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

Jan Sommerfeld Leonard Petereit Benedikt Schäfer

4. 12. 2018

Inhaltsverzeichnis

1 Entwurf und Umsetzung der Netzwerk-Schnittstelle

1.1 Grundlegende Theorie zur Kommunikation in einem Netzwerk

Unter dem Begriff Netzwerk im Zusammenhang mit dieser Arbeit wird, sofern es nicht ausdrücklich anders definiert ist, die Ansammlung mehrerer, untereinander verbundener Computer verstanden. Diese haben die Möglichkeit mit standardisierten Internet-Netzwerkprotokollen miteinander zu kommunizieren.
Diese Kommunikation wird beschrieben durch das ISO/OSI Modell (s. ***ISO/OSI Modell***) der Netzwerkkommunikation, auf dem auch die Arbeitsweise unseres Programmes aufbaut. Dabei kann man den Prozess, welcher bei einer Netzwerkübertragung abläuft und die verwendeten Informationen in Schichten nach ihrem Abstraktionsgrad einteilen, 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 einer Schicht führt, obwohl das gesamte System überaus komplex ist.

1.2 Notwendigkeit, Anforderungen und Spezifikation eines eigenen Netzwerkprotokolls

Um das Programm zu dritt 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 Anforderungen werden durch die Trennung der Programmiersprachen Lua und C++ einfach geregelt, aber in reinem C++ Kontext muss es dennoch eine Unterteilung zwischen Netzwerkschnittstelle und Benutzeroberfläche geben, weshalb die gesamte Netzwerkkommunikation über eine statische Bibliothek in die ausgeführte Datei eingebunden ist, in dem die Benutzeroberfläche und die Verbindung der Programmteile gemacht werden. Die Aufgabe der Bibliothek ist, Funktionen und Klassen bereitzustellen, die eine verlässliche Verbindung zwischen zwei Rechnern aufbauen (Start und Ziel), über die ein fester Satz von Anweisungen mit optionalen Argumenten sowie Dateien bidirektional übertragen werden können. 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 und
- die Übertragung auch von größeren Dateien mit zusätzlichen Informationen fehlerfrei ablaufen

Diese Bedingungen implizieren bereits, dass eine bestimmte Regelung und einen Ablauf für die Kommunikation zwischen den beiden Computern geben muss: ein Protokoll (s. ***Protokoll***).

Alle oben genannten Anforderungen müssen durch dieses Protokoll erfüllt sein und geregelt werden; daher umfasst es sowohl einen bestimmten Datensatz, der in den Header (s. ***Header***) eines Pakets (s. ***Paket***) geschrieben wird als auch einen festgelegten Ablauf der Kommunikation, der sicherstellt, dass alle Daten korrekt ankommen sind und verarbeitet werden.

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 auch sogar schon eine verschlüsselte Verbindung bereitstellt), also der Transportschicht des ISO/OSI-Modells, auf der bereits genau das implementiert und standardisiert ist - praktisch ist unser Protokoll also auf der Anwendungsschicht des Modells. Da das Protokoll sowohl für den Versand von Anweisungen als auch für den von Dateien geeignet sein muss und dabei so strukturiert und einfach wie möglich soll, gibt es zwei verschiedene Teilprotokolle, was in folgendem Aufbau der Kommunikationsstruktur resultiert:

Im ersten Schritt werden zwei Informationen, als Zahlen codiert, zum Typ der Übertragung, also Anweisung oder Datei, und zu ihrer Gesamtgröße in den Header des ersten Pakets der Übertragung geschrieben. Abhängig davon, was der Übertragungstyp ist, wird nun entweder die Informationen für eine Dateiübertragung oder eine Anweisungsübertragung angehängt. Dabei sind alle Werte, sofern es möglich ist, als Zahlen und nicht als Zeichenketten vorhanden, um die Größe der Informationen zu minimieren.

Betrachtet man zunächst den Header für die Anweisungen, so braucht man einen Anweisungscode, der am Zielort eindeutig zu einer Anweisung zugeordnet werden kann; Teil zwei sind Standardargumente, die binär auf 32 Bit Speicherbreite gespeichert werden. Dabei soll zum Beispiel, wenn das erste Bit auf eins gesetzt wurde, die Anweisung „Rufe eine Liste der Daten ab, auf die das Programm Zugriff hat“ nur die Ordnerstruktur herüberschicken, ist das zweite Bit auf eins gesetzt soll es zusätzliche Informationen zur Größe oder den Zugriffsrechten auf die Dateien mitsenden; dafür wären anderenfalls eigene Anweisungen nötig, was der Übersichtlichkeit bei der Implementierung abträglich wäre und deshalb bereits im Protokoll integriert ist.

Der dritte Wert im Headersegment für die Anweisungen ist eine Zahl, die ein Programm identifiziert, auf dass die Anweisung angewendet werden soll, was standardmäßig das Eigene ist.

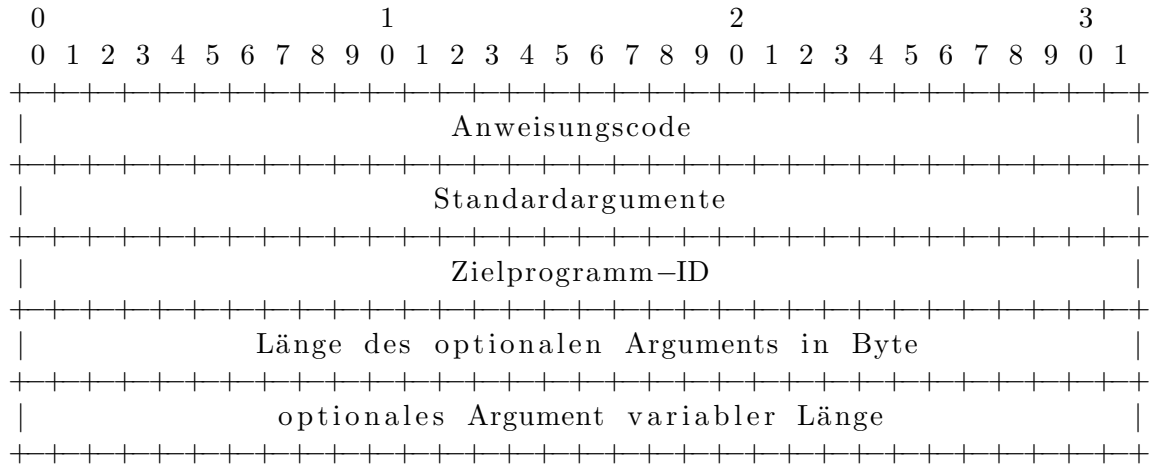


Abbildung 1: Header für Anweisungen

Der vierte Wert ist eine Längenangabe in Bytes. Er bezeichnet die Länge eines optionalen Argumentes, welches an den Header angehängt werden kann, beispielsweise ein Dateiname, eine Chat-Nachricht oder ein Bash-Befehl.

Um die ganze Anweisung klein zu halten wird intern festgelegt, dass die Gesamtgröße dieses Headersegments inklusive des optionalen Arguments nicht größer als 2^{16} Byte sein darf - größere Anhänge müssen als Datei verschickt werden. Aus Kompatibilitätsgründen mit verschiedenen Compilern und Systemen sind alle Werte Integer mit 32 bzw. 64 Bit Speicherbreite, um damit

sowohl genügend Platz für alle Informationen zu bieten als auch Probleme mit verschiedenen Rechnerarchitektur-spezifischen Anpassungen der Speicherbreite zu vermeiden. Insgesamt ergibt sich also die in Abbildung 1 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.

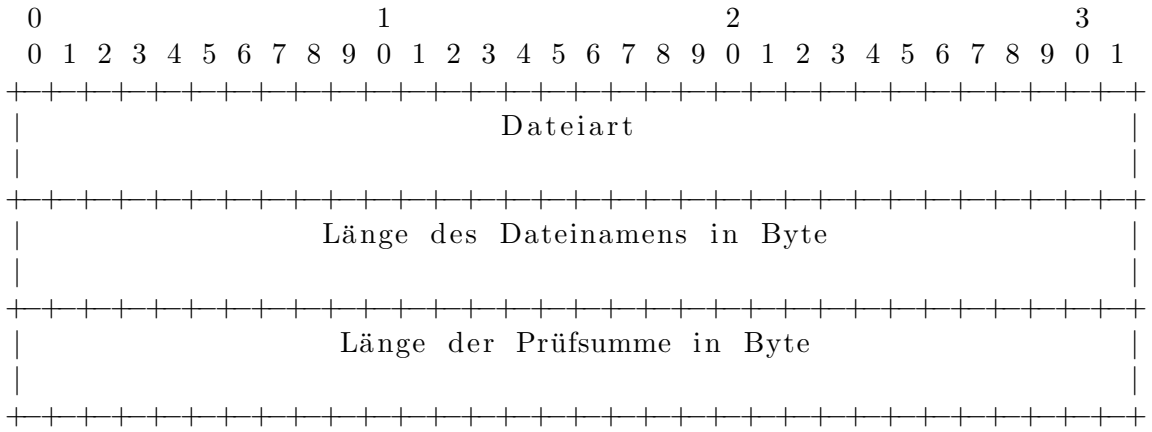


Abbildung 2: Header für Dateiübertragungen

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

Für jede Dateiübertragung wird, im Gegensatz zu den Anweisungen, jeweils eine neue TLS-Verbindung aufgebaut(Schritt 0), die unabhängig von vorhergehenden und nachfolgenden ist, es gibt also für jede Datei einen abgeschotteten „Kanal“, in dem Fehler leicht behandelbar und konkurrierende Übertragungen unmöglich sind; außerdem wird die Kennzeichnung einzelner Datenpakete gespart, was wiederum zu insgesamt weniger Datenverkehr führt. Danach wird ein Paket mit Informationen zu der Datei zum Zielcomputer geschickt(Schritt 1), welcher, um die Integrität der Übertragung zu verifizieren, dasselbe Paket zurücksendet(Schritt 2).

Dieses Informationspaket ist in Abbildung 2 schematisch dargestellt. Es besteht aus der Art der Datei (z.B. Video, Audio, Text, ausführbare Dateien etc.), was in der weiteren Einordnung im Zielcomputer nützlich ist, dem ursprünglichen Dateinamen und einer Prüfsumme, mit der die fehlerfreie Ankunft verifiziert werden kann.
Das Headersegment ist daher so aufgebaut, dass zunächst die drei Ganzzahlwerte für Typ, Länge des Dateinamen in Byte und Länge der Prüfsumme in Byte in den Header geschrieben

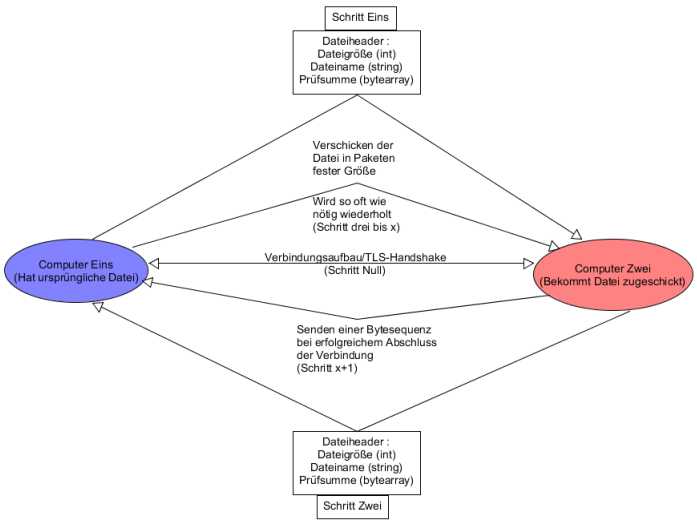


Abbildung 3: Schematische Darstellung des Ablaufs bei der Dateiübertragung

werden. Dann werden Dateinamenslänge und 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 kleinen Abschnitten und ohne weitere Kennzeichnung in mehreren kleinen Paketen geschieht(Schritt 3 bis x). 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 der Managerklasse ausgesendet und die Verbindung getrennt. Diese Schrittfolge ist schematisch in Abbildung 3 dargestellt. Nach dem erfolgreichen Übertragen wird noch mit der anfänglich verschickten Prüfsumme der Datei die Integrität der Datei geprüft, um Fehler insgesamt auszuschließen.

1.3 Funktionsumfang der selbstgeschriebenen Klassenbibliothek

Die Bibliothek stellt auf oberster Ebene eine Schnittstelle zum Übertragen von Dateien und Anweisungen zu einer bekannten Adresse, welche in angemessener Form codiert sind. Das heißt konkret, es werden Methoden bereitgestellt, welche die einzelnen Informationen in das passende Format das einfach verschickt werden kann umwandeln, dargestellt durch zwei Klassen, die die nötigen Informationen, also z.B. Dateiprüfsummen, Dateinamen, Anweisungen usw., so verpacken, dass diese dann einfach vom darunterliegenden System verarbeitet werden können. Zusätzlich dazu ist natürlich eine entsprechende Fehlerbehandlung bei Verbindungsabbrüchen oder fehlerhaften Übertragungen und die Möglichkeit eine Fortschrittsmeldung bei der Dateiübertragung implementiert. Insgesamt muss der Programmierer nur mit zwei Klassen interagieren, um die volle Funktionalität der Bibliothek ausnutzen, was das gesamte Konstrukt sehr gut für weitere Verwendungen in flexiblen Kontexten der modularen Programmierung eignet und gute Modularität bietet.

1.4 Implementierung und Funktionsweise der Klassenbibliothek

In der Bibliothek sind neben den beiden oben beschriebenen Manager-Klassen (MngThManager für Dateien und MngFileManager für Dateien) mehrere andere Klassen definiert, welche 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 denen die Datenstrukturen und Datentypen, die in den Header der Pakete geschrieben werden, als C-Structs definiert sind und einige Funktionen, die häufig gebraucht werden (z.B. String-Vervielfältigung, eine hexadezimale Ausgabe und eine Funktion zum errechnen der Dateiprüfsumme) deklariert; die Definition erfolgt in einer zugehörigen Datei.

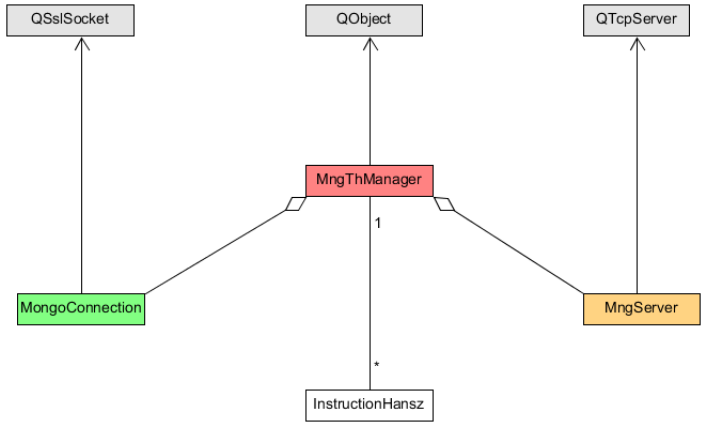


Abbildung 4: Schematisch: Anweisungsversand

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: (*FileHansz* bzw. *InstructionHansz*), sowie andere Klassen zum Verbindungsaufbau und dem Versand: Eine Klasse, die für das Annehmen von hereinkommenden Verbindungen zuständig ist und eine(bei Dateien zwei) Socket-Klassen, die sich um das Senden bzw. Empfangen von Daten kümmern, was in Abb. 4 bzw. 5 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 dann nur noch ausgelesen und verschickt werden muss - somit ist ein gleicher Aufbau der Verbindung in beide Richtungen sichergestellt.

Das tatsächliche Versenden, sowie der Verbindungsaufbau wird von einer „Server“-Klasse und den Socket-Klassen übernommen. Dabei horcht die Serverklasse, 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 inklusive 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 Zielcomputer gesendet(1), welcher genau dasselbe Paket dann zurückschickt(2) um sicherzustellen, dass alle Steuerinformationen korrekt angekommen sind. Jetzt wird die eigentliche Übertragung der Datei gestartet, welche mit dem Auslesen eines Datenblockes fester Größe beginnt(3). Dieser wird in einen Pufferspeicher geschrieben, der, wenn er voll ist von der Socket-Klasse ausgelesen und übertragen wird(4); diese Anweisungsfolge(3 und 4) 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 der Datei-Prüfsumme benannten Datei im Dateisystem 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 dann weitergehend speichert. Sind alle Pakete versandt worden und angekommen, wofür die TCP-basierte Verbindung sorgt, wird vom Empfangenden Rechner eine Bytesequenz

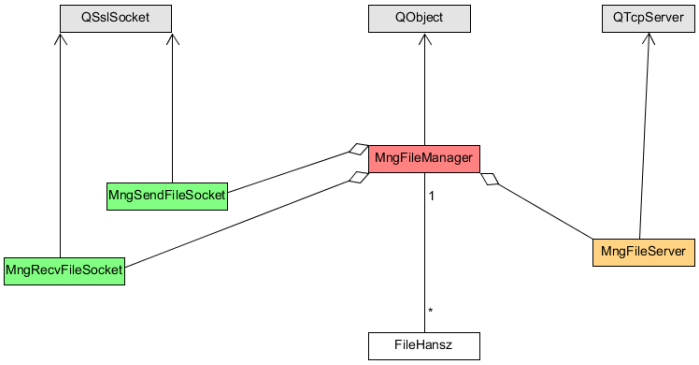


Abbildung 5: Schematisch: Dateiversand

übertragen, welche die Erfolgreiche Übertragung kennzeichnet.

Danach wird die Verbindung abgeschlossen und die beiden Sockets gelöscht.

2 Funktionsumfang und Eingabeverarbeitung auf der Anwendungsebene

2.1 Entwicklung einer eigenen Scriptsprache zur Anwendungssteuerung

2.1.1 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, indem 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, oder 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 nach welcher Art 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.

2.1.2 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

[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	-

Maßen 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	ip_address file_name file_type
send_comm	ip_address comm_name [arguments]
get_file	ip_address file_name file_type
root_mode	[duration]
shutdown [send_comm]	ip_address
open [send_file]	ip_address file_name file_type

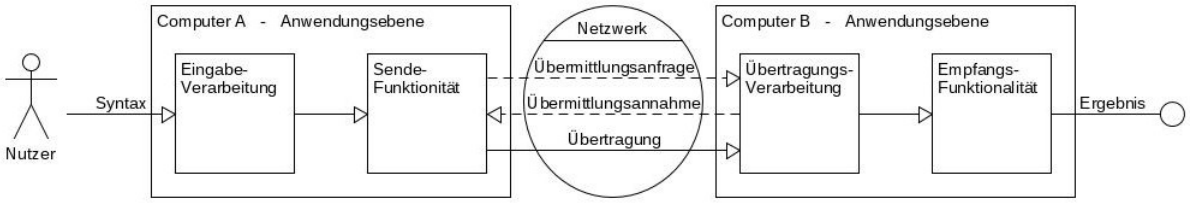
Die Befehle *send_file* und *send_comm* sind hierbei die allgemeinen Funktionen zum Senden einer Datei oder eines Befehls an die betriebssystemeigene Kommandozeile, jeglicher Art. Das Kommando *get_file* ist identisch zu *send_file* mit dem Unterschied, dass eine Datei von dem angefragten PC auf den anfragenden übertragen werden soll. Durch die $root_mode$ Befehl werden alle Übermittlungsanfragen automatisch als akzeptiert gewertet, was das Fernsteuern von Computern ermöglichen soll. *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 symbolisch für die Erweiterbarkeit des Funktionsumfangs unserer Software. In dem Fall, dass eine neue Verbindung zu einem anderen Computer aufgebaut wird, muss zwangsweise mit jeder getätigten Anweisung an das Programm eine IP-Adresse des Empfängercomputers der Form IPv4 oder IPv6 als Argument übergeben werden. Zusätzlich für Befehle, welche mit Dateien arbeiten sollen, müssen sowohl der Dateiname als auch das Dateiformat als Zeichenkette angefügt werden. Die Übermittlungsfunktionen für Anweisungen benötigt neben dem Anweisungsnamen als Zeichenkette auch optionale Argumente, mit welchem die Anweisung in der Kommandozeile ausgeführt werden soll.
Um unsere formale Sprache zu implementieren, entwickelten wir zuerst 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.

2.2 Implementierung der Anwendungssteuerung in der Skriptsprache Lua

2.2.1 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 werden zum einen auf dem

Rechner der sendenden Person die nach der in Kapitel 3.1 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.



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

Wie in Kapitel 3.2.1 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 diese Softwareebene in einer Skriptsprache zu implementieren. Diese Klasse von Programmiersprachen kennzeichnet sich dadurch aus, dass der vom Menschen lesbare Quelltext eines Programmes erst bei seiner Ausführung in Anweisungen übersetzt werden, welche für den Computer ausführbar sind. Im Gegensatz zu kompilierten Sprachen hat dies den Vorteil, dass Programmcode unkompliziert korrigiert und modifiziert werden kann.
In diesem Zusammenhang ergibt sich außerdem eine leichte Wartbarkeit, also die Möglichkeit eventuell auftretende Fehler zu beseitigen, als ein weiterer erfüllter Anspruch. Da wir stets zu dritt an unserer Software programmiert haben, sollte es jedem von uns mit möglichst geringem Aufwand möglich sein, Programmteile zu verbessern, auch wenn es sich nicht um den ursprünglichen Autor handelt.
Den Vorteil von kompilierten Programmiersprachen, dass sie sich durch eine höhere Leistungsfähigkeit besonders bei komplexeren Problemen auszeichnen, haben wir uns durch die Implementierung unserer Netzwerkteils in der Sprache C++ zu nutzen gemacht. Dies stellt den Anspruch, dass die gewählte Skriptsprache problemlos in Software der Sprache C++ einzubetten ist, und keine Probleme bei der Kommunikation zwischen Programmen der zwei 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 Computersystemen und wir haben mobile Geräte vorerst außen vor gelassen. Zu guter Letzt ist der Zeitraum der Seminarfacharbeit auch nur

auf eineinhalb Jahre begrenzt, weshalb die gewählte Sprache mit einem geringem 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 vor, 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.

2.2.3 Erläuterung wesentlicher Elemente der Implementierung

Eine zentrale Rolle in der Umsetzung unserer Eingabeverarbeitung spielt die Interpreter Funktion. Ihre Aufgabe besteht darin, die von dem Nutzer in Form einer Zeichenkette gelieferten Eingabe, in eine für das Programm verständliche Anweisung zu übertragen. Sie ruft außerdem die Funktionen auf, welche die gewünschte Funktionalität implementieren, und übergibt ihnen die eingegebenen Parameter zur weiterführenden Überprüfung.

Listing 1: Interpreter Funktion

```
function interpret_input(ui_input)
    local content = split_input(ui_input)
    local commands = {
        ["send_file"]=0,
        ["send_comm"]=0,
        ["get"]=0,
        ["open"]=0,
        ["shutdown"]=0}
    if commands[content[1]]~=NIL then
        local result = _G[content[1]](content)
        print(result)
    else
        local name = debug.getinfo(1, "n").name..": "
        local subject = string.format("%q",content[1])
        print("ERROR: "..name..subject.." - unbekannter Befehl!")
    end
end
```

Die Variable *content* 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 nur, dass die Variablen initialisiert sind und nicht keinen zugeordneten Wert haben. In der folgenden $If - Else$-Bedingung 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 Befehlssatzes 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

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

$.G[STRING](ARGUMENTS)$, 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 Anzahl der eingegebenen Parameter mit der für den Befehl benötigten übereinstimmt. Ist dies der Fall, so wird eine erste Übertragung mit dem Zweck der Autorisierung gestartet. Es wird eine zu bestätigende Anfrage an den Empfänger-PC geschickt, welche standardmäßig nach 10 Sekunden ausläuft. Wenn eine Bestätigung in besagtem Zeitraum stattfindet, so wird die eigentliche Übertragung initialisiert. Schlägt eine der Bedingungen fehl, so wird der Vorgang abgebrochen und eine Fehlermeldung entsprechen der oben dargestellten Form ausgegeben.

Listing 2: Beispielhafte Sende Funktion

```
function send_NAME(args)
local name = debug.getinfo(1, "n").name
    if get_length(args)==INTEGER then
        local ARG1 = args[2]
        ...
        local ARGN = args[N+1]
        if authenticate(ip) then
            local blank = "c_sendNAME(ARG1, ..., ARGN))"
        else
            local type = " - Uebertragung wurde abgelehnt"
            print("ERROR: "..name..type)
        end
    else
        local type = " - Argumentenzahl unpassend"
        print("ERROR: "..name..type)
    end
end
```

Anhang

Glossar

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 ist.
Paket, Netzwerkpaket:	Informationseinheit, die von Computer zu Computer in einem Netzwerk versendet 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
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.