichen haben sich zwei Übertragungsmodi etabliert. Zum einen der *Ascii-Mode* Filetransfer und zum anderen der *Binary-Mode* Filetransfer. Es gibt noch andere Modi, die aber fast ausschließlich historische Hintergründe haben und deshalb nicht von Belang sind.

Der *Binary-Mode* Filetransfer ist der Einfachste. Bei diesem werden die Daten Byte für Byte übertragen und am anderen Ende genauso wie sie eintreffen in einer Datei gespeichert.

Beim *Ascii-Mode* Filetransfer geht man davon aus, dass man Textdateien zu transferieren hat, bei denen Übersetzungsarbeit zu leisten ist. Der Grund hierfür liegt in verschiedenen Eigenheiten der Systeme. So werden z.B. unter Windows Zeilenumbrüche in Textdateien durch die Bytekombination `0x0D`, `0x0A` (`\r\n`) dargestellt. Das ist historisch bedingt den Schreibmaschinen zuzuschreiben, die ein *Carriage Return* und anschließend erst den richtigen Zeilenumbruch brauchten. Auf unixoiden Betriebssystemen reicht allerdings das Byte `0x0A` (`\n`) als Zeilenumbruch aus. Heutzutage ist dies die wichtigste zu erfüllende Umsetzung bei der Übertragung von Daten in *Ascii-Mode*. Allerdings bedeutet *Ascii-Mode* eigentlich noch etwas mehr, nämlich die Konvertierung der Repräsentation von Textdateien auf dem eigenen System (z.B. von anderen *Charsets*) in das standardisierte `NVT-ASCII` (auch *Telnetascii* genannt). Nach der Übertragung werden die Textdaten auf der jeweils anderen Seite wieder in die hauseigene Repräsentation überführt.

Die Festlegung der Übertragungsart erfolgt über das `TYPE`-Kommando:

```
TYPE I
200 Type set to I
```

`I` (Image) steht dabei für Binary, `A` wäre Ascii. Ascii-Mode ist bei FTP-Servern die Voreinstellung. Übertragungen von Verzeichnislisten erfolgen grundsätzlich in *Ascii-Mode*.

1 Diese Erweiterungem sind bisher kein offizieller Standard sondern nur ein Entwurf der Internet Engineering Taskforce (IETF), der unter der Bezeichnung draft-ietf-ftpext-mlst-16 bekannt ist.

Das Betriebssystem

Um mit Verzeichnislisten etwas anfangen zu können, ist es wichtig, das Betriebssystem des Servers zu kennen. Dafür gibt es das System (`SYST`) Kommando:

```
SYST
215 UNIX Type: L8
```

Das erste Wort der Antwort dieses Befehls gibt das System an. Neben `UNIX` gibt es z.B. `VMS`, `MACOS`, `WINDOWS_NT`, `OS/400` und etliche weitere. Eine Reihe von Servern ist auch nicht ganz ehrlich und gibt grundsätzlich UNIX an, was aber nicht weiter relevant ist, da sie dann meist in den Verzeichnislistings ein unixkompatibles Format verwenden.

Wiederaufnahe abgebrochener Übertragungen

Um Techniken wie *Resuming* zu ermöglichen, kann dem Server über das Restart (`REST`) Kommando mitgeteilt werden, dass die Übertragung erst bei einem bestimmten Byte beginnen soll:

```
REST 2048
350 Restarting at 2048. Send STORE or RETRIEVE to initiate transfer
```

In diesem Beispiel werden die ersten zwei Kibibyte übersprungen, d.h. wenn der Client nun den Dateiinhalt sendet, werden diese Daten an die Zieldatei ab Byte 2048 angefügt.

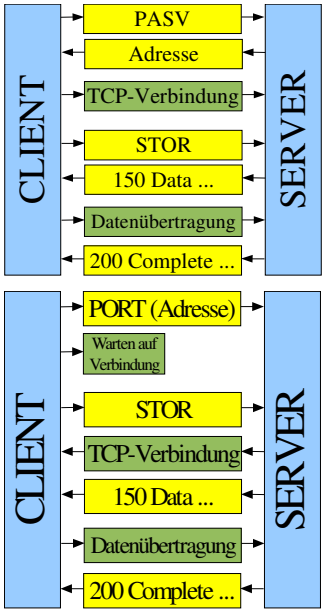
Das `REST`-Kommando ist aber keine Pflichtimplementierung und wird somit nicht von allen Servern unterstützt. Manchmal wird die Benutzung auch untersagt, da dadurch unter Umständen Dateien kompromittiert werden könnten. In diesem Fall gibt es auch einige Server, die dabei nicht die Antwort des `REST`-Kommandos mit einem Fehler versehen, sondern erst auf das Store Kommando mit einem Fehler antworten, was für die Clients dann wiederum nur sehr schwer interpretiert werden kann.

3.5. Datenübertragungen

Datenverbindungen

Eine besondere Eigenheit des FTP-Protokolls ist es, dass für die Datenübertragung neben der Kontrollverbindung eine weitere Verbindung auf einem anderen Port aufgebaut wird. Dabei unterscheidet man zwei Verfahren:

- Beim *Passive Mode* Verfahren erstellt der Client eine Verbindung zu einer Adresse, die der Server ihm gerade mitgeteilt hat. (Passiv bezieht sich dabei auf den Server, der nicht selbst aktiv wird, sondern nur auf eine Verbindung wartet.)
- Beim *Port Mode* Verfahren läuft das genau umgekehrt. Der Client wartet auf einem Port und sendet dem Server dessen Adresse, woraufhin dieser die Verbindung erstellt.



Ursprünglich war Port 20 für FTP-Datenverbindungen vorgesehen. Da ein statischer Port allerdings Probleme bereitet und Ports unter 1024 von

normalen Benutzern unter unixoiden Betriebssystem nicht belegt werden dürfen, wird meist ein zufälliger Port gewählt. Das wiederum bringt oftmals Probleme, wenn Client oder Server hinter einem Gateway sitzen: Ist das Portforwarding nur für den Server auf Port 21 aktiviert, so wird *Passive Mode* nicht funktionieren, da der Server einerseits mit hoher Wahrscheinlichkeit seine LAN-interne IP-Adresse angeben wird und andererseits der Port durch den Gateway vermutlich nicht weitergeleitet wird und somit nicht verbindbar ist. Andersherum funktioniert *Port Mode* aus den selben Gründen nicht, wenn sich der Client hinter einem Gateway befindet. Kombiniert man beides, wird eine Datenverbindung unmöglich.

Der Initialisierung einer Datenverbindung beginnt also mit einem der folgenden Kommandos:

```
PASV
227 Entering Passive Mode (192,168,0,1,175,199).
```

```
PORT 192,168,2,2,5,60
200 PORT command successful
```

Ohne jetzt eine Wertung vornehmen zu wollen, ist dieses Format der Repräsentation der Zieladresse ein wenig ungeeignet. Sowohl für den Benutzer als auch für den Programmierer wäre ein Format wie `192.168.2.2:1340`, welches in vielen anderen Bereichen üblich ist, wenn es um Adressangaben geht, wesentlich praktischer.

Die IP-Adresse besteht normalerweise aus vier durch Punkte getrennten Zahlen aus dem Bereich 0-255, also aus vier Bytes. Der Port umfasst einen Wertebereich von 0 bis 65535 (0xFFFF), besteht also aus zwei Bytes. Somit ergeben sich sechs Bytes, deren dezimale Repräsentation durch Kommas getrennt übergeben wird.

Mit diesen Informationen sind nun Client oder Server in der Lage eine Verbindung aufzubauen, über die Daten übertragen werden können.

Datenübertragung

Nachdem alle für den Transfer benötigten Vorbereitungen getroffen worden sind und auch die Datenverbindung aufgebaut wurde, wird dem Server über das Store (`STOR`) Kommando mitgeteilt, um welche Datei es sich handelt. Dazu öffnet der Server die Datei, bestätigt dies über eine vorläufige Meldung und schreibt dann die auf der Datenverbindung eintreffenden Daten in die Datei, bis der Client die Datenverbindung schließt oder auf der Kontrollverbindung das Abort (`ABOR`) Kommando sendet, das zum Abbruch führt.

Weitere Kommandos / Verzeichnislisten

Neben dem `STOR`-Kommando gibt es noch das Retrieve (`RETR`) Kommando, das die selben Mechanismen benutzt und die Datei herunterlädt, was für die Zwecke von Wput aber nicht interessant ist. Wichtig ist allerdings das `LIST`-Kommando, mit dem über die Datenverbindung eine Verzeichnisliste heruntergeladen werden kann:

```
PORT 192,168,2,2,5,60
200 PORT command successful
LIST
150 Opening ASCII mode data connection for file list
226 Transfer complete.
```

Ein typische unixoides System würde dabei z.B. die folgende Verzeichnisliste übertragen:

```
-rw-r--r--   1 ftp      nogroup  14526354 Mar 23  2004 kryptologie.tbz
drwxr-xr-x  64 ftp      nogroup      4096 Oct 26 15:15 phrack
-rw-r--r--   1 root     root      7236633 Oct 26 15:16 phrack.tbz
```

Diese enthält den Dateinamen, die Modifizierungszeit, die Dateigröße, die Besitzer und die dazugehörigen Zugriffsrechte für jede Datei. Für Wput sind dabei bisher nur die Größe und das Datum von Bedeutung.

3.6. Verschlüsselung

Das FTP-Protokoll ist ein sehr altes Protokoll, das einige Schwächen hat. Ein wesentliches Problem ist die Tatsache, dass die Anmeldeinformationen völlig unverschlüsselt übertragen werden und somit sehr leicht von anderen gelesen werden können. Auch die Übertragung der eigentlichen Nutzdaten – also der Dateien – erfolgt unverschlüsselt. Dem versucht man mit Protokollerweiterungen[1] zu begegnen, aber leider fehlen diese Implementationen noch bei den meisten Servern und Clients.

Das Prinzip der Verschlüsselung ist denkbar einfach. Nach dem Kommando zur Verschlüsselung wird die Verbindung mit einer Bibliothek gekapselt, sodass jedes Senden und Empfangen über diese Bibliothek abläuft, die die eigentliche Verschlüsselung übernimmt. Wput benutzt hierbei die OpenSSL-Bibliothek.

Noch vor dem Anmelden versucht Wput deshalb eine Verschlüsselung zu initialisieren:

```
AUTH TLS
234 AUTH TLS successful
```

Wenn das Kommando erfolgreich ist, also der FTP-Server es unterstützt und auch selbst bereit ist, eine verschlüsselte Verbindung aufzubauen, wird nun die Kontrolle an die SSL-Bibliothek übergeben, die die Verschlüsselungseinstellungen aushandelt und die verschlüsselte Verbindung initialisiert (Handshake). Alle weitere Kommunikation ist dabei für Dritte nicht lesbar.

Datenverbindungen werden dann aber nicht automatisch verschlüsselt, sondern müssen noch einmal extra initialisiert werden. Über das Protection Buffer Size (`PBSZ`) Kommando muss dem Server mitgeteilt werden, wie groß die maximale Puffergröße ist. Da Wput die SSL-Bibliothek benutzt, der die Puffergröße egal ist, wird 0 angegeben:

```
PBSZ 0
200 PBSZ 0 successful
```

Als nächstes muss noch der Typ der Verschlüsselungsstufe für die Datenverbindung gesetzt werden. Dabei gibt es zwei Flags: *Integritätsgeschützt* und *Vertraulich*, woraus sich folgende Tabelle ergibt:

Kommando	***Integritätsgeschützt***	***Vertraulich***
PROT C (Clear)	O	O
PROT S (Save)	X	O
PROT E (Confidential)	O	X

1 Die Protokollerweiterungen für sichere Kommunikation über FTP-Kanäle werden im RFC2228 (FTP Security Extensions) spezifiziert.

Kommando	*Integritätsgeschützt*	*Vertraulich*
PROT P (Private)	X	X

Wput implementiert dabei nur das Protection Level Private (`PROT P`), was auch vollkommen ausreichend ist.

```
PROT P
200 Protection set to Private
```

Wird nun eine Datenverbindung aufgebaut, wird auf dieser Verbindung genau wie vorher auf der Kontrollverbindung die Verschlüsselung über die SSL-Bibliothek initialisiert.

3.7. Kritik und mit dem FTP-Protokoll verbundene Probleme

Das größte Problem ist die Verschlüsselung, weil diese in den meisten FTP-Servern und Clients noch gar nicht implementiert ist. Auch sind die Verschlüsselungseinstellungen nicht so einfach zu konfigurieren und viele Benutzer wissen gar nicht davon, dass die Übertragung normalerweise unverschlüsselt stattfindet oder interessieren sich einfach nicht für die Sicherheit ihrer Daten.

Das Protokoll ist antwortbasierend. Theoretisch könnte man zwar alle Kommandos hintereinander senden, praktisch gesehen ist es aber notwendig, die Antwort auf ein Kommando abzuwarten, um entsprechend reagieren zu können. Gerade auf Verbindungen, die lange Antwortzeiten haben, ist das Warten allerdings ineffizient. Wo z.B. das HTTP-Protokoll meist mit einer Anfrage pro Datei auskommt, müssen beim FTP-Protokoll eine Reihe von Verabredungen getroffen werden.

Das Protokoll weiß nicht genau, was es eigentlich will. Einerseits ist es ein einfaches Protokoll für die Dateiübertragung, während es andererseits aber noch viel mehr ermöglicht. So kann der Benutzer Verzeichnisse erstellen, Dateien umbenennen, Dateien löschen, die Zugriffsrechte ändern und somit könnte man schon fast von einem Netzwerkdateisystem sprechen, wobei das Protokoll für diesen Zweck aber auch nicht wirklich brauchbar ist, da die Zugriffsmechanismen dafür nicht ausreichend sind. So kann z.B. auf eine Datei nicht an einer beliebigen Stelle zugegriffen werden, sondern es muss immer die gesamte Datei hoch- bzw heruntergeladen werden. Viele andere Protokolle sind dabei wesentlich flexibler.

4. Strukturprinzipien

4.1. Die Notwendigkeit eines Eingabemechanismus

Allein die Kenntnis des FTP-Protokolls reicht nicht aus, um das Programm fertigzustellen. Es fehlt im Wesentlichen eine Idee, in welcher Form Eingabedaten vorliegen müssen, damit aus ihnen der Ablauf der Übertragung abgeleitet werden kann. Dieses Prinzip wird hier als Eingabemechanismus bezeichnet.

Um das Programm an sich funktionsfähig zu machen, würde es ausreichen, auf der Kommandozeile über Parameter den Hostnamen, Benutzername und Kennwort, das Zielverzeichnis, den lokalen Dateinamen und den Zieldateinamen sowie noch ein paar Kleinigkeiten, wie die Wahl zwischen Ascii oder Binary Mode anzugeben. So ähnlich sahen die ersten Versionen von Wput auch aus, allerdings entspricht das nicht den Anforderungen, die sich das Projekt selbst stellt. So gehören eine gewisse intuitive Bedienung, ein gewisser Grad an Benutzerfreundlichkeit sowie das Erfüllen komplexerer Aufgaben dazu. Das vorher beschriebene Szenario wäre auch mit einem kleinen Script für das Konsolenprogramm `ftp` erreichbar.

Im Folgenden soll der von Wput verwendete Eingabemechanismus erläutert und aufgezeigt werden, wie aus den Eingabedaten einzelne *fsessions* erzeugt werden.

4.2. *fsessions*

Der mehrfach erwähnte Begriff der *fsession* bedarf einiger Klärung, da diese als ein Grundbestandteil von Wput das essentielle Bindeglied zwischen dem FTP-Protokoll und der Benutzerebene darstellt.

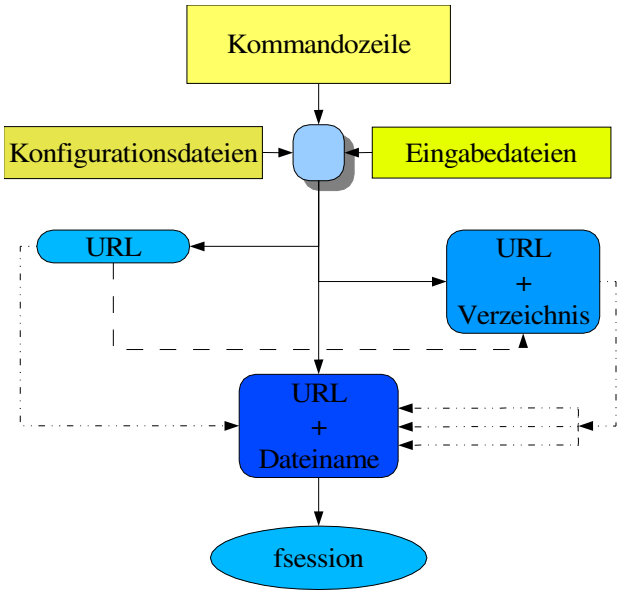
Die *fsessions* enthalten Wputs einzelne Arbeitsaufträge. Jede *fsession* repräsentiert eine Datei und enthält alle für ihren Transfer notwendigen Daten sowie alle Daten, die ansonsten zu der Datei gehören.

Im Wesentlichen sind das folgende Informationen:

- Hostname bzw. Ip-Adresse
- Port
- Benutzername und Kennwort
- Zielverzeichnis und Zieldateiname
- Quelldateiname
- Zeitpunkte der letzten Änderung
- Dateigrößen
- Übertragunsmodus

4.3. Der Eingabemechanismus in Wput

Wie der folgende Graph illustriert, stammen die Informationen, die Wput einliest, aus unterschiedlichen Quellen. Allgemeine Optionen werden aus Konfigurationsdateien oder von der Kommandozeile gelesen. Diese Informationen überlagern die in Wput voreingestellten Werte wie Timeoutintervalle, Proxyeinstellungen, *Ascii/Binary Mode*, *Port/Passive Mode* usw.



Die wesentlich wichtigeren Daten sind allerdings Dateinamen und URLs, die von der Kommandozeile (oder, wenn eine Eingabedatei spezifiziert wurde, aus dieser) gelesen werden. Die URLs sind allgemeingültige Adressangaben, die die wichtigsten Informationen (Host, Benutzer, Kennwort, Pfad, Dateiname) enthalten (siehe 7.3. URL-Parsing).

Normalerweise erwartet Wput eine URL und einen Dateinamen, damit eine *fsession* entstehen kann. Allerdings ist es oft auch ausreichend, dass die URL ohne einen dazugehörigen Dateinamen angegeben wird, wenn sich die lokale Datei daraus irgendwie[1] ableiten lässt. Stellt sich heraus, dass es sich bei dem Dateinamen in Wirklichkeit um ein Verzeichnis handelt, so wird das gesamte Verzeichnis traversiert und für jede Datei ein neues Paar aus Datei und URL erstellt. Aus jedem Paar von URL und Dateiname wird dann eine *fsession* generiert, die danach bereit ist, übertragen zu werden.

1 Wput prüft für jeden Bestandteil des Pfades, ob eine entsprechende Datei existiert. Für `ftp://host/pub/src/c.tgz` sucht Wput nach `pub/src/c.tgz`, `src/c.tgz` und letztendlich nach `c.tgz`. Die Implementierung dieses Prinzips wird in 7.4. Bau einer fsession besprochen.

Teil III – Implementierung

Der Implementierungsbereich wird versuchen exemplarisch Einblicke in den Quellcode von Wput zu verschaffen. Es werden die zugrunde liegenden Bibliotheken beschrieben und einige Funktionen detailliert erläutert. Bei den Erläuterungen werden die Funktionen teilweise in vereinfachten Formen dargestellt, da die Realität gelegentlich doch etwas komplexer ist und weitere Sonderfälle beachtet werden müssen.

Wie bereits erwähnt können die Funktionen im Originalzusammenhang im LXR-System online betrachtet werden.

5. Die Socket-Bibliothek (socketlib.c, socketlib.h)

Die Socketbibliothek stellt allgemeine Funktionen zur Kommunikation über Sockets bereit. Die Bibliothek ist halbwegs objektorientiert geschrieben, was in C wie in fast jeder Sprache möglich ist, auch wenn diese an sich keine objektorientierte Programmiersprache ist. Die Socketbibliothek ist somit ein eigenständiges Modul, welches keine weiteren Abhängigkeiten von Wput hat.[1]

5.1. Das Socket-Objekt

Normalerweise sind Sockets nur Dateideskriptoren (`fds`), also Zahlen, die auf ein Dateihandle verweisen. In der Bibliothek wird aber ein Socketobjekt (`wput_socket`) definiert, das zwar einerseits diesen Dateideskriptor enthält, das andererseits - zumindest wenn die OpenSSL-Bibliothek verfügbar ist - noch zwei zu dieser Bibliothek gehörende Objekt enthält, sodass die Arbeit mit einem Socketobjekt transparent die Verschlüsselungsmöglichkeiten nutzen kann.

```
typedef struct _wput_socket {
        int fd;
#ifdef HAVE_SSL
        SSL     * ssl;
        SSL_CTX * ctx;
#endif
} wput_socket;
```

5.2. Die Socket-API

```
wput_socket * socket_new();
void          socket_set_default_timeout(int timeout);
wput_socket * socket_connect(const unsigned int ip, const unsigned short port);
wput_socket * socket_listen(unsigned bindaddr, unsigned short * s_port);
wput_socket * socket_accept(wput_socket * sock);
void          socket_close(wput_socket * sock);
#ifdef HAVE_SSL
int           socket_transform_to_ssl(wput_socket * sock);
#endif

char * socket_read_line(wput_socket * sock);
int    socket_read (wput_socket * sock, void *buf, size_t len);
int    socket_write(wput_socket * sock, void *buf, size_t len);

int socket_is_data_writeable(int sock, int timeout);
```

1 Das stimmt zwar nicht ganz, da z.B. die `printout()`-Funktion für die Ausgabe genutzt wird, welche aber durch eine kleine `#define` Direktive auf eine normale `printf(3)`-Funktion umgeleitet werden kann, sodass das Modul durchaus auch extern nutzbar ist.

```
int socket_is_data_readable(int sock, int timeout);
```

Diese Funktionen bilden die API, auf der die gesamte Netzwerkkommunikation aufbaut. Die Implementierung dieser Funktionen wird nicht näher beschrieben, da sie sich fast ausschließlich auf Betriebssystemfunktionen stützen, deren Verwendung zu komplex zu erläutern wäre und den Rahmen dieses Dokumentes deutlich sprengen würde.

`socket_new()` ist eine hauptsächlich intern verwendete Funktion, die ein neues Socketobjekt alloziert und die Referenz darauf zurück gibt. Objektorientiert betrachtet wird also eine neue Instanz des Socket-Objekts erzeugt.

Die typischen Socket-Funktionen sind `connect`, `bind(listen)`, `accept`, `close`, `read` und `write`:

`socket_connect()` erstellt eine neue Socket, die sofern möglich mit dem Zielhost verbunden wird. Alles was für ein Verbindungsziel in diesem Fall notwendig ist, ist die vier Byte große IP-Adresse, die folglich wunderbar in einen `unsigned int` passt und der zwei Byte große Port.

`socket_listen()` erstellt eine Serversocket, die auf einem Port auf eine Verbindung wartet. Der Port wird hierbei als Zeiger übergeben, sodass dessen Wert von der Funktion selbst gesetzt werden kann. Der Wert kann selbst nicht der Rückgabewert sein, da dieser ja bereits für ein Socketobjekt vergeben ist.

`socket_accept()` empfängt eine eingehende Verbindung auf einer Serversocket und gibt dementsprechend ein neues Socketobjekt für diese eingehende Verbindung zurück.

`socket_close()` fährt die Socket herunter und schließt den Dateideskriptor. Wenn der Socket SSL-Objekte zugeordnet sind, so werden auch diese deinitialisiert. Die `wput_socket` wird dealloziert und jeder Verweis auf diese wird ungültig.

`socket_read()` und `socket_write()` lesen bzw. schreiben jeweils eine Anzahl an Zeichen in einen übergebenen Puffer. Es besteht kein großer Unterschied zu den Systemfunktionen `read(2)` und `write(2)`, jedoch wird, sofern ein SSL-Objekt initialisiert wurde, dieses zum Lesen bzw. Schreiben verwendet, sodass die Verschlüsselung auf höheren Ebenen sehr transparent verlaufen kann.

Um die Verschlüsselung aufzubauen, reicht es aus, die Funktion `socket_transform_to_ssl()` aufzurufen, die die SSL-Objekte entsprechend erstellt und mit der Socket verbindet.

Eine weitere wichtige Funktion ist `socket_read_line()`, die genau eine Textzeile von der Socket liest. Ein sehr unangenehmer Aspekt des FTP-Protokolls ist es nämlich, dass es zeilenbasiert ist und die Länge einer Zeile unbekannt ist. Normalerweise wird eine Textzeile in genau ein TCP-Paket gepackt und verschickt, sodass das Lesen von der Socket mit einem ausreichend großen Puffer genau eine Textzeile zu Tage bringt. Wie sich herausstellte, kann es aber auf sehr langsamen Verbindungen vorkommen, dass einzelne TCP-Pakete fragmentiert werden, also in mehrere kleinere Pakete aufgeteilt werden, sodass eine Zeile nun z.B. aus zwei Paketen besteht. Andererseits kommt es bei sehr schnellen Verbindungen in bestimmten Situationen dazu, dass mehrere Zeilen zu einem Paket zusammengefasst werden.

Um das Problem zu umgehen gäbe es nun zwei Möglichkeiten: Entweder liest man Daten immer in einen ausreichend großen Puffer und sobald ein Zeilenumbruch zu finden ist, wird die erste Zeile zurückgegeben, während der Rest im Puffer verbleibt. Beim nächsten Aufruf wird dann entweder sofort

eine weitere Zeile zurückgegeben oder es werden zuvor weitere Daten von der Socket gelesen. Diese Methode ist jedoch etwas aufwendiger und bedarf eines weiteren Datenpuffers, welche auf Betriebssystemebene jedoch bereits für die ankommenden Daten verwendet werden. Die andere Möglichkeit ist, so lange jeweils genau ein Byte von der Socket zu lesen, bis ein Zeilenumbruch gelesen wird und dann die Zeile zurückzugeben. Das erzeugt zwar ein wenig mehr Overhead was die Funktionsaufrufe angeht, braucht aber etwas weniger Speicher, da nicht noch ein weiterer Puffer benötigt wird.

Für die Zeile selbst wird dann natürlich schon ein Puffer benötigt. Da die Zeilenlänge aber variabel ist, ist bei keiner Puffergröße gewährleistet, dass die Zeile garantiert in den Puffer passt, weswegen hierbei ein dynamischer Puffer verwendet wird, der bei Bedarf vergrößert wird. Dasselbe Prinzip findet auch in der `read_line()` Funktion für das Auslesen der Konfigurationsdateien Anwendung. Durch diese Funktion wird sichergestellt, dass nicht wesentlich mehr Speicher als nötig verwendet wird und dass der Puffer nie überlaufen kann, da er beständig erweitert wird.

Die Funktionen `socket_is_data_readable()` und `socket_is_data_writeable()` sind kleine Hilfsfunktionen, die über die Betriebssystemfunktion `select(2)` prüfen, ob auf der Socket innerhalb des Timeoutintervalls Daten angekommen sind, bzw. ob die Socket beschreibbar ist. Der Test auf die Beschreibbarkeit ist z.B. im Zusammenhang mit dem Verbindungsaufbau sinnvoll, da eine Socket erst dann beschreibbar ist, wenn der TCP-Handshake mit dem Zielrechner komplett ist, wodurch also festgestellt werden kann, ob die Verbindung der Socket in dem Timeoutintvall zustande gekommen ist oder nicht.

5.3. Die Proxy-API

Neben den Socketfunktionen implementiert die Socketbibliothek noch eine Reihe von Proxyfunktionen, damit Verbindungen über Proxyserver möglich sind, was in vielen Umgebungen leider immer noch notwendig ist.

```
#define PROXY_OFF   0
#define PROXY_SOCKS 1
#define PROXY_HTTP  2

typedef struct _proxy_settings {
        unsigned int   ip;
        unsigned short port;
        char *         user;
        char *         pass;
        unsigned int   bind:1;
        unsigned int   type:2;
} proxy_settings;


wput_socket * proxy_init(proxy_settings * ps);
wput_socket * proxy_listen(proxy_settings * ps, unsigned int * ip, unsigned short * port);
wput_socket * proxy_accept(wput_socket * server);
wput_socket * proxy_connect(proxy_settings * ps, unsigned int ip, unsigned short port, const char *
hostname);
```

Das `proxy_settings` Objekt enthält ein paar grundlegende Einstellung für die Proxies. Neben der Adresse und den Anmeldedaten enthält es noch den Proxytyp, der spezifiziert, um welche Art von Proxy es sich handelt.

Im Wesentlichen gibt es zwei verschiedene Proxytypen: den Socks-Proxy und den HTTP-Proxy,

welche durch die Konstanten `PROXY_SOCKS` und `PROXY_HTTP` symbolisiert werden. Beim HTTP-Proxy wird eine Anfrage an den Proxyserver gesendet, die auf dem HTTP-Protokoll basiert, das eigentlich für Webseiten gedacht ist. Eine entsprechende Verbindungsanfrage sieht in der Regel wie folgt aus:

```
CONNECT www.ftpserver.com:21 HTTP/1.0

HTTP/1.0 200 Connection established
```

Nach der letzten Zeile leitet dann der Proxyserver die jeweils auf der einen Verbindung eingehenden Daten an die andere weiter, was vollkommen transparent für andere Funktionen verläuft, sodass sich mit dieser Socket genauso wie mit jeder anderen arbeiten lässt.
Da der Proxyserver oft eine Benutzeranmeldung erfordert, kann dem `CONNECT` noch eine weitere Zeile folgen:

```
Proxy-Authorization: Basic [base64-encode-username-and-password]
```

Wobei der Teil in den eckigen Klammern durch die `base64`-encodierte Version von `Benutzername\0Kennwort\0` ersetzt wird, was im HTTP-Protokoll für die Anmeldung auf Webseiten entsprechend genauso verwendet wird. Die Socketbibliothek enthält zu diesem Zweck auch eine kleine `base64`-Implementation.

Bei den Socks-Proxyservern gibt es die Versionen 4 und 5, wobei Wput nur die neuere Version 5 unterstützt. Das Socks-Protokoll ist diesmal kein *human-readable*-Protokoll sondern ein sehr einfaches binärbasiertes Protokoll, von dem Wput nur die essentiellsten Teile implementiert. So gibt es eine Vielzahl von Authentifizierungsmethoden, von denen Wput lediglich die Klartextauthentifizierung unterstützt.

Eine Hexadezimale Repräsentation zeigt die Abfolge der Initialisierung:

```
05 01 00
05 00
```

`05` ist die Versionsnummer, die stimmen muss, damit die Verbindung zustande kommen kann. `01` ist die Anzahl an Authentifizierungsmöglichkeiten, die anschließend folgen. Ist bei Wput kein Benutzername für den Proxy hinterlegt, wird ausschließlich die `00` angegeben, was einer anonymen Benutzung des Proxies ohne Anmeldung entspricht. Ist ein Benutzername verfügbar, bietet Wput auch die Klartextauthentifizierung (`02`) an:

```
05 02 02 00
05 02
05 CC [Benutzer] CC [Kennwort]
05 00
```

Der Server bestätigt dann die Initialisierung und wählt eine der Möglichkeiten aus. Wird die Authentifizierung gewählt, so werden noch Benutzername und Kennwort gesendet, wobei `CC` jeweils für die Anzahl an Bytes steht, die der Benutzername bzw. das Kennwort benötigen. Die Null der zweiten Antwort entspricht dabei der Bestätigung für eine erfolgreiche Authentifizierung. Ein anderer Code ist einem Fehler gleichzusetzen.

Da dieser Teil für alle Socks-Proxy-Funktionen identisch ist, wird er in der Funktion `proxy_init` zusammengefasst.

Der tatsächliche Verbindungsaufbau sieht dann folgendermaßen aus:

```
05 01 00 01 IP IP IP IP PP PP
05 00 00 01 IP IP IP IP PP PP
```

Hierbei bedeutet die erste `01`, dass eine normale *Connect*-Verbindung erstellt werden soll. Die zweite `01` gibt an, dass die Zieladresse als IPv4-Adresse spezifiziert wird, was dann die vierbytige Repräsentation selbiger ist, die hier als `IP IP IP IP` dargestellt wurde. Es folgen zwei Bytes für den Zielport. Statt einer `01` für die IP-Adresse kann auch `03` angegeben werden, was bewirkt, dass ein Hostname statt einer IP-Adresse gelesen wird:

```
05 01 00 03 CC [Hostname]  PP PP
05 00 00 01 IP IP IP IP PP PP
```

Wobei `[Hostname]` dann die Hexadezimale Repräsentation des Hostnamens wäre, also z.B. `77 77 77 2e 66 74 70 73 65 72 76 65 72 2e 63 6f 6d` für `www.ftpserver.com`. `CC` steht für die Anzahl an Zeichen aus denen der Hostname besteht, also in diesem Fall `11` (17).

Die Antwort ist fast genauso aufgebaut, wie die Anfrage. `05` ist die Versionsnummer und das nächste Byte ist der Antwortcode, wobei `00` mit einem erfolgreichen Verbindungsaufbau gleichzusetzen ist. Die zurückgegebene Adresse (IP und Port) ist die Adresse, an der der Server die Verbindung aufgebaut hat, was aber für die meisten Zwecke nicht weiter von Bedeutung ist.

Nach dem Verbindungsaufbau kann die Socket ganz normal genutzt werden.

Socks-Proxyserver bieten neben dem Verbindungsaufbau auch noch die Möglichkeit an, auf dem Proxy-Server einen Verbindungsserver zu erstellen (also eine *listening* Socket), was für Portmode-FTP von Bedeutung ist, da sich in dem Fall der Server mit dem Proxy verbinden muss, welcher die Verbindung an den Client weiterleitet. Dafür gibt es dann die `bind`-Funktion im Socks-Protokoll, die in der Funktion `proxy_bind()` implementiert wird. Die eingehende Verbindung wird dann über `proxy_accept()` aufgenommen.

Die Implementierung dieser Funktionen ist dabei ziemlich einfach gehalten. Die meisten Aktionen erfordern nur ein kleines Bytefeld (`char`-Array), bei dem die einzelnen Felder gesetzt und durch die Socket geschickt werden. Beim Empfang erfolgt eine entsprechende Auswertung und nach einem erfolgreichen Verbindungsaufbau wird die zur Proxyverbindung gehörende Socket zurückgegeben.

6. Die FTP-Bibliothek (ftplib.c, ftplib.h)

Die FTP-Bibliothek ist ebenfalls wie die Socketbibliothek recht objektorientiert gehalten und könnte somit theoretisch auch außerhalb von Wput für andere Projekte Verwendung finden.

Die FTP-Bibliothek stützt sich bewusst nicht auf die in Wget verwendeten Funktionen, da der Wget-Quellcode unnötig komplex ist und zu einem gänzlich anderen Zweck geschaffen wurde.

6.1. Das FTP-Verbindungs-Objekt (`ftp_con`)

Das wichtigste Objekt ist die FTP-Verbindung:

```
typedef struct _ftp_connection {
        host_t      * host;
        char        * user;
        char        * pass;
        wput_socket * sock;
        wput_socket * datasock;
        wput_socket * servsock;
        ftp_reply     r;
        proxy_settings  * ps;

        directory_list  * directorylist;
        char            * current_directory;

        unsigned int  local_ip;
        unsigned int  bindaddr;

        unsigned char needcwd     :1;
        unsigned char loggedin    :1;
        unsigned char portmode    :1;
                 char current_type:2; /* -1 (undefined), 0 (ascii), 1 binary */
        unsigned char secure      :1; /* 1 tls required */
        unsigned char datatls     :1;

        enum stype OS;
} ftp_con;
```

Dieses Objekt enthält alle zu einer FTP-Verbindung gehörenden Daten.

Der Zielrechner wird über die `host_t` Struktur spezifiziert:

```
typedef struct _host_type {
        unsigned int ip;
        char * hostname;
        unsigned short port;
} host_t;
```

Der Hostname ist nur dann belegt, wenn die IP nicht existiert. In Umgebungen, in denen man mit Proxies arbeitet, lässt sich häufig der Hostname nicht zu einer IP auflösen, weswegen dann der Hostname als einzige Identifizierungmöglichkeit bleibt.

Die `wput_sockets` sind die Kommunikationswege. `sock` selbst ist die Kontrollverbindung, `datasock` die Datenverbindung. `servsock` wird nur temporär für den Fall benötigt, dass im Portmode FTP eine *listening* Socket geöffnet werden muss, von der erst später eine eingehende Verbindung empfangen wird.

`bindaddr` ist die IP-Adresse des Interfaces an dem die *listening* Socket den Port öffnen soll. Wenn alle Interfaces benutzt werden, ist `bindaddr` 0.0.0.0 (`INADDR_ANY`).

Bedeutung der Flags:

Flag	*Bedeutung*
`needcwd`	Der Server befindet sich in einer Situation, in der auf jeden Fall das Verzeichnis gewechselt werden muss (z.B. nach einem Verbindungsaufbau).
`loggedin`	Der Loginprozess ist erfolgreich abgelaufen. Der Benutzer ist angemeldet.
`portmode`	Gibt die Starteinstellung für die Datenübertragung an. Schlägt eine der Möglichkeiten fehl, so wird diese Einstellung entsprechend verändert, sodass beim nächsten Mal gleich die richtige Methode ausgewählt wird.
`current_type`	Übertragungsmodus. Zum Zeitpunkt der Initialisierung ist `current_type` 0, nach `TYPE A` ist es 1 und nach `TYPE I` 2.
`secure`	TLS ist notwendig, um eine Verbindung aufzubauen. Normalerweise überträgt Wput die Daten unverschlüsselt, wenn der Aufbau einer TLS-Verbindung fehlschlägt. Ist `secure` gesetzt, wird in bei einem TLS-Fehlschlag die Verbindung abgebrochen.
`datatls`	Die Initialisierung der Datenverbindung verlief erfolgreich.
`OS`	Gibt das Betriebssystem des Servers an.

Für den internen Gebrauch ist die Struktur der `ftp_reply r` von Bedeutung, die genaue Informationen über die letzte Antwort des FTP-Servers enthält:

```
typedef struct _ftp_reply {
        unsigned short int code;
        char * reply;
        char * message;
} ftp_reply;
```

`code` ist dabei der Antwortcode des FTP-Servers. Bei `200 Successful`, wäre `code` `200`, `reply` die gesamte gelesene Zeile und `message` nur die Textnachricht (`Successful`).

Für den Fall, dass die FTP-Erweiterungen wie `SIZE` und `MDTM` nicht verfügbar sind, greift die Bibliothek auf die `LIST`-Funktion zurück. Da nicht für jede Datei eine komplette Verzeichnisliste heruntergeladen werden soll, werden die Ergebnisse in einer verketteten Liste gespeichert. Der Einstiegspunkt ist `directorylist`. Die Datenstrukturen sehen wie folgt aus.

```
struct fileinfo
{
  enum ftype type;                /* file type */
  char * name;                    /* file name */
  off_t size;                     /* file size */
  time_t tstamp;                  /* time-stamp */
  int perms;
  char * linkto;
  struct fileinfo *prev;        /* ...and next structure. */
  struct fileinfo *next;        /* ...and next structure. */
};

typedef struct _directory_list {
```

```
  char * name;
  struct fileinfo * list;
  struct _directory_list * next;
} directory_list;
```

Somit gibt es eine einfach verkettete Liste von Verzeichnissen, die jeweils den Namen des Verzeichnisses und den Einstiegspunkt für die Dateiliste enthalten. Die Dateiliste wiederum ist eine doppelt verkettete Liste, die original aus externen Quellen übernommen wurde, da die FTP-LS-Bibliothek genau diese Datenstruktur benutzt. Daher finden sich auch einige für Wput unwichtige Informationen in dieser Struktur.

6.2. Die FTP-API - Grundbestandteile

Um überhaupt auf einem FTP-Objekt arbeiten zu können, muss dieses zunächst einmal existieren. Dafür sind die Konstruktoren da, die die Datenstruktur allozieren und bereits einige wenige Eigenschaften initialisieren.

```
/* konstruktor */
host_t  * ftp_new_host(unsigned ip, char * hostname, unsigned short port);
ftp_con * ftp_new(host_t * host, int secure);

/* dekonstruktor */
void ftp_free_host(host_t * host);
void ftp_quit(ftp_con * self);
```

Die Dekonstruktoren machen dabei genau das Gegenteil und geben den von dem Objekt belegten Speicher wieder frei. `ftp_quit()` geht noch einen Schritt weiter und beendet die FTP-Verbindung, sofern sie noch besteht, ordnungsgemäß, schließt auch die zur Verbindung gehörenden Sockets und gibt die Verzeichnislistings wieder frei.

```
/* basic send/recv-api */
int  ftp_get_msg(ftp_con * self);
void ftp_issue_cmd(ftp_con * self, char * cmd, char * value);
```

Diese beiden Funktionen implementieren den wichtigsten Grundbestandteil der API. `ftp_issue_cmd()` ist eine an sich kleine Funktion, die die zwei Zeichenfolgen, die ihr übergeben werden, durch ein Leerzeichen verbindet und an die Sockets weiterreicht. Da solche Stringmanipulationen in C aber eher kompliziert zu realisieren sind, wäre es zu aufwendig, wenn dies jede Funktion, die etwas senden muss, selbst tun würde.

`ftp_get_msg()` ist der Kern der Bibliothek. Die Funktion liest eine Zeile von der Socket aus und prüft deren Validität.

Nun zum detaillierten Aufbau der Funktion. Ein Grundkonstrukt könnte so aussehen:

```
int ftp_get_msg(ftp_con * self) {
    char * msg = socket_read_line(self->sock);
    msg[3] = 0;
    self->r.code    = atoi(msg);
    self->r.reply   = msg;
    self->r.message = msg+4;
    return 0;
}
```

Es wird also eine Zeile gelesen und das vierte Zeichen auf `\0` gesetzt, sodass aus `200 Successful\0` plötzlich `200\0Successful\0` wird. Da Zeichenfolgen ja nichts anderes als `\0`-terminierte Bytefolgen sind, entstehen somit zwei Zeichenfolgen. Die erste davon ist `200`, die über die Funktion `atoi(3)` in

einen Integer umgewandelt wird. Die zweite Zeichenfolge beginnt an der Adresse der ersten plus vier Bytes und enthält die Textnachricht des Kommandos. Die ganzen Informationen werden dann in die `ftp_reply` Struktur gepackt und liegen zur Verwendung durch andere bereit.

Das allein ist aber bei weitem nicht ausreichend: Das erste Problem ist, dass `socket_read_line()` nicht immer eine Zeile zurück gibt, sondern bei einem Lesefehler auch `NULL` oder `ERR_TIMEOUT` zurück geben kann, Fehler die also entsprechend abgefangen werden müssen:

```
if(!msg[1]) {
    printout(vLESS, "Receive-Error: Connection broke down.\n");
    return ERR_RECONNECT;
}
if(msg == (char *)[2] ERR_TIMEOUT)
    return ERR_TIMEOUT;
```

Des Weiteren gilt es zu prüfen, ob die Nachricht den Anforderungen an eine FTP-Antwortzeile genügt, da sonst das Parsing durcheinander kommt, was zu undefiniertem Verhalten führen kann. Die wichtigste Voraussetzung ist, dass die Nachricht aus mindestens vier Zeichen besteht, wovon die ersten drei jeweils Ziffern sind. Somit ergibt sich folgende Prüfung:

```
if(strlen(msg) < 4 || !ISDIGIT[3](msg[0]) || !ISDIGIT(msg[1]) || !ISDIGIT(msg[2])) {
    printout(vLESS, "Receive-Error: Invalid FTP-answer (%d bytes): %s\n", strlen(msg), msg);
    free[4](msg);
    printout(vLESS, "Reconnecting to be sure, nothing went wrong\n");
    return ERR_RECONNECT;
}
```

Mit diesem Code ist die Funktion schon recht robust, doch leider werden nicht alle Fälle abgedeckt, die das RFC ermöglicht. So sind Antworten von FTP-Servern nicht immer nur auf den dreistelligen Zifferncode und die Antwort beschränkt, sondern können auch z.B. aus mehreren Zeilen bestehen, wobei mehrzeilige Antworten mit einer Zeile wie `200-Multiline Message` beginnen, also statt dem Leerzeichen mit einen Bindestrich versehen sind. Die letzte Zeile einer solchen mehrzeiligen Nachricht muss dann wieder ein Leerzeichen haben. So kann also eine mehrzeilige Nachricht durchaus wie folgt aussehen:

```
123-First line
Second line
 234 A line beginning with numbers
123 The last line
```

Die Idee ist folglich, dass, sobald ein Bindestrich als viertes Zeichen entdeckt wird, von einer mehrzeiligen Nachricht ausgegangen wird, also beliebig viele Zeilen folgen können, bis wieder eine

---

1 `msg` ist normalerweise ein Char-Pointer der eine Adresse hat. Wenn msg NULL (`0x00000000`) ist, ist `!msg` wahr. In allen anderen Fällen ist `!msg` falsch.

2 `ERR_TIMEOUT` ist eine Konstante, die knapp unter Null liegt. Da `socket_read_line()` allerdings einen Pointer auf ein Char-Array zurück gibt, kann man nicht ohne weiteres diesen Pointer mit einer Ganzzahl vergleichen, weswegen die Ganzzahl vorher auf einen Char-Pointer gecastet wird, damit die zu vergleichenden Zahlen vom selben Typ sind. Da die Errorlevels immer nur knapp unter Null liegen (also im Bereich von -10 bis 0) kann man sicher sein, dass es nie vorkommen wird, dass eine Zeichenfolge einmal die reale Adresse von z.B. `0xFFFFFFF6` (-10) annehmen wird, schon allein, weil sie an der Grenze des Datentyps liegt.

3 Das Makro `ISDIGIT(x)` prüft einfach nur, ob das Zeichen `x` eine Ziffer ist.

4 `socket_read_line()` alloziert einen Puffer für die Zeile. Es ist Aufgabe der lesenden Funktion, diesen Speicherbereich wieder freizugeben, sobald dieser nicht mehr benötigt wird. Das Freigeben erfolgt über die `free(3)`-Funktion.

Nachricht kommt, die gänzlich den Voraussetzungen entspricht. Für die Implementierung bedeutet dies, dass es eine funktionsinterne *Flag* gibt, die angibt, ob es sich um eine mehrzeilige Nachricht handelt:

```
static[1] int multi_line = 0;
```

Wird nun ein Bindestrich entdeckt, wird diese Flag auf `1` gesetzt und `ftp_get_msg()` selbst aufgerufen. Im nächsten Kontext ist dann die Flag immer noch gesetzt, was erlaubt, dass auch Zeilen, die nicht mit dem Zifferncode beginnen, akzeptiert werden. Zur Information für den Benutzer werden die Textnachrichten dieser mehrzeiligen Meldungen ausgegeben:

```
if(msg[3] == '-') {
    multi_line = 1;
    printout(vMORE, "# %s\n", msg+4);
    free(msg);
    return ftp_get_msg(self);
}
```

Und für nicht valide Zeilen ergibt sich folgende Modifikation:

```
if(strlen(msg) < 4 || !ISDIGIT(msg[0]) || !ISDIGIT(msg[1]) || !ISDIGIT(msg[2])) {
    if(multi_line) {
        printout(vMORE, "# %s\n", msg);
        free(msg);
        return ftp_get_msg(self);
    }
    printout(vLESS, "Receive-Error: Invalid FTP-answer (%d bytes): %s\n", strlen(msg), msg);
    free(msg);
    printout(vLESS, "Reconnecting to be sure, nothing went wrong\n");
    return ERR_RECONNECT;
}
```

Anschließend muss am Ende der Funktion die Flag wieder auf `0` gesetzt werden, damit beim nächsten Aufruf wieder von einer normalen Nachricht ausgegangen werden kann:

```
multi_line = 0;
```

Als letztes ergibt sich der Wunsch nach einer automatischen Klassifizierung der Antwortcodes, sodass ein entsprechender Rückgabewert möglich ist. Dazu wird einfach das erste Zeichen der Antwort geprüft und anhand dessen ein entsprechender Rückgabewert erzeugt. Aus all dem ergibt sich nun folgende Gesamtfunktion:

```
int ftp_get_msg(ftp_con * self) {
    char * msg = socket_read_line(self->sock);
    static int multi_line = 0;
    if(self->r.reply) {
        free(self->r.reply);
        self->r.reply = NULL;
    }
    if(!msg) {
        printout(vLESS, "Receive-Error: Connection broke down.\n");
        return ERR_RECONNECT;
    }
    if(msg == (char *) ERR_TIMEOUT)
        return ERR_TIMEOUT;
    if(strlen(msg) < 4 || !ISDIGIT(msg[0]) || !ISDIGIT(msg[1]) || !ISDIGIT(msg[2])) {
```

1 `static` bedeutet, dass diese Variable nicht für jeden Funktionsaufruf eindeutig ist, sondern für jeden Funktionsaufruf genau dieselbe Variable ist, deren Wert auch zwischen Funktionsaufrufen erhalten bleibt. Man kann diesen Variablentyp mit globalen Variablen vergleichen, jedoch ist bei `static` der Geltungsbereich auf die Funktion selbst beschränkt und inteferiert somit nicht mit anderen Codebereichen.
Die Variable wird mit dem Wert `0` initialisiert, was aber nur beim ersten Funktionsaufruf der Fall ist.

```
            if(multi_line) {
                printout(vMORE, "# %s\n", msg);
                free(msg);
                return ftp_get_msg(self);
            }
            printout(vLESS, "Receive-Error: Invalid FTP-answer (%d bytes): %s\n", strlen(msg), msg);
            free(msg);
            printout(vLESS, "Reconnecting to be sure, nothing went wrong\n");
            return ERR_RECONNECT;
        }
        if(msg[3] == '-') {
            multi_line = 1;
            printout(vMORE, "# %s\n", msg+4);
            free(msg);
            return ftp_get_msg(self);
        }
        multi_line = 0;
        msg[3] = 0;
        self->r.code    = atoi(msg);
        self->r.reply   = msg;
        self->r.message = msg+4;
        printout(vDEBUG, "[%d] '%s'\n", self->r.code, self->r.message);

        /* check errors that may occur to every process and return a specific error number */
        if(self->r.reply[0] == '1')
            return ERR_POSITIVE_PRELIMARY;

        if(self->r.reply[0] == '2' || self->r.reply[0] == '3')
            return 0;

        /* rfc says that on 4xx errors, the command can be retried as it was */
        if(self->r.reply[0] == '4')
            return ERR_RETRY;

        if(self->r.reply[0] == '5')
            return ERR_PERMANENT;

        /* when this is reached there must be something wrong */
        printout(vLESS, "Dead code reached by FTP-reply:\n%d %s\n",
             self->r.code, self->r.message);
        return 0;
    }
```

6.3. Die FTP-API – FTP-Funktionen

Die Bibliothek implementiert zahlreiche Funktionen, die im Wesentlichen die in Kapitel 3 beschriebenen FTP-Funktionen erfüllen und daher hier nicht jede einzeln erläutert wird.

```
/* ftp-functions */
int  ftp_connect(ftp_con * self, proxy_settings * ps);
int  ftp_login(ftp_con * self, char * user, char * pass);
#ifdef HAVE_SSL
int  ftp_auth_tls(ftp_con * self);
int  ftp_set_protection_level(ftp_con * self);
#endif
int  ftp_do_syst(ftp_con * self);
int  ftp_do_abor(ftp_con * self);
void ftp_do_quit(ftp_con * self);
int  ftp_do_cwd(ftp_con * self, char * directory);
int  ftp_do_mkd(ftp_con * self, char * directory);

int  ftp_get_modification_time(ftp_con * self, char * filename, time_t * timestamp);
int  ftp_get_filesize(ftp_con * self, char * filename, off_t * filesize);
int  ftp_set_type(ftp_con * self, int type);

int  ftp_do_list(ftp_con * self);
int  ftp_get_list(ftp_con * self);
```

```
int  ftp_do_rest(ftp_con * self, off_t filesize);
int  ftp_do_stor(ftp_con * self, char * filename);

int  ftp_establish_data_connection(ftp_con * self);
int  ftp_complete_data_connection(ftp_con * self);

int  ftp_do_passive(ftp_con * self);
int  ftp_do_port(ftp_con * self);
```

Der allgemeine Aufbau dieser Funktionen soll hierbei am Beispiel von `int ftp_login(ftp_con * self, char * user, char * pass)` beschrieben werden. Voraussetzung der Funktion ist es, dass `self` ein FTP-Objekt ist, bei dem bereits die Kontrollverbindung aufgebaut ist. Der Benutzername (`user`) und das Kennwort (`pass`) werden übergeben. Dabei darf `pass` auch `NULL` sein, wobei in diesem Fall angenommen wird, dass das Benutzerkonto kein Kennwort benötigt. Nach dem Aufruf der Funktion soll der Benutzer eingeloggt sein. Gibt es einen Fehler (z.B. falsches Kennwort), so wird der Fehlercode `ERR_FAILED` zurückgegeben. Gibt es einen Datenkommunikationsfehler (`SOCKET_ERROR`) so wird dieser zurückgegeben.

Als erstes erfolgt ein erneuter Blick auf die Spezifikation des Anmeldevorgangs: Zum Anmelden wird das `USER`-Kommando gesendet und als Antwort sind drei Rückgabewerte auszuwerten:

```
331 need password
332 need account
230 logged in
```

Das Senden des `USER`-Kommandos ist dank der API sehr einfach:

```
ftp_issue_cmd(self, "USER", user);
```

Nun wird die Rückmeldung des Servers erwartet, die über die `ftp_get_msg()` Funktion gelesen wird:

```
res = ftp_get_msg(self);
if(SOCKET_ERROR(res)) return res;
```

Das Makro `SOCKET_ERROR(x)` prüft dabei, ob `x` `ERR_TIMEOUT` oder `ERR_RECONNECT` ist. In diesem Fall besteht ein Verbindungsproblem, wogegen die Loginfunktion machtlos ist. Der Fehler wird an die nächst höhere Ebene weitergereicht, die sich dann darum bemühen muss, die Verbindung erneut aufzubauen.

Anhand des Fehlercodes werden nun die einzelnen Möglichkeiten traversiert. Die Accountmethode wird von Wput nicht implementiert, weswegen also die `332`er Nachricht in einer Fehlerausgabe endet:

```
if(self->r.code == 332) {
    printout(vLESS, "Error: Server requires account login, which is not supported\n");
    return ERR_FAILED;
}
```

Für die `331`er Nachricht wird ein Kennwort benötigt, wobei hier zunächst geprüft werden muss, ob überhaupt ein Kennwort zum Senden vorhanden ist. Ist das nicht der Fall, wird versucht, ein leeres Kennwort zu benutzen, wobei sich ein leeres Kennwort sehr stark von dem nichtexistenten Kennwort unterscheidet, denn `NULL != ""`.

```
if(self->r.code == 331) {
    if(!pass)
        printout(vMORE, "Warning: remote server requires a password, but none set. Using an
empty one\n");
    ftp_issue_cmd(self, "PASS", pass ? pass : "");
    res = ftp_get_msg(self);
    if(SOCKET_ERROR(res)) return res;
}
```

Nun müssen auch von diesem Kommando die Fehlercodes ausgewertet werden. Hierbei sind wieder die Fehlercodes `230` (*logged in*) und `332` (*account required*) zulässig. Da dies dieselben sind, wie beim `USER`-Kommando, kann man sich durch geschickte Strukturierung hierbei weiteren auswertenden Code sparen.

Als letztes werden alle anderen Fehlercodes gefiltert und als Fehler betrachtet. Nur `230` führt zum Erfolg:

```
if(self->r.code != 230) {
    printout(vLESS, "Error: Login-Sequence failed (%s)\n", self->r.message);
    return ERR_FAILED;
}

printout(vNORMAL, "Logged in!\n");
return 0;
```

Damit wäre die Funktion an sich komplett, aber um ein wenig Overhead zu vermeiden, soll sich die FTP-Verbindung noch merken, unter welchem Benutzername sie eingeloggt ist und für den Fall, dass derselbe Benutzer noch einmal angemeldet werden soll, soll die Loginprozedur dann einfach übersprungen werden, da der Benutzer ja bereits angemeldet ist.
Somit werden am Ende der Funktion der Benutzername und das Kennwort in dem FTP-Verbindungs-Objekt gespeichert und außerdem die `loggedin`-Flag gesetzt:

```
self->loggedin = 1;
self->user     = user;
self->pass     = pass;
```

Beim Funktionsaufruf wird dann anhand dieser Informationen geprüft, ob der Benutzer schon eingeloggt ist und gegebenenfalls die Funktion direkt verlassen:

```
if(self->loggedin && SAVE_STRCMP[1](user, self->user) && SAVE_STRCMP(pass, self->pass))
    return 0;
```

Hier noch einmal das Gesamtbild der `ftp_login()`-Funktion:

```
int ftp_login(ftp_con * self, char * user, char * pass){
    int res = 0;

    if(self->loggedin && SAVE_STRCMP(user, self->user) && SAVE_STRCMP(pass, self->pass))
        return 0;

    printout(vNORMAL, "Logging in as %s ... ", user);

    ftp_issue_cmd(self, "USER", user);
    res = ftp_get_msg(self);
    if(SOCKET_ERROR(res)) return res;

    if(self->r.code == 331) {
        if(!pass)
            printout(vMORE, "Warning: remote server requires a password, but none set. Using an
empty one\n");
        ftp_issue_cmd(self, "PASS", pass ? pass : "");
        res = ftp_get_msg(self);
        if(SOCKET_ERROR(res)) return res;
    }

    if(self->r.code == 332) {
        printout(vLESS, "Error: Server requires account login, which is not supported\n");
```

1 `SAVE_STRCMP` ist ein Makro, das dafür sorgt, dass keine Zeichenfolgen verglichen werden die `NULL` sind, was in einem Segmentation Fault enden würde. Ansonsten gibt es `1` zurück, wenn die beiden Zeichenfolgen übereinstimmen oder `0`, wenn dies nicht der Fall ist.

```
        return ERR_FAILED;
    }

    if(self->r.code != 230) {
        printout(vLESS, "Error: Login-Sequence failed (%d %s)\n", self->r.code, self->r.message);
        return ERR_FAILED;
    }

    printout(vNORMAL, "Logged in!\n");
    self->loggedin = 1;
    self->user     = user;
    self->pass     = pass;
    return 0;
}
```

Das grundsätzliche Prinzip ist für die meisten dieser Bibliotheksfunktionen gleich. Es wird ein Kommando übermittelt und das Ergebnis ausgewertet, um daraus Schlüsse für die Umgebung bzw. den weiteren Verlauf zu ziehen, die sich dann in konkreten Aktionen bzw. in Rückgabewerten manifestieren.

In der nächst höheren Ebene wird mit Hilfe dieser Bibliothek die Verbindung gesteuert und letztendlich eine Datei hochgeladen.

7. Grundfunktionen

Die meisten höheren Funktionen die mit der Kommunikation oder dem Transfer zu tun haben, benutzen das *fsession*-Objekt. Dieses enthält – wie bereits in Kapitel 4 erwähnt – alle Daten, die zum Transfer einer Datei notwendig sind. Zusätzlich enthält eine *fsession* in ihrer aktiven Phase, d.h. wenn sie bearbeitet bzw. übermittelt wird und mit dem FTP-Server kommunizieren muss, einen Verweis auf ein `ftp_con` Objekt, auf dem die FTP-Steuerung über die FTP-Bibliothek aus Kapitel 6 erfolgen kann.

7.1. Verzeichniswechsel

Das Wechseln des Verzeichnisses ist eine der komplizierteren Aufgaben von Wput. Am Anfang befindet man sich irgendwo im Verzeichnisbaum des FTP-Servers. Man könnte den Ort über das Print Working Directory (`PWD`) Kommando herausfinden, was einem allerdings recht wenig nützt. Die Aufgabe von Wput ist es lediglich, in das Zielverzeichnis zu wechseln (wie unter 3.4. beschrieben).

Im Idealfall übermittelt Wput also das Zielverzeichnis und gelangt prompt dorthin. Für diesen Fall wurde die Funktion `do_cwd()` geschrieben:

```
int do_cwd(_fsession * fsession){
    int res;
    char * unescaped = unescape(cpy(fsession->target_dname));
    res = ftp_do_cwd(fsession->ftp, unescaped);
    free(unescaped);
    return res;
}
```

Da der Zielpfadname sozusagen noch im Rohmaterial vorliegt, muss dieser zunächst über die `unescape()` Funktion bearbeitet werden. Da bei diesem Vorgang die Originaldaten verändert werden, wird über die Funktion `cpy()` eine Kopie der Zeichenfolge erstellt. Nach der Verwendung des bearbeiteten Verzeichnisnamens, wird der für die Kopie reservierte Speicher wieder freigegeben.

Aber diese Funktion reicht nicht aus, um eine vollständige Funktionalität zu gewährleisten. Angenommen, es würden zwei Dateien übertragen werden: Die erste nach `/pub/usr/`, die zweite nach `/pub/src/`. Da Wput unnötigen Overhead zu vermeiden versucht, wird nicht für jede dieser Dateien eine neue FTP-Verbindung aufgebaut, sondern die alte, sofern das möglich ist, nach dem Transfer weitergenutzt. Für die erste Datei wechselt Wput also in das Verzeichnis `pub/usr/` und versucht danach in das Verzeichnis `pub/src/` zu wechseln. Zusammen sieht das so aus:

```
CWD pub
CWD usr
[Übertragung]
CWD pub
CWD src
```

Wäre dieser Vorgang erfolgreich, so befände sich Wput nun im Verzeichnis `/pub/usr/pub/src`, was bestimmt nicht das gewünschte Verhalten wäre. Da Wput allerdings nicht weiß, in welchem Verzeichnis es gestartet wurde (Es handelt sich dabei leider nicht immer um das Rootverzeichnis (`/`)), brächte ein Wechseln nach `/` und anschließend nach `pub/src` auch nicht das gewünschte Ergebnis. Wünschenswert wäre eine automatische Erkennung des Unterschieds, sodass nur noch der relative Pfad für die Änderung übrig bleibt. Genau diese Funktion wird in `get_relative_path()` implementiert. Aus `pub/usr` und `pub/src` wird demzufolge `../src`, wobei `..` in das höher liegende Verzeichnis

wechselt. Nach dem erfolgreichen Wechseln in ein Verzeichnis muss dieses Verzeichnis im FTP-Verbindungs-Objekt als aktuelles Verzeichnis gespeichert werden, damit sich spätere Verzeichniswechseloperationen wieder darauf beziehen können. Entsprechend modifiziert sieht die Routine dann folgendermaßen aus:

```
int do_cwd(_fsession * fsession){
    int res;
    char * unescaped = unescape(cpy(fsession->target_dname));
    char * relative;
    clear_path[1](unescaped);
    if(fsession->ftp->current_directory) clear_path(fsession->ftp->current_directory);

    if(fsession->ftp->current_directory) {
        relative = get_relative_path(fsession->ftp->current_directory, unescaped);
        free(unescaped);
        unescaped = relative;
    }

    res = ftp_do_cwd(fsession->ftp, unescaped);
    free(unescaped);
    return res;
}
```

Nur wenn bereits ein aktuelles Verzeichnis für diese FTP-Verbindung gesetzt wurde, wird aus diesem und dem Zielverzeichnis ein relativer Pfad gewonnen, der dann übermittelt wird.

In vielen anderen Situationen schlägt dieser direkte Weg fehl und es muss jedes Verzeichnis einzeln gewechselt und bei Bedarf sogar erstellt werden. Dazu dient die Funktion `long_do_cwd()`, die ebenfalls wie `do_cwd()` als erstes den relativen Pfad ermittelt. Für jedes Element des Pfades (`/` ist der Delimiter) wird dieser Pfadbestandteil über die `unescape()`-Funktion bearbeitet und dann an `try_do_cwd()` übergeben. Diese Funktion implementiert die Möglichkeit das Verzeichnis zu wechseln, bei einem Fehlschlag das Verzeichnis zu erstellen, um anschließend erneut zu versuchen, in dieses Verzeichnis zu gelangen.

```
int try_do_cwd(ftp_con * ftp, char * path) {
    int res;
    res = ftp_do_cwd(ftp, path);
    if(SOCKET_ERROR(res))
        return ERR_RECONNECT;
    if(res < 0 && opt.no_directories)
        return ERR_FAILED;

    if(res < 0) {
        res = ftp_do_mkd(ftp, path);
        if(SOCKET_ERROR(res))
            return ERR_RECONNECT;
        if(res < 0)
            return ERR_FAILED;

        res = ftp_do_cwd(ftp, path);
        if(SOCKET_ERROR(res))
            return ERR_RECONNECT;
        if(res < 0)
            return ERR_FAILED;
```

1 Die Funktion `get_relative_path()` kann nur dann zufriedenstellend arbeiten, wenn beide Pfadnamen auf ein Minimum reduziert sind. Pfadangaben, die `.` oder `..` enthalten, sind zwar überall sonst zulässig, bringen aber die Funktion durcheinander. Um das zu verhindern, entfernt `clear_path()` (eine weitere geniale Funktion des Autors) überflüssige Pfadargumente zuverlässig. Aus `pub/./src` wird `pub/src`, aus `pub/usr/../src` wird ebenfalls `pub/src`. Aus `../..gaga/welt../../../.fool/../../new.`, wird `../new.` um zu beweisen, dass die Funktion auch unter Extrembedingungen einwandfrei funktioniert.

```
    }
    return 0;
}
```

Dabei wird nach jedem FTP-Kommando überprüft, ob ein Verbindungsfehler besteht und dieser gegebenenfalls weitergereicht. Wenn das Wechseln in ein Verzeichnis fehlschlägt, wird zudem noch geprüft, ob nicht durch den Benutzer verboten wurde, Verzeichnisse zu erstellen, was dieser über die Kommandozeilenoption `--no-directories` tun kann, welche die *Flag* `opt.no_directories` setzt.

Als nächstes gilt es wieder unnötigen Overhead zu vermeiden. Für den Fall, dass in das Verzeichnis `/pub/usr/share/doc/` gewechselt werden soll und bereits `usr/` nicht existiert, ist es überflüssig zu probieren, ins Verzeichnis `share` zu wechseln, da dieses garantiert auch nicht existieren wird. Also wird eine *Flag* benötigt, die sobald ein Verzeichnis erstellt wurde auf `1` gesetzt wird und in folgenden Durchläufen verhindert, dass ein weiteres Verzeichnis erstellt wird:

```
int try_do_cwd(ftp_con * ftp, char * path, int mkd) {
    if(!mkd) {
        res = ftp_do_cwd(ftp, path);
        if(SOCKET_ERROR(res))
            return ERR_RECONNECT;
    }
    [...]
    mkd = 1;
    [...]
    return mkd;
```

Die Flag `mkd` wird zurückgegeben. Von der aufrufenden Funktion (`long_do_cwd()`) wird erwartet, dass diese `try_do_cwd()` anfangs mit `mkd=0` aufruft, aber sobald `try_do_cwd() 1` zurückgegeben hat, weitere Funktionsaufrufe auch mit einer `1` versieht.

Nun ergibt sich auch noch der spezielle Fall, dass in das Verzeichnis `/pub` gewechselt werden soll und `/pub` gar nicht existiert. In so einem Fall muss zuerst in das Rootverzeichnis (`/`) gewechselt werden und dort das Verzeichnis `pub` erstellt werden. Zu diesem Zwecke wird geprüft, ob das erste Zeichen des Zielverzeichnisses ein Slash ist und in diesem Falle versucht, in das Rootverzeichnis zu wechseln. Schlägt selbst dies fehl, so wird davon ausgegangen, dass es unmöglich ist, auf dem FTP-Server in irgendein Verzeichnis zu wechseln und der gesamte Host als nicht nutzbar gekennzeichnet, sodass gar nicht erst versucht wird, weitere Dateien hochzuladen. Um `/pub` zu `pub` zu transformieren, wird einfach nur der Zeiger auf `/pub` inkrementiert, der nun auf die nächst folgende Speicherzelle zeigt, also auf `pub`.

```
    if(path[0] == '/') {
        res = ftp_do_cwd(ftp, "/");
        if(SOCKET_ERROR(res))
            return ERR_RECONNECT;

        if(res < 0)
            return ERR_SKIP;
        path++;
    }
```

Nach dem letztendlich erfolgreichen Wechsel in ein Verzeichnis, muss dieses Verzeichnis noch in `ftp_con.current_directory` gespeichert werden.

7.2. Die FTP-Steuerungsfunktion

```
int fsession_transmit_file(_fsession * fsession, ftp_con * ftp);
```

Die Ansprüche an die FTP-Steuerungsfunktion sind denkbar einfach. Gegeben sind die globalen Optionen sowie eine *fsession* und sofern vorhanden, die vorherige FTP-Verbindung. Nun soll mit Hilfe dieser Daten sowie der FTP-Bibliothek und einigen Hilfsfunktionen die gesamte FTP-Transaktion abgewickelt werden. Als Rückgabewert gibt es drei Möglichkeiten: `ERR_FAILED` für den Fall, dass die Datei aufgrund eines Fehlers nicht übermittelt wurde, `ERR_SKIPPED`, falls die Datei übersprungen wurde (z.B. weil die Zieldatei neuer ist) oder `0` falls die Datei erfolgreich übermittelt werden konnte.

Als erstes gilt es zu prüfen, ob bereits eine FTP-Verbindung besteht und wenn ja, ob diese mit der Zieladresse der *fsession* übereinstimmt. Ist das nicht der Fall, muss die bestehende Verbindung geschlossen und eine neue initialisiert werden:

```
if(ftp) {
    if(ftp->host->ip       == fsession->host->ip &&
       ftp->host->port     == fsession->host->port &&
       SAVE_STRCMP(ftp->host->hostname, fsession->host->hostname))
        fsession->ftp = ftp;
    else
        ftp_quit(ftp);
}
if(!fsession->ftp)
    fsession->ftp = ftp_new(fsession->host, opt.secure);
```

Als nächstes muss, sofern das FTP-Objekt noch nicht verbunden ist, die Verbindung aufgebaut werden. Dabei werden auch bestimmte globale Optionen für das FTP-Objekt gesetzt.

```
if(!fsession->ftp->sock) {
    ftp_connect(fsession->ftp, &opt.ps);
    fsession->ftp->portmode = opt.portmode;
    fsession->ftp->bindaddr = opt.bindaddr;
}
```

Anschließend muss der Benutzer angemeldet werden. Da `ftp_login()` selbst prüft, ob der Benutzer bereits angemeldet ist, muss dies nicht hier geschehen.

```
res = ftp_login(fsession->ftp, fsession->user, fsession->pass);
```

Die nächste Aufgabe ist das Wechseln des Verzeichnisses. Hierbei wird zuerst das Zielverzeichnis über die Funktion `clear_path()` von überflüssigen Angaben bereinigt (siehe 7.1.). Als erstes wird versucht über die schnelle `do_cwd()` Funktion zum Ziel zu kommen. Falls das fehlschlägt, wird `long_do_cwd()` verwendet und falls diese ebenfalls fehlschlägt, kann die Funktion gleich verlassen werden:

```
if(fsession->target_dname)
    clear_path(fsession->target_dname);

if(fsession->target_dname && (
    fsession->ftp->needcwd || (fsession->ftp->current_directory &&
    strcmp(fsession->ftp->current_directory, fsession->target_dname))))
{
    res = do_cwd(fsession);
    if( res == ERR_FAILED ) {
        res = long_do_cwd(fsession);
        if(res == ERR_FAILED) {
            fsession->ftp->needcwd = 1;
            printout(vLESS, "Failed to change to target directory. Skipping this file/dir.\n");
            return ERR_FAILED;
        }
    }
```

```
        fsession->ftp->needcwd = 0;
        if(fsession->ftp->current_directory)
            free(fsession->ftp->current_directory);
        fsession->ftp->current_directory = cpy(fsession->target_dname);
    }
```

Im Erfolgsfall wird das aktuelle Verzeichnis zur späteren Verwendung gespeichert.

Als nächstes gilt es, die Dateigröße der Zieldatei zu ermitteln, sofern diese existiert. Anhand der Größe ergeben sich vier Möglichkeiten für das Verhältnis von Quell- und Zieldatei.

a) Die Zieldatei existiert nicht
b) Die Zieldatei ist kleiner als die Quelldatei
c) Die Zieldatei ist genauso groß wie die Quelldatei
d) Die Zieldatei ist größer als die Quelldatei

Für den Fall a) wird Wput die Datei immer hochladen. Ein anderes Verhalten würde keinen Sinn machen. Für die anderen Fälle ist das Verhalten änderbar. Die Informationen dazu werden im `resume_table` gespeichert:

```
   typedef struct __resume_table {
   #    define RESUME_TABLE_SKIP    0
   #    define RESUME_TABLE_UPLOAD  1
   #    define RESUME_TABLE_RESUME  2
b)     unsigned char large_small:2;
c)     unsigned char large_large:1;
d)     unsigned char small_large:1;
   } _resume_table;
```

Dabei entsprechen die drei Einträge den Fällen b), c) und d). Für c) und d) gibt es nur zwei Behandlungsmöglichkeiten: Erneut Hochladen (`RESUME_TABLE_UPLOAD`) oder Überspringen (`RESUME_TABLE_SKIP`). Für a) gibt es noch die dritte Möglichkeit den Upload fortzusetzen (`RESUME_TABLE_UPLOAD`), sprich die fehlenden Bytes an die Zieldatei anzufügen.

Damit der Benutzer die Möglichkeit hat das Verhalten nach seinen Wünschen zu gestalten, gibt es ein Defaultverhalten, das mit Hilfe von Kommandozeilenparametern überschrieben werden kann. Die folgende Tabelle soll dies illustrieren, wobei die Zahlen bei Quelle und Ziel für imaginäre Dateigrößen stehen.

Quelle	*Ziel*	*Default*	*--dont-continue*	*--reupload*	*--skip-larger*	*--skip-existing*
2	1	resume	upload			skip
2	2	skip		upload		skip
1	2	upload			skip	skip

Durch Kombination der *Flags* kann fast jeder mögliche Zustand für jedes Feld erreicht werden.

Nun gibt es genau eine Situation, in der es völlig überflüssig ist, die Dateigröße zu ermitteln: Für den Fall, dass alle Felder auf `upload` stehen, wird die Datei garantiert hochgeladen. Ansonsten muss aber die Größe ermittelt werden:

```
if(fsession->resume_table.small_large == RESUME_TABLE_UPLOAD &&
   fsession->resume_table.large_large == RESUME_TABLE_UPLOAD &&
   fsession->resume_table.large_small == RESUME_TABLE_UPLOAD)
```

```
        fsession->target_fsize = 0;
    else if(fsession->local_fname) {
        res = ftp_get_filesize(fsession->ftp, fsession->target_fname, &fsession->target_fsize);
        if(res == ERR_FAILED)
            fsession->target_fsize = 0;
    }
```

Da die Größe der Zieldatei bekannt ist, wird nun, sofern feststeht, dass die Datei übersprungen wird, der Vorgang abgebrochen:

```
    if( fsession->local_fname && (
       (fsession->local_fsize <  fsession->target_fsize &&
                 fsession->resume_table.small_large == RESUME_TABLE_SKIP) ||
       (fsession->local_fsize == fsession->target_fsize &&
                 fsession->resume_table.large_large == RESUME_TABLE_SKIP) ||
       (fsession->local_fsize  > fsession->target_fsize &&
                 fsession->resume_table.large_small == RESUME_TABLE_SKIP)))
    {
        printout(vMORE, "Skipping this file due to resume/upload/skip rules.\n");
        printout(vLESS, "-- Skipping file: %s\n", fsession->local_fname);
        return ERR_SKIP;
    }
```

In der Funktion `set_resuming()` wird als nächstes die Zieldateigröße auf `0` gesetzt, wenn der entsprechende Eintrag des `resume_table` einen Upload vorsieht.

Sofern Timestamping für den Vergleich der Änderungszeitpunkte erwünscht ist, werden diese über `check_timestamp()` verglichen und die Datei gegebenenfalls auch übersprungen:

```
    if(opt.timestamping)
        if(check_timestamp(fsession))
            return ERR_SKIP;
```

Als nächstes wird versucht, den Übertragungsmodus der Datei zu setzen. Normalerweise ist die Voreinstellung `TYPE_UNDEFINED` und in diesem Fall wird der Übertragungsmodus anhand des Dateityps festgemacht. Die derzeit festgelegten *Ascii*dateitypen sind folgende:

```
txt, c, java, cpp, sh, f, f90, f77, f95, bas, pro, csh, ksh, conf, htm, html,
php, pl, cgi, inf, js, asp, bat, cfm, css, dhtml, diz, h, hpp, ini, mak, nfo,
shtml, shtm, tcl, pas
```

Alle anderen Dateitypen werden als *Binary* behandelt. Für den Fall, dass über eine Kommandozeilenoption der Übertragungsmodus festgelegt wird, wird die Voreinstellung zu `TYPE_A` (*Ascii*) oder `TYPE_I` (*Binary*), sodass keine automatische Bestimmung notwendig ist. Über `ftp_set_type()` wird letztendlich dem FTP-Server der Übertragungsmodus mitgeteilt.

```
    if(fsession->binary == TYPE_UNDEFINED)
        fsession->binary = get_filemode(fsession->target_fname);

    ftp_set_type(fsession->ftp, fsession->binary);
```

Nun steht der Übermittlung nicht mehr viel im Weg. In der Funktion `do_send()` wird nun die Datenverbindung aufgebaut, über den `REST` Befehl das Resuming sofern notwendig eingeschaltet und anschließend werden die Daten in Paketen zu einem Kibibyte über die Datenverbindung geschickt.

Das sind die wesentlichsten Funktionen von `fsession_transmit_file()`. Die in den Erläuterungen nicht beachteten Teile sind die Überprüfung der Rückgabewerte, sodass bei Fehlern die Verbindung neu aufgebaut wird. Zu diesem Zweck laufen die beschriebenen Prozesse in einer Schleife ab, die entweder bei einer erfolgreichen Übermittlung, bei einem permanenten Fehler oder nach dem Ablauf

der Wiederholungsversuche abbricht.

7.3. Das URL-Parsing

```
int parse_url(_fsession * fsession, char *url);
```

Eine FTP-Url hat folgende Bestandteile:

- Benutzername
- Kennwort
- Hostname
- Port
- Verzeichnis
- Dateiname

Davon sind alle bis auf den Hostnamen optional. Typische URLs könnten z.B. folgende sein.

```
a) ftp://www.ftpserver.com
b) ftp://www.ftpserver.com/pub/src/
c) ftp://donald:entenbrust@www.ftpserver.com:21/pub/src/README
```

Das `ftp://`-Präfix identifiziert die URL als FTP-Url und ist für das Parsing an sich irrelevant. Diese 6 Zeichen können also in jedem Fall übersprungen werden. Da auf der URL ein bisschen gearbeitet wird, wird eine Kopie erstellt, um die Originaldaten nicht zu kompromittieren:

```
url = cpy(url+6);
```

Nun wird die URL an den markantesten Punkten aufgeteilt. Der vorerst wichtigste ist der Slash. Durch den die URL in zwei Teile geteilt wird. Für c) wären das folglich:

```
d) donald:entenbrust@www.ftpserver.com:21
k) pub/src/README
```

In der Implementation wird über die Funktion `strchr(3)` ein Pointer auf das erste Auftreten des Slashs gefunden. Das Byte an dieser Stelle wird `0` gesetzt, wodurch zwei Teilstrings entstehen. Die Position des zweiten Teilstrings (also k)) wird in der Variablen `path` für die spätere Verwendung gespeichert. Wird kein Slash gefunden, bleibt auch `path` auf `NULL` gesetzt:

```
char * path = NULL;
char * d;
d = strchr(url, '/');
if(d) *d = 0, path = d + 1;
```

Der nächste Delimiter im ersten Teilstring ist das `@`, welches auch wieder durch einen Nullchar ersetzt wird. Somit wird aus d) Folgendes:

```
e) donald:entenbrust
h) www.ftpserver.com:21
```

Da Benutzer häufiger das `@` im Benutzernamen oder Kennwort verwenden, das `@` aber garantiert nie im Hostnamen oder Port auftreten wird, wird mit der `strrchr(3)` das letzte Auftreten dieses Zeichens aufgespürt. Der zweite Teilstring (h)) wird in `host` gespeichert:

```
d = strrchr(url, '@');
if(d) *d = 0, host = d + 1;
else  host = url;
```

Sofern überhaupt ein @ gefunden wurde, erfolgt nun dieselbe Prozedur erneut für den Benutzernamen und das Kennwort, wobei der Doppelpunkt der Delimiter ist:

```
f) donald
g) entenbrust
```

Wird der Doppelpunkt nicht gefunden, so ist nur der Benutzername angegeben (wie z.B. bei `ftp://user@host/`). Die bereitstehenden Daten werden aus der URL kopiert und in den Informationen zur *fsession* gespeichert:

```
if(d) {
    d = strchr(url, ':');
    if(d) {
        *d = 0;
        fsession->user = cpy(unescape(url));
        fsession->pass = cpy(unescape(d+1));
    } else
        fsession->user = cpy(unescape(url));
}
```

Nun wird dieselbe Aktion noch für den Hostteil durchgeführt, bei dem der Doppelpunkt den Hostnamen und den Port teilt:

```
i) www.ftpserver.com
j) 21
```

Sofern ein Port gefunden wurde, wird dieser über die `atoi(3)` Funktion in eine Zahl umgewandelt.

```
d = strchr(host, ':');
if(d)
    *d = 0,
    fsession->host->port = atoi(d + 1);
```

Über die Funktion `get_ip_addr()` wird die zu dem Hostnamen gehörige IP-Adresse ermittelt:

```
get_ip_addr(host, &fsession->host->ip);
```

Nachdem der Hostname gefunden wurde, kann, sofern kein Kennwort spezifiziert wurde, in der Kennwortliste nach einem zu diesem Host und Benutzer gehörigem Kennwort gesucht werden. In vielen Umgebungen ist es nämlich unsicher, auf der Kommandozeile ein Kennwort im Klartext anzugeben, da dies meist in der Prozessübersicht auch von anderen Benutzern des Systems gelesen werden kann. Deshalb existiert in Wput seit Version 0.5 eine Kennwortdatei in die Benutzernamen und Kennwörter zu FTP-Servern eingetragen werden können, die Wput dann automatisch ergänzt. Diese Liste wird bei Programmstart eingelesen und ein zu einem Host und Benutzer gehörendes Element kann über die Funktion `password_list_find()` gefunden werden. Wurden in der Kennwortdatei keine Einträge zu dem Host gefunden und ist auch der Benutzer nicht angegeben (wie z.B. in a) oder b)), so wird der Benutzer gemäß Spezifikation auf `anonymous` gesetzt:

```
if(!fsession->pass) {
        password_list * P = password_list_find(opt.pl, host, fsession->user);
        if(P) {
            if(!fsession->user) fsession->user = cpy(P->user);
            fsession->pass = cpy(P->pass);
        } else if(!fsession->user) {
            fsession->user = cpy("anonymous");
            fsession->pass = cpy(opt.email_address);
        }
}
```

Als letztes Element bleibt der Pfadanteil zu parsen. Dabei ist das Zielverzeichnis alles, was vor dem

letzten Slash kommt. Der danach folgende Teil ist dann der Dateiname:

```
    if(!path) {
        free(url);
        return 0;
    }

    /* last '/' is where the filename starts */
    d = strrchr(path, '/');
    if(d)
        *d = 0,
        fsession->target_dname = cpy(path);
    else
        d = path-1;

    if(*++d)
        fsession->target_fname = cpy(unescape(d));

    free(url);
    return 0;
}
```

Für das Beispiel c) würden somit folgende Attribute der *fsession* gesetzt:

```
host->ip            ip von 'www.ftpserver.com'
host->hostname      'www.ftpserver.com'
host->port          21
user                'donald'
pass                'entenbrust'
target_dname        'pub/src'
target_fname        'README'
```

7.4. Der Bau einer fsession

```
_fsession * build_fsession(char * file, char * url);
```

Aus den Informationen, die die globalen Optionen, eine URL und u.U. eine dazugehörige Datei bieten, sollen brauchbare Werte für eine *fsession* erstellt werden. `build_fsession()` ist somit der Konstruktor für die *fsessions*, der diese alloziert und mit Daten belegt. Als erstes werden alle Daten mit `0` initialisiert, um anschließend bekannte Optionen zu setzen und danach über `parse_url()` die aus der URL ableitbaren Daten zu belegen:

```
    _fsession * fsession = malloc(sizeof(_fsession));

    memset(fsession, 0, sizeof(_fsession));
    fsession->binary       = opt.binary;
    fsession->retry        = opt.retry;
    fsession->resume_table =&opt.resume_table;

    if( parse_url(fsession, url) == ERR_FAILED) {
        printout(vLESS, "Error: error while parsing url (%s)\n", url);
        free_fsession(fsession);
        return NULL;
    }
```

Als nächstes erfolgt die aufwendige Aufgabe, den Dateinamen aus den URL-Bestandteilen abzuleiten, sofern dieser nicht in `file` spezifiziert wurde.

Für den Sonderfall, dass weder ein Zielverzeichnis noch ein Zieldateiname angegeben wurde, wird das aktuelle Verzeichnis als Zielverzeichnis angenommen, wobei eine Warnung des Benutzers in diesem Fall durchaus angebracht ist:

```
if(!fsession->target_dname && !fsession->target_fname) {
    printout(vLESS, "Warning: Neither a remote location nor a local filename "
        "has been specified. Assuming you want to upload the "
        "current working directory to the remote server.\n");
    file = cpy(".");
}
```

In den anderen Fällen wird nun aus Zielverzeichnis und Zieldateiname mit ein wenig zauberhaftem C-Code eine zusammengesetzte Zeichenfolge erstellt:

```
tmp[1] = file = (char *) malloc(
     (fsession->target_dname ? strlen(fsession->target_dname) + 1 : 0) +
     (fsession->target_fname ? strlen(fsession->target_fname)     : 0) + 1);
file[0] = 0;
if(fsession->target_dname) {
    strcpy(file, fsession->target_dname);
    strcat(file, "/");
}
if(fsession->target_fname)
    strcat(file, fsession->target_fname);
```

Da diese noch fast in der Originalform der URL besteht, muss diese über die `unescape()` Funktion bearbeitet werden.

```
 unescape(file);
```

Unter unixoiden Betriebssystemen ist `file` nun etwas wie `pub/src/README` und verweist u.U. auf eine im lokalen Dateisystem real existierende Datei. Unter Windows ist die Zeichenfolge völlig unbrauchbar und es müssen zunächst die Backslashs durch Slashs ersetzt werden, damit `pub\src\README` herauskommt:

```
#ifdef WIN32
    do if(*file == '/') *file = dirsep;
    while(*file++);
    if(*(file-=2) == dirsep) *file = 0;
    file = tmp;
#endif
```

Über die Funktion `stat(2)` können Informationen über eine Datei herausgefunden werden und da es einen Fehler gibt, wenn die Datei nicht existiert, kann so auch indirekt überprüft werden, ob eine Datei vorhanden ist. Genau dieser Vorgang wird nun mehrfach wiederholt, bis entweder eine Datei gefunden wurde oder die letzte Möglichkeit durchprobiert wurde. Dabei wird mit `file` angefangen (`pub/src/README`), von dort der erste Slash gesucht und an dessen Stelle (`src/README`) erneut versucht eine Datei zu finden.

```
while( stat(file, &statbuf) != 0 ) {
    file = strchr(file, dirsep);
    if(!file) {
         file = "";
         break;
    } else
         file++; /* we don't want a leading dirsep */
}
file = cpy(file);
free(tmp);
```

An dieser Stelle sind nun gleiche Voraussetzungen geschaffen. Auch wenn `build_fsession()` ohne das Argument `file` aufgerufen wurde, ist dieses nun definiert. Nun wird noch einmal vollkommen

1 Da `file` selbst u.U. als Pointer noch verändert wird, wird `tmp` benötigt, um ein Backup auf den echten Puffer zu haben. Nicht zuletzt weil der mit `malloc(3)` reservierte Speicherbereich am Ende auch wieder freigegeben werden muss.

unabhängig von dem Vorhergegangenen die Existenz dieser Datei überprüft:

```
    if(stat(file, &statbuf) != 0) {
        printout(vLESS, "Error: File `%s' does not exist. Don't know what to do about this URL.\n",
file);
        free(file);
        free_fsession(fsession);
        return NULL;
    }
```

Existiert die Datei, so gibt es noch die Möglichkeit, dass es sich bei der Datei um ein Verzeichnis handelt. In diesem Fall fügt die Funktion `queue_add_dir()` für jede Datei in dem Verzeichnis eine neue Kombination aus URL und Datei in die Warteschlange. Die *fsession* selbst wird wieder freigegeben:

```
    if( S_ISDIR[1](statbuf.st_mode) ) {
        queue_add_dir(file, url, fsession);
        free_fsession(fsession);
        free(file);
        return (void *) -2;
    }
```

Da inzwischen bekannt ist, dass es sich um eine Datei handelt, können nun basierend auf den Informationen, die `stat(2)` zurückgegeben hat, weitere Eigenschaften der *fsession* gesetzt werden:

```
    fsession->local_fname = file;
    fsession->local_fsize = statbuf.st_size;
    fsession->local_ftime = statbuf.st_mtime;
```

Für den Fall, dass noch kein Zieldateiname spezifiziert wurde, wird dieser über ein bisschen magische Pointerarithmetik vom Quelldateinamen abgeleitet. Auch das Zielverzeichnis wird u.U. angepasst. Wenn die URL also `ftp://host/pub/usr/` ist und `file src/README`, so wird `target_dname` von `pub/usr` in `pub/usr/src` verwandelt und `target_fname` auf `README` gesetzt:

```
    if(!fsession->target_fname && strchr(fsession->local_fname, dirsep)) {
        int slashlen = strrchr(fsession->local_fname, dirsep) - fsession->local_fname;
        if(fsession->target_dname) {
            fsession->target_dname = realloc(fsession->target_dname,
                strlen(fsession->target_dname) + 1 +
                slashlen + 1);
            strcat(fsession->target_dname, "/");
        } else {
            fsession->target_dname = malloc(slashlen + 1);
            *fsession->target_dname = 0;
        }
        strncat(fsession->target_dname, fsession->local_fname, slashlen);
        fsession->target_fname = cpy(basename(fsession->local_fname));
    } else if(!fsession->target_fname)
        fsession->target_fname = cpy(fsession->local_fname);
```

Zu guter Letzt wird die nun fertige *fsession* zurückgegeben:

```
    return fsession;
```

1 `S_ISDIR` ist ein Makro, das anhand der Informationen, die der Aufruf von `stat(2)` zurückgegeben hat, prüft, ob die Datei ein Verzeichnis ist.

8. Main und andere Kleinigkeiten

Diese Sektion widmet sich einigen restlichen Funktionen, zeigt Grundprinzipien der Implementierung auf und versucht, Licht in weitere Bereiche von Wput zu werfen.

8.1. `main`

```
int main(int argc, char *argv[]);
```

`main` ist die Hauptfunktion. Diese wird aufgerufen, wenn das Programm ausgeführt wird. In ihr werden globale Optionen sowie andere Bibliotheken (OpenSSL oder die Socketlibrary unter Windows) initialisiert. Des Weiteren wird ein grober Programmablauf skizziert, in dem zentrale Funktionen aufgerufen werden. So werden die Konfigurationsdateien über die Funktion `readwputrc()` eingelesen und anschließend die Kommadozeilenoptionen in `commandlineoptions()` gesetzt. Danach werden anhand einiger Optionen weitere Einstellungen getroffen. So wird für den Fall, dass Wput im Hintergrund laufen soll, der Prozess von Wput per `fork(2)` in einen von seinem *parent* unabhängigen Prozess transformiert und anschließend die Ein- und Ausgabedeskriptoren (`stdin`, `stdout`, `stderr`) freigegeben, sodass sich der Prozess von der Konsole lösen kann.

Mit den wichtigsten Informationen ausgestattet werden nun alle bereits bekannten Kombinationen aus URL und Datei übermittelt. Ist per Kommandozeilenoption angegeben worden, dass die URLs auch aus einer Eingabedatei gelesen werden, so geschieht dies in `read_urls()`. Alle verbleibenden einzelnen URLs oder Dateinamen werden anschließend in `process_missing()` bearbeitet (siehe 8.2.).

Nachdem alle Dateien übermittelt worden sind, wird die letzte bestehende FTP-Verbindung geschlossen, sofern dies noch nicht geschehen ist. Zu guter Letzt werden die Aufräumarbeiten gestartet, in denen andere noch reservierte Speicherbereiche freigegeben werden.

8.2. Die Queue-Struktur

Eine Kombination aus URL und Dateiname ist nicht automatisch vorhanden, weswegen die Queue aus diesen beiden Einträgen flexibel gestaltet ist. Von der Eingabedatei bzw. der Kommandozeile wird entweder eine URL oder ein Dateiname gelesen. Werden fünf Dateinamen gelesen, so werden fünf Queueobjekte erstellt, in denen die URL auf `NULL` gesetzt ist. Werden nun z.B. sechs URLs eingelesen, so werden zuerst die ersten fünf Queueobjekt mit URLs gefüllt und zum Schluss ein neues Queueobjekt angefügt, in dem diesmal der Dateiname auf `NULL` gesetzt ist. Sobald eine Kombination aus URL und Dateiname fertig ist, wird daraus eine *fsession* zu erstellen versucht.

Die letzte verwendete URL merkt sich Wput immer. Für den Fall, dass nach dem Einlesen aller Quellinformationen noch Einträge ohne URL übrig sind, wird in der Funktion `process_missing()` diesen Einträgen die letzte verwendete URL hinzugefügt.

Für den Aufruf von `wput 1.txt 2.txt ftp://host.de/` würden sich folgende Aufrufe und entsprechende Queues ergeben:

```
queue_add_file('1.txt')
   # ('1.txt', NULL) => NULL
queue_add_file('2.txt')
   # ('1.txt', NULL) => ('2.txt', NULL) => NULL
```

```
queue_add_url('ftp://host.de/')
    # ('1.txt', 'ftp://host.de/') => ('2.txt', NULL) => NULL
    # Übermittlung des ersten Eintrags:
    # ('2.txt', NULL) => NULL
process_missing()
    # ('2.txt', 'ftp://host.de/') => NULL
    # Übermittlung des zweiten Eintrags
```

Sind am Ende einzelne URLs übrig, so ist das nicht weiter störend. Diese werden direkt zur Übermittlung weitergegeben, wobei der Dateiname wie in 7.4. beschrieben aus der URL abgeleitet wird.

8.3. Parsing der Kommandozeilenparameter

```
void commandlineoptions(int argc, char * argv[]);
```

In der Funktion `commandlineoptions()` werden die Kommandozeilenargumente ausgewertet und in entsprechende Aktionen umgewandelt. `argc` und `argv` sind die vom Betriebssystem übergebenen Argumente. `argc` ist die Anzahl und `argv` ist ein Array von Zeichenfolgen. Für das Parsen dieser Argumente gibt es die `getopt(3)`-Bibliothek, die in sehr vielen Konsolenanwendungen genutzt wird.

Im Wesentlichen gibt man an, welche Optionen zugelassen sind, und ob diese Optionen ein Argument benötigen. `getopt_long(3)` gibt dann für jede in den Parametern erkannte Option einen dazugehörigen Nummerncode mit und setzt die globale Variable `optarg` auf das entsprechende (sofern vorhandene) Argument. Anhand der Nummerncodes werden dann Schlüsse gezogen. So werden *Flags* überschrieben und Optionen gesetzt, die das Verhalten von Wput beeinflussen.

8.4. `printout`

Die Funktion `printout()` ist die zentrale Ausgabefunktion. Für nahezu jede Textmeldung, die der Benutzer zu sehen bekommt, ist `printout()` verantwortlich. `printout()` sorgt insbesondere für zwei Dinge: Einerseits werden die Informationen in die über die Kommandozeilenparameter spezifizierte Ausgabedatei geschrieben (was, sofern nicht anders angegeben, `stdout` ist). Andererseits werden die Meldungen nur dann ausgegeben, wenn ein bestimmtes *verbosity level* erreicht ist. Es gibt fünf verschiedene Stufen der Ausgabe:

Bezeichnung	***Beschreibung***
`vDEBUG`	Alle Ausgaben werden zugelassen. Dieses Level ist vor allem sinnvoll. um genau nachvollziehen zu können, was das Programm wann tut, um daraus ein eventuell auftretendes Fehlverhalten aufspüren zu können.
`vMORE`	Mehr Ausgaben als normal. Es werden insbesondere ein paar weniger wichtige Warnungen ausgegeben sowie zu fast allen FTP-Befehlen Nachrichten erzeugt.
`vNORMAL`	Die Standardausgaben zeigen dem Nutzer die wichtigsten Meldungen und geben einen guten Überblick über die von Wput durchgeführen Aktionen.
`vLESS`	So wenig Ausgaben wie möglich. Es werden nur gravierende Warnungen und keine Statusmeldungen ausgegeben. Zu jeder transferierten Datei werden nur die anfänglichen Informationen sowie eine Erfolgs- oder Fehlermeldung ausgegeben.

Bezeichnung	*Beschreibung*
`quite`	Alle Ausgaben werden unterdrückt.

Um dies möglich zu machen, wird bei jedem Aufruf von `printout()` in Wput das mindeste *verbosity level* angegeben.

Durch diesen Mechanismus lassen sich z.B. für den weiteren Gebrauch in Scripts, überflüssige Ausgaben herausfiltern oder zur Fehlerdiagnose oder zur Dokumentation der Arbeitsschritte weitere Ausgaben aktivieren.

8.5. Die FTP-LS-Bibliothek

Es gibt immer noch genug alte FTP-Server, die Kommandos wie `MDTM` oder `SIZE` nicht verstehen und bei denen Dateiinformationen nur über das Verzeichnislisting erhalten werden können. Da diese Verzeichnislistings nie standardisiert wurden, sehen diese auf nahezu jedem Betriebssystem unterschiedlich aus. Kluge Köpfe haben sich eines Tages hingesetzt und eine kleine Bibliothek geschrieben, die die am häufigsten verwendeten Formate erkennt und daraus für den Programmierer brauchbare Informationen liefert.

Die Bibliothek wurde nahezu 1:1 aus Wget übernommen. Da Wget aber im Gegensatz zu Wput das Verzeichnislisting in einer Datei speichert (Wput behält es ausschließlich im Speicher), sind ein paar kleine Modifikationen notwendig, die sich größtenteils über `#define` Direktiven ausdrücken lassen und in `wget.h` verzeichnet sind.

8.6. Memorydebugging

Neben Syntaxfehlern sind Pufferüberläufe, falsch freigegebene Puffer oder fehlende Freigaben eine der häufigsten Fehlerursachen. Da sich diese allerdings nur mit sehr viel Mühe aufspüren lassen, werden in Wput zum Zweck des Debugging die wichtigsten Speicherfunktionen wie `malloc(3)`, `realloc(3)`, `free(3)`, `strcat(3)` und `cpy()` durch entsprechende Debugfunktionen ersetzt, die entsprechende Aufrufe protokollieren und dabei auch die Quelle (also Quelldatei und Zeilennummer) angeben. `dbg_malloc()` verwaltet dabei eine Liste aller allozierten Puffer und `dbg_free()` prüft anhand dieser Liste vor jeder Freigabe, ob an der freizugebenden Adresse wirklich ein Speicherbereich reserviert wurde. Dadurch lassen sich fehlerhafte Aufrufe von `free(3)` herausfiltern.

Am Ende eines Durchlaufs von Wput wird darüber hinaus eine Liste aller nicht freigegebenen Speicher ausgegeben, sodass gezielt nach Memoryleaks gesucht werden kann, um diese zu beseitigen.

Im normalen Gebrauch von Wput sind diese Funktionen allerdings absolut überflüssig und für die Performance nur hinderlich, weswegen die Funktionen auch nur zu Debugzwecken einkompiliert werden. Hierzu lässt sich dem beiliegenden `configure`-Script die Option `--enable-memdbg` übergeben, sodass beim Kompilieren die entsprechenden Funktionsaufrufe umgeleitet werden.

8.7. Skiplists

Für den Fall, dass die Verbindung zu einem Host fehlschlägt, wäre es unsinnig die Verbindung für die nächste Datei erneut aufzubauen. Dafür wurde eine im Code als `skiplist` bezeichnete, einfach

verkettete Liste eingebaut:

```
typedef struct _skiplist {
    int                   ip;
    char *                host;
    unsigned short int    port;
    char *                user;
    char *                pass;
    char *                dir;
    struct _skiplist *  next;
} skiplist;
```

Für jede fehlgeschlagene Datei wird anhand der Ursache des Fehlschlags ein entsprechender Eintrag in die `skiplist` eingefügt. Schlägt die Verbindung zum Host fehl, so werden Hostname bzw. IP und Port belegt, während die anderen Felder undefiniert bleiben.

Schlägt der Login-Prozess für einen bestimmten Benutzer fehl, so wird neben dem Host auch noch der Benutzername und das Kennwort gesetzt, sodass Wput gar nicht erst versucht, sich mit denselben Benutzerdaten erneut einzuloggen.

Schlägt das Wechseln des Verzeichnisses fehl, so wird entsprechend auch das Zielverzeichnis gesetzt, sodass alle Dateien, die innerhalb dieses Zielverzeichnisses liegen, übersprungen werden.

9. Praktische Anwendung

9.1. Spezifikation

```
Usage: wput [options] [file]... [url]...
  url        ftp://[username[:password]@]hostname[:port][/[path/][file]]

Startup:
  -V, --version         Display the version of wput and exit.
  -h, --help            Print this help-screen
  -b, --background      go to background after startup

Logging and input file:
  -o,  --output-file=FILE      log messages to FILE
  -a,  --append-output=FILE    append log messages to FILE
  -q,  --quiet                 quiet (no output)
  -v,  --verbose               be verbose
  -d,  --debug                 debug output
  -nv, --less-verbose          be less verbose
  -i,  --input-file=FILE       read the URLs from FILE
  -s,  --sort                  sorts all input URLs by server-ip and path
  -I,  --input-pipe=COMMAND    take the output of COMMAND as data-source
  -R,  --remove-source-files   unlink files upon successful upload

Upload:
       --bind-address=ADDR     bind to ADDR (hostname or IP) on local host
  -t,  --tries=NUMBER          set retry count to NUMBER (-1 means infinite)
  -nc, --dont-continue         do not resume partially-uploaded files
  -u,  --reupload              do not skip already completed files
       --skip-larger           do not upload files if remote size is larger
       --skip-existing         do not upload files that exist remotely
  -N,  --timestamping          don't re-upload files unless newer than remote
  -T,  --timeout=10th-SECONDS  set various timeouts to 10th-SECONDS
  -w,  --wait=10th-SECONDS     wait 10th-SECONDS between uploads. (default: 0)
       --random-wait           wait from 0...2*WAIT secs between uploads.
       --waitretry=SECONDS     wait SECONDS between retries of an upload
  -l,  --limit-rate=RATE       limit upload rate to RATE
  -nd, --no-directories        do not create any directories
  -Y,  --proxy=http/socks/off  set proxy type or turn off
       --proxy-user=NAME       set the proxy-username to NAME
       --proxy-pass=PASS       set the proxy-password to PASS

FTP-Options:
  -p,  --port-mode             no-passive, turn on port mode ftp (def. pasv)
  -A,  --ascii                 force ASCII  mode-transfer
  -B,  --binary                force BINARY mode-transfer
       --force-tls             force the useage of TLS

See the wput(1) for more detailed descriptions of the options.
Mail bug reports and suggestions to <itooktheredpill@gmx.de>
```

Eine genauere Beschreibung dieser Optionen findet sich in der Manual Page zu Wput unter `wput(1)` oder http://itooktheredpill.dyndns.org/wput/wput.1.html.

9.2. Beispiel 1 – Eine Datei

Die wohl häufigste Anwendung von Wput, ist die zum Upload einer einzelnen Datei:

```
# wput ftp://ftp:shamous@rootboard.org/incoming/ wput-devel.tgz
--19:40:57-- `wput-devel.tgz'
    => ftp://ftp:xxxxx@217.172.179.237:21/incoming/wput-devel.tgz
Connecting to 217.172.179.237:21... connected!
Logging in as ftp ... Logged in!
Length: 373,754
    0K ......... ......... ......... ......... .........  13% (null)
   50K ......... ......... ......... ......... .........  27% 71.00 KiB/s
  100K ......... ......... ......... ......... .........  41% 55.00 KiB/s
  150K ......... ......... ......... ......... .........  54% 47.25 KiB/s
  200K ......... ......... ......... ......... .........  68% 44.20 KiB/s
  250K ......... ......... ......... ......... .........  82% 41.00 KiB/s
  300K ......... ......... ......... ......... .........  95% 40.12 KiB/s
  350K ......... ....
19:41:10 (wput-devel.tgz) - `30.52K/s' [373754]
```

Oder die Fortsetzung eines abgebrochenen Uploads, wobei bei diesem Beispiel der Dateiname aus der URL abgeleitet wird:

```
# wput ftp://ftp:shamous@rootboard.org/incoming/wput-devel.tgz
--19:44:52-- `wput-devel.tgz'
    => ftp://ftp:xxxxx@217.172.179.237:21/incoming/wput-devel.tgz
Connecting to 217.172.179.237:21... connected!
Logging in as ftp ... Logged in!
Length: 373,754 [205,818 to go]
       [ skipped 164K ]
  150K ,,,,,,,,,, ,,,,..... ......... ......... .........  54% (null)
  200K ......... ......... ......... ......... .........  68% 71.00 KiB/s
  250K ......... ......... ......... ......... .........  82% 55.00 KiB/s
  300K ......... ......... ......... ......... .........  95% 46.00 KiB/s
  350K ......... ....
19:45:00 (wput-devel.tgz) - `30.39K/s' [373754]
```

9.3. Beispiel 2 – Ein ganzes Verzeichnis

Hierbei wird ein komplettes Verzeichnis übermittelt. Das Zielverzeichnis existiert noch nicht und wird entsprechend erstellt. Die Verbindung verläuft verschlüsselt. Die Ausgabe ist etwas wortreicher (*more verbose*).

```
# wput -v ftp://ftp:shamous@rootboard.org/incoming/src/getopt
--22:04:36-- `src/getopt/getopt.c'
    => ftp://ftp:xxxxx@217.172.179.237:21/incoming/src/getopt/getopt.c
Connecting to 217.172.179.237:21... connected!
==> AUTH TLS ... done (communication is now encrypted!)
Logging in as ftp ... Logged in!
==> CWD incoming/src/getopt failed (incoming/src/getopt: No such file or
directory).
==> CWD incoming
==> CWD src failed (src: No such file or directory).
==> MKD src
==> CWD src
==> MKD getopt
```

```
==> CWD getopt
Remote file size is smaller than local size. Restarting at 0
==> TYPE A ... done.
Setting data protection level to private... done.
==> PASV done.
==> STOR getopt.c done.
Length: 34,560
    0K ......... ......... ......... ...
22:04:39 (getopt.c) - `26.99K/s' [34560]

--22:04:39-- `src/getopt/getopt.h'
    => ftp://ftp:xxxxx@217.172.179.237:21/incoming/src/getopt/getopt.h
Remote file size is smaller than local size. Restarting at 0
==> TYPE I ... done.
Setting data protection level to private... done.
==> PASV done.
==> STOR getopt.h done.
Length: 6,424
    0K ......
22:04:40 (getopt.h) - `22.05K/s' [6424]

FINISHED --22:04:41--
Transfered 40,984 bytes in 2 files ` 8.85K/s'
```

9.3. Beispiel 3 – Automatisiertes Backup über eine Pipe

Bei diesem Beispiel erzeugt der Befehl `tar cz wput/` ein tar.gz Archiv des `wput/` Verzeichnisses, welches von `tar` auf die Standardausgabe geschrieben wird. Wput hat keine Eingabedatei, aber dafür wurde eine *input-pipe* angegeben. Eine *input-pipe* ist ein Befehl, den Wput ausführt und von dessen Ausgabe Wput Daten einliest, um diese anschließend hochzuladen. Da Wput dem Kommando noch einige Informationen übergibt (Benutzername, Hostname, Port, Verzeichnis und Dateiname), diese aber in diesem Beispiel nicht benötigt werden, werden diese über den `echo`-Befehl nach `/dev/null` (also ins Nirgendwo) verfrachtet. Der `cat` Befehl wiederum liest einfach nur Daten von der Standardeingabe ein und gibt diese auf seiner Standardausgabe wieder aus. Da er im Kontext von Wput aufgerufen wird, liest er also die Eingabedaten, die an Wput übermittelt werden und gibt diese aus, sodass Wput diese lesen und hochladen kann:

```
# tar cz wput/ | wput -I "cat; echo > /dev/null" \
  ftp://ftp:shamous@rootboard.org/incoming/wput-snapshot.tgz

--22:24:59-- `'
    => ftp://ftp:xxxxx@217.172.179.237:21/incoming/wput-snapshot.tgz
Connecting to 217.172.179.237:21... connected!
Logging in as ftp ... Logged in!

    0K ......... ......... ......... ......... .........   0% (null)
   50K ......... ......... ......... ......... .........   0% 60.00 KiB/s
  100K ......... ......... ......... ......... .........   0% 70.50 KiB/s
  150K ......... ......... .........
22:25:28 (wput-snapshot.tgz) - `30.44K/s' [184320]
```

10. Auswertung

10.1. Erreichtes

Als die Entwicklung von Wput begann, war nicht abzusehen, wo dies hinführen würde und mit dem jetzt entstanden Ergebnis hat sicherlich niemand gerechnet. Insbesondere der Punkt der Verschlüsselung war lange Zeit nur ein Punkt auf der `TODO` Liste, von dem angenommen wurde, dass er sowieso nie implementiert werden würde und doch hat sich das Gegenteil herausgestellt.

In der nun schon etwas längeren Zeit, die Wput existiert, hat sich das Programm immer weiterentwickelt und ist inzwischen ein recht bekanntes, robustes Tool, das sich zunehmender Beliebtheit erfreut. Mit Version 0.5 wurde Wput sehr stabil und benutzbar und nicht zuletzt deswegen meldeten sich Maintainer verschiedenster Distributionen und erstellten Ports und Binärpakete. So ist Wput u.a. mit Binärpaketen für Debian, Gentoo, MacOS, FreeBSD, SuSE und Windows verfügbar und läuft auf verschiedensten Rechnerarchitekturen.

Wput wird international genutzt. So hat der Support bereits mit Leuten aus Deutschland, Belgien, den Niederlanden, Dänemark, Polen, Russland, Frankreich, den USA und Kanada zu tun gehabt, was auch nur einen Bruchteil der tatsächlichen Benutzung repräsentiert.

Wput ist nicht das Voodooprogramm, das plötzlich alle Probleme der Welt löst und es hat auch nie den Anspruch, ein Programm zu sein, dass jeder benötigt und das Rekorddownloadzahlen schreibt. Wput ist für einen bestimmten Zweck geschaffen worden, den es zuverlässig erfüllt. Und die Downloadzahlen die zwischen 10 und 200 täglich schwanken, zeugen auch von einer andauernden Beliebtheit.

10.2. ... und Probleme

Die Entwicklung von Wput war ein langer Weg. Durch die steigende Komplexität des Programms, wird es immer schwieriger das gesamte Projekt zu überblicken. Das größte Problem sind bei der Programmierung in C neben den üblichen logischen Fehlern die Fehler, die im Zusammenhang mit Zeichenfolgen und jeglichen Speicherreservierungen auftreten. In bestimmten Situationen wird halt ein Puffer plötzlich nicht freigegeben, mehrmals freigegeben oder über seine Grenzen hinaus geschrieben, was zu Programmabstürzen führt. Diese Fehler zu lokalisieren ist ziemlich aufwendig und viele treten nur in sehr speziellen Situationen auf, sodass diese auch sehr oft unentdeckt bleiben. Im schlimmsten Fall führen solche Fehler nicht zu Programmabstürzen, sondern beinflussen den Programmablauf in unbeabsichtigter Weise. So wurde z.B. in `ftp_login()` anfangs `self->user` auf `user` gesetzt und `self->pass` auf `pass`. An sich kein Problem, jedoch stammt `user` aus einer *fsession* und wenn deren Speicher freigegeben wird, verweist `user` auf einen Speicherbereich in dem sich alles befinden kann und natürlich wird dieser Speicherbereich nicht sofort überschrieben, sondern behält den richtigen Inhalt noch eine Weile. Das führt erst später dazu, dass Wput versucht, sich erneut einzuloggen, obwohl es bereits angemeldet ist. Das Problem wurde dann umgangen, indem `self->user` entsprechend auf `cpy(user)` gesetzt wurde, was eine Kopie der Zeichenfolge im Speicher erstellt, die dann unter der Verwaltung des `ftp_con` Objektes liegt. Selbiges gilt für `pass`, nur kann `pass` in bestimmten Situationen ja auch `NULL` sein und `cpy(NULL)` erzeugt einen Segmentation Fault, also ist `self->pass = pass ? cpy(pass) : NULL`.

Viel komplizierter sind dann noch die Probleme, die sich durch kleine Inkonsistenzen und Rechenfehler bei der Verbindung von URLs und Verzeichnissen ergeben. So wird an der falschen Stelle plötzlich ein Slash abgeschnitten oder wenn der Slash fehlt, ein Buchstabe abgeschnitten und schon ist das Verhalten überhaupt nicht mehr vorhersehbar.

Während der Implementierung sind noch eine Reihe von weiteren Problemen aufgetreten, deren Ursachen teilweise bis heute nicht aufgespürt sind. So hat der `select(2)` call in `socket_is_data_readable()` in bestimmten, nicht definierbaren Umgebungen versagt und behauptet, es wären keine Daten vorhanden, obwohl genau das der Fall war. Ein anderes Beispiel ist das Problem, das sich gleich am Anfang der `main()` Funktion befindet, bei dem die Socketbibliothek für Windows zweimal initialisiert werden muss, damit sie überhaupt benutzt werden kann. Solche Fehlerursachen herauszufinden und zu beheben nimmt unglaublich viel Zeit in Anspruch.

Auf exotischen Systemen mit anderen Endianformen, falsch herum laufenden Stacks oder 64bit-Systemen gibt es noch einige Probleme, deren Ursachen um so schwerer zu ergründen sind, da der Entwickler gerade mal die üblichen x86er Systeme zur Verfügung hat. Aber auch dafür wird sich eine Lösung finden.

Mit steigenden Nutzerzahlen steigt auch der Bedarf an Support für ein Programm. Einige wenige Anfragen zu beantworten, ist für einen Programmierer kein Problem, aber wenn sich Fragen häufen, insbesondere, wenn sich dieselben Fragen wiederholen, ist die Supporttätigkeit eher hinderlich.

Andere Projekte haben Mailinglisten für Benutzer und Entwickler, in denen die Benutzer bei den kleinen Problemen helfen und die Entwickler nur dann tätig werden, wenn es wirklich um Dinge geht, bei denen ein tiefer Blick in den Quellcode geworfen werden muss. Wput hat dafür aber noch keine konstante Benutzerbasis und gibt anscheinend auch nicht genug Diskussionsstoff, um eine Mailingliste zu rechtfertigen, zumal auch bisher nur ein Entwickler an dem Projekt arbeitet, was nicht zuletzt auch daran liegt, dass sich Wput noch auf keiner Entwicklerplattform befindet, auf denen anderen die Mitarbeit sehr leicht gemacht wird, wie das z.B. mit CVS auf Plattformen wie SourceForge der Fall ist.

10.3. Ausblick

Dass ein ähnlich großer Punkt der `TODO` Liste wie die SSL-Implementierung in der nächsten Zeit abgearbeitet wird, ist eher unwahrscheinlich, aber in der Ausblick-Sektion darf man ja optimistisch in die Zukunft schauen.

Wünschenswert wäre eine breitere Unterstützung von Protokollen, bei denen es darum geht, Dateien zu übertragen (z.B. Samba, Fish, HTTP PUT etc.).

Sofern man sich zunächst auf das FTP-Protokoll beschränkt, fehlt noch ein Scripting Feature, bei dem es dem Benutzer möglich ist, kleine Funktionen zu schreiben, die bestimmte Aktionen auf dem Server ausführen, wie z.B. das automatische Umbenennen oder Löschen von Dateien. Eine Funktion zur Übernahme der Dateiberechtigungen wäre insbesondere für Webauftritte, die `cgi`-Scripts mit Ausführungsrechten hochladen müssen, wünschenswert.

Immer mehr Anwendungen und Webserver sind IPv6 fähig. Auch wenn es noch Jahre dauern wird, bis sich IPv6 endgültig durchsetzt, wäre eine entsprechende Erweiterung für Wput früher oder später notwendig.

Wput ist bisher nur in der Lage, Dateien zu übertragen, die kleiner als 2GB sind. Das liegt an der Begrenzung der Datentypen, denn ein `signed integer` fasst genau 2^31 Bit. Also ist 2GB die höchste Zahl, die so genutzt werden kann. Da es aber keinen systemunabhängigen und funktionierenden Standard gibt (und diverse Experimente mit solche großen Dateien fehlgeschlagen sind, zumal solche Dateien auch mit anderen Anwendungen nur schwer zu bearbeiten sind), fehlt es hier noch an Möglichkeiten.

Es gibt noch eine Reihe von Kleinigkeiten, die in Wput nicht zur vollständigen Zufriedenheit funktionieren oder ausgebessert werden könnten. Einige wurden in der Datei `TODO` notiert, die meisten stehen aber direkt als Kommentar im Source, sind mit einer `TODO` Direktive versehen und können wie folgt angezeigt werden:

```
cd wput/src
grep TODO *.c
```

Als größeres noch nicht fertig gestelltes Projekt stellt sich derzeit die Portierung von Wput auf das VMS System dar. Ein Ziel, das wohl in naher Zukunft erreicht wird.

Um eine möglichst große Zahl von Benutzern erreichen zu können, wurde die Sprache Englisch für das Interface von Wput gewählt. Allerdings wäre die Unterstützung anderer Sprachen ein wünschenswertes Ziel. Diese *Internationalisierung* erfolgt durch ausgelagerte Sprachdaten, die Anhand der Umgebungsvariablen im Programm dynamisch geladen werden. Die Übersetzung von Wput würde dann durch Benutzer aus anderen Länder geschehen.

Eine immer größer werdende Bedrohung für freie Software stellen Softwarepatente dar, die, sofern sie denn von der EU durchgebracht werden, wohl auch das Aus für Wput bedeuten würden, da selbst trivialste Vorgänge, wie das Anzeigen eines Fortschrittsbalkens[1], durch Patente geschützt sind.

1 Das Patent auf den Fortschrittsbalken war acht Jahre lang patentiert, bis IBM das Patent im Jahre 2003 auslaufen ließ.

11. Quellen

Die wichtigsten Grundlagen der Spezifikation sind das RFC 959, in dem das FTP-Protokoll spezifiziert wurde sowie RFC 943 (Assigned Names and Numbers), RFC 2228 (FTP Security Extensions) und der Entwurf der IETF (http://www.ietf.org/internet-drafts/draft-ietf-ftpext-mlst-16.txt).

Aus dem Wget-Quellcode wurden die FTP-LS-Bibliothek sowie einige Funktionen aus `progress.c`, die sich mit der Betriebssystemunabhängigen Zeitmessung beschäftigen, übernommen. Solch "gestohlener Code" ist in den Quelldateien durch entsprechende Kommentare gekennzeichnet.

Diverse *Manual Pages* zu den einzelnen Systemfunktionen und -bibliotheken sowie Beispielimplementationen von Socket-Funktionen und diverse Beiträge in Mailinglisten haben geholfen sehr viele Fehler zu beseitigen.
Des Weiteren haben Bug Reports von Benutzern von Wput viele Fehler aufgespürt und geholfen, diese zu beheben. Die Kommunikation ist unter http://itooktheredpill.dyndns.org/wput/wail/ einsehbar.
Wenn Patches anderer eingeflossen sind, so ist dies entweder in der entsprechenden Stelle des Quellcodes oder in der ChangeLog des Projektes vermerkt.

12. Anlagen

12.1. Wput

Das Projektergebnis als Quellcode wird nicht angefügt. Wput selbst ist unter folgender Adresse zum Download verfügbar:

http://itooktheredpill.dyndns.org/wput/

Dort finden sich Dokumentation und Quellcode. Der Quellcode selbst ist zusätzlich online im praktischen LXR-System zu begutachten:

http://itooktheredpill.dyndns.org/wput/lxr/source/

Die im Source verwendeten Referenzen zu Funktionen von Wput führen über folgende Adresse zum Quellcode der Funktion, wobei `%s` durch den entsprechenden Funktionsnamen zu ersetzen ist:

http://itooktheredpill.dyndns.org/wput/lxr/ident?v=devel;i=%s

12.2. Funktionsdiagramm

Zur besseren Orientierung im Quelltext ist ein Funktionsdiagramm angefügt, das die Beziehungen zwischen den einzelnen Funktionen sowie den detaillierten Programmaufbau darstellt.
Einige Funktionen wurden außen vor gelassen, da sie sehr häufig verwendet werden und/oder die Übersichtlichkeit des Graphen erheblich stören:

```
printout(), cpy(), printip(), int64toa(), Abort(), socket_close(), legible(),
legible_1(), time_str(), ftp_free_host(), unescape(), hextoi(), retry_wait(),
set_option()
```

Aus denselben Gründen wurden die Funktionen `ftp_issue_cmd()` und `ftp_get_msg()` zu `FTP Framework` zusammengefasst.