Implementation eines Routingverfahrens auf Basis eines LORA-Hardwaremodul

Kay Valentin Birth

Matrikelnummer: 578026

Angewandte Informatik WiSe 22/23 Technik Mobiler Systeme, Hochschule für Wirtschaft und Technik Berlin

Kay.Birth@Student.HTW-Berlin.de

Inhaltsverzeichnis

1 Einleitung

Diese Arbeit entsteht im Rahmen des Semesterprojektes im Kurs Drahtlose Netzwerke der HTW Berlin bei Herrn Prof. Dr. Huhn. Ziel soll es sein einen Kommunikationsstack zu implementieren, welcher LoRa Hardwaremodule beinhaltet und ein Routing Protokoll auf Basis von AODV [1] nutzt. Die Implementation erfolgt in Python und nutzt eine Interaktive Shell als Nutzerinterface.

Der Programmcode ist als GitHub Repository öffentlich verfügbar [2].

2 Routing Protokoll auf Basis von AODV

Die Basis des Routing Protokolls stellt das Ad-hoc on-demand Distance Vector Routing Protokoll(AODV) [1] bereit. Dieses wurde an die Gegebenheiten von LoRa angepasst. Teil der Änderungen sind die Definierung der Nachrichtenpakete und die damit verbundenen Änderungen an der Verarbeitung dieser.

2.1 Nachrichtenformate

2.1.1 User Data

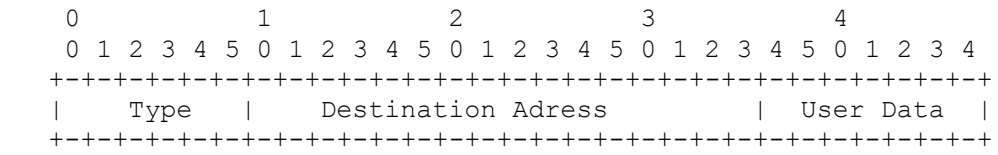

Abbildung 1- User Data Format

Das Format für Nutzdaten befindet sich in Abbildung 1, eine Erklärung deren Felder in Tabelle 1.

Tabelle 1 - User Data Felder

Typ	0
Destination Adress	Adresse des Zielknoten
User Data	Nutzdaten der Länge 1-228

2.1.2 Route Request

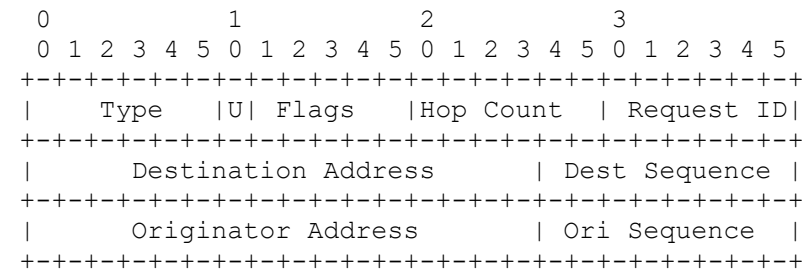

Abbildung 2 - Route Request Format

Das Format für ein Route Request befindet sich in Abbildung 2, eine Erklärung deren Felder in Tabelle 2.

Tabelle 2 - Route Request Felder

Typ	1
Flags	1.Bit – Unknown Sequence Number (U) Rest reserviert
Hop Count	Anzahl der Hops vom Originator zum Knoten welcher den Request bearbeitet
Request ID	Eindeutige Identifikation des Request im Zusammenhang mit der Originator Adresse
Destination Adress	Zieladresse der gewünschten Route
Destination Sequence Number	Letze bekannte Sequenznummer der Zieladresse
Originator Adress	Adresse des Knotens welcher die Request ursprünglich erstellte
Originator Sequence Number	Sequenznummer des Originator Knotens

2.1.3 Route Reply

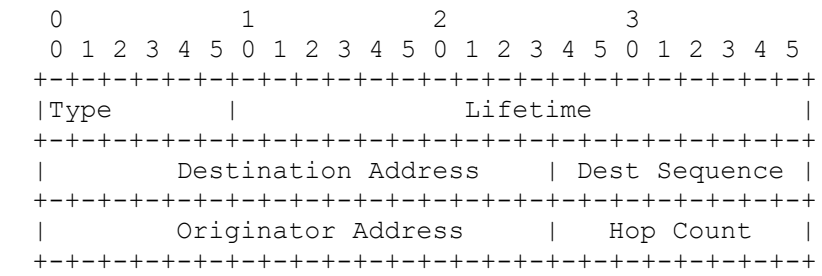

Abbildung 3 - Route Reply Format

Das Format für ein Route Request befindet sich in Abbildung 3, eine Erklärung deren Felder in Tabelle 3.

Tabelle 3 - Route Reply Felder

Typ	2
Lifetime	Die Zeit in Millisekunden, für die die Knoten, die das RREP empfangen, die Route als gültig betrachten.
Destination Adress	Zieladresse der gewünschten Route
Destination Sequence Number	Letze bekannte Sequenznummer der Zieladresse
Originator Adress	Adresse des Knoten welcher die Request ursprünglich erstellte
Originator Sequence Number	Sequenznummer des Originator Knotens

2.2 Konstanten

Tabelle 4 - AODV Konstanten

ACTIVE_ROUTE_TIMEOUT	13 000 Millisekunden
NODE_TRAVERSAL_TIME	40 Millisekunden
NET_DIAMETER	35
RREQ_RETRIES	2
MY_ROUTE_TIMEOUT	2* ACTIVE_ROUTE_TIMEOUT

NET_TRAVERSAL_TIME	2*NODE_TRAVERSAL_TIME*NET_DIAMETER
PATH_DISCOVERY_TIME	2 * NET_TRAVERSAL_TIME

3 Komponenten

Das Programm besteht aus einer interaktiven Shell als User Interface, einem Client für die Interaktionen mit der seriellen Schnittstelle und der Protokollimplementation.

3.1 User Interface

Python bietet für interaktive Shells die Standardbibliothek cmd [3] an. Es wird eine eigene Klasse erstellt, welche von „cmd“ erbt. Innerhalb dieser Klasse können nun Methoden mit dem Präfix „do_“ implementiert werden, welche dann als Befehl in der Shell verfügbar sind. Weitere Parameter, welche nach dem Befehle folgen, werden ebenfalls an die Methode übergeben. Gibt man in der Shell „write a“ ein, so wird die Methode „do_write“ mit dem Parameter „a“ aufgerufen. Durch das Hinzufügen von Python Docstrings werden entsprechende Methoden automatisch dokumentiert und sind über den Befehl „help“ auffindbar. Mit „help cmd“ wird der Docstring für die „do_cmd“ Methode ausgegeben. Die Shell wird durch das Aufrufen der cmdloop() Methode, auf der selbst erstellten Klasse, gestartet. Diese Methode wurde von „cmd“ geerbt.

3.2 Client für Serielle Schnittstelle

Die Kommunikation mit dem Lora-Modul erfolgt über Bluetooth beziehungsweise über die dadurch bereitgestellte serielle Schnittstelle. Auch hier wird innerhalb einer Klasse alle Funktionalität diesbezüglich gebündelt. Die PySerial [4] Bibliothek unterstützt die Interaktion mit der Schnittstelle.

Bevor mit der Schnittstelle interagiert werden kann, muss diese Eingerichtet werden. Dieser Vorgang wird in der „setUp“ Methode durchgeführt. Dabei werden dem Nutzer alle möglichen seriellen Schnittstellen, sowie die Möglichkeit eine manuell anzugeben, zur Auswahl angeboten. Mit der ausgewählten Schnittstelle wird ein Serial Objekt aus der PySerial [4] Bibliothek erstellt und dient nun als Interaktionsobjekt. Das Lesen und Schreiben in diese Schnittstelle wird über 2 Queues und Threads realisiert.

Der Thread zum Schreiben prüft zunächst ob in der zugehörigen Queue etwas zum Schreiben ist. Sobald etwas vorhanden ist, entnimmt der Thread die Nachricht aus der Queue, bereitet sie für das Modul auf, indem Leerzeichen am Anfang und am Ende entfernt werden und anschließend ein „\r\n“ angefügt werden. Letzteres signalisiert dem Modul das Ende eines Befehls. Nach der Aufbereitung wird die Nachricht entsprechend „UTF-8“ enkodiert und an das Modul übertragen.

Der Thread zum Lesen prüft zunächst, ob etwas zum Auslesen vorhanden ist. Sollte dies der Fall sein, wird zeilenweise ausgelesen und die Nachrichten für die Weitergabe an das Protokoll aufgearbeitet. Das erfolgt insofern, dass nur die Nachrichten, die für das Protokoll relevant sind, also all jene Nachrichten, welche von einem anderen Modul stammen berücksichtigt werden. Die Unterscheidung wird mit REGEX vollführt.

Die Aufgabe des Clients ist es ebenfalls, das angeschlossene Hardwaremodul nicht zu überlasten. Sollte dies eintreten kann nun ein Neustart, durch trennen der Stromzufuhr, das Modul wieder funktionsfähig machen. Solch eine Überlastung entsteht, wenn mehrere Befehle übermittelt werden bevor das Modul den vorherigen Befehl bestätigt. Deshalb signalisiert dem Schreiben Thread ein Bool wann nicht geschrieben werden darf. Dieses Bool wird vom Lese Thread aktuell gehalten, in dem es bei dem Erhalt von Befehl Bestätigungen dieses auf False setzt. Das Filtern der Befehl Bestätigungen wird ebenfalls mit REGEX bewerkstelligt.

3.3 Implementation des Protokolls

Das in Kapitel 2 dargestellte Protokoll wird innerhalb eines Python Moduls implementiert und ist im Ordner AODV zu finden vgl. [2].

Die dort genannten Datenpakete werden in Klassen übersetzt und sind in der Models.py [2] gebündelt. Das Enkodieren der Datenpakete in ihre Base64 Repräsentation, und umgekehrt werden Mithilfe der Python Bibliothek Bitstring [5] und der Standardbibliothek Base64 [6] realisiert. Dafür wird für jedes Packet ein Format String definiert, welcher die Datentypen und die zugehörigen Klassenvaribalen enthält. Bitstring ist nun in der Lage mithilfe dieses Strings und dem Objekt selbst, die darin gespeicherten Daten in ein Bit Array zu wandeln, welches dann von der Standard Bibliothek Base64 [6] verwendet wird um die Bits in Base64 zu kodieren. Der Umgekehrte Vorgang befüllt ein entsprechendes Objekt mithilfe eines Base64 Strings.

Die Routen sowie die Routing Tabellen sind ebenfalls eigene Klassen. Routingtabellen sind prinzipiell Dictionaries, wobei die Zieladresse als Schlüssel und das Routen Objekt als Wert hinterlegt ist. Zusätzlich sind weitere Klassenmethoden zum Interagieren mit der Tabelle implementiert. Unter anderem besteht die Möglichkeit zu prüfen, ob eine Tabelle einen Eintrag für ein Ziel hat. Auch der Sequenznummernvergleich zur Bestimmung der Aktualität zwischen Einträgen und Datenpaketen wird als Klassenmethode realisiert. Dabei werden die als unsigned Integer gespeicherten Werte zunächst in signed umgewandelt und dann entsprechend dem Protokoll [1, Kap. 6.1] verglichen. Die Umwandlung erfolgt erneut mit der Bitstring [5] Bibliothek, indem der gespeicherte unsigned Integer Wert in ein Bit Array gewandelt wird, welches dann als signed Integer interpretiert wird. Der Ablauf für das Updaten von Routen in Abhängigkeit vom Auslösenden Datenpaket ist ebenfalls in separate Methoden getrennt und entsprechend dem Protokoll implementiert. Das Prüfen der Lifetime von Routen übernimmt ein eigener Thread, welcher bei der Erstellung des Routing Table Objektes gestartet wird. Sofern etwas in der Tabelle vorhanden ist, wird die Lifetime geprüft und sofern abgelaufen, wird der zugehörige Routeneintrag verworfen. Die Threadsicherheit bei der Interaktion mit dem Dictionary wird durch Locks sichergestellt. Damit der Thread nicht in einer enormen

Geschwindigkeit praktisch immer die Tabelle prüft und sie somit sperrt, wartet der Thread nach jedem Loop mindestes die Standard Lifetime einer Route ab.

Die Protokolllogik selbst befindet sich in der AODV.py vgl. [2] und ist auch in der Klasse AODV gebündelt. Bei der Instanziierung wird die Queue, welche mit dem Schreiben Thread des Seriellen Clients kommuniziert, übergeben damit es eigenständig Nachrichten an das Hardwaremodul schicken kann. Entsprechende Hilfsmethoden, welche die Zieladresse des Hardwaremoduls vor dem Senden der Nachricht anpasst, sind vorhanden.

Die parse Methode der Klasse ist Anlaufstelle für den Seriellen Client, um Nachrichten anderer Module an das Protokoll zu leiten. Die Methode identifiziert die Nachrichten und erstellt zugehörige Datenpaket Objekte und befüllt diese. Insgesamt gibt es drei Verarbeitungsmethoden: für Nutzdaten bei denen geprüft wird, ob die Daten für das eigene Modul vorgesehen sind oder ob diese entsprechend der Routing Tabelle weitergeleitet werden müssen; für die Route Requests und für die Replies welche entsprechend dem Protokoll mit den Routing Tabellen interagieren und weitere Datenpakete an andere Module senden. Bereits bearbeitete Route Requests müssen entsprechend des Protokolls zwischen gespeichert werden um eine doppelte Behandlung zu verhindert. Das aktuell Halten dieses Zwischenspeichers, welcher eine Dictionary ist, übernimmt wieder ein Thread, welcher identisch eine Lifetime prüft wie jener Thread in den Routingtabellen. Nutzdaten, welche versendet werden sollen, müssen ebenfalls zwischengespeichert werden da potenziell mehrere Versuche nötigt sind, um eine Route zu finden. Dieser Vorgang wird ebenfalls in einem gesonderten Thread ausgeführt. Der Thread prüft, ob etwas im Zwischenspeicher, einem Dictionary, vorhanden ist. Sofern dies gegeben ist, prüft der Thread ob die Nutzdaten nicht bereits die maximale Anzahl an Sendeversuchen erreicht hat. Ist das der Fall werden die Daten verworfen und der Vorfall geloggt. Sind noch Versuche frei, wird geprüft, ob eine Route vorhanden ist, sofern ja werden die Daten über die Route versendet. Wenn nicht wird ein Route Request initiiert, der gescheiterte Versuch vermerkt und die Wartezeit entsprechend einer Binären-Backoffs angepasst.

4 Emulation des Funkmoduls

Die Verfügbarkeit der Hardwaremodule beschränkt sich auf die Einrichtungen der Hochschule. Um ein Unabhängiges Testkonzept zu entwickelt wurde ein Emulator für das Funkmodul entworfen.

Das Threadbasierte Funktionsprinzip, um mit der seriellen Schnittstelle zu interagieren, ist gleich dem eigentlich Programm. Die einkommenden Befehle werden mithilfe von REGEX analysiert und genutzt, um die zugehörigen Methoden aufzurufen, welche eine Klasse bereitstellt. Dieser Emulator ist in der Lage die Grundlegenden Funktionen des LoRa Hardwaremoduls zu simulieren und kann auf das Routingprotokoll ausgebaut werden.

Literaturverzeichnis

[1] S. R. Das, C. E. Perkins, und E. M. Belding-Royer, „Ad hoc On-Demand Distance Vector (AODV) Routing“, Internet Engineering Task Force, Request for Comments RFC 3561, Juli 2003. doi: 10.17487/RFC3561.

[2] K. V. Birth, „ValentinBirth/SerielleKommunikation“, *GitHub*. https://github.com/ValentinBirth/SerielleKommunikation (zugegriffen 1. Februar 2023).

[3] „cmd — Support for line-oriented command interpreters“, *Python documentation*. https://docs.python.org/3/library/cmd.html (zugegriffen 1. Februar 2023).

[4] „pySerial“. pySerial. Zugegriffen: 1. Februar 2023. [Online]. Verfügbar unter: https://github.com/pyserial/pyserial

[5] S. Griffiths, „scott-griffiths/bitstring“. Zugegriffen: 1. Februar 2023. [Online]. Verfügbar unter: https://github.com/scott-griffiths/bitstring

[6] „base64 — Base16, Base32, Base64, Base85 Data Encodings“, *Python documentation*. https://docs.python.org/3/library/base64.html (zugegriffen 1. Februar 2023).