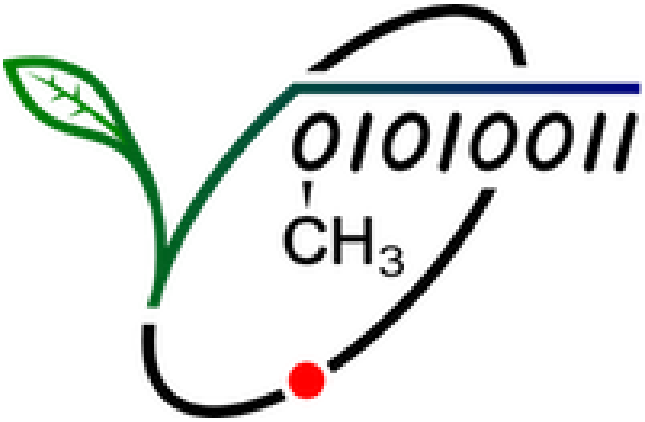
Entwicklung und Implementierung eines Netzwerkprotokolls zur bidirektionalen Kommunikation und Fernsteuerung eines Systems unter Zuhilfenahme einer selbstentwickelten Sprache zur Programmsteuerung



Leonard Petereit
Benedikt Schäfer

Fachbetreuung - Herr J. Süpke
Seminarfachbetreuung - Frau Dr. M. Moor

Januar 2019

Inhaltsverzeichnis

Einleitung

Daten speichern und verwalten ist eine komplexe Aufgabe, die viele kleine Probleme aufwirft. So muss zum Beispiel genug Speicherplatz bereitgestellt werden und das Verschieben, Kopieren, Umbenennen, Auslesen oder Sichern der Daten ist ebenfalls nicht selbstverständlich; letzteres soll oft von jedem beliebigen Ort aus geschehen, da man nicht nur zu Hause auf z.B. Urlaubsbilder oder wichtige Dokumente zugreifen will.
Die einfache Lösung bestünde darin, den Computer zu Hause fernzusteuern - das ist aber entweder nur umständlich mit Funktionen des Betriebssystems zu realisieren - was bei verschiedenen Betriebssystemen noch problematischer wird - oder mit schlecht erweiterbaren grafischen Systemen, die einen fortgeschrittenen Nutzer oft nicht zufrieden stellen. Ein auf einer Ebene dazwischen angesiedeltes System zur Fernsteuerung scheint sinnvoll - also eine betriebssystemunabhängige Software zur Versendung von Steuerbefehlen und Dateien an einen über ein Computernetzwerk verbundenen Zielcomputer, welche dem Nutzer die Möglichkeit gibt, sie nach den seinen spezifischen Ansprüchen selbständig zu erweitern.

Dieses grob formulierte Ziel schlüsselt sich auf in mehrere Unterprobleme. Als Grundlage soll ein Netzwerkprotokoll entwickelt werden, welches unseren gestellten Ansprüchen entsprechend erstellt werden muss. Es sollte optimiert sein für die Übertragung von Anweisungen eines vom Nutzer erweiterbaren Befehlssatzes und Dateien jeglichen Typs; dabei muss es sicherstellen, dass die Daten sicher, fehlerfrei und im richtigen Format ankommen und sowohl Dateien als auch Befehle bei der Übertragung umfassen.
Die Kommunikation mit dem Nutzer geschieht über ein Terminal; die Eingabebefehle sind Teil einer selbst definierten formalen Sprache - damit werden unkomplizierte Bedienung und eine Möglichkeit zur Erweiterung kombiniert. Der Nutzer soll in der Lage sein, den vorhandenen Kanon von Befehlen beliebig zu erweitern und mit eigenen Funktionen zu ergänzen, ohne das gesamte Programm neu kompilieren zu müssen. Deshalb wird die gesamte Eingabeverarbeitung intern von einer plattformunabhängigen Skriptsprache geregelt. Dadurch soll der Nutzer außerdem in der Lage sein, das Programm mit eigenen Skripten auszubauen und Vorgänge im Programm je nach Bedarf zu automatisieren.
Zu guter Letzt muss das Programm modular aufgebaut sein, damit es von allen drei Entwicklern abgesehen von Schnittstellen dazwischen, unabhängig entwickelt werden kann und niemand alle Details überblicken muss. Insgesamt ist unser Ziel also, den Anwender mit einer gering einschränkenden Software bei der Fernsteuerung von Computern zu unterstützen, indem sich um Verbindungsaufbau, sowie Befehls- und Dateiübertragung gekümmert wird und eine Möglichkeit zur nutzerseitigen, systematischen Erweiterung besteht.

1 Funktionsumfang und Eingabeverarbeitung auf der Anwendungsebene

1.1 Entwicklung einer eigenen Scriptsprache zur Anwendungssteuerung

1.1.1 Theoretische Grundlagen formaler Sprachen

Wir als Menschen haben die gesprochenen Sprachen entwickelt, um uns untereinander zu verständigen und Informationen auszutauschen. Um Missverständnisse zu vermeiden, ist unsere Kommunikation durch strikte grammatikalische und semantische Regeln definiert. Es werden strukturelle Merkmale, wie zum Beispiel die Reihenfolge einzelner Sprachteile festgelegt, außerdem gibt es einen begrenzten Wortschatz, in dem jedes Wort eine bestimmte Bedeutung hat. Um auch Computern Informationen zu übermitteln, und eine maschinelle Verarbeitung dieser zu ermöglichen, benutzt man formale Sprachen. Diese gleichen von den grundlegenden strukturellen Ansprüchen den natürlichen Sprachen, verfügen also auch über eine Syntax und eine Semantik, definieren sich allerdings dadurch, dass diese für einen Rechner eindeutig verständlich sein müssen, und demnach kein Interpretationsfreiraum gelassen wird. Ziel ist es klare Anweisungen in einer für den Computer verständlichen Weise zu übermitteln.

Da Sprachen in der Regel durch das Aneinanderreihen von einzelnen Sprachelementen über eine Unendlichkeit verfügen, nutzt man künstliche Grammatiken der Form $G = (N, T, P, s)$, um diese zu beschreiben. Eine Grammatik G dieser Art verfügt über eine Menge von nichtterminalen Symbolen N, auch Variablen genannt, welche im Verlauf der Wortbildung durch die Menge der Produktionsregeln P der Form $U \to V$ durch eine Kombination aus terminalen Zeichen der Menge T ersetzt werden. Das Startsymbol s ist zwingend die erste Variable, welche durch eine Produktion der Form $s \to V$ ersetzt wird. Ein Wort gehört einer Grammatik G an, wenn es nur noch aus terminalen Symbolen der Menge T_G besteht und ausgehend von dem Startsymbol S_G mit den Produktionen der Menge P_G gebildet werden kann.[1]

Je nachdem, wie die Produktionsregeln einer Grammatik aufgebaut sind, lässt sich eine formale Sprache nach dem Informatiker Noam Chomsky in vier Typen einteilen. Grundsätzlich gehört jede Sprache dem Typ 0 der allgemeinen Sprachen an. Eine Sprache gehört immer einem Typ X an, wenn alle Bedingungen der Typen $\leq X$ für alle Produktionsregeln der Form $U \to V$ erfüllt sind. Die Bedingung für eine kontextsensitive Sprache des Typs 1 besagt, dass durch eine Produktionsregel das Wort nicht verkürzt werden kann, also $|U| \leq |V|$. Für eine kontextfreie Sprache des Typs 2 gilt, dass durch eine Produktionsregel nur jeweils eine Variable ersetzt werden kann, also $|U| = 1$. Eine Sprache ist eine reguläre Sprache des Typs 3, wenn ein nichtterminales Symbol mit einer Produktionsregel durch ein terminales Symbol und ein optionales nichtterminales Symbol ersetzt wird, jedoch nicht durch mehre, also $U \leq a|aB$. [2]
Um die Darstellung von Sprachen zu vereinheitlichen, wurde die Erweiterte-Backus-Nauer-Form, kurz EBNF entwickelt. Die oben gegebenen Zeichen sind Teil des EBNF ISO-Standards und durch sie lässt sich jede beliebige formale Sprache beschreiben.

[1] $https://www.uni-ulm.de/fileadmin/website_uni_ulm/iui.inst.040/Formale_Methoden_der_Informatik/Vorlesungsskripte/FMdI-06--2010-01-10--FormaleSprachen_Vorlesung.pdf$; 19.11.2018
[2] I. Wegener; Theoretische Informatik; Kap.5

Verwendung	Zeichen	Verwendung	Zeichen
Definition	=	Aufzählung	,
Endezeichen	;	Alternative	—
Option	[...]	Optionale Wiederholung	{...}
Gruppierung	(....)	Anführungszeichen 1. Variante	"..."
Anführungszeichen 2. Variante	'...'	Kommentar	(*...*)
Spezielle Sequenz	?...?	Ausnahme	-

Tabelle 1: Inhalt der Erweiterten-Backus-Nauer-Form

1.1.2 Spezifikation unserer Skriptsprache anhand unserer Ansprüche an den Funktionsumfang

Für unser Programm haben wir uns grundlegend vorgenommen Funktionen für das Versenden von Dateien sowie Befehlen an einen zweiten PC bereitzustellen. Es erfordert also in gewissem Maße eine Kommunikation zwischen dem Nutzer und dem Programm, um die Wünsche des Bedieners genauer zu spezifizieren und der Software zu übermitteln. Um dies zu bewerkstelligen entschieden wir uns, eine eigene formale Sprache zu entwickeln, mit dem Ziel Befehle an das Programm mit den entsprechenden Argumenten zu übermitteln, welche über ein Konsolenfenster eingegeben werden. Unsere Sprache sollte Folgende Anweisungen beinhalten:

Befehl	Argumente
send_file	file_name
send_comm	return command_name [arguments]
get_file	file_name
connect	ip_address
disconnect	
reconnect	ip_address
shutdown [send_comm]	
open [send_file]	file_name
chat	chat_msg
squit	

Tabelle 2: Befehlssatz der Eingabesyntax

Die Befehle *send_file* und *send_comm* sind hierbei die allgemeinen Funktionen zum Senden einer Datei oder eines Befehls an die betriebssystemeigene Kommandozeile jeglicher Art. Das Kommando *get_file* ist identisch zu *send_file* mit dem Unterschied, dass eine Datei von dem angefragten PC auf den anfragenden übertragen werden soll. Mit den Anweisungen *(dis−/re−)connect* wird der grundlegende Verbindungsaufbau gesteuert. Durch den *connect*-Befehl wird versucht eine Verbindung zu der als Argument übergebenen IP-Adresse zu errichten. *disconnect* bewirkt die Beendigung einer bestehenden Verbindung, *reconnect* versucht zusätzlich eine neue Verbindung zu erstellen. *shutdown* und *open* sind spezielle versendete Anweisungen zum Herunterfahren des Zielcomputers und Öffnen einer Datei, bei denen wir es aufgrund ihrer frequentierten Nutzung für sinnvoll hielten sie eigenständig in unsere formale Sprache zu integrieren. Sie basieren auf den zwei Grundfunktionen und stehen beispielhaft für die Erweiterbarkeit des Funktionsumfangs unserer Software. Zusätzlich für Befehle, welche mit Dateien arbeiten sollen, müssen sowohl der Dateiname als auch das Dateiformat als Zeichenkette angefügt werden. Die Übermittlungsfunktionen für Anweisungen benötigen neben dem Anweisungsnamen als Zeichenkette auch optionale Argumente, mit welchen die Anweisung in

der Kommandozeile ausgeführt werden soll.

Um unsere formale Sprache zu implementieren entwickelten wir zuerst als Grundlage eine Darstellung der Eweiterten-Backus-Nauer-Form. Diese ist im Anhang einzusehen. Eine solche Darstellung hat den Vorteil einer unkomplizierten Übertragung der Produktionsregeln in die Implementierung, welche die Nutzereingaben überprüft. Im späteren Verlauf der Entwicklung wurden einzelne Befehle und Argumente angefügt und verändert, um eine Praxistauglichkeit zu bewirken.

1.2 Implementierung der Anwendungssteuerung in der Skriptsprache Lua

1.2.1 Darstellung des grundlegenden Programmaufbaus und Programmablaufs

Grundlegende Aufgabe der Anwendungssteuerung ist es die Eingaben, die der Nutzer unserer Software tätigt, in Aktionen des Programms umzusetzen. Dazu wird zum einen auf dem Startrechner die nach der in Kapitel 1.1 beschriebenen Syntax eingegebene Anweisung in der Eingabeverarbeitung auf ihre Korrektheit überprüft um anschließend aufgeschlüsselt zu werden und die gewünschte Sendefunktion zu initialisieren.
Nachdem die Daten mithilfe des Netzwerkprotokolls übermittelt wurden, werden sie von der Übertragungsverarbeitung auf dem Empfängercomputer weiter bearbeitet. Je nach spezifiziertem Typ der Übermittlung wird entweder die enthaltene Datei auf Fehler überprüft und anschließend endgültig gespeichert, oder die auszuführende Anweisung wird an die betriebsystemeigene Kommandozeile weitergegeben und ausgeführt.
Um keine ungewollten Übertragungen von potenziell schädlichen Dateien oder Anweisungen zu ermöglichen wird für die Ausführung jeder Übermittlungsanfrage die Bestätigung des Empfängers benötigt, sofern nicht der $root-Modus$ aktiviert ist. Als *root*-Nutzer hat man uneingeschränkte Kontrolle über den Ziel-Computer, was besonders für administrative Tätigkeiten sinnvoll ist.

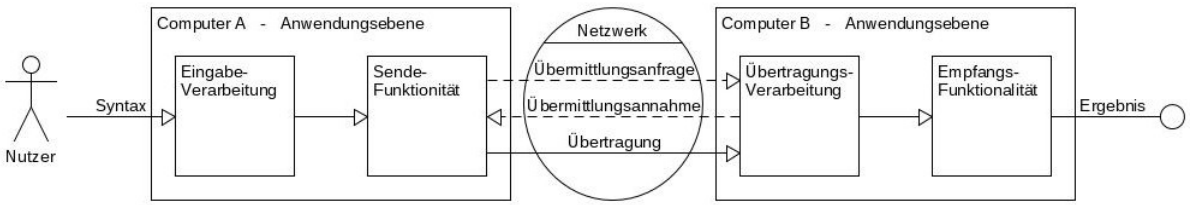


Abbildung 1: Schematische Darstellung der Ablaufenden Prozesse im Programm

1.2.2 Ansprüche an die Implementierung und die daraus resultierende Wahl der Programmiersprache

Wie in Kapitel 1.2.1 erläutert, wird auf der Anwendungsebene unseres Programmes, also dem Teil, welcher nicht für die Kommunikation im Netzwerk zuständig ist, die grundlegende Funktionalität implementiert. Da unser selbstentwickeltes Netzwerkprotokoll, welches die eigentliche Kommunikation zwischen 2 Computern regelt, relativ universell auf verschiedenartige Daten anwendbar ist, stellt dieses im Bezug auf den Funktionsumfang der Software nicht den limitierenden Faktor da. Um das Protokoll eventuell auch im Zeitraum nach der eigentlichen Arbeit ausnutzen zu können, und nicht durch einen festgelegten Satz von Befehlen beschränkt zu werden, entschieden wir uns dazu diesen Programmteil in einer Skriptsprache zu implementieren.

Diese Klasse von Programmiersprachen zeichnet sich dadurch aus, dass der vom Menschen lesbare Quelltext eines Programmes erst bei seiner Ausführung in Anweisungen übersetzt wird, welche für den Computer ausführbar sind. Im Gegensatz zu kompilierten Sprachen hat dies den Vorteil, dass der Programmcode unkompliziert korrigiert und modifiziert werden kann und sofort lauffähig ist.

In diesem Zusammenhang ergibt sich außerdem eine leichte Wartbarkeit, also die Möglichkeit eventuell auftretende Fehler zu beseitigen, als ein weiterer erfüllter Anspruch.
Den Vorteil von kompilierten Programmiersprachen, dass sie sich durch eine höhere Leistungsfähigkeit besonders bei komplexeren Problemen auszeichnen, haben wir uns durch die Implementierung unseres Netzwerkteils in der Sprache C++ zu Nutz gemacht. Dies stellt den Anspruch, dass die gewählte Skriptsprache problemlos in Software der Sprache C++ einzubetten ist, und keine Probleme an der Schnittstelle der beiden Programmteile mit zwei unterschiedlichen Sprachen auftreten.

Da wir als Gruppenmitglieder unterschiedliche Betriebsysteme auf unseren Arbeitsrechnern benutzen, entstand als Nebenprodukt die Anforderung, dass das finale Programm systemunabhängig sein muss, also ohne erheblichen Aufwand auf neue Betriebssysteme portierbar ist. Unser Fokus lag dabei für die Arbeit auf nicht mobilen Computersystemen. Zu guter Letzt ist der Zeitraum der Seminarfacharbeit auch nur auf eineinhalb Jahre begrenzt, weshalb die gewählte Sprache mit einem geringen Lernaufwand benutzbar sein muss.

Als Kompromiss zwischen allen beschriebenen Ansprüchen entschieden wir uns für die Programmiersprache Lua. Da sie in reinem C geschrieben ist, weist sie eine uneingeschränkte Integrationmöglichkeit in bereits vorhandenen C++ - Kontext auf, und lässt sich außerdem auf die am weitesten verbreiteten Betriebssytemarchitekturen wie Windows und Unix portieren.[3]
Ein positiver Nebeneffekt dieser Wahl ist die hohe Leistungsfähigkeit von Lua, verglichen zu anderen weit verbreiteten Skriptsprachen. Der Sprachaufbau orientiert sich sehr nah an der Englischen Sprache, was ein schnelles Erlernen ermöglichte.

1.2.3 Erläuterung wesentlicher Elemente der Implementierung

Eine zentrale Rolle in der Umsetzung unserer Eingabeverarbeitung spielt die Interpreter-Funktion. Sie überträgt die von dem Nutzer in Form einer Zeichenkette gelieferte Eingabe in eine für das Programm verständliche Form. Außerdem ruft sie die Funktionen auf, welche die gewünschte Funktionalität implementieren, und übergibt ihnen die eingegebenen Parameter zur weiterführenden Überprüfung.

Die Variable *content* (in der Abbildung 2 *Eingabe* genannt) ist eine Liste der eingegebenen Wörter. Sie wird durch die Funktion *split_input* erstellt, welche die rohe eingegebene Zeichenkette an den Leerzeichen teilt, und demnach wortweise abspeichert. In der Liste *commands* wird der Befehlssatz gespeichert. Hierbei wird dem Namen von jedem verfügbaren Befehl ein Wert zugeordnet. Es spielt keine Rolle, welcher Wert es ist, wichtig ist, dass die Variablen initialisiert sind und nicht keinen zugeordneten Wert haben. In der folgenden $If-Else$-Struktur wird überprüft, ob der eingegebene Befehl ein Teil des Befehlssatzes ist, sprich, ob in der Liste *commands* ein Wert für den gewünschten Befehl vorliegt. Ist dies nicht der Fall, gibt die Funktion einen Fehler der Form „$ERROR : interpret\_content : EINGABE - unbekannterBefehl$!“ aus, und es wird keine Übertragung eingeleitet. Wenn die eingegebene Anweisung Teil des Be-

[3] www.lua.org/about.html, 17.11.2018

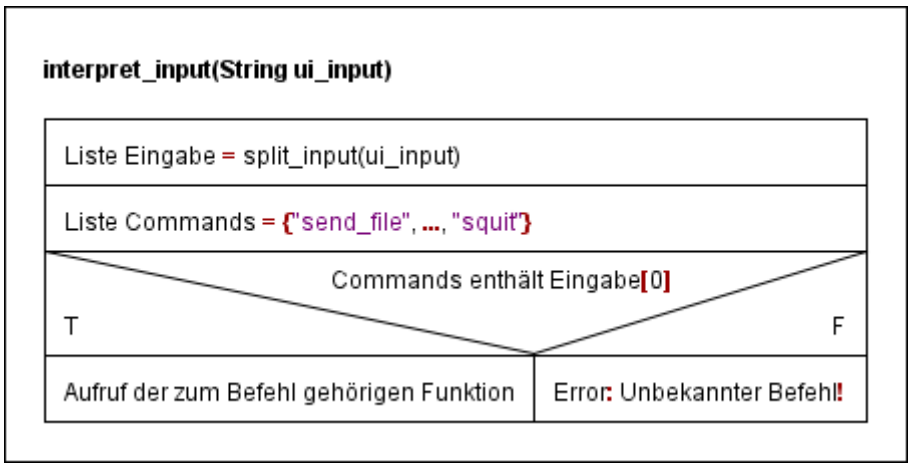


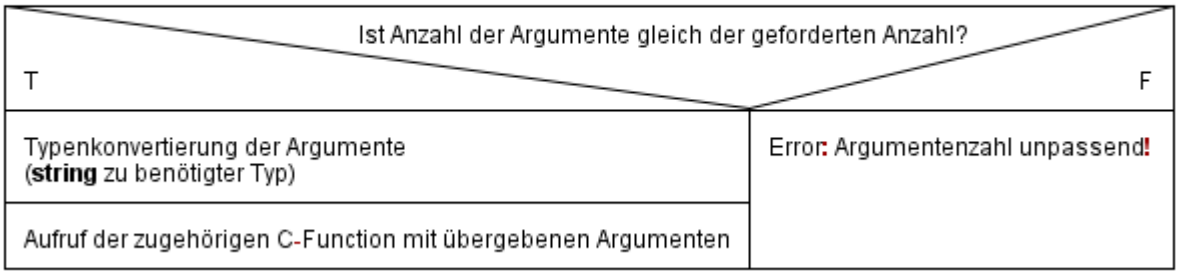
Abbildung 2: Grundlegende Struktur der Interpreterfunktion

fehlssatzes ist, so wird die entsprechende Funktion mit den übergebenen Parametern aufgerufen, und das Ergebnis in der Variable *result* gespeichert. Der Aufruf geschieht durch die Lua-Interne Funktion $.G[STRING](ARGUMENTS)$, welche eine Zeichenkette als Input fordert, und eine Funktion dieses Namens mit den gewünschten Argumenten $ARGUMENTS$ aufruft.

Die gewählte Form der Implementierung, dass der Befehlssatz in einer Liste, bestehend aus den Namen der zugehörigen Funktionen, gespeichert ist, ermöglicht eine unkomplizierte Erweiterung der grundlegenden Befehle durch neue Funktionalität, welche möglicherweise spezifischere Ansprüche an das Protokoll fordert. Beispiel hierfür ist die Implementierung von Streaming, die einen ständigen Erhalt der Netzwerkverbindung zwischen den zwei Rechnern benötigen würde. Die eigentlichen Funktionen sind alle nach einem ähnlichen Schema aufgebaut. Es gibt zwei verschachtelte $If - Else$-Bedingungen, die erfüllt werden müssen um die eigentliche Übertragung auszulösen. In der ersten wird überprüft, ob die gewünschte Übertragung autorisiert ist. Dies ist der Fall, wenn auf dem Computer, an den die Übertragung gerichtet ist, der Verbindung zugestimmt wurde. Die zweite Bedingung prüft, ob die Anzahl der gegebenen Argumente mit der Anzahl an geforderten Argumenten übereinstimmt. Schlägt eine der Bedingungen fehl, so wird der Vorgang abgebrochen und eine Fehlermeldung entsprechend der oben dargestellten Form ausgegeben. Sind beide Forderungen erfüllt, werden die Argumente, welche als Zeichenkette eingegeben wurden, in die notwendigen Datentypen konvertiert und die zu den Funktionen zugehörigen Anbindungen an das Netzwerkprotokoll initialisiert.

Abbildung 3: Strukturelle Darstellung der grundlegenden Funktionsstruktur

Sende-Befehl-Template(Liste Argumente)



2 Entwurf und Umsetzung der Netzwerk-Schnittstelle

2.1 Grundlegende Theorie zur Kommunikation in einem Netzwerk

Unter dem Begriff Netzwerk im Zusammenhang mit dieser Arbeit wird, sofern nicht ausdrücklich anders definiert, die Ansammlung mehrerer, untereinander verbundener Computer verstanden. Diese haben die Möglichkeit mit durch die ISO standardisierten Internet-Netzwerkprotokollen miteinander zu kommunizieren.

Diese Kommunikation wird durch das ISO/OSI Modell beschrieben, auf dem auch die Arbeitsweise unseres Programmes aufbaut. Innerhalb dieses Modells ist der Prozess einer Netzwerkübertragung sowie die dabei verwendeten Informationen bzw. Protokolle in Schichten nach ihrem Abstraktionsgrad einteilbar, von Nullen und Einsen in einem Kabel bis hin zu z.B. verschiedensten Verschlüsselungsprotokollen; was in unteren Schichten abgesichert ist, muss in denen darüber nicht implementiert werden, was wiederum zu einfachen und verlässlichen einzelnen Algorithmen in den Schichten führt, obwohl das gesamte System überaus komplex ist. Das Modell ist in Abbildung 4 dargestellt.

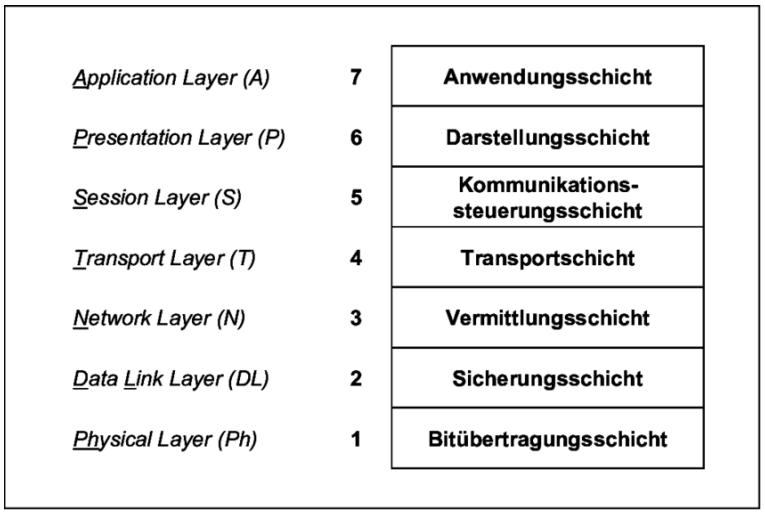


Abbildung 4: Schematischer Aufbau des ISO/OSI-Modells[4]

2.2 Notwendigkeit, Anforderungen und Spezifikation eines eigenen Netzwerkprotokolls

Um das Programm mit mehreren Programmierern zu entwickeln, muss eine gewisse Modularität gewährleistet werden. Außerdem müssen die Sinnabschnitte klar nach ihrer Funktion gegliedert sein und leicht benutzbare sowie wiederverwendbare Schnittstellen besitzen, damit nicht das gesamte Programm angepasst werden muss, wenn ein Teil geändert wird und jeder sich auf seinen Teil konzentrieren kann, ohne die anderen vollständig zu kennen. Diese Anforderung wird durch die Trennung der verwendeten Programmiersprachen Lua und C++ einfach geregelt. Im C++ - Kontext muss es dennoch eine Unterteilung zwischen Netzwerkschnittstelle und Benutzeroberfläche geben, weshalb die gesamte Netzwerkkommunikation über eine statische Bibliothek in die ausführbare Datei eingebunden ist. Dort sind die Benutzeroberfläche und die Verbindung der Programmteile implementiert. Beide Programmteile nutzen die Funktionen und Klassen des frei zugänglichen Programmier-Frameworks Qt[5].

Diese Netzwerkbibliothek stellt Funktionen und Klassen bereit, die eine verlässliche Verbindung zwischen zwei Rechnern aufbauen (Start und Ziel), über die ein erweiterbarer Satz von Anweisungen mit optionalen Argumenten sowie Dateien bidirektional übertragen werden können.

[4] https://www.researchgate.net/profile/Sebastian_Lempert/publication/202268228/figure/fig1/AS:393996175200256@1470947416022/Abbildung-1-Die-7-Schichten-des-ISO-OSI-Referenzmodells-Der-geschichtete-Aufbau-eines.png [Zugriff: 20.12.2018 14:20 Uhr]

[5] https://www.qt.io/ [21.12.2018 16:30 Uhr]

Startrechner bezeichnet daher immer den Rechner, vor dem der Anwender sitzt, Zielrechner denjenigen, der über das Programm gesteuert wird und Anweisungen erhält.
Dafür stellen sich folgende Anforderungen. Es muss:

- ein gesicherter Transport von beliebigen Daten möglich sein,
- eine verschlüsselte Kommunikation vorliegen,
- der Datenverkehr so klein wie möglich sein,
- der Kommunikationskanal in beide Richtungen gleich aufgebaut sein,
- die Übertragung von Dateien und Anweisungen möglich sein,
- ein erweiterbares System zur Definition von Anweisungen existieren und
- die Übertragung auch von größeren Dateien mit zusätzlichen Informationen fehlerfrei ablaufen.

Diese Bedingungen implizieren bereits, dass es eine bestimmte Regelung des Ablaufs für die Kommunikation zwischen den beiden Computern geben muss: ein Protokoll, also eine Kommunikationsvorschrift zwischen zwei Computern, das mit bestimmten Datenformaten und Abläufen das Verhalten der Computer während der Kommunikation festlegt.

Um nicht ebenfalls für die Ankunft der Daten selbst sorgen zu müssen, baut das Protokoll auf TCP/IP auf, genauer auf dem Protokoll TLS, welches schon eine verschlüsselte Verbindung bereitstellt, also der Sicherungsschicht des ISO/OSI-Modells. Demnach ist unser Protokoll zwischen der fünften und sechsten Schicht des Modells(s. Abb. 4) einzuordnen, da es auf TLS aufbaut, aber noch nicht zur Darstellung von Informationen genutzt wird, wie es beispielsweise mit HTML der Fall wäre, sondern deren formatierte Übertragung und Aufbereitung zur Steuerung des Programms regelt. Man kann die verschickten Daten in Pakete einteilen; das sind kleinere Datenabschnitte, die sequentiell, aber unabhängig voneinander verschickt und am Zielort zusammengesetzt werden. Um diese Pakete innerhalb des Netzwerks zu navigieren, wird am Startcomputer an den Anfang jedes Pakets (*Header*) ein Datensatz geschrieben, der Informationen zum Zielort, dem Weg oder der Behandlung der Daten des Pakets enthält.

Auf dieser Technik baut unser Protokoll auf. Es schreibt an den Anfang jeder Übertragung einen Datensatz mit einem festen Format, welcher die Art des Pakets beschreibt und am Zielort ausgelesen wird. Das bestimmt dann die weitere Behandlung der Daten. Besonders im Fall von Anwendungen ist der Unterschied zwischen Header und Datenteil des Pakets fließend, da hier eine feste Struktur sowohl im Bezug auf die Größe als auch die Verarbeitungsgeschwindigkeit von Vorteil ist und die meisten Informationen zur Anweisung schon im Header integriert sind. Da das Protokoll sowohl für den Versand von Anweisungen als auch von Dateien geeignet sein muss und dabei so strukturiert und einfach wie möglich soll, gibt es zwei verschiedene Teilprotokolle, was in dem folgenden Aufbau der Kommunikationsstruktur resultiert:

Im ersten Schritt werden zwei Informationen - nämlich der Typ der Übertragung, also Anweisung oder Datei, und ihre Gesamtgröße - in den Header des ersten Pakets der Übertragung geschrieben.
Abhängig davon, was der Übertragungstyp ist, werden nun entweder die Informationen für eine Datei- oder eine Anweisungsübertragung angehängt; die Gesamtgröße ermöglicht es die Vollständigkeit einer Übertragung zu verifizieren. Dabei sind alle Werte, sofern es möglich ist, als Zahlen und nicht als Zeichenketten vorhanden, um die Größe der Informationen zu minimieren, und außerdem, besonders im Falle der Anweisungen, eine systematische Erweiterbarkeit

zu gewährleisten.

Betrachtet man den Header für die Anweisungen, so ist zunächst ein Anweisungscode vonnöten, also eine vorher festgelegte Zahl für jede Anweisung. Diese kann am Zielort eindeutig einer Handlungsfolge, d.h. einer bestimmten Anweisung zugeordnet werden.
Teil zwei sind Standardargumente, die binär auf 32 Bit Speicherbreite gesetzt werden können. Diese Standardargumente sind dazu da, ein Zusammenfassen mehrerer ähnlicher Anweisungen zu gewährleisten und dadurch das Protokoll sinnvoll zu strukturieren, anstatt es mit Anweisungen zu überladen; so gibt es beispielsweise nur eine Anweisung um Ordnerstrukturen anzufordern. Mit ihr ist es möglich, die Dateien, die bereits auf dem Zielcomputer vorhanden sind, abzurufen und anzuzeigen. Je nachdem, welche Informationen zu welchen Dateien geschickt werden sollen, können jetzt Standardargumente als Bitflags gesetzt werden - das bedeutet, die binäre Zahlendarstellung von Integern (Ganzzahlen) wird insofern ausgenutzt, dass einzelne Bits einer Ganzzahl mit bestimmter Speicherbreite auf eins oder null gesetzt werden können. Das kann am Zielort wieder ausgelesen werden und so dienen die Bits als „Signale“, die die genaue Auslegung einer Anweisung bestimmen und dabei extrem platzsparend sind. So können neben den Dateinamen auch Dateigrößen, Zugriffsrechte oder andere Metainformationen mitgeschickt werden, wenn die entsprechenden Bits auf eins gesetzt worden sind - andernfalls wären dafür zusätzliche Anweisungen nötig, was das Protokoll unnötig vergrößern würde.

Der dritte Wert im Headersegment für die Anweisungen ist eine Zahl, die ein Programm identifiziert, auf das die Anweisung angewandt werden soll, was im Standardfall unsere Software ist.

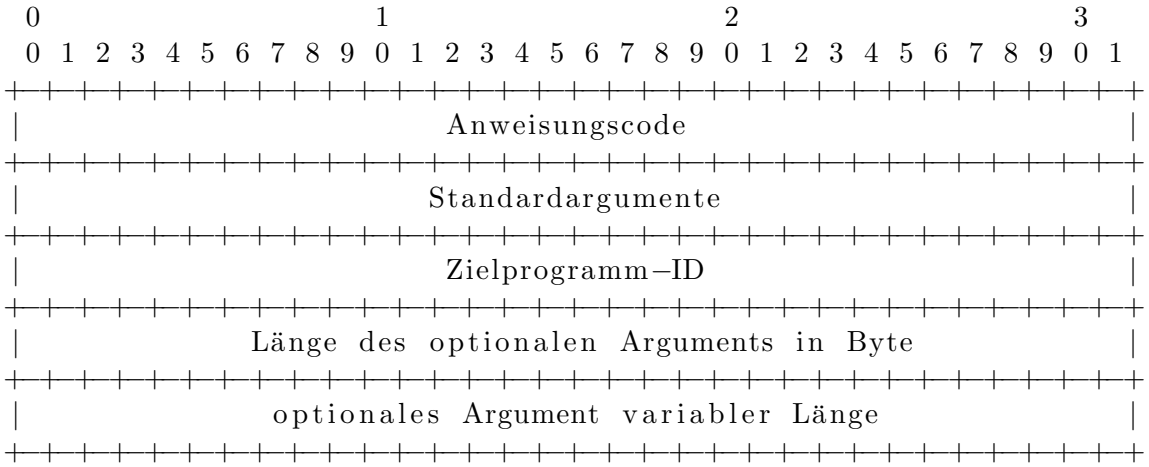


Abbildung 5: Header für Anweisungen

Der vierte Wert ist eine Längenangabe in Byte. Er bezeichnet die Länge des optionalen Paketinhalts beliebigen Formats, welches an den Header angehängt werden kann, beispielsweise ein Dateiname, ein Prüfungssummenwert oder ein Kommandozeilenbefehl.

Um die ganze Anweisung klein zu halten wird intern festgelegt, dass die Gesamtgröße dieses Headersegments inklusive des optionalen Arguments nicht größer als $2^{16} - 1$ Byte sein darf - größere Anhänge müssen als Datei verschickt werden. Aus Kompatibilitätsgründen mit verschiedenen Compilern und Systemen sind alle Zahlenwerte Integer(Ganzzahlen) mit 32 bzw. 64 Bit Speicherbreite angegeben, um damit sowohl genügend Platz für alle Informationen zu bieten als auch Probleme mit verschiedenen Anpassungen der Speicherbreite abhängig von der unterliegenden Rechnerarchitektur zu vermeiden, die während der Entwicklung ein Problem darstellten können. Für den Nutzer spielen die tatsächlichen Kodierungen, also welches Programm oder welche Anweisung welchen Code erhält, keine Rolle. Sie werden intern festgelegt

und bei Interaktionen mit dem Nutzer ausgeführt oder in ihre Entsprechungen in von Menschen lesbare Form umgewandelt. Die genauen Werte für Anweisung und Programme werden nur in seltenen Fällen im Zusammenhang mit der Nutzerseitigen Erweiterung des Anwendungskanons relevant, und sind im Anhang einsehbar.[s.S. 18] Insgesamt ergibt sich die in Abbildung 5 zu sehende Spezifikation des Headers für die Versendung von Anweisungen. Dabei ist die Länge der Argumente in (pro Zeile 32) Bit angegeben, sofern nicht anders spezifiziert.

Das Versenden von Dateien hat andere Anforderungen, was in einem geänderten Aufbau des Header-Segments für die Übertragung, sowie in einem anderen Ablauf resultiert.

Für jede Dateiübertragung wird, im Gegensatz zu den Anweisungen, jeweils eine neue TLS-Verbindung aufgebaut, die unabhängig von den vorhergehenden und nachfolgenden ist. Es gibt also für jede Datei einen abgeschotteten „Kanal“, in dem Fehler leicht behandelbar und konkurrierende Übertragungen unmöglich sind. Außerdem bleibt die Kennzeichnung einzelner Datenpakete erspart, was wiederum zu insgesamt weniger Datenverkehr führt. Darauf folgend wird ein Paket mit Informationen zu der Datei zum Zielcomputer geschickt, welcher, um die Integrität der Übertragung zu verifizieren, dasselbe Paket zurücksendet.
Dieses Informationspaket ist in Abbildung 6 schematisch dargestellt. Es besteht aus der Art der Datei (z.B. Video, Audio, Text, ausführbare Dateien etc.), was für die weitere Einordnung im Zielcomputer nützlich ist, dem ursprünglichen Dateinamen und einer Prüfsumme, mit der die fehlerfreie Ankunft verifiziert werden kann. Auch hier spielt der tatsächliche Zahlenwert, der hinter dem Typ steht, nur intern eine Rolle, abgesehen von benutzerdefinierten Erweiterungen des Befehlssatzes, wie im Kapitel zur Eingabeverarbeitung nachzulesen ist. Deshalb sind die Codierungstabllen im Anhang enthalten. [s. S. 18]

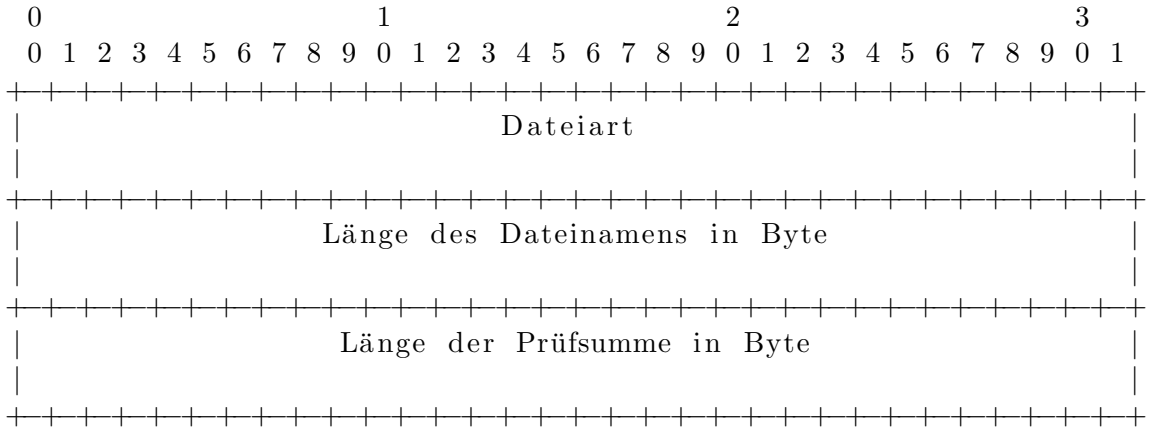


Abbildung 6: Header für Dateiübertragungen

Das Headersegment ist so aufgebaut, dass zunächst die drei Ganzzahlwerte für Typ, Länge des Dateinamens in Byte und Länge der Prüfsumme in Byte in den Speicher geschrieben werden. Dann wird der Dateiname und die Prüfsumme angehängt und dieses gesamte Paket wird an den Zielcomputer übermittelt.

Ist die Verbindung verifiziert, wird mit der eigentlichen Übertragung der Datei begonnen, die in Abschnitten und ohne weitere Kennzeichnung in mehreren kleinen Paketen geschieht. Das funktioniert, da unser Standard zum Einen auf TCP/IP aufbaut, welcher bereits die richtige Reihenfolge und Vollständigkeit der Pakete weitgehend sicherstellt und zum Anderen nur eine Verbindung pro Dateiübertragung aktiv ist. Am Zielcomputer wird die Datei schließlich in einem temporären Standardverzeichnis abgespeichert und Paket für Paket zusammengesetzt. Ist

die Übertragung nach dem Senden einer bestimmten Bytesequenz vom Zielcomputer beendet, wird ein Signal aus einer der Managerklassen ausgesendet und die Verbindung getrennt. Diese Schrittfolge ist schematisch in Abbildung 7 dargestellt. Die Aufgaben und Funktionen der Managerklassen werden im nächsten Kapitel genauer erläutert. Nach der erfolgreichen Übertragung wird mit der anfänglich verschickten Prüfsumme der Datei ihre Integrität geprüft, um Fehler insgesamt auszuschließen, und die weitere Verarbeitung an die anderen Programmteile abgegeben.

Alle Anforderungen an das Protokoll werden erfüllt; der gesicherte und verschlüsselte Transport von Daten ist durch die Verwendung einer TLS-Verbindung und dem Aufbau eigener Kanäle erfüllt, die fehlerfreie Übertragung von Daten wird am Zielort über Gesamtgröße und Prüfsummen verifiziert, der Kommunikationskanal ist symmetrisch aufgebaut, da die Bibliothek sowohl für den Start- als auch für den Zielcomputer die nötige Funktionalität bereitstellt; die Codierung von allen Informationen als Zahlen oder als Bitflags sowie die Nutzung eigener Kanäle für Dateien reduziert den Datenverkehr drastisch, und damit ist auch dieser Punkt erfüllt, außerdem ist die Verwendung von Zahlencodes systematisch erweiterbar ohne die Speicherbreite verändern zu müssen, also wurden alle Ziele erreicht.

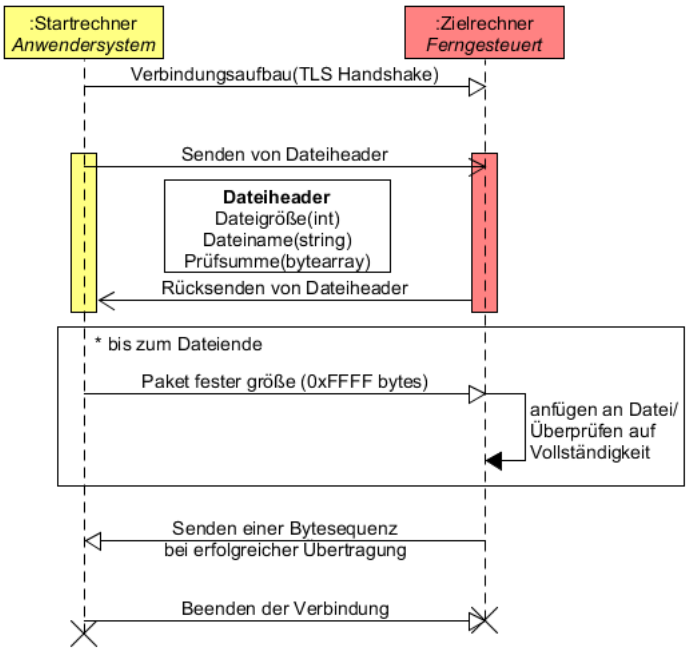


Abbildung 7: Schematische Darstellung des Ablaufs bei der Dateiübertragung

2.3 Funktionsumfang der selbstgeschriebenen Netzwerkbibliothek

Wie schon erwähnt, wird die Netzwerkkommunikation über eine statisch eingebundene Bibliothek vom Rest des Programmes abgetrennt. Solch eine Bibliothek stellt die gesamte darin implementierte Funktionalität mit einigen wenigen, für den weitergehenden Gebrauch freigegebenen Funktionen und Klassen bereit. Die Bibliothek stellt auf oberster Ebene eine Schnittstelle zum Übertragen von Dateien und Anweisungen zu einer bekannten Netzwerkadresse bereit. Das heißt konkret, es werden Methoden der beiden Klassen MngThManager und MngFileManager bereitgestellt, welche die einzelnen Informationen intern in die zuvor spezifizierte Form umwandeln und verschicken sowie am Zielcomputer wieder entpacken und an die weiteren Programmteile übergeben. Zusätzlich dazu ist eine entsprechende Fehlerbehandlung bei Verbindungsabbrüchen, fehlerhaften Übertragungen oder internen Kommunikationsproblemen durch falsche Eingaben sowie die Möglichkeit eine Fortschrittsmeldung bei der Dateiübertragung implementiert. Insgesamt muss der Programmierer also nur mit zwei Klassen interagieren, um die volle Funktionalität der Bibliothek auszunutzen, weshalb sich das gesamte Konstrukt sehr gut für weitere Verwendungen in flexiblen Kontexten der modularen Programmierung eignet.

2.4 Implementierung und Funktionsweise der Klassenbibliothek

In der Bibliothek sind neben den beiden oben genannten Manager-Klassen (MngThManager für Anweisungen und MngFileManager für Dateien) weitere Klassen definiert, die von den jeweiligen Manager-Klassen aus aufgerufen und benutzt werden und dadurch eine darunterliegende Schicht bilden, die durch eine fest definierte Schnittstelle an den Rest des Programms angebunden werden. Außerdem gibt es eine von allen Klassen benutzte Datei, in der die Datenstrukturen und -typen, die in den Header der Pakete geschrieben werden, als C-Structs (s. ***C-Structs***) definiert und einige Funktionen, die häufig gebraucht werden (z.B. String-Vervielfältigung, oder eine Funktion zum errechnen der Dateiprüfsumme) deklariert sind.

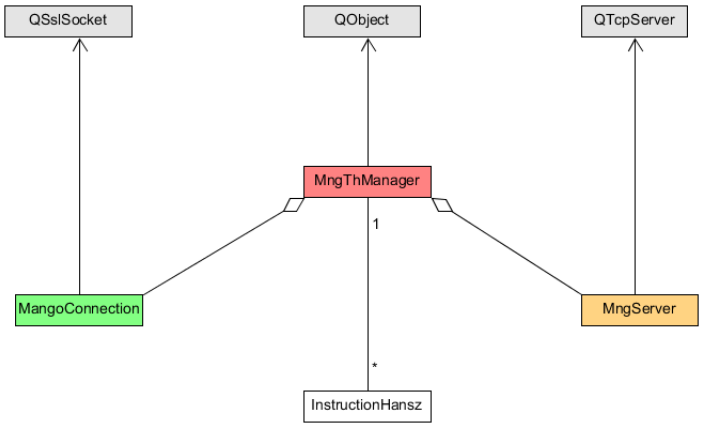


Abbildung 8: Schematisch: Klassen zum Anweisungsversand

Der Aufbau der Übertragung beider Datentypen, die verschickt werden, ähnelt sich bis zu einem gewissen Punkt. So haben beide eine eigene Klasse, in der intern die Objekte (Anweisungen und Dateizugriffsobjekte) gespeichert sind und verwaltet werden (*InstructionHansz* bzw. *FileHansz*), sowie andere Klassen zum Verbindungsaufbau und dem Versand. So gibt es eine Klasse, die für das Annehmen von hereinkommenden Verbindungen zuständig ist und eine (bei Dateien zwei) Socket-Klassen (s. ***Socket***), die sich um das Senden bzw. Empfangen von Daten kümmern, was in Abb. 8 bzw. 9 vereinfacht schematisch dargestellt ist.

Die beiden Speicherklassen werden dabei sowohl für den Empfang als auch das Senden eines Objektes genutzt - im ersten Fall werden sie mit den Teildaten (also Anweisungscode und zusätzlichen Argumenten oder Dateiname und Dateityp) initialisiert und erstellen daraus einen Pufferspeicher, der bereits den korrekten Header und das Datenpaket enthält, dessen Inhalt nur noch ausgelesen und verschickt werden muss - somit ist ein gleicher Aufbau der Verbindung in beide Richtungen sichergestellt.

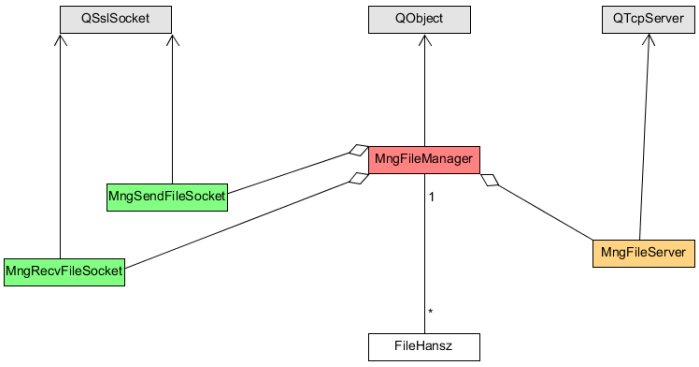


Abbildung 9: Schematisch: Klassen zum Dateiversand

Das tatsächliche Versenden sowie der Verbindungsaufbau wird von einer Server-Klasse und den Socket-Klassen übernommen. Dabei horcht die Server-Klasse, sowohl für Dateien als auch für Anweisungen, auf der empfangenden Seite auf einem zuvor bestimmten Port auf eingehende Verbindungen. Auf der sendenden Seite wird eine neue Klasse erstellt, welche ein TLS-Socket initialisiert und zu der gegebenen Adresse verbindet. Ist die Verbindungsanfrage eingegangen, wird eine normale TLS-gesicherte Verbindung aufgebaut, sofern das möglich ist. Danach weichen die Abläufe von Dateiversand und Anweisungsübertragung ab: Anweisungen werden jetzt, in der Reihenfolge in der sie eingehen, von der Managerklasse an das Socket getaktet weitergeleitet, wo sie dann in-

klusive Header und Datenpaket verschickt werden - bei Dateien ist der Ablauf etwas komplexer.

Dabei muss zunächst erwähnt werden, dass für Dateien, im Gegensatz zu Anweisungen, jedes Mal eine neue Verbindung aufgebaut wird. In der Verbindung für eine Datei wird, sobald sie aufgebaut ist, zuerst der Header für die Dateiübertragung an den Zielrechner gesendet, welcher genau dasselbe Paket dann zurückschickt um sicherzustellen, dass alle Steuerinformationen korrekt angekommen sind. Jetzt startet die eigentliche Übertragung der Datei, welche mit dem Auslesen eines Datenblockes fester Größe beginnt. Dieser wird in einen Pufferspeicher geschrieben, der, sobald er voll ist, von der Socket-Klasse ausgelesen und übertragen wird. Diese Anweisungsfolge wird so lange wiederholt, bis das Ende der Datei erreicht ist. Der letzte Datenblock enthält oft weniger als die Maximalgröße, was aber in der Praxis kein Problem ist - das letzte Datenpaket auf der Seite des verschickenden Rechners ist nur etwas kleiner als der Rest. Auf der empfangenden Seite werden die ankommenden Datenpakete über ihre Repräsentation als Filehansz-Objekt in eine nach ihrer Prüfsumme benannten Datei in einem temporären Verzeichnis gespeichert. Das verhindert die konkurrierende Benennung von ungleichen Dateien und vereinfacht das Überprüfen auf korrekte Übertragung. Die ursprünglichen Namen werden zu Beginn der Übertragung per Signal an das Programm außerhalb der Bibliothek geschickt, welches diese speichert - somit geht auch der Überblick nicht verloren. Wurden alle Pakete verschickt und sind angekommen, wofür die TCP-basierte Verbindung sorgt, wird vom empfangenden Rechner eine Bytesequenz versendet, welche die erfolgreiche Übertragung kennzeichnet.

Danach wird die Verbindung beendet und die beiden Sockets gelöscht. Der gesamte Prozess ist in Abb. 7 (S. 14) als Sequenzdiagramm dargestellt.
Die Verbindung für Anweisungen wird erst am Programmende oder bei einem disconnect-Befehl getrennt; sie läuft auf Port 16962, die Dateiübertragung auf Port 16963. Greift man über das Internet auf einen Computer in einem lokalen Netzwerk zu, so muss zuvor beim Standardgateway für beide Ports Port-Forwarding eingestellt werden.

Insgesamt arbeiten die für den Benutzer der Schnittstellen nicht sichtbaren Klassen an der reibungslosen Funktion eines komplexen Protokolls, das sich durch extrem speichereffiziente Datenstrukturen und einen mehrstufigen Aufbau auszeichnet. Dabei sind sie in ihrer Implementierung auf eine möglichst geringe Fehlerquote und Stabilität ausgelegt, insbesondere durch die Verwendung von einem „Kanal“ pro Datei oder eine Taktung bei der Übertragung von Anweisungen, was das Abstürzen des Programms, das durch zu viele gleichzeitig geschickte Anweisungen entstünde, verhindert. So entsteht ein stabiles und verlässliches System, welches die weitere Funktion unseres Programmes stützt.

3 Zusammenfassung

Unsere grundlegende Zielstellung der Arbeit war es ein Programm zu entwickeln und zu implementieren, welches die Möglichkeiten bereitstellt sowohl Anweisungen als auch Dateien über ein Netzwerk zwischen zwei Computern zu verschicken.

Zur Verwirklichung dieses Ziels haben wir uns mit dem Design eines eigenen Netzwerkprotokolls beschäftigt, welches die ablaufenden Kommunikationsprozesse zwischen den zwei Rechnern definiert. Es baut auf dem gängigen Protokollstandard TLS auf und gliedert die zu übertragenden Daten in eine für unsere Anwendung optimierte Form. Zur Nutzereingabe entwickelten wir eine eigene Eingabesyntax auf der Basis einer selbst entwickelten formalen Sprache. Im Gegensatz zu natürlichen Sprachen kennzeichnen sich diese grundsätzlich durch ihre Eindeutigkeit, ausgehend von einer klar definierten Grammatik. Dies ermöglicht eine problemlose, automatisierte Interpretation der Nutzereingaben durch das Programm und damit eine reibungslose Kommunikation mit einem Benutzer. Neben den zwei größeren Unterzielen prägten auch mehre kleinere Zielstellungen unsere Entwicklung. So sollte das Endprodukt möglichst universell anwendbar sein, also beispielsweise auf jegliche Datentypen, und es sollte auf unterschiedlichen Betriebssytemarchitekturen lauffähig sein, was beides mit der entwickelten Software erfüllt ist. Außerdem stellt die Erweiterbarkeit und Anpassungsmöglichkeit an die Ansprüche unterschiedlicher Benutzer ein Prinzip da, nach welchem wir unsere Entwicklung richteten; deshalb sind sowohl das Protokoll als auch die Eingabesyntax so implementiert, dass durch geringen Aufwand neue Programmfunktionen eingegliedert werden können, ohne das Programm neu kompilieren zu müssen.

Eine in unseren Augen sinnvolle Weiterführung der Arbeit wäre die Portierung der Software auf mobile Geräte, wie Handys und Tablets, da diese stetig voranschreitend die klassischen Computer bei alltäglichen Aufgaben ersetzen und größere Mobilität gewährleisten. So würde eine solche Erweiterung unsere anfängliche Motivation, einen Computer unabhängig von seiner physischen Position steuern zu können, weiter auslegen, da es eine Fernsteuerung von überall her ermöglicht. Die Benutzeroberfläche betreffend sind einige kleine Erweiterungen sinnvoll, welche sich positiv auf die Bedienerfreundlichkeit auswirken würden. So ließe sich beispielsweise eine mächtigere Konsole entwickeln, welche durch Features wie die automatische Vervollständigung von Eingaben, das schreiben von Batch-ähnlichen Skripten oder das Erlauben von Drag'n'Drop für Dateien dem Benutzer die Arbeit mit unserem Programm erleichtert. Als Funktionserweiterung für das Protokoll wäre in Zukunft eine Auslegung auf Streaming möglich, mit der zum Beispiel eine grafische Übertragung oder das Ansehen einer Videodatei ohne vorheriges Übertragen möglich wäre.

Unser zu Beginn harmonisches Gruppen- und Arbeitsklima wurde kurz vor der regulären Abgabe durch den Ausfall eines Gruppenmitglieds drastisch beeinflusst. Da die Kommunikation mit besagtem Gruppenmitglied nach mehreren Anläufen fehlschlug, und der vereinbarte Arbeitsanteil nicht erbracht wurde, war eine spontane Neuverteilung der Aufgaben zwischen den verbleibenden Gruppenmitglieder nötig. Die ab diesem Punkt weiterführende Arbeit war geprägt durch eine hohe Produktivität, resultierend aus einer starken Kommunikation. Gestützt durch die Projektentwicklungsplattform GitHub entstand eine dynamische Zusammenarbeit mit der Möglichkeit spontan Aufgabenbereiche umzuverteilen, so dass die Arbeit trotz verringerter Gruppengröße erfolgreich fertig gestellt werden konnte.

Anhang

Glossar

Bibliothek/statische Bibliothek

ISO/OSI Modell: Ein informatisches Modell zur Darstellung der Kommunikation in einem Netzwerk. Dabei werden verschiedene „Schichten“ definiert, die jeweils unterschiedliche Aufgaben bei der Kommunikation übernehmen und aufeinander aufbauen. So gibt es die unteren, noch sehr hardwarenahen Schichten, die sich um die korrekte Übertragung von einzelnen Bits kümmern; Schichten darüber, welche für die Ankunft einzelner ***Paket***e am richtigen Ort zuständig sind bis hin zu Schichten, die Abläufe und Formate enthalten, die für die Darstellung und Übertragung von Websites zuständig sind.

Header: Ein Datensatz, der an den Anfang eines im Netzwerk verschickten Paketes geschrieben wird um dessen sichere Ankunft sicherzustellen oder ein Teil des Quelltextes, in dem bestimmte Datentypen und Strukturen definiert aber nicht deklariert sind

Paket, Netzwerkpaket: Informationseinheit, die von Computer zu Computer in einem Netzwerk verschickt wird

Protokoll, Netzwerkprotokoll: Spezifikation eines Teils des Headers eines Pakets; kann außerdem auch einen Ablauf für die Kommunikation zwischen den an der Verbindung beteiligten Rechnern beinhalten

Codereferenz

Übertragungstyp

Zahlcode	Interner Name	Funktion
0x10	MANGO_TYPE_INST	Anweisung
0x20	MANGO_TYPE_FILE	Datei

Vorimplementierte Dateitypen

Zahlcode	Interner Name	Funktion
1	Undefined	Undefinierter Typ
2	Movie	Film
3	Picture	Bild
4	Text	Text
5	Audio	Audio
6	Broken	für ungültige Übertragungen reserviert

Vorimplementierte Anweisungstypen

Zahlcode	Interner Name	Funktion
1	Exit	dieses Programm schließen und Verbindungen abbrechen
2	Kill	schließe ein bestimmtes Programm
3	GetFileList	Abrufen von Ordner/Dateistrukturen
4	GetPrgmList	Liste von bekannten Programmen abrufen
5	RetrieveFile	bestimmte Datei von Ziel nach Start senden
6	Execute	bestimmtes Programm starten, Argumente im Paket
7	Chat	Paket enthält Textnachricht
8	FileToBeSent	signalisiert beginnende Dateiübertragung
9	InvalidInstr	für ungültige Übertragungen reserviert

Vorimplementierte Programmcodes

Zahlcode	Interner Name	Funktion
1	This	dieses Programm
2	InvalidPrgm	für ungültige Übertragungen reserviert