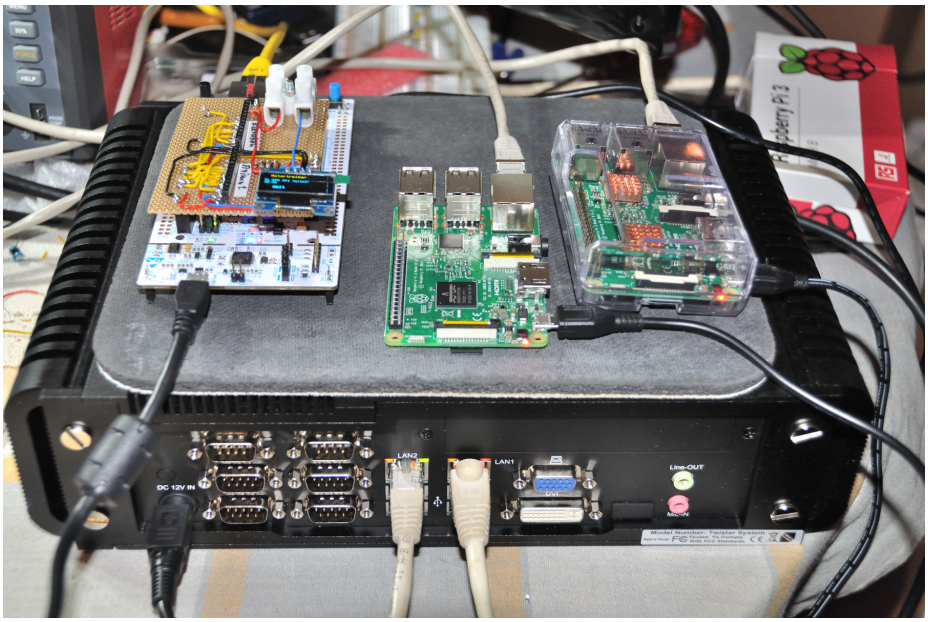
Jugend Forscht 2017

Christian-Gymnasium Hermannsburg

Strukturierte systemübergreifende Kommunikation mit TCP/IP

Tim Rambousky



betreut durch
StD Thomas Biedermann

15. Januar 2017

Inhaltsverzeichnis

1 Einleitung

1.1 Problemstellung

Die Radioastronomie-AG an unserer Schule beschäftigt sich schon seit mehreren Jahren mit dem Bau und der Inbetriebnahme eines Radioteleskops. Mit diesem Radioteleskop wollen wir elektromagnetische Strahlung von Objekten außerhalb unseres Sonnensystems untersuchen.

Deshalb wurden in den vergangenen Jahren bereits eine Reihe von Jugend Forscht Projekten erarbeitet, die Grundlagen in der für das Projekt notwendigen Messtechnik und (Motor-)Steuerung legten. Auch wurden zwei Radioteleskope gebaut, jeweils mit einem Reflektordurchmesser von 1,8 m und 2,6 m, um Erfahrungen mit einem solchen System zu sammeln und geplante Konzepte zu testen. Das finale Radioteleskop hat einen Reflektordurchmesser von 4 m und wurde Ende 2016 auf dem Sportplatz des Christian Gymnasiums Hermannsburg aufgebaut. Es soll später dauerhaft durch den Verein *Sternwarte Südheide e.V.* betrieben werden.

Da die Mechanik zur Bewegung des Teleskops nun einsatzbereit ist, hat sich unsere AG zum Ziel gesetzt, Messtechnik, Motorsteuerung und Steuerungssoftware selbst zu entwickeln. Die Idee zu meinem Projekte kam aus der Überlegung, wie man langfristig effektiv Messungen mit dem Radioteleskop vornehmen kann. Dafür muss der Computer der die Messung steuert mit der Motorsteuerung und dem Messsystem, welche jeweils auf einzelne Steuereinheiten ausgelagert sind, kommunizieren können. Diese Kommunikation muss über eine standardisierte Schnittstelle laufen, da wir unterschiedliche Typen von Steuereinheiten benutzen und zusätzlich in der AG auch mit unterschiedlichen Betriebssystemen (Windows und Linux) arbeiten.

Bedingt durch den Einsatz der beiden Betriebssysteme verwenden wir in unserer AG die zwei Programmiersprachen *C#* und *Python*. Die *Visual C#*-Umgebung läuft nicht unter Linux. Eine standardisierte Schnittstelle ermöglicht unseren AG-Mitgliedern weiterhin Softwarekomponenten, die in der Programmiersprache *C#* geschrieben werden, in das Radioteleskop-Projekt einzubringen.

Deshalb haben wir uns für eine Kommunikation mit TCP/IP über ein Netzwerk entschieden. Damit ist die Kommunikation stark standardisiert und für alle Betriebssysteme, Programmiersprachen und Steuereinheiten verfügbar.

1.2 Projektidee

In meinem Projekt möchte ich die Grundlage für eine Kommunikation über TCP/IP für unser Radioteleskop-System entwickeln. Dafür schreibe ich in den Programmiersprachen *C#* und *Python* ähnliche Klassen, die das Senden und Empfangen von Textnachrichten über TCP/IP auch auf verschiedenen Betriebssystemen ermöglichen. Diese Klassen implementiere ich in meine Testprogramme oder in Projekte anderer AG-Mitglieder, die damit extern auf unsere Steuereinheiten zugreifen können. Außerdem erarbeite und teste ich eine Netzwerkstruktur, die nicht nur für ein grundlegendes Senden von Textnachrichten sorgt, sondern auch langfristig für eine strukturierte Kommunikation zwischen

den verschiedenen Komponenten unseres Radioteleskop-Systems ausgelegt ist.

2 Grundlagen

2.1 TCP/IP

Transmission Control Protocol/Internet Protocol (TCP/IP) ist eine Familie aus grundlegenden verbindungsorientierten Netzwerkprotokollen, die den Datentransport über ein Netzwerk regeln. Dabei werden die Rechner/Systeme in einem Netzwerk anhand ihrer IP-Adressen identifiziert. Eine IP-Adresse (IPv4) hat folgende Form: 192.168.178.39.

Dabei ist jeder der vier Zahlenwerte ein Byte und kann somit Werte zwischen 0 und 255 annehmen. Zu jeder IP-Adresse gehört auch eine Netzwerkmaske, die angibt, welcher Teil der Adresse zum Netzwerk und welcher zum System gehört. Ein Beispiel für eine Netzwerkmaske wäre 255.255.255.0, zusammen schreibt man dann 192.168.178.39/255.255.255.0. Diese Netzwerkmaske gibt an, dass die ersten drei Zahlenwerte der IP-Adresse zum Netzwerk gehören, der letzte Zahlenwert spezifiziert das System. 192.168.178.xxx ist also die Adresse des Netzwerks, die 39 verweist auf das System. Alternativ kann man als Netzwerkmaske auch die Anzahl der Bits, die die Netzwerkmaske darstellt, angeben: 192.168.178.39/24

2.2 Sockets

In *Python* und in *C#*, sowie in den meisten anderen Programmiersprachen, benutzt man Sockets für die Netzwerkkommunikation. Das Prinzip dahinter funktioniert wie folgt: Wenn ein Programm Daten über eine Netzwerkschnittstelle empfangen oder senden möchte, dann meldet es dies beim Betriebssystem an und bekommt ein sogenanntes Socket (dt. Steckdose) zugewiesen. Mit diesem Socket kann das Programm eine Netzwerkverbindung zu einem anderen Socket aufbauen. Um Sockets zu identifizieren werden die IP-Adressen der anderen Systeme benutzt. Da aber auf einem System mehrere Sockets laufen können, werden die Netzwerkverbindungen zusätzlich an einen Port gebunden. Beim Port handelt es sich um eine 16-Bit Zahl, grundsätzlich stehen also 65.536 Ports zur Verfügung. Allerdings sind einige für spezielle Protokolle und Anwendungen reserviert.

2.3 Client-Server-Systeme

Die beiden Kommunikationspartner einer Netzwerkverbindung heißen Server und Client. Der Server ist der passive Kommunikationspartner und im Netzwerk unter einer festen IP-Adresse und einem festen Port zu erreichen. Er bindet sich an diese, wartet auf Verbindungsanfragen und akzeptiert diese. Diese Verbindungsanfragen kommen vom Client, dem aktiven Kommunikationspartner. Nachdem eine Verbindung hergestellt wurde, können beide untereinander Daten senden und empfangen.

direction	*device*	*param*	*response*	*Bemerkungen, Beispiel*
GET	AZM	POS	AZM < pos>, <stat>	<pos> (float) aktuelle Position <stat> RUN | RDY | ERR „GET AZM POS“ → „AZM 115.0, RDY
SET	AZM	PAR <par>, <value>	AZM PAR <par>, <errno>	<par> Kürzel des Parameters <value> Wert des Parameters <errno> 0: Parameter und Wert in Ordnung 1: ungültiger Parameter 2: ungültiger Wert „SET AZM PAR MAX, 63“ → „AZM PAR MAX, 0“ “SET AZM PAR MOX, 63” → “AZM PAR MOX, 1” “SET AZM PAR MAX, 1.3” → “AZM PAR MAX, 2”
SET	AZM	TXT <zeile>, <spalte>, <size>, <text>	AZM TXT <ermo>	<zeile> Startposition Pixelzeile <spalte> Startposition Pixelspalte <size> Schriftgröße in Pixel <text> Auszugebender Text <errno> 0: kein Fehler aufgetreten 1: ungültiger Parameter „SET AZM TXT 1,1,8,Hello World“ → “AZM TXT 0”

Abbildung 1: Tabelle mit Kommandos an den Azimutmotor nach dem erstellten Kommandosyntax

2.4 Kommandosyntax

Um die Kommunikation zwischen den Komponenten unseres Systems weiter zu strukturieren haben wir uns im Rahmen unserer AG auf einen einheitlichen Kommandosyntax für die von uns verwendeten Textnachrichten erstellt. Dieser geht davon aus, dass sich jede Komponente unserer Steuereinheiten als Eigenschaft auffassen lässt. Diese lassen sich auslesen (GET) und setzen (SET). Die Kommandos setzen sich aus folgenden Teilen zusammen.

Syntax: <direction> <device> [<param 1> [, <param 2> ...]]

direction: GET oder SET (bzw. GTB und STB für “blocked”-Ausführung)

device: anzusprechendes Device

param n: optionale(r) Parameter

Jedes Kommando wird mit einer Quittung <response> beantwortet. Ersetzt man in den folgenden Beispielen GET bzw. SET durch GTB bzw. STB, wird die Ausführung des nächsten Kommandos erst fortgesetzt, wenn das vorhergehende Kommando vollständig ausgeführt worden ist. Damit ist es möglich, auf Warteschleifen oder Statusabfragen zu verzichten. Dies ist vor allem dann entscheidend, wenn mehrere Kommandos in einem Script zusammengefasst werden sollen, um eine automatische Messung durchzuführen.

Die Abb. 1 zeigt einige Beispiele für Kommandos für den Azimutmotor(AZM), der Teil der Motorsteuerungseinheit ist.

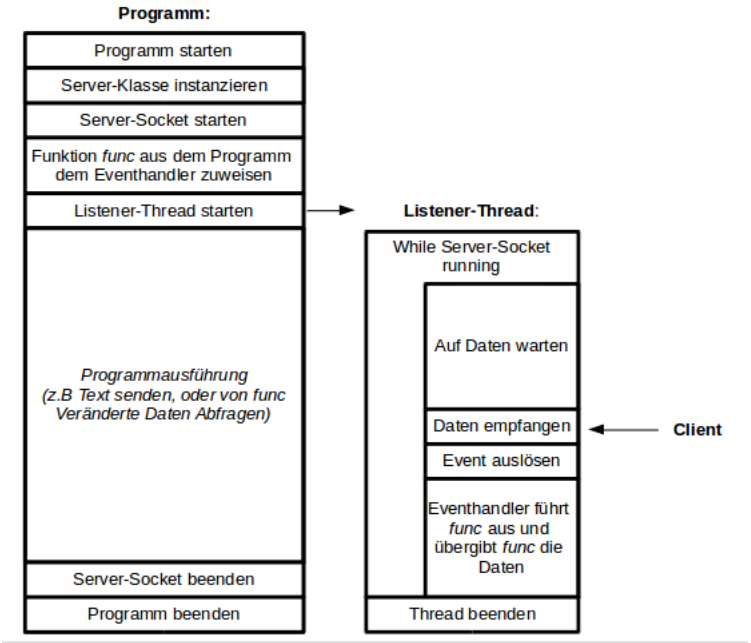


Abbildung 2: Struktogramm eines Programms mit implementierter Server-Klasse

3 Die Server- und Client-Klassen

Die wichtigste Aufgabe meiner Arbeit ist es Klassen zu schreiben, die eine Netzwerkkommunikation ermöglichen. Diese müssen in den Programmiersprachen *Python* und *C#* angefertigt werden.

3.1 Struktur der Klassen

Die Struktur der Klassen ist in *C#* und *Python* fast identisch und soll hier genauer betrachtet werden. Es gibt jeweils eine Klasse für Server-Sockets und Client-Sockets. Beiden wird beim Instanzieren über einen Konstruktor eine IP-Adresse und ein Port übergeben. Dem Server-Socket wird die IP-Adresse übergeben unter der er im Netzwerk erreichbar sein soll, dem Client-Socket die des Server-Sockets mit dem er sich verbinden soll.

Der einzige Unterschied zwischen den Klassen besteht in der Art wie die Verbindungen aufgebaut und geschlossen werden. Während der Client-Socket sich mit einem Server-Socket verbindet und diese Verbindung wieder beenden kann, wartet der Server-Socket nur auf eine Verbindung. Zusätzlich zum kompletten Beenden des Server-Sockets kann auch nur die aktuelle Verbindung beendet werden, sodass der Server-Socket wieder auf eingehende Verbindungen wartet. Ansonsten sind die Server- und Client-Klassen identisch.

Zur weiteren Erläuterung der Klassen zeigt Abb. 2 ein Struktogramm, das ein Beispielprogramm darstellt in das die Serverklasse implementiert wurde. Nachdem das Programm gestartet wurde wird die die Serverklasse instanziert und der Server-Socket ge-

startet. Darauf wird dem Eventhandler der Serverklasse eine Funktion *func* zugewiesen, die immer dann ausgeführt wird, wenn der Server-Socket Daten empfängt. Ein mögliches Beispiel für eine solche Funktion wäre das Bearbeiten der empfangenen und das Zurücksenden der modifizierten Daten. Nachdem eine Verbindung eingegangen und akzeptiert wurde, wird der Listener-Thread gestartet. Ein Thread ist immer ein zum Hauptprogramm parallel ablaufender Prozess. Während das Programm weiter läuft und eigene Funktionen ausführt, die hier nicht weiter relevant sind, wartet der Listener-Thread in einer While-Schleife auf eingehende Daten. Empfängt er solche führt er mittels des bereits erwähnten Eventhandlers die ihm zugewiesene Funktion *func* aus. Wird das Programm, und somit der Server-Socket, beendet, verlässt der Listener-Thread seine Schleife und beendet sich somit selbst.

3.2 Testprogramme

Um meine Klassen zu testen benutze ich Programme mit einer graphischen Oberfläche. In *C#* arbeite ich mit *Visual Studios*, in *Python* mit dem *QT-Designer* um die Oberflächen zu entwerfen. Die Oberflächen beider Programme sind nahezu identisch.

In dem Abschnitt „Connection“ lassen sich IP-Adresse und Port übergeben und mit dem „Set“-Knopf speichern (siehe Abbildungen 3 und 4). Mit den Knöpfen „Connect“ und „Disconnect“ lassen sich Verbindung aufbauen und beenden. Bei der Oberfläche für einen Server steht dort entsprechend „Start“ und „Stop“. Ob eine Verbindung besteht wird in der Statusleiste angezeigt.

Im Abschnitt „Communication“ lassen sich Textnachrichten eingeben und mit dem „Send“-Knopf senden. Nachrichten die gesendet wurden werden im Textfeld unter Client ausgegeben, empfangene Nachrichten im Textfeld unter Server. Bei der Server-Oberfläche ist dies entsprechend umgekehrt. Die Einbindung der Klasse in das Programm mit der graphischen Oberfläche funktioniert wie in *Abschnitt 3.1* beschrieben. Dem Eventhandler wurde eine Funktion zugewiesen, die eine Nachricht in einem Textfeld ausgibt.

Für das plattformübergreifende Testen habe ich unter Windows einen *C#*-Client laufen lassen und unter Linux einen *Python*-Server (eine leere IP-Adresse bezeichnet hier die IP des aktuellen Systems). Zuerst wird der Server gestartet, danach versucht der Client sich zu verbinden. Nachdem die Verbindung hergestellt wurde, werden von beiden Seiten Nachrichten gesendet. In den Abbildungen 3 und 4 ist zu erkennen, dass die versendeten Textnachrichten beim Partner angekommen sind. Damit ist gezeigt, dass die Klassen funktionieren und die Anforderungen für unser Projekt erfüllen.

4 Netzwerkstruktur

Unser Radioteleskop-System benötigt eine Netzwerkstruktur in der die einzelnen Komponenten sicher und einfach miteinander kommunizieren können. In diese Struktur wollen wir einen Server einbauen, den wir hier, um Verwechslungen zu vermeiden, *Server-PC* nennen. Dieser soll zu einem späteren Zeitpunkt einige Berechnungen übernehmen. Daher muss die gesamte Kommunikation über ihn erfolgen. In späteren Versionen soll zu-

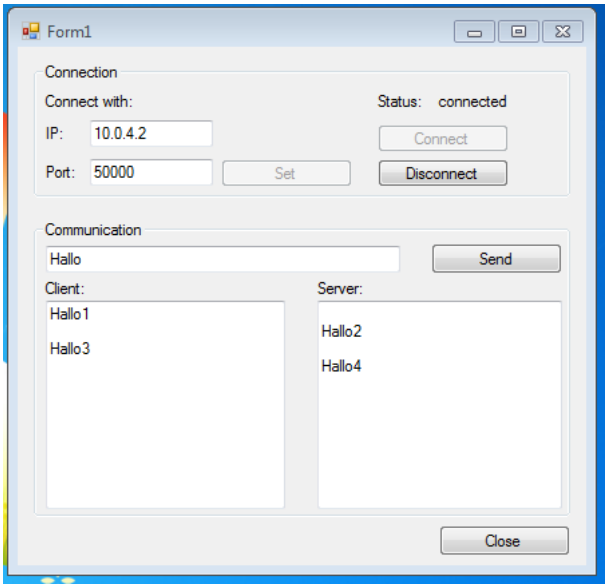


Abbildung 3: Testoberfläche der Client-Klasse in *C#* in einer Windows Umgebung

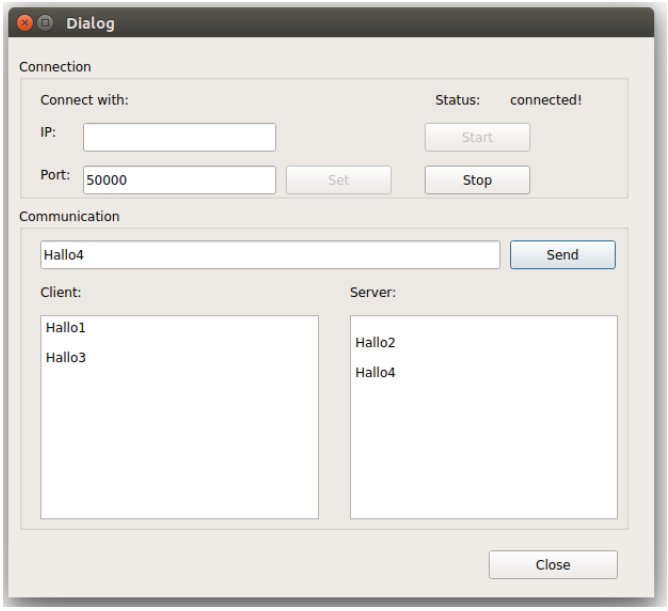


Abbildung 4: Testoberfläche der Server-Klasse in *Python* in einer Linux Umgebung

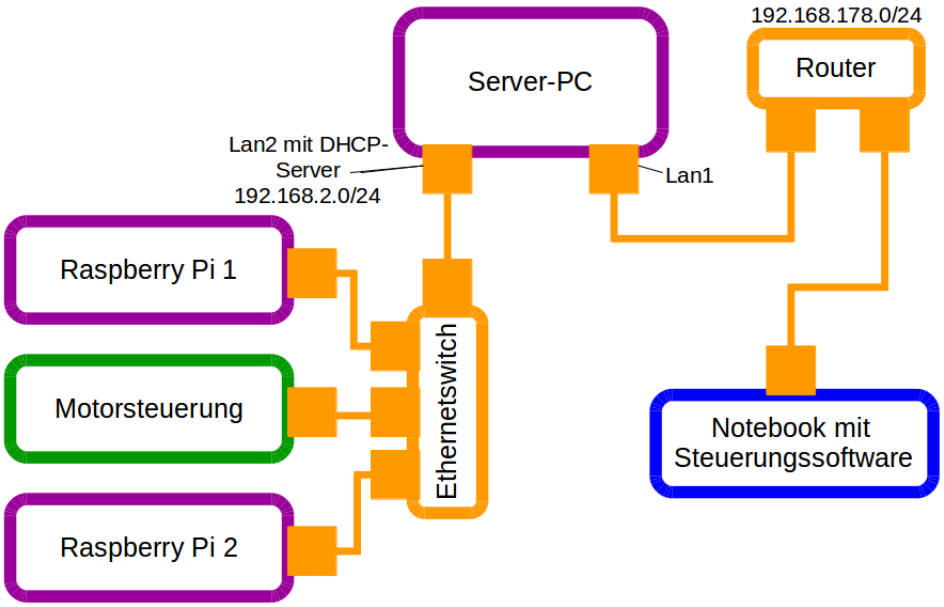


Abbildung 5: Struktur des Netzwerkes zur Steuerung des Radioteleskop-Systems

sätzlich die Kommunikation selbst als Textdatei aufgezeichnet und gespeichert werden können. Die künftige Implementation, die ich auch im weiteren Verlauf dieser Arbeit verwenden werde, zeigt Abb. 5.

Der Server-PC hat zwei Netzwerkschnittstellen (*Lan1* und *Lan2*). Mit *Lan1* ist er an das Netzwerk eines Routers mit Internetzugang angebunden, auf *Lan2* sind in einem Subnetz Steuereinheiten des Radioteleskops angeschlossen. Auf dem Notebook im Routernetzwerk (siehe Abb. 5) befindet sich die Steuerungssoftware, die Befehle über den Server an die Steuereinheiten sendet und anschließend eine Rückmeldung erhält.

Diese Struktur hat den Vorteil, dass Komplexität aus der Steuerungssoftware genommen wird und diese damit einfacher an verschiedene Aufgaben angepasst werden kann. Außerdem lässt sich die Struktur leichter um zusätzliche (künftige) Module und Funktionen erweitern, wie beispielsweise eine manuelle Steuerung der Motoren, eine Videoüberwachung des Radioteleskops oder eine Steuerung des Systems über das Internet. Zusätzlich wird durch die Überwachung der Kommunikation die Fehlersuche erleichtert.

Der Server-PC soll die Daten später verarbeiten und speichern. Daher reicht es nicht aus die beiden Netzwerke mit einer Bridge zwischen den beiden Netzwerkschnittstellen des Server-PCs zu verbinden, um Daten zwischen den Netzwerken zu versenden. Es muss ein eigenes Programm geschrieben werden, welches diese Aufgabe übernimmt.

4.1 Der Server-PC

Bei dem Server-PC handelt es sich um einen Industrierechner der Marke *Ivy Bridge IPC* des Herstellers *Amerry*. Dieser läuft mit der Linux Distribution *Ubuntu-Server* als Betriebssystem. Folglich werden alle Programme, die auf dem Server-PC laufen sollen,

in *Python* geschrieben.

4.1.1 DHCP-Server

Im Subnetzwerk der Netzwerkschnittstelle *Lan2* des Server-PCs befindet sich kein Router. Somit bekommen die Steuereinheiten keine IP-Adresse zugewiesen, wie es bei einem Netzwerk mit Router der Fall ist. Deshalb richte ich an *Lan2* einen DHCP-Server ein. Dieser wird von den Steuereinheiten automatisch erkannt und sie schicken eine Anfrage für eine IP-Adresse an den DHCP-Server, welcher ihnen dann eine aus seinem gültigen Adressbereich zuweist.

Da wir die Steuereinheiten unter einer festen IP-Adresse erreichen wollen wurde der DHCP-Server so eingerichtet, dass er die MAC-Adresse (eine Adresse, die für jedes Gerät einzigartig ist) der Steuereinheiten erkennt und ihnen immer die gleiche IP-Adresse zuweist.

4.1.2 Das Server-PC-Programm

Im Server-PC-Programm laufen mittels der Server- und Client-Klassen parallel ein Server-Socket und drei Client-Sockets in Server-und Client-Threads. Der Client-Socket auf dem Notebook mit der Steuerungssoftware verbindet sich mit dem Server-Socket, die drei Client-Sockets verbinden sich je mit einem Server-Socket auf einer der Steuereinheiten. Aufgrund der Kommandosyntax der Textnachrichten kann der Server-Thread entscheiden an welchen der drei Client-Threads er die Nachricht weiterleiten soll. Für den Austausch zwischen den Threads auf dem Server-PC gibt es pro Client-Thread eine Variable in der Daten zwischengespeichert werden und ein Event, welches dazu benutzt wird dem parallel laufenden Server-Thread das Vorhandensein von Daten zu signalisieren, welcher diese anschließend abfragen und weiter senden sollen. Da der Server zwischen den drei Client-Threads unterscheiden muss, sind für ihn zum Austausch ebenfalls eine Variable, aber 3 Events (eins pro Client-Thread) eingebaut. Diese Events sind nicht Teil des Eventhandlers der Server-/Client-Klasse, sondern des für die Threads benutzten Threading-Modules der *Python*-Standartbibliothek.

Die Abb. 6 zeigt ein Struktogramm, welches den Ablauf des Server-Threads und eines der Client-Threads zeigt. Die Unterscheidung zwischen anderen Client-Threads wird hier vernachlässigt und im Vergleich zu Abb. 2 wurden Vereinfachungen vorgenommen, um das Struktogramm übersichtlich zu halten.

Das Hauptprogramm startet den Server- und den Client-Thread. Diese starten ihrerseits den Server- und Client-Socket und die zugehörigen Listener-Threads. Danach wartet das Hauptprogramm nur noch auf das Beenden der Threads. Zur Veranschaulichung wird der Fall betrachtet, dass vom Notebook eine Nachricht gesendet wird. Diese wird vom Listener-Thread des Server-Sockets empfangen. Dieser speichert wiederum diese Daten in der Variable „StoC“ und setzt das „Event S“. Dieses Event wird vom Client-Thread in einer Schleife abgefragt. Folglich werden die Daten aus „StoC“ ausgelesen. Entsprechen sie dem Kill-Kommando, so wird das Kill Ereignis gesetzt und Server- und Client-Thread werden mit ihren Listener-Threads beendet. Daraufhin beendet sich auch das Hauptpro-

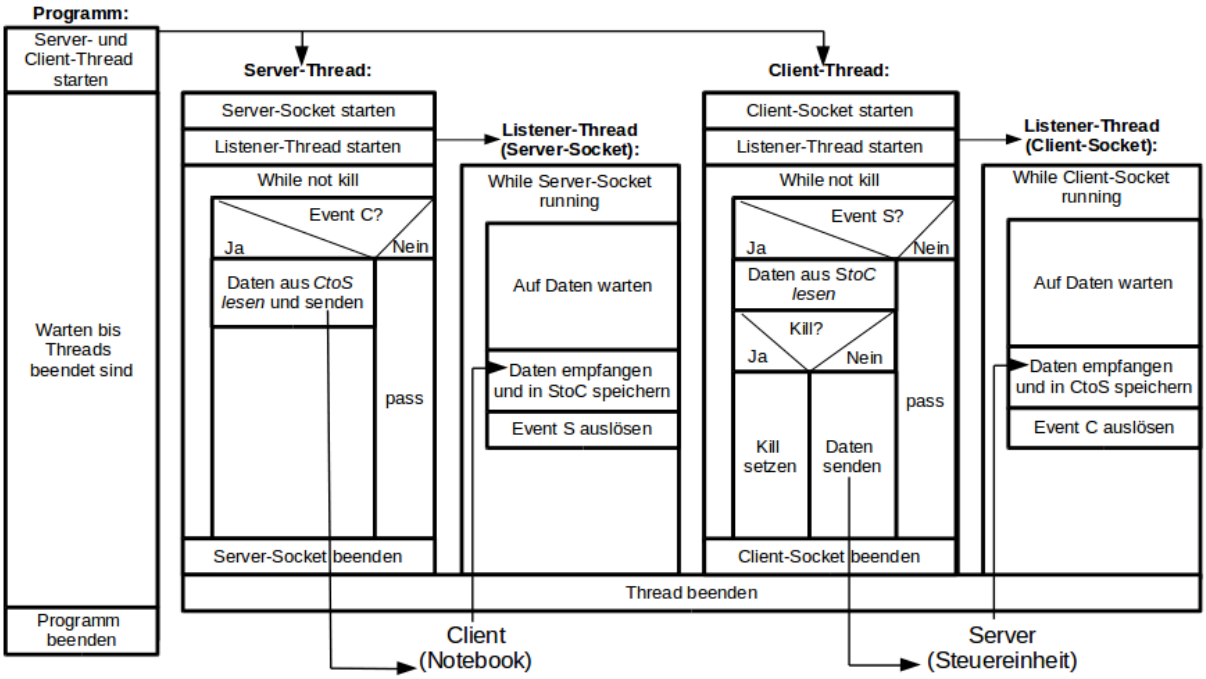


Abbildung 6: Struktogramm (vereinfacht) des Server-PC-Programms

gramm. Diese Möglichkeit wurde eingebaut da im Normalbetrieb nicht direkt auf den Server zugegriffen werden kann. Da dies in diesem Beispiel aber nicht der Fall ist, werden die Daten an die zugehörige Steuereinheit weitergesendet. Diese empfängt die Daten und schickt als Bestätigung eine Antwort zurück. Nach dem selben Prinzip wie eben dargestellt, werden nun die Daten vom Listner-Thread des Client-Sockets empfangen und an den Server-Thread weitergeleitet, der sie schlussendlich an das Notebook weiter sendet.

4.2 Test des Systems

Zum Test des Server-PC-Programms habe ich die in Abb. 5 beschriebene Struktur aufgebaut, allerdings noch ohne die Motorsteuerung. Diese ist für den Test der Funktionsweise auch nicht notwendig. Für den Zugriff auf den Server-PC und die Raspberry Pi's habe ich von meinem Notebook eine Verbindung über einen SSH-Client (Secure Shell Client) aufgebaut. Somit konnte ich über das Terminal die Computer steuern.

Im Test schicke ich über das Testprogramm mit graphischer Oberfläche für die *Python*-Client-Klasse eine Nachricht an den Server. Dieser unterscheidet aufgrund der Kommandosyntax der Nachricht an welche der Komponenten diese weitergeleitet werden soll. Die Komponenten antworten darauf mit einem „OK“. Die Screenshots der Terminals und der graphischen Oberfläche sind in den Abbildungen 7 und 8 dargestellt. Um eine bessere Übersichtlichkeit zu erzielen, wurden in Abb. 7 die einzelnen Terminals zusätzlich gekennzeichnet.

Somit ist gezeigt, dass auch das Serverprogramm und die zugehörige Netzwerkstruktur die benötigte Form der Kommunikation ermöglichen.

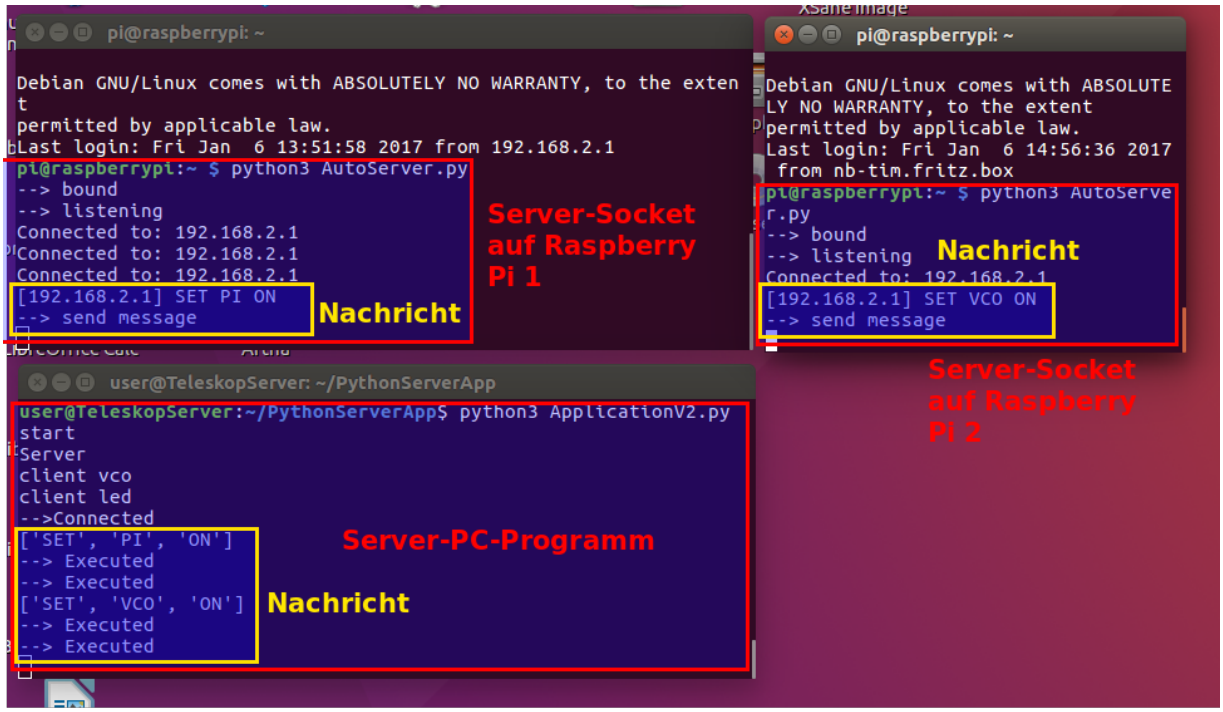


Abbildung 7: Screenshot der Terminals mit SSH-Zugriff auf andere Systeme

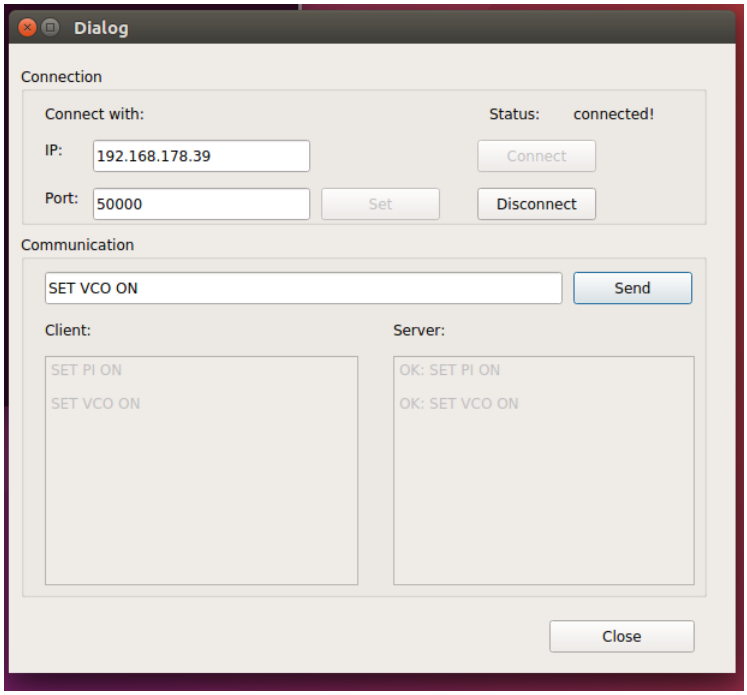


Abbildung 8: Screenshot des Testprogramms für die Client-Klasse in *Python* in einer Linux-Umgebung

5 Fazit

Mit den entwickelten Server- und Client-Klassen habe ich eine Netzwerkkommunikation über TCP/IP ermöglicht. Sie lassen sich sowohl in *C#*- als auch in *Python*-Programme einfach integrieren. Dadurch können wir auch über verschiedene Betriebssysteme hinaus Daten versenden und empfangen.

Dies versetzt unsere AG in die Lage, die verschieden Komponenten in unserem Radioteleskop-System extern anzusteuern und von diesen Informationen als Antwort zurück zu erhalten. Im Hinblick auf diese vorhandene Systemstruktur des Radioteleskops habe ich eine Netzwerkstruktur mit einem Server-PC entwickelt und getestet, die die Kommunikation zwischen externem Steuergerät und anderen Komponenten in unserem System strukturiert. Diese kann künftig ohne großen Aufwand um zusätzliche Funktionen erweitert werden. Damit habe ich eine solide Grundlage für die Steuerung unseres Radioteleskop-Systems gelegt. Diese wird in anderen laufenden Projekten in unserer AG bereits genutzt und auch die künftigen Arbeiten an diesem Projekt können darauf aufbauen.

6 Ausblick

Auf diesem Projekt aufbauend kann nun weiter an der Inbetriebnahme des Radioteleskop-Systems gearbeitet werden. Schwerpunktmäßig muss das Server-PC-Programm erweitert werden. Dabei gilt es in erster Linie die Steuerung von Außerhalb auszubauen, wie das Setzen von Parametern und das Abfragen des monentanen Status mittels Befehle unserer Kommandosyntax. Dies würd vorallem für die in *Abschnitt 2.4* angeprochenen Scricpt-Befehle wichtig, damit auch automatische Messungen vorgenommen werden können. Zusätlich wird auch das Aufzeichnen der Kommunikation implementiert. Weiterhin muss das Systems mit den endgültigen Steuereinheiten getestet werden. Schließlich müssen Funktionen der Steuerungssoftware auf den Server-PC ausgelagert werden, um darauf basierend eine manuelle Steuerung für das Radioteleskop zu aufbauen zu können, die dann direkt beim Teleskop platziert werden kann.

Nach Integration der anderen laufenden Projekte unserer AG werden wir einen Test des gesamten Radioteleskop-Systems durchführen um dann anschließend unsere ersten Messungen der elektromagnetischen Strahlung des interstellaren Wasserstoffs durchführen zu können.

Literatur

[1] Ernesti, Johannes und Peter Kaiser: *Python 3 - Das umfassende Handbuch.* Rheinwerk Verlag GmbH, Bonn, 2016.

[2] Louis, Dirk, Shinja Strassner und Thorsten Kansy: *Microsoft Visual C# 2010 - Das Entwicklerbuch.* O'Reilly Verlag GmbH, Köln, 2010.

[3] Soest, Daniel van und Charly Kühnast: *Ubuntu Server 16.04 LTS - Das umfassende Handbuch*. Rheinwerk Verlag GmbH, Bonn, 2016.

Danksagung

Mein ganz besonderer Dank gilt meinem Betreuer StD Thomas Biedermann, der sich stets Zeit für mein Projekt genommen hat und mir bei der Planung und bei Problemen helfend zur Seite stand. Außerdem bedanke ich mich bei Lucas Jürgens, der mich in das Einrichten von Linux-Systemen eingeführt hat und durch dessen Hilfe mir so manche Stunde an Fehlersuche erspart blieb. Schließlich danke ich Susanne Biedermann, die in unserer AG immer mit Keksen und Getränken für unser leibliches Wohlergehen gesorgt hat.