TIANE

Ein offener Smart-Home Sprachassistent

FERDINAND und KLARA KRÄMER

Bundeswettbewerb Jugend Forscht 2019

Mathematik/Informatik

Inhaltsangabe

Kurzfassung

Ziel unseres Projektes ist die Entwicklung eines universellen Sprachassistenten, der anders als Alexa, Google Home & Co. möglichst vollständig als Open Source verfügbar sein soll. Zudem wird er durchaus einige zusätzliche innovative Funktionen enthalten. Unsere TIANE soll als eine Art universelle Plattform für das sprachgesteuerte Smart-Home im ganzen Haus fungieren. Dafür nutzt sie neben Mikrofonen mit Spracherkennung auch Kameras mit Bilderkennung, um ihre Nutzer im Haus wiederfinden und ihnen so personalisierte Informationen und Funktionen zur Verfügung stellen zu können. TIANE verfügt von Haus aus über zahlreiche Kernfunktionen eines modernen Sprachassistenten. Dank einer sowohl einsteigerfreundlichen als auch sehr tief reichenden Schnittstelle lässt sich das gesamte System ganz einfach modular erweitern. Bei Bedarf können auch Kernkomponenten wie die Spracherkennung ausgetauscht werden.

TIANE ist im Einsatz als Server-Client-System aufgebaut: Ein zentraler Home-Server – zum Beispiel ein ausrangierter Arbeitscomputer – dient als Zentrale für die Verwaltung des Netzwerks. Ebenso laufen über ihn rechenintensive Aufgaben, wie beispielsweise die Bilderkennung. In den mit TIANE ausgestatteten Räumen genügt als Hardware beispielsweise ein Raspberry Pi, an den Mikrofon, Kamera, Lautsprecher und die Smart-Home-Hardware für den jeweiligen Raum angeschlossen werden. Um unseren Sprachassistenten möglichst leicht nachvollziehbar und vor allem erweiterbar zu gestalten, wurde das gesamte System in der einsteigerfreundlichen Programmiersprache Python programmiert.

Unser Projekt umfasst die Entwicklung der für TIANE benötigten Konzepte und vor allem der Programme für Server, Netzwerk, Client, grundlegende Module etc.

Der TIANE-Code ist über GitHub frei zugänglich (siehe https://github.com/FerdiKr/TIANE).
Zur Verfügung gestellt werden darüber hinaus eine umfassende Dokumentation der Kernkomponenten, der Modulschnittstelle und eine Installationsanleitung, um TIANE für interessierte Anwender zu einer echten Alternative zu bestehenden kommerziellen Systemen zu machen.

Einleitung

Spätestens seit dem Film „Iron Man“ ist er wohl für viele ein Wunschtraum: JARVIS, der intelligente Computer des Protagonisten, der nicht nur im gesamten Haushalt immer hilfreich zugegen ist, sondern auch bei der Entwicklung und Steuerung futuristischer Kampfanzüge. Er scheint nicht an eine sichtbare Hardware gebunden zu sein, sondern begleitet seinen Schöpfer in Form einer angenehmen Computerstimme einfach überall hin.

Nun erobern seit einigen Jahren sogenannte Smart-Speaker oder Sprachassistenten, die ihren Nutzern genau einen Teil dieses JARVIS-Feelings versprechen, immer mehr Haushalte. Das sind meist kleine, einzeln stehende Lautsprecher mit Mikrofonarray und Netzwerkverbindung, die auf an sie gerichtete Sprachbefehle des Nutzers (meist eingeleitet von bestimmten Schlüsselwörtern) horchen und so zum Beispiel Musik abspielen, Smart-Home-Elektronik steuern oder Informationen liefern können. Zu den bekanntesten Vertretern dieser Gattung zählen Alexa, Siri und Google Home.

Allerdings mehrt sich mit der Verbreitung dieser Geräte auch die bereits zu Anfang stets deutliche Kritik an ihnen: Von Firmen wie Amazon, Apple und Google entwickelt und hergestellt, zu deren Kerngeschäft nun mal der Handel mit Nutzerdaten gehört, ziehen die freundlichen Assistenten einigen berechtigten Argwohn auf sich. Vollkommen intransparente „Blackboxen“ mit ultrafeinen Mikrofonen, dauernder Netzwerkverbindung und mittlerweile sogar vereinzelt schon Kameras im eigenen Wohnzimmer wirken naturgemäß gerade auf technisch versierte Nutzer nicht unbedingt vertrauenserweckend, zumal nie wirklich klar ist, wann diese Mikrofone überhaupt aufnehmen und wo und zu welchen Zwecken diese Tonaufnahmen dann gespeichert und ausgewertet werden. Dies ist an zahllosen aktuellen Fällen zu sehen, in denen Nutzer von Alexa und Co. beispielsweise die Daten fremder Personen abrufen konnten ([1], [2]).

Zudem können die Assistenten unserer persönlichen Meinung nach nicht einmal ihr ursprünglichstes Versprechen einlösen: Beim Gespräch mit einem solchen Lautsprecher spricht man nun mal genau mit diesem Lautsprecher, der fest in einem Raum steht, und eben nicht mit einem omnipräsenten „Guten Geist des Hauses“ wie JARVIS. Und selbst wenn man jeden Raum der Wohnung mit einem solchen Gerät ausstattet, agieren diese mitnichten wie eine Einheit, sondern teilen sich lediglich die Daten des Nutzers, Unterhaltungen müssen stets von Anfang bis Ende in *einem* Raum, mit *einem* Gerät geführt werden. Und überhaupt, wer will sich denn bitte nur mit der 1.000.000sten „Alexa“ unterhalten? Bevor nicht wenigstens der Name frei wählbar ist, kann einfach kein wirkliches „JARVIS-Feeling“ aufkommen; so ist auch der Name TIANE nicht festgelegt, sondern kann jederzeit nach Belieben geändert werden.

Ziel unseres TIANE Projektes ist es also, einen eigenen Sprachassistenten zu entwerfen, der die JARVIS-Versprechen einlöst und mit den Daten seiner Nutzer vertrauenswürdig umgeht.

Im Einzelnen sieht die Zielsetzung wie folgt aus:

- TIANE soll innerhalb eines Haushalts als einheitliches System agieren und nicht nur als Ansammlung von unabhängigen Geräten. Das beinhaltet insbesondere die Fähigkeit, Unterhaltungen über mehrere Räume hinweg „mitzunehmen“ und andere Personen im ganzen Haus finden und ansprechen zu können.

- Dafür soll TIANE mittels Stimm- und Gesichtserkennung den aktuellen Aufenthaltsort ihrer Nutzer bestimmen können.
- TIANE soll in möglichst vielen Aspekten frei anpassbar sein: Eine umfangreiche und gut dokumentierte Modulschnittstelle soll das einfache Entwickeln neuer Fähigkeiten ermöglichen und auf Wunsch sollen auch Kernkomponenten wie die Spracherkennung und -ausgabe leicht ausgetauscht werden können. Außerdem soll sich auch der Name ohne Mehraufwand ändern lassen,
- Dafür wird der gesamte TIANE-Code open-source auf der Internetplattform GitHub veröffentlicht, um das System leicht anpassbar und vor allem voll vertrauenswürdig zu machen.

Verwendete Hardware- und Softwarekomponenten

Die Hardware

Um das erste Ziel erfüllen zu können und TIANE also im ganzen Haushalt raumübergreifend einheitlich verfügbar zu machen, haben wir uns schon früh entschlossen, auf ein Server-Client-System zu setzen, bei dem zwar jeder Raum einen eigenen, unabhängigen Client bekommt, um jegliche Technik in diesem Raum anzusteuern, aber die Clients stets in Kontakt mit einem kleinen Heimserver stehen, der rechenintensive Aufgaben übernehmen und Aktionen über mehrere Räume hinweg koordinieren kann. Für die Raumclients empfehlen wir aufgrund der einfachen Einrichtung, der brauchbaren Rechenleistung und vor allem der umfassenden Möglichkeiten, mithilfe von WLAN, Bluetooth und der GPIO-Ports beliebige Smart-Home-Elektronik anzusteuern, ganz eindeutig Raspberry Pis; in unseren eigenen Aufbauten verwenden wir das Modell 3B+, mit jedem Modell der 2er-Serie sollte TIANE aber ebenfalls ohne Probleme kompatibel sein.

An die einzelnen TIANE-Raum-Clients wird nun jegliche Hardware angeschlossen, die im jeweiligen Raum für TIANE zur Verfügung stehen bzw. mit TIANE gesteuert werden soll: Das umfasst Mikrofon, Kamera und Lautsprecher für die Sprachsteuerung und Bilderkennung genauso wie jegliche Smart-Home-Elektronik, die TIANE später regeln können soll.

Als Mikrofone nutzen wir billige Konferenz-Mikrofone, die zwar eine miserable Tonqualität liefern, dafür aber, meist in Regalen in der Mitte der langen Seite eines Raumes platziert, recht gut den ganzen Raum abdecken. In einigen Experimenten, vor allem in kleinen oder weniger schallschluckenden Räumen, haben auch schon die integrierten Mikrofone unserer ebenfalls sehr billig gehaltenen Webcams problemlos für die Spracherkennung ausgereicht, letztlich wird man wohl weiterhin für jeden Raum einzeln die passende Kombination von Gerät und Position im Raum ausprobieren müssen. Die Webcams selbst (in unserem Fall zwei Logitech C270 pro Raum, wobei aber ebenfalls auch beliebige andere Modelle funktionieren sollten) sind für die Gesichtserkennung und die damit verbundene Positionsbestimmung von Nutzern innerhalb des Hauses gedacht und werden deshalb so angebracht, dass sie einen möglichst großen Teil des Raumes auf Kopfhöhe gut überblicken. Es hat sich bewährt, sie für optimalen Überblick auch in Räumen mit vielen Ein- und Ausgängen auf Schränken in zwei diagonal gegenüberliegenden Raumecken

anzubringen, für die meisten Räume dürfte aber auch eine einzige, auf den Eingangsbereich gerichtete Kamera völlig ausreichen. Damit TIANE in einem Raum nicht nur zuschauen und –hören, sondern auch mit dem Nutzer sprechen kann, muss noch irgendeine Form von Audio-Ausgabe an den RasPi angeschlossen werden: Das ist natürlich optimalerweise eine im Raum bereits vorhandene Stereo- oder gar Surround-Anlage, ein kleiner, billiger USB-Lautsprecher funktioniert aber genauso gut.

Das Steuern von Smart-Home-Hardware in den einzelnen Räumen gehört zu TIANEs Hauptaufgaben, und der Kreativität für die Ansteuerung der Geräte sind dabei kaum Grenzen gesetzt: Alles, was sich irgendwie an einen Raspberry Pi anschließen lässt (und im Idealfall bereits über eine Python-API verfügt), lässt sich sehr einfach auch mit TIANE steuern, seien es die beliebten (aber teuren!) Philips Hue-WLAN-Lampen oder einfache Steckdosenrelais.

Ein interessantes Thema ist auch der Server, der später in diesem gesamten Raum-Netzwerk den Überblick bewahren soll. Wir verwenden dafür einen älteren, ausrangierten Büro-PC (Prozessor aus der Intel Core2-Generation, also ca. 10 Jahre alt), den wir mit einem aktuellen Ubuntu-Betriebssystem (aktuell ist Linux für alle TIANE-Komponenten Pflicht) sowie für Bilderkennungszwecke noch mit einer CUDA-fähigen Grafikkarte ausgestattet haben. Dieser Computer ist zwar etwas laut, steht aber in einem Raum, wo er niemanden stört, und verrichtet tadellos seinen Dienst. Eine bessere Alternative zu einer alten, lauten Maschine wäre sicher ein moderner, lüfterloser Intel NUC-Kleincomputer, und in einigen Experimenten sind wir sogar schon ganz ohne eigenständigen Server ausgekommen: Solange man auf die aufwändige Gesichtserkennung (und damit aber leider auch auf einen der großen Vorteile des TIANE-Systems) verzichten kann, lässt sich das Server-Programm problemlos parallel zum Client-Programm auf einem der Raum-RasPis ausführen, da die reine Verwaltung des TIANE-Netzwerks nicht allzu viel Leistung benötigt. Damit sind sogar theoretisch TIANE-standalone-Aufbauten mit nur einem einzigen Gerät möglich, was aber natürlich ebenfalls einen enormen Verlust von Funktionen zum Nachteil hat.

Sobald man sich für den Standard-TIANE-Aufbau mit mehr als einem Endgerät entscheidet, sollte man sich dringend auch die Frage stellen, wie man die Geräte überhaupt mit einem gemeinsamen Netzwerk verbindet. Wir persönlich setzen dafür stark auf flache LAN-Kabel hinter Fußleisten und einen kleinen Switch bei jedem Raum-RasPi, was aber auch mit entsprechend viel Aufwand verbunden ist. Einfacher ist natürlich die Verbindung via WLAN, besonders da die Raspberry Pis schon seit Jahren über eingebaute Drahtlosmodule verfügen. Grundsätzlich reicht die Übertragungsrate moderner Router für den Betrieb von TIANE auch völlig aus, gerade bei umfangreicheren Aufbauten mit vielen Kameras in den Räumen empfehlen wir jedoch, für das TIANE-Netzwerk einen eigenen Router einzuplanen: TIANE tauscht von Natur aus nicht viele Daten mit dem Internet aus, das lokale Netzwerk wird dafür aber umso mehr belastet, da sämtliche Räume permanent ihre Kamerabilder an den Server schicken, was unter Umständen dazu führen kann, dass man mit einem Smartphone oder Tablet im selben Netzwerk nicht mehr allzu viel Bandbreite abbekommt.

Die Software

Das Hauptaugenmerk liegt bei TIANE eindeutig auf der Software. Als Programmiersprache kommt dabei aufgrund der besseren Verständlichkeit und der umfangreichen Verfügbarkeit von Libraries zur Ansteuerung verschiedenster (Smart-Home-)Hardware Python zum Einsatz. Interessant sind aber natürlich besonders die von TIANEs Kernkomponenten verwendeten Libraries, deren Auswahl teilweise durchaus Kopfzerbrechen bereitet hat.

Am schwersten haben wir uns mit der Spracherkennung getan, die für einen Sprachassistenten leider nun mal unabkömmlich ist. Als gängigste Open-source-Variante auf diesem Gebiet ist eindeutig „CMUSphinx“ bzw. „PocketSphinx“ zu nennen, aber um es kurz zu machen: Wir haben es getestet und für TIANE als absolut ungeeignet empfunden. Nach der unnötig aufwendigen Suche und Installation von deutschen Sprachpaketen konnte Sphinx selbst aus eigens erstellten Audiodateien mit perfekter Sprachqualität kaum einen Satz korrekt erkennen – der Alltagstest fiel entsprechend vernichtend aus. Nach langer, verzweifelter Suche, die leider meist rein englischsprachige Programme zutage förderte, mussten wir uns schließlich schweren Herzens für „Google Speech Recognition“ entscheiden. Um das Konzept des offenen, datenschutzbewussten Sprachassistenten nicht ganz zunichte zu machen, nutzen wir Googles Online-Dienst immerhin mit einigen Einschränkungen:

- Die Spracherkennung wird nur aktiviert, wenn zuvor die Hotworderkennung angeschlagen hat (und dies mit einem gut hörbaren akustischen Signal anzeigt), und diese läuft vollständig offline. Es werden also definitiv keine zufälligen Gespräche aufgezeichnet und an Google gesendet, das kann bei uns jeder selbst im Quellcode nachprüfen.
- Wir nutzen Google Speech Recognition, nicht Google *Cloud* Speech Recognition. Google hat in letzter Zeit mit der „Cloud Console“ einen neuen Zugang zu seinen APIs für Entwickler geschaffen – mit dem großen Nachteil, dass dort Registrierungszwang herrscht, wodurch sich sämtliche Daten eines Nutzers stets verknüpfen lassen, und manche Dienste – wie unter anderem auch Cloud Speech – sogar kostenpflichtig sind. Die ältere API, Google Speech Recognition, existiert aber offenbar immer noch irgendwo: Zwar werden keine neuen Keys mehr vergeben (früher konnte sich jeder einen kostenlosen API-Key, der 50 freie Aufrufe der Speech Recognition pro Tag beinhaltete, abholen), aber der Zugang mit alten Keys (falls Sie noch einen besitzen sollten) und vor allem mit öffentlichen Keys funktioniert immer noch. Einen solchen öffentlichen Key ("`AIzaSyBOti4mM-6x9WDnZIjIeyEU21OpBXqWBgw`") verwendet auch TIANE standardmäßig. Er wurde von Google zu Testzwecken ausgestellt, hat quasi keine Beschränkungen auf die Zahl der Aufrufe und den großen Vorteil, dass er eben öffentlich ist: Dieser Key wird seit Jahren von tausenden von Menschen weltweit verwendet, Rückschlüsse auf einzelne Nutzer sind somit kaum möglich, und für absolute Anonymität können wir zum Beispiel die Nutzung eines VPNs empfehlen. Ein großer Nachteil ist, dass es jederzeit passieren kann, dass Google die überholte Speech Recognition-API oder zumindest diesen Key einstellt.
- TIANE ist ein offenes und modifizierbares System. Es steht jedem, der die Installation und die schlechte Erkennungsrate nicht scheut, stets frei, stattdessen die Sphinx-Spracherkennung zu nutzen, oder vielleicht sogar eine ganz andere, uns noch unbekannte Engine. Außerdem haben auch wir selbst schon eine potentielle zukünftige Alternative im Visier (siehe Abschnitt „Ergebnisse und Diskussion“).

Für eine einfachere Einbindung der Spracherkennung nutzen wir die Open-source Python-Library „SpeechRecognition“ [3].

Wie bereits erwähnt nutzen wir eine der Spracherkennung vorgeschaltete Offline-Hotworderkennung, die permanent auf ein bestimmtes, frei wählbares Aktivierungswort (oder eine Wortfolge) lauscht. Für Hobbyprojekte hat sich auf diesem Feld „snowboy“ [4] als eine

Art Quasi-Standard etabliert: Es kann nach mehreren Hotwords auf einmal suchen, arbeitet dabei auch auf dem Raspberry Pi sehr ressourcenschonend, das Training geschieht einfach online und die Erkennungsrate ist absolut brauchbar. Für unsere Zwecke mussten wir allerdings die Datei „snowboydecoder.py“ leicht anpassen, dazu später mehr.

Ebenso wichtig wie die Spracherkennung ist auch die Sprachausgabe für die Kommunikation mit dem Nutzer. Auf diesem Gebiet gibt es zum Glück eine recht hochwertige Open-source-Lösung: „SVOX PicoTTS“, eine angenehme, weibliche Stimme mit halbwegs natürlicher Aussprache und Betonung, die frühere Standard-Sprachausgabe auf Android-Smartphones. Da die deutsche Stimme in unseren Ohren etwas zu tief klang, haben wir den Klang durch Erhöhung der Abtastrate bei der Wiedergabe der generierten Audiodateien angepasst, um unsere charakteristische TIANE-Stimme zu erzeugen; ein Parameter, mit dem es sich eindeutig zu „spielen“ lohnt. PicoTTS funktioniert vollständig offline und verbraucht auch auf einem Raspberry Pi kaum Leistung. Als Wrapper nutzen wir die Library „py-picotts“ [5]. Wer lieber eine andere Stimme möchte, kann diese bei TIANE natürlich auch sehr leicht anpassen; mögliche Alternativen wären zum Beispiel das etwas in die Jahre gekommene „espeak“ (monotone, nuschelnde Roboterstimme, dafür aber auch als männliche Variante verfügbar) und natürlich „Google Cloud Text-To-Speech“ (überragende Qualität und zwei weibliche sowie zwei männliche Stimmen zur Auswahl, dafür nur als registrierungs- und kostenpflichtiger Onlinedienst verfügbar).

Für die allgemeine Verarbeitung von Audiosignalen, den Umgang mit Audiodateien und für die Ansteuerung von Mikrofon und Lautsprecher kommt „pyaudio“ [6] zum Einsatz. Die Kamera- und Gesichtserkennungsmodule nutzen außerdem noch die Libraries „opencv“ [7], „imutils“ [8], „dlib“ [9] und „face_recognition“ [10]. Eine Übersicht über sämtliche verwendete Fremdsoftware findet sich auch in den Quellenangaben.

Integration zu einem Gesamtsystem

Netzwerkverbindung

Nachdem Hardware und Libraries ausgewählt und getestet waren, galt es zunächst, die Hardware in der gewünschten Weise zu verbinden. Zur möglichst einfachen Kommunikation zwischen den Clients und dem Server entschieden wir uns nach kurzer Sichtung der vorhandenen Alternativen für die Programmierung eines eigenen kleinen Netzwerkmoduls. Dieses ist auf Server- und Clientseite in der Datei „TNetwork.py“ zu finden, baut auf Pythons low-level socket-API auf und stellt Funktionen zum Herstellen einer Verbindung sowie zum einfachen, aber gut verschlüsseltem Senden und Empfangen von beliebigen Daten in Form von Dictionaries bereit. Intern besitzt dieses Modul nach dem Start auf Server und Client jeweils einen eigenen Thread, in dem es immer wieder auf den Empfang von pickle-codierten Daten wartet, diese zu einem Dictionary decodiert, damit ein lokales Buffer-Dictionary ergänzt und danach die Daten aus einem separaten Sende-Buffer-Dictionary auf gleiche Weise zurückschickt. Liegen einmal keine Daten vor, wird ein leeres Dictionary gesendet, um das verbundene Gerät wenigstens wissen zu lassen, dass die Verbindung noch existiert. Auf diese Weise müssen zum Senden und Empfangen von beliebigen Daten nur die Sende- bzw. Empfangs-Buffer-Dictionaries gefüllt bzw. gelesen werden, was eine

sehr komfortable Netzwerkkommunikation ermöglicht. Der Server unterhält zu jedem Client eine eigene Netzwerkverbindung, die Clients sind jeweils nur mit dem Server verbunden.

Start des TIANE-Servers

Nach dem Start des TIANE-Programms auf dem Server („TIANE_server.py“) initialisiert dieses zunächst alle benötigten Variablen und Objekte (auf die wir später nach und nach näher eingehen), unter anderem eine Klasse namens `TIANE`, die im späteren Verlauf den Kern des Systems darstellt, und eine Klasse namens `Modules`. Diese letztere Klasse ist, wie der Name schon andeutet, für das Laden der im TIANE-System sehr wichtigen Module verantwortlich.

Anmerkung zum folgenden Abschnitt: Für Details zu den Arten und Funktionsweisen von Modulen und somit für besseres Verständnis dieses Abschnittes empfehlen wir unseren „Guide zur Modulentwicklung“ auf GitHub (https://github.com/FerdiKr/TIANE/blob/master/TIANE - Guide zur Modulentwicklung.pdf)

Bei der Initialisierung der Klasse wird zuerst die Funktion `Modules.get_modules()` aufgerufen, um mithilfe des Python-eigenen Tools „pkgutil“ die common_modules in ihrem Ordner (meist „modules/common“) zu finden und zu laden, d.h. in dem Fall an eine Liste namens `Modules.common_modules` anzuhängen. Python legt bei diesem Vorgehen tatsächlich einfach den gesamten Inhalt der .py-Dateien der Module als Objekte in dieser Liste ab, der enthaltene Code ist direkt ausführbar. Beim Laden werden die Module bereits automatisch auf Syntaxfehler überprüft und gegebenenfalls übersprungen. Zum Schluss werden die Module in der Liste noch nach ihrem `PRIORITY`-Parameter sortiert (falls kein solcher gesetzt ist, wird stattdessen 0 angenommen). Auf gleiche Weise wird verfahren, um die Liste `Modules.common_continuous_modules` zu füllen; die Funktion `Modules.get_user_modules()` lädt außerdem noch die Nutzermodule in die Dictionaries `Modules.user_modules` und `modules.user_continuous_modules`, wobei jeweils unter dem Key des Nutzernamens die jeweilige Liste mit den Modulobjekten zu finden ist.

Nachdem alle wichtigen Klassen initialisiert und die Module geladen sind, setzt der Hauptthread des Serverprogramms einen socket auf und wartet auf Verbindungen. Sobald ein Verbindungsversuch erfolgt, initialisiert der Server sofort ein Objekt der Klasse `Network_Device` für diesen Client, welches fortan im Grunde genommen die Repräsentation dieses Gerätes im Server darstellt. Dieses `Network_Device`-Objekt wiederum startet unmittelbar darauf einen eigenen Thread, in dem die Funktion `Network_Device.start_connection()` ausgeführt wird. Auf diese Weise kann der Hauptthread des Servers sofort auf die nächste Verbindung warten.

Der neue Thread dagegen wartet zunächst auf eine bestimmte Begrüßungssequenz von seinem gerade erlangten und noch unbekannten Gesprächspartner: Kommt diese nicht innerhalb einer bestimmten Zeit oder fällt sie anders aus als erwartet, so stammt dieser Verbindungsversuch nicht von einem TIANE-kompatiblen Gerät; die Verbindung wird beendet und das `Network_Device`-Objekt wieder entfernt. Ist die Begrüßung dagegen korrekt, antwortet der Server dem Gerät auf ähnliche Weise, damit auch dieses weiß, dass es mit dem richtigen Partner spricht, und startet die oben beschriebene, verschlüsselte „TNetwork“-Kommunikation. Aus der Begrüßung kann der Server außerdem entnehmen, ob es sich um einen TIANE-Raum-Client oder ein anderes TNetwork-Gerät handelt.

Die Möglichkeit, beliebige Programme mittels unseres einfachen, aber mächtigen TNetwork-Protokolls mit dem TIANE-Server kommunizieren zu lassen, ohne dass diese gleich alle

Funktionen eines TIANE-Raum-Clients unterstützen müssen, wurde geschaffen, um TIANE *wirklich* universell zu gestalten: So ist es möglich, beliebige andere DIY-Smart-Home-Hardware, z.B. einen Smart Mirror oder ähnliches, an TIANE anzuschließen und so nach Programmierung eines entsprechenden, meist recht einfachen Moduls für den TIANE-Server vom ganzen Haus aus per Sprachbefehl zu steuern. Dafür müssen solche Geräte dem Server unter den TNetwork-Keys ‚`DEVICE_TYPE`' und ‚`DEVICE_NAME`' nur einen beliebigen Typ und einen eindeutigen Namen nennen; mit diesem Namen als Key wird das entsprechende Network_Device-Objekt, das die TNetwork-Verbindung unterhält, dann im Dictionary Other_devices gespeichert, welches von jedem Modul auf dem Server aus zugänglich ist.

Handelt es sich dagegen laut Begrüßung um einen TIANE-Raum-Client, sind deutlich komplexere Funktionen gefragt: Diese stellt die Klasse `Room_Dock` zur Verfügung, in die das `Network_Device`-Objekt an dieser Stelle umgewandelt wird (der Name rührt daher, dass die Funktionen dieser Klasse quasi den Ort darstellen, an dem der Raum „andockt"). Dann werden zunächst die wichtigen Daten ausgetauscht: Der Raum teilt dem Server seinen Raumnamen mit (z.B. ‚Wohnzimmer'), woraufhin dieser Name an eine Liste aller Raumnamen (`room_list`) angehängt wird und außerdem als neuer Key im Dictionary `Rooms` dient; unter diesem Key wird nun eine Referenz auf die `Room_Dock`-Instanz für diesen Raum gespeichert. So kann der Server später alle Räume sehr einfach verwalten, sämtliche Informationen über einen Raum sowie die TNetwork-Instanz zur Kommunikation mit diesem lassen sich unter Angabe des Namens des gewünschten Raumes aus dem `Rooms`-Dictionary ablesen. Dann erst antwortet der Server mit seinen wichtigen Daten: Er schickt dem Raum seinen Namen (der aber weitgehend irrelevant ist) sowie alle Daten über Nutzer und andere Räume im Netzwerk, die im `local_storage`-Dictionary des Servers gespeichert sind.

Das `local_storage`-Dictionary ist, wie der Name schon andeutet, eine Art gemeinsamer, lokaler Speicherbereich für sämtliche Komponenten und Module von TIANE auf einem Gerät. Module können hier beliebige Daten speichern und abrufen, für TIANE-Kernkomponenten sind aber besonders die Bereiche hinter den Keys ‚`users`' und ‚`rooms`' interessant: Diese sind nämlich jeweils wieder Dictionaries, die unter dem Namen eines Nutzers bzw. Raumes sämtliche Informationen über diesen enthalten. So können diese Daten stets zentral verwaltet und sogar verteilt werden: Bei jeder Änderung in den Bereichen ‚`users`' oder ‚`rooms`' oder auch unter beliebigen anderen, von Modulerstellern festlegbaren Keys, werden die neuen Daten sofort an sämtliche Räume verteilt. Eine genauere Beschreibung des `local_storage` und seiner Funktionen findet sich in unserem „Guide zur Modulentwicklung".

Nachdem diese ersten Informationen ausgetauscht sind, geht der `Room_Dock`-Thread in der Funktion `Room_Dock.handle_online_requests()` in eine Schleife, in der er fortan auf Anfragen des Raumes, auf die wir später näher eingehen, wartet und diese bearbeitet. Falls der Server einen Verbindungsabbruch registriert (feststellbar dadurch, dass TNetwork noch nicht einmal leere Dictionaries empfängt, sondern ein Timeout anzeigt), wird das `Room_Dock`-Objekt aus dem `Rooms`-Dictionary sowie der Raumname aus der `room_list` und die entsprechende Sektion aus dem `local_storage` entfernt. Auf diese Weise funktionieren Räume im TIANE-System quasi „plug-and-play": Es kann sich jederzeit ein völlig neuer Raum mit dem Server verbinden, und sollte einmal ein Raum-Raspberry

abstürzen, läuft der restliche Betrieb ohne Probleme weiter, und der Raum kann sich nach Behebung des Fehlers sofort wieder anschließen.

Erst wenn mindestens ein Raum verbunden ist, nehmen auch die anderen Komponenten des Servers den Betrieb auf (ein früherer Start ist nicht nötig, weil ohne Räume auch keine Möglichkeit zur Ein-/Ausgabe besteht), allen voran die Module: Die continuous_modules wurden vorhin zwar bereits geladen, aber eben noch nicht gestartet. Dies geschieht durch eine Aufruf der Funktion `Modules.start_continuous()`, sobald der erste Raum verbunden ist. Diese Funktion startet gleich mehrere neue Threads: Einen für die common_continuous_modules (sofern vorhanden) und jeweils einen weiteren pro Nutzer, der user_continuous_modules verwendet. In jedem dieser Threads läuft die Funktion `Modules.run_continuous()` für die jeweilige Modulliste. Diese Funktion legt zunächst für jedes Modul in der Liste ein Element im Dictionary contiunuous_modules in der vorhin instanziierten Hauptklasse `TIANE` an: Unter dem Namen des Moduls (Dateiname der .py-Datei, als String) als Key wird dort eine Instanz der Klasse `Modulewrapper_continuous` gespeichert, mit dem vom Modul per Hyperparameter festgelegten Zeitintervall, in dem es aufgerufen werden soll (falls nicht festgelegt, wird 0 angenommen), und dem Namen des Nutzers, dem das Modul gehört (falls es sich um ein user_continuous_module handelt) als Parameter.

Diese `Modulewrapper_continuous`-Klasse (und eine ähnliche namens „Modulewrapper“ für nicht-continuous_modules) bildet den Kern der gesamten TIANE-Modulschnittstelle, wie sie in unserem „Guide zur Modulentwicklung“ beschrieben wird: Sämtliche dort erwähnten Attribute und Funktionen liegen nämlich in dieser Klasse, die auch das `tiane`-Objekt darstellt, das jedes Modul beim Aufruf übergeben bekommt (auch wenn die meisten Attribute der Modulewrapper nur Weiterleitungen auf die entsprechenden Daten in der Hauptklasse TIANE enthalten). Auf diese Weise erhalten Module nur Zugriff auf Attribute und Funktionen, die sie auch betreffen: Im `Modulewrapper_continuous` fehlen zum Beispiel die Funktionen `say()` und `listen()`, da continuous_modules nicht selbst mit dem Nutzer kommunizieren sollen, weil sie sonst den Betrieb aufhalten würden. Außerdem erleichtern die Modulewrapper die Programmierung von Modulen ungemein: Durch das Sammeln aller relevanten Objekte an einem Ort und, wie wir später noch sehen werden, durch das automatische Ergänzen von Funktionsparametern.

Nachdem für jedes Modul ein Wrapper instanziiert wurde, werden der Reihe nach die `start()`-Funktionen der einzelnen Module aufgerufen (sofern sie eine solche besitzen und mit einer Referenz auf den entsprechenden Modulewrapper und auf den `local_storage` als Parameter), in denen diese ihrerseits Variablen oder Objekte initialisieren und im `local_storage` ablegen können. Wenn alle Module gestartet sind, geht der Thread in eine Endlosschleife, in der er immer wieder die Liste der Module durchgeht, zu jedem Modul den passenden Wrapper findet, das vom Modul gewünschte Zeitintervall mit der Differenz aus der aktuellen Zeit und einer im Wrapper gespeicherten letzten Aufrufzeit (`last_call`) vergleicht und gegebenenfalls die `run()`-Funktion des Moduls (mit dem Wrapper und `local_storage` als Parameter) ausführt und die letzte Aufrufzeit überschreibt. Wenn ein Modul bei Start oder Ausführung eine Exception auslöst, wird diese im Übrigen automatisch aufgefangen, ein entsprechendes traceback ausgegeben und das Modul danach schlicht und einfach aus der Liste entfernt und nicht mehr ausgeführt, der restliche Betrieb wird nicht beeinträchtigt.

Start eines TIANE-Raum-Clients

Auch das TIANE-Raum-Client-Programm („TIANE-room.py“) initialisiert nach seinem Start zunächst diverse Variablen und Objekte, unter anderem einen (zu Beginn hier leeren) `local_storage`, eine `Modules`- und eine `TIANE`-Klasse. Die beiden letzteren verhalten sich aber ein wenig anders als auf dem Server: Die `Modules`-Klasse lädt nur eine modules- und eine continuous_modules-Liste, da sich hier nicht um user_modules gekümmert werden muss, dementsprechend wird auch nur ein continuous_modules-Thread gestartet. Dafür ist im Raum-Programm die `TIANE`-Klasse ein aktives Element: Sobald der Raum (nach dem vorhin gezeigten Schema) eine Verbindung zum Server hergestellt hat, wird durch Aufruf der Funktion `TIANE.start()` ein neuer Thread eröffnet, der die Funktion `TIANE.handle_online_requests()` ausführt. Diese wiederum ist das Gegenstück zur `handle_online_requests()`-Funktion in den `Room_Dock`-Objekten des Servers; im Raum liegt sie in der Hauptklasse, weil hier schließlich nur eine Verbindung bearbeitet werden muss (nämlich die zum Server). Wenn diese Verbindung abreißt, wird der Raum sich übrigens auch im Gegensatz zum Server sofort mit einem Fehler beenden: Das ist sinnvoll, da ein Raumclient ohne den Server nicht lange arbeiten kann, und es auch wenig ergiebig erscheint, bei einem Netzwerk- oder Serverfehler permanent weiter „ins Leere zu funken“ und auf eine Verbindung zu hoffen.

Außerdem initialisiert der Raum-Client aber noch zwei weitere Objekte, die auf dem Server fehlen: `Audio_Input` und `Audio_Output`. Diese Klassen werden beide aus der Datei „TIANE_Audio.py“ importiert und sind, wie der Name schon verrät, für die Audio-Ein- bzw. Ausgabe verantwortlich, also für die Verwaltung von Mikrofon und Lautsprecher in diesem Raum inklusive der dazugehörigen Spracherkennung und -Synthese. Dafür importieren die Klassen wiederum jeweils eine kleine Klasse aus der Datei „stt.py“ bzw. „tts.py“. Diese Klassen enthalten die eigentlichen Aufrufe der Spracherkennungs- oder Sprachausgabeengine, um deren Austausch durch versierte Nutzer bei Bedarf so einfach wie möglich zu gestalten.

Sobald eine Verbindung zum Server besteht, werden die Audio-Klassen durch Aufruf der Funktionen `Audio_Output.start()` bzw. `Audio_Input.start_hotword_detection` gestartet. `Audio_Output.start()` startet einen neuen Thread für die Funktion `Audio_Output.putout()`. Diese Funktion ist für die Verwaltung des „pyaudio-output-streams“ verantwortlich, also die Ausgabe eines Stroms von Audiodaten über den Lautsprecher. Damit diese Audiodaten nicht in Echtzeit bereitgestellt werden müssen, was zu Rucklern in der Wiedergabe führen könnte, werden sie in einem Buffer gespeichert und dem Stream in kleinen Stücken („Chunks“) übergeben. So kann zum Beispiel ein Musikwiedergabe-Modul direkt ein ganzes Lied in diesen Buffer schreiben und muss sich nicht weiter darum kümmern. Was aber, wenn während der Wiedergabe des Liedes eine wichtige Benachrichtigung eines anderen Moduls ansteht (per Sprachausgabe)? Und was, wenn diese Audiodaten ein ganz anderes Format haben als die des Liedes? Aus diesen Gründen nutzt die `Audio_Output`-Klasse ein System aus zwei Buffern und der Funktion `putout()`, die den Stream verwaltet. `putout()` beobachtet stets beide Buffer: Den playback_audio_buffer mit niedriger Priorität, z.B. für die Musikwiedergabe, und den notification_audio_buffer mit hoher Priorität für Sprachausgaben und ähnliche Töne, die möglichst sofort abgespielt werden sollen. Solange nur in maximal einem dieser Buffer Daten vorliegen, werden diese einfach Chunk für Chunk an den Stream durchgereicht, falls aber beide gefüllt sind, kann die Wiedergabe aus dem playback_audio_buffer jederzeit zugunsten des notification_audio_buffers unterbrochen und nach dessen Leerung lückenlos fortgesetzt

werden. Da hierfür kurzzeitig der pyaudio-Stream terminiert und neu aufgebaut wird, können sogar Formatunterschiede berücksichtigt werden: Dazu muss beim Befüllen eines der Buffer lediglich ein entsprechendes Format-Dictionary (playback_audio_format bzw. notification_audio_format) angepasst werden. Konkret werden die Keys ‚`format`' (Allgemeine Codierung der Audiodaten, siehe pyaudio-Dokumentation für Details), ‚`rate`' (Abtast- bzw. Samplerate in Hz), ‚`channels`' (Zahl der Audiokanäle) und ‚`chunk`' (Größe der Chunks) benötigt.

Befüllt werden die Buffer am besten mit den dafür bereitgestellten Wiedergabefunktionen: `Audio_Output.play(audiodaten)` akzeptiert eine Liste von Audio-Chunks als Argument und spielt diese als playback_audio ab, `Audio_Output.say(text)` übergibt den gegebenen `text` an die Funktion `say()` der `tts`-Klasse (austauschbar!), erwartet von dieser ein Format-Dictionary und eine Liste von wav-Audio-Chunks und spielt sie ab, spricht also den Text mit der Sprachausgabe aus.

`Audio_Input.start_hotword_detection()` ist für den Start der snowboy-Hotworterkennung zuständig, die TIANE nutzt, um auf ihr Schlüsselwort (z.B. „Hey Tiane“) zur Aufnahme von Sprachbefehlen zu lauschen. Snowboy nutzt dafür kleine machine-learning-Modelle, die Nutzer auf der snowboy-Homepage durch dreimaliges Vorsprechen auf ihr gewünschtes Aktivierungswort trainieren und dann von dort herunterladen können. Dementsprechend besteht die erste Aufgabe von `Audio_Input.start_hotword_detection()` darin, diese Hotword-Modelle (im Ordner „hotword_models“ im TIANE-Verzeichnis des Raumes) zu finden und anschließend nach Nutzern zu sortieren: Die Snowboy-Trainings-Website bietet universelle Modelle, die sofort auf jeden beliebigen Nutzer reagieren können, nämlich nur bei Schlüsselwörtern an, für die genügend Trainingsdaten vorliegen, die also schon von mehreren tausend Nutzern trainiert und heruntergeladen wurden. Das trifft auf beliebte Aktivierungswörter wie „Computer“ zu, nicht aber auf (noch) seltene wie „Hey Tiane“. Für solche „jungen“ Wörter können nur „persönliche Modelle“ heruntergeladen werden, die zwar gut auf die Stimme des Nutzers reagieren, der sie trainiert hat, aber quasi gar nicht auf andere. TIANE zieht aus ebendieser Einschränkung jedoch einen extremen Vorteil: Wenn nämlich für jeden Nutzer ein eigenes Hotword-Modell trainiert werden muss, lässt sich anhand der Tatsache, welches dieser Modelle „anspringt“, leicht und bei gutem Training erstaunlich zuverlässig ermitteln, welcher Nutzer denn gerade zu TIANE spricht! Dafür erwartet TIANE, dass die Hotword-Modelle jeweils den Namen des Nutzers, der sie trainiert hat, im Dateinamen tragen; wird ein Hotword-Modell ohne Nutzername verwendet (z.B. ein universelles Modell), kann TIANE bei dessen Aktivierung selbstverständlich nicht auf diesem Wege feststellen, wer ihr Gesprächspartner ist. `Audio_Input.start_hotword_detection()` übergibt nun snowboy eine Liste aller Hotword-Modelle und eine Liste mit einer callback_function für jedes Modell, die von Snowboy bei Aktivierung automatisch aufgerufen wird und den Namen des Nutzers, zu dem das gerade aktivierte Modell gehört, unter dem Key ‚`TIANE_Hotword_detected`' in den `local_storage` des Raumes schreibt. Die Snowboy-Hotworderkennung wird dann in einem eigenen Thread in der Funktion `Audio_Input.run_hotword_detection()` permanent im Hintergrund ausgeführt und wartet auf ihre Schlüsselwörter.

Die Klasse `Audio_Input` hat aber neben dem reinen Warten auf Aktivierungswörter auch noch eine andere wichtige Aufgabe: Bei Aktivierung eines dieser Aktivierungswörter muss die nachfolgende Sequenz aufgezeichnet und von einer Spracherkennung ausgewertet werden, um den eigentlichen Befehl zu erhalten. Snowboy bringt auch hierfür eine praktische

Funktion gleich mit: Beim Start akzeptiert die Engine als zusätzliches Argument eine audio_recorder_callback-Funktion, die automatisch aufgerufen wird, sobald snowboy ein Hotword und das Ende des darauffolgenden Nutzerbefehls erkannt hat, und die dazwischen aufgezeichneten Audiodaten als Parameter übergeben bekommt. Diese Callback-Funktion, `Audio_Input.audioRecorderCallback()`, übergibt die Audiodaten sogleich der Funktion `recognize()` der `stt`-Klasse und erwartet von dieser das Transkript als String zurück, um es unter dem Key ‚`TIANE_recognized_text`' in den `local_storage` zu schreiben.

Allerdings soll TIANE ja nicht nur nach einem Aktivierungswort zuhören können, sondern auch mitten im Gespräch, wenn sie dem Nutzer zum Beispiel gerade eine Rückfrage gestellt hat. Das gestaltete sich leider schwieriger als gedacht: Bei allen Vorteilen und der einfachen Bedienung hat snowboy nämlich den großen Nachteil, dass es den „pyaudio-input-stream", also die vom Mikrofon empfangenen Audiodaten, exklusiv belegt und nicht von außen zugänglich macht. Wir mussten daher die Datei „snowboydecoder.py" leicht dahingehend anpassen, dass wir nun von außen das Signal zum Aufzeichnen einer Passage geben können, was wiederum den Vorteil hat, dass sich die Snowboy-Engine automatisch darum kümmert, zu erkennen, wann der Nutzer seine Eingabe denn beendet hat. Und da wir TIANE also ohnehin eine modifizierte snowboy-Version beilegen mussten, konnten wir auch noch gleich eine kleine Funktion integrieren, die die Hotworderkennung während der Sprachausgabe deaktiviert, um Falscherkennungen zu vermeiden.

Die Funktion, die für das Zuhören ohne vorheriges Schlüsselwort verantwortlich ist, heißt `Audio_Input.listen()`. Sie sendet das Signal zum Aufzeichnen an snowboy und wartet dann, bis der fertige, von `audioRecorderCallback()` übersetzte Text im `local_storage` vorliegt, um diesen zu returnen.

Der Weg eines Sprachkommandos

Nachdem die Serververbindung hergestellt und die Hotworderkennung gestartet ist, geht der Hauptthread des TIANE-Raumclient-Programms in eine Schleife, in der er einfach immer wieder über den `local_storage` überprüft, ob in dem Raum ein Aktivierungswort eines Nutzers erkannt wurde. Ist das der Fall, fragt der Raum beim Server nach, um welchen Nutzer es sich am wahrscheinlichsten handelt (dort festgelegt vom continuous_modules „assign_users"). Der Hauptthread wartet noch, bis der Nutzername und das fertig in Text übersetzte Kommando vorliegen, und übergibt dann Text und Nutzername an die Funktion `TIANE.handle_voice_call()`.

Diese Funktion ruft zunächst die Funktion `begin()` der ebenfalls anfangs initialisierten Klasse `Conversation` auf. `Conversation` ist für die Verwaltung der „Konversation" in dem jeweiligen Raum zuständig: Weil es für den Nutzer schnell unübersichtlich werden würde, wenn TIANE mit verschiedensten Anliegen gleichzeitig auf ihn einredet, sorgt diese Klasse dafür, dass in jedem Raum immer nur eine Konversation nach der anderen geführt werden kann. Das heißt z.B., dass ein Sprachkommando erst mit allen Rückfragen und Antworten abgearbeitet werden muss, bevor eine Benachrichtigung zu Wort kommen kann. Deshalb muss vor jedem Start einer Konversation oder jedem Aussprechen eines Textes über die Sprachausgabe in dem Raum die Funktion `Conversation.begin(original_command)` mit dem Kommando, mit dem die jeweilige Konversation begonnen wurde (oder im Fall von Benachrichtigungen mit einem zufällig generierten String), als Argument aufgerufen werden. Die Funktion prüft hiernach, ob die

Konversation, die mit diesem Kommando begonnen wurde, gerade an der Reihe ist, und lässt den aufrufenden Thread ansonsten so lange warten, bis das der Fall ist.

Nachdem `TIANE.handle_voice_call()` von `Conversation.begin()` zurückgekehrt ist, also jetzt die alleinige Berechtigung hat, in diesem Raum eine Konversation zu führen, gilt es, das passende Modul zu finden, das sich für das gegebene Kommando „zuständig fühlt", sprich dem Nutzer in geeigneter Weise antworten oder die gewünschte Aktion ausführen kann.

Die eigentliche Verarbeitung von Kommandos geschieht im TIANE-System ausschließlich durch Module, um TIANEs Fähigkeiten möglichst einfach und universell erweiterbar zu machen: Um TIANE eine neue Funktion zu verpassen, genügt es, diese Funktion in ein speziell gestaltetes Python-Skript (ein Modul) zu packen, das Skript in den passenden Ordner zu legen und die Module neu laden zu lassen (entweder durch einen Neustart des TIANE-Programms oder, einfacher, durch das Kommando „Hey TIANE, lade die Module neu", welches ähnliches bewirkt).

Im TIANE-System kommen für die Verarbeitung eines Kommandos immer zunächst die Module auf dem Server in Frage, damit diese ggf. andere Module „überschreiben" oder die Anfrage gar an einen ganz anderen Raum umleiten können (z.B. wenn das im Wohnzimmer geäußerte Kommando lautet: „Hey TIANE, mach das Licht in der Küche aus"). Deshalb schickt der Raum-Client als erstes mithilfe der Funktion `TIANE.request_query_modules()` via TNetwork-Verbindung eine Anfrage (mit Text und user als Inhalt) an den Server, die dort von `Room_Dock.handle_online_requests()` empfangen wird.

Bei Empfang der Anfrage wird auf dem Server die Funktion `TIANE.route_query_modules()` aufgerufen. Diese Funktion ist bei dieser Anfrage noch relativ „ratlos" und leitet sie daher nur an `Modules.query_threaded()` (ebenfalls auf dem Server) weiter. Dort wiederum wird geprüft, ob es sich um einen Direktaufruf eines Moduls über seinen Namen aus einem anderen Modul heraus handelt, was in dem oben genannten Beispiel allerdings nicht der Fall ist. Da also kein Direktaufruf vorliegt, muss es sich um ein Sprachkommando eines Nutzers handeln, und um weitere Informationen zu erhalten, die möglicherweise dabei helfen können, das richtige Modul zur Behandlung des Kommandos zu finden, wird dieses nun zunächst einmal analysiert. Das geschieht mit der Funktion `analyze()`, angesiedelt in der Klasse `Analyzer` und zu finden in der Datei „analyze.py".

Die Funktion `analyze(text)` dient dazu, aus der oft komplexen Nutzereingabe die relevanten Informationen herauszufiltern. Sie findet heraus, ob der Benutzer einen Ort, einen Raum und/oder eine Zeit nennt und gibt all diese Informationen als Dictionary aus. Sollte über ein Element keine Angabe im Text enthalten sein, bekommt der entsprechende key des dictionarys den value `None`.

Um einen Raum zu erkennen, lässt `analyze()` jedes einzelne Wort der Eingabe mit der `room_list` abgleichen und prüft, ob es darin enthalten ist. Sollte dies der Fall sein, wird der Raum unter dem key ‚`room`' im `analysis`-Dictionary gespeichert.

Um aus einer vom Benutzer genannten Zeit eine weiterverwendbare Zeitangabe zu extrahieren, beinhaltet die Funktion `analyze()` mehrere kleine Funktionen, die aus dem Text die genannten Zeitelemente herausfiltern. Hierbei wird allgemein zwischen „_abs" und „_rel" Funktionen unterschieden. „_rel" steht für **rel**ative Zeitangaben wie beispielsweise „übermorgen" oder „nächsten Monat" und „_abs" beschreibt **abs**olute Angaben wie „am 1.

Juli“ oder „im Januar“. Für komplexere Erwähnungen von Zeiten, wie beispielsweise „in 54 Tagen“ wird außerdem eine Funktion verwendet, die auch monatsübergreifend den richtigen Tag finden kann, wobei alle Eventualitäten – Monate mit 31 Tagen, Monate mit 30 Tagen, Februar und Schaltjahre – berücksichtigt werden.

Diese Analyse des Satzes ist wichtig, weil sie jedem per Sprachkommando gestarteten Modul beim Start zur Verfügung gestellt wird, um Modulentwicklern insbesondere das Einbeziehen von Zeitangaben in eigenen Modulen zu erleichtern. Um nun ein passendes Modul für die Behandlung des Kommandos zu finden, ruft TIANE an dieser Stelle einfach der Reihe nach die `isValid()`-Funktionen aller common_modules sowie der user_modules des entsprechenden Nutzers auf (falls vorhanden), in der Hoffnung, dass eines der Module den Befehl verarbeiten kann. Ist das der Fall, wird das betreffende Modul in einem neuen Thread ausgeführt (in der Funktion `Modules.run_threaded_module()` und mit dem entsprechenden Modulewrapper); die Meldung, dass ein passendes Modul gefunden wurde, wird zurückgereicht bis an die Funktion `TIANE.handle_voice_call()` im Ursprungsraum, die sich daraufhin beendet.

Konnte auf dem Server kein passendes Modul für dieses Kommando gefunden werden, wie zum Beispiel für „Mach das Licht in der Küche aus“, muss die Anfrage an einen passenden Raum weitergeleitet werden. Hier kommt das Analysis-Dictionary ins Spiel: Mit dessen ‚`room`‘-Key lässt sich nämlich der korrekte Zielraum für derartige Kommandos ermitteln… Falls dieser Key ‚`None`‘ enthält, also im Kommando kein konkreter Raum genannt wurde, wird wieder eine Meldung bis an die Funktion `TIANE.handle_voice_call()` im Ursprungsraum zurückgereicht, die diesmal allerdings veranlasst, dass der Raum zunächst einmal mithilfe von `Modules.query_threaded()` seine eigenen Module durchkämmt und, falls er dort ebenfalls nicht fündig wurde, dem Nutzer mit einer Standard-Fehlermeldung (dem berühmten „Das habe ich leider nicht verstanden“) antwortet.

Falls aber ein konkreter Raum ermittelt werden konnte, wird das Programm erst richtig aktiv: `Modules.query_threaded()` auf dem Server ruft wiederum `TIANE.route_query_modules()` auf, diesmal jedoch mit dem Namen des Zielraums als zusätzliches Argument. Mit dieser Angabe sucht die Funktion im `Rooms`-Dictionary nach dem entsprechenden `Room_Dock`-Objekt, um über dessen Funktion `request_query_modules()` eine entsprechende Anfrage an den Raum zu senden, die dort wiederum von `TIANE.handle_online_requests()` empfangen und bearbeitet wird. Dazu wird hier die Funktion `Modules.query_threaded()` aufgerufen, die sämtliche Module dieses Raumes absucht und ein passendes Modul gegebenenfalls in einem eigenen Thread ausführt, eine Erfolgs- oder Misserfolgsmeldung wird wieder bis an den ursprünglich aufrufenden Raum (also an den Raum, in dem das Kommando ausgesprochen wurde) zurückgeleitet.

Mithilfe der Funktion `start_module()` in jedem Modulewrapper kann der Prozess der Modulsuche auch von einem anderen Modul aus angestoßen werden, die Funktion akzeptiert außer einem Text, auf den alle Module getestet werden sollen, auch den Namen und Raum eines Moduls als Parameter, um direkte Aufrufe zu ermöglichen. So lassen sich mithilfe der TIANE-Modulschnittstelle auch hochkomplexe Ablaufketten über das gesamte Haus hinweg realisieren (Details hierzu finden Sie wieder in unserem „Guide zur Modulentwicklung“).

Nach ziemlich ähnlichem Schema werden auch Befehle zur Sprachausgabe bzw. zum Empfang einer Spracheingabe während eines Gespräches durch das TIANE-Netzwerk

geleitet: Sämtliche entsprechenden Befehle, die ein Modul erteilt, werden im Modulewrapper mit zusätzlichen Informationen (z.B. Name des Nutzers, der das Modul gestartet hat, mit dem also auch geredet werden soll) versehen und gehen dann zunächst über den Server. Dieser schaut im `local_storage` nach, wo sich der Nutzer gerade aufhält; eine Information, die dort von speziellen Modulen (z.B. dem mitgelieferten „assign_users“) hinterlegt wird, die ihre Daten wiederum von anderen Modulen (z.B. Gesichtserkennung) oder von den vorhin erwähnten Stimminformationen bei einem Sprachkommando beziehen. Eine Funktion namens `TIANE.route_say()` bzw. `route_listen()` auf dem Server leitet den Befehl an den entsprechenden Raum weiter, wo er von `TIANE.handle_online_requests()` aufgefangen wird. In diesem Fall wird allerdings für jeden einzelnen Befehl direkt ein neuer Thread gestartet (in der Funktion `TIANE.say()` bzw. `listen()`): Sprachein- oder Ausgabebefehle bedürfen schließlich der Freigabe durch `Conversation.begin()`, worauf `TIANE.handle_online_requests()` nicht warten kann. Ist die Freigabe dann erteilt, wird die entsprechende Funktion der Audio-Klasse aufgerufen, zum Schluss wird eine Erfolgsmeldung bzw. bei der Spracheingabe der empfangene Satz bis an das aufrufende Modul zurückgereicht.

Ergebnisse und Diskussion

Somit erfüllt TIANE sämtliche zuvor gesteckten Ziele: Sie dient als vollwertiges, offenes, vernetztes Heimassistenzsystem, das sich von der grundsätzlichen Bedienung her in keiner Weise von den kommerziellen Lösungen unterscheidet. Hinzu kommen nützliche Features wie die Erkennung der Nutzerpräsenz mithilfe von Kameras und Gesichtserkennung und vor allem das lückenlose Zusammenspiel sämtlicher TIANE-Geräte im System: Durch den zentralen Server, der jedes Kommando an den entsprechenden Raum zustellt und jede Sprachausgabe stets aktuell dem Nutzer folgen lässt, ist TIANE deutlich besser vernetzt als jedes vergleichbare System, das wir finden konnten. Geplant ist sogar eine weitere Optimierung an dieser Stelle, die dafür sorgen soll, dass TIANE eine Sprachausgabe mitten im Satz unterbrechen und in einem anderen Raum fortsetzen kann; so könnte sie selbst mit Nutzern, die sich gerade zügig durchs Haus bewegen, eine gute Kommunikation aufrechterhalten.

TIANE bringt auch schon von Haus aus viele der nützlichen Grundfunktionen moderner Sprachassistenten mit: Vom einfachen „Wirf einen Würfel“ über „Wie wird das Wetter morgen in meiner Stadt“ bis hin zu „Erinnere mich nächsten Donnerstag daran, meiner Schwester zum Geburtstag zu gratulieren“ funktionieren alle gängigen Kommandos.

Besonderen Wert haben wir dabei auf unsere tiefreichende Modulschnittstelle sowie deren umfassende Dokumentation gelegt: Erst eine solche Schnittstelle erlaubt es schließlich, TIANE mit weiteren derartigen Funktionen zu erweitern (und diese gerne auf unserer GitHub-Seite einzureichen!) sowie vor allem eigene SmartHome-Hardware anzusteuern. Unser „Guide zur Modulentwicklung“ (ebenfalls auf GitHub) leistet dabei die benötigte Hilfestellung.

Ein großer Kritikpunkt ist natürlich, dass TIANE trotz all ihrer schönen Open-source-Komponenten an einer so zentralen Stelle wie der Spracherkennung auf einen unfreien Online-Dienst zurückgreifen muss: Das Konzept des voll vertrauenswürdigen Sprachassistenten wird dadurch teilweise in Frage gestellt. Leider sahen wir uns zu diesem Schritt gezwungen, um TIANE überhaupt alltagstauglich zu machen; vorhandene Open-source-Lösungen (Pocketsphinx…) lieferten schlicht und einfach eine viel zu geringe

Erkennungsquote. Aus diesem Grund ist die Spracherkennung über die Datei „stt.py“ aber auch frei austauschbar, eine alternative Version mit Pocketsphinx statt Google findet sich sogar schon fertig in unserem GitHub-Repository.

Unsere Hoffnungen setzen wir an dieser Stelle aber voll auf Mozillas Projekt „Deep Speech“ [11], das auf Basis von „TensorFlow“ eine hochwertige Spracherkennung rein lokal und vollständig open-source bieten soll. TIANEs Kern-Code haben wir an einigen Stellen sogar bereits ganz bewusst so konzipiert, dass die für Deep Speech nötigen Änderungen (aufgrund des voraussichtlich deutlich höheren Ressourcenbedarfs müsste die Spracherkennung dann vermutlich auf dem Server ablaufen) leichter implementiert werden können. Leider ist das dazugehörige Projekt „Common Voice“ zur Sammlung von Trainingsdaten für die Spracherkennung in Deutschland erst im Juni 2018 gestartet [12], dementsprechend unbrauchbar für den Realbetrieb ist auch diese Lösung derzeit noch. Der Stand der englischen Version lässt jedoch durchaus hoffen. Mit der Implementierung von Deep Speech als Spracherkennungsengine würde TIANE endlich vollständig auf open-source-Code setzen und damit zur echten Alternative für alle werden, die zwar die Vorzüge eines Sprachassistenten schätzen, bisher aber aufgrund von Datenschutzbedenken die Anschaffung gescheut haben.

Quellenverzeichnis

Jeglicher von uns geschriebener Quellcode für TIANE sowie weitere Dokumentation findet sich in unserem GitHub-Repository: https://github.com/FerdiKr/TIANE.

Ein Direktlink zu unserem in dieser Arbeit oft erwähnten „TIANE-Guide zur Modulentwicklung“ findet sich hier (https://github.com/FerdiKr/TIANE/blob/master/TIANE - Guide zur Modulentwicklung.pdf)

Teile des Codes einiger Module stammen aus folgenden Quellen:

- Gesichtserkennung: https://www.pyimagesearch.com/2018/06/18/face-recognition-with-opencv-python-and-deep-learning/
- GUI: https://github.com/Jo-Dan/The-Machine

TIANE verwendet folgende fremde Bibliotheken:

- [3] speech_recognition (https://github.com/Uberi/speech_recognition)
- [4] snowboy (https://snowboy.kitt.ai/)
- [5] py-picotts (https://github.com/gooofy/py-picotts)
- [6] pyaudio (https://github.com/jleb/pyaudio)
- [7] OpenCV (https://github.com/opencv/opencv)
- [8] imutils (https://github.com/jrosebr1/imutils)
- [9] dlib (https://github.com/davisking/dlib)
- [10] face_recognition (https://github.com/ageitgey/face_recognition)

Sonstige Quellen:

- [1]:https://www.heise.de/newsticker/meldung/Alexa-Tondateien-preisgegeben-Haette-Amazon-Nutzer-informieren-muessen-4257970.html
- [2]:https://www.zeit.de/digital/datenschutz/2018-12/amazon-datenschutz-alexa-nutzerdaten
- [11] https://github.com/mozilla/DeepSpeech
- [12] https://blog.mozilla.org/press-de/2018/06/07/common-voice-wird-mehrsprachig/

Die Wetterdaten bezieht TIANE aus der API von open-weather-map.com

Sämtliche in diesem Text verlinkten Webseiten wurden am 30.03.2018 aufgerufen.

Bei unseren Tests während der Entwicklung von TIANE sind keine Roboter zu Schaden gekommen ☺