Design and Implementation of a Distributed Tree-based Frequency Selection Protocol

Gabriel Behrendt und Jonas Kuß

Ein Bericht für das Modul
Protocol Programming Lab

Technische Universität Berlin
21. August 2022

Inhaltsverzeichnis

Kapitel 1

Einführung und Motivation

Die kabellose Kommunikation hat den Vorteil, dass Kosten erspart werden, da keine zusätzlichen Kabel und Switches gelegt und gewartet werden müssen. Auf diese Weise können Netzwerke spontan und einfach aufgebaut werden. Zudem eröffnet sich die Möglichkeit der Mobilität, eben weil die feste Vernetzung nicht benötigt wird. Rechner können sich, solange sie in Reichweite bleiben, frei bewegen und sind dennoch weiterhin ansprechbar. Diese Gründe erklären die weite Verbreitung von kleineren Rechnern wie Internet of Things Geräte, eingebettete Systeme, Handys und Wireless Sensor Networks. Allein die Anzahl an zusammengeschalteten Internet of Things Geräten wird zum Jahre 2030 bis auf 29 Milliarden geschätzt[1]. Diese Entwicklung wäre ohne kabellose Kommunikation unvorstellbar.

Allerdings ist diese Form der Kommunikation weniger zuverlässig. Da Nachrichten *over-the-air* übertragen werden, kann es häufiger zu Interferenzen kommen, die die Daten fälschen. Wenn eine solche Interferenz durch das Übertragen mehrerer Geräte über den selben Kanal ausgelöst wird, spricht man von Kollisionen. Anders als bei der verkabelten Kommunikation, wo durch Switches dieses Problem nicht mehr existiert, sind bei der kabellosen Kommunikation Kollisionen im Allgemeinen nicht vermeidbar.

Es gibt verschiedene Lösungen, um das Problem zu umgehen. Ein Ansatz wäre ein synchrones Kommunikationsmodell. Synchronisierte Netzwerke haben die Möglichkeit Sendezeiten einzuteilen, damit zu jedem Zeitpunkt höchstens ein Gerät Daten überträgt. Eine solche Lösung hat aber einen hohen Bedarf an Koordination, unter anderem durch Zeitsynchronisation, die teuer und ungenau sein kann. Außerdem erzielt man bei unregelmäßiger und unvorhersehbarer Kommunikation einen schlechten Datendurchsatz. Ein Netzwerkteilnehmer, im folgenden *Node* genannt, der bereit ist zu schicken, muss immer auf die eigene Sendezeit warten, unabhängig davon, ob andere Nodes im Moment gerade im Übertragen begriffen sind.

Diese Problematik bestehet bei einem asynchronen Kommunikationsmodell nicht. Die Nodes können durch *random access* jederzeit auf den Kanal zugreifen. Kollisionen werden dann durch *Collision Detection/Avoidance* gelöst. Kommt es zu einer Kollision, muss gewährleistet sein, dass die kommunizierenden Nodes in der Lage sind, diese zu erkennen und die Senderoperation an einem späteren Zeitpunkt zu wiederholen, was häufig in zufälligen, exponentiell wachsenden Abständen (*expo-*

[1] www.statista.com/statistics/1183457/iot-connected-devices-worldwide/, abgerufen am 18. August 2022.

nential Backoff) stattfindet. Dieser Ansatz erzielt also eine effizientere Einteilung des Zugriffs auf das Kommunikationsmedium.

Jedoch kann es bei vielen gleichzeitig aktiven Nodes zu sehr großen Backoffzeiten kommen, wodurch sich wiederum der Durchsatz verschlechtert. Hat man mehrere Frequenzen zur Verfügung, bietet sich hierfür das Prinzip des *Frequency Division Multiplexing* an, wodurch die Nodes auf dem Frequenzband verteilt werden können.

Es kann also in einem kabellosen Netzwerk aber auch zu einem Szenario kommen, in dem Übertragungen häufig, aber nicht vorhersehbar stattfinden. Ein Node könnte jederzeit versuchen, mit einem anderen beliebigen Node zu kommunizieren. In einem solchen Fall ist also ein asynchroner Ansatz erforderlich. Jedoch kann die Anzahl an Nodes so groß sein, dass die Kommunikation auf einer einzigen Frequenz nicht praktikabel ist, weshalb die Nodes auf verschiedene Frequenzen aufgeteilt werden müssen.

Die Herausforderung ist nun, die Kommunikation transparent zwischen allen Paaren von Nodes zu ermöglichen, obwohl diese so verteilt sein können, dass der eine Node nicht weiß, auf welcher Frequenz sich sein Gesprächspartner befindet. Die triviale Lösung wäre das Durchsuchen des ganzen Frequenzbandes. Längere Suchen führen jedoch zu erhöhten Latenzen, die man eventuell vermeiden könnte.

In diesem Bericht stellen wir ein Protokoll vor, welches genau diese Problematik umgehen kann und es Nodes in einem solchen Umfeld erlaubt, frei miteinander zu kommunizieren. Der vorliegende Bericht ist folgendermaßen aufgebaut: In Kapitel 2 führen wir unseren Ansatz ein, indem wir unsere Zielsetzungen bei Beginn des Projekts erläutern. Die tatsächliche Implementierung des Protokolls wird in Kapitel 3 beschrieben. Schließlich dient Kapitel 4 der Analyse und der Diskussion des Protokolls. Hierbei geben wir auch einen Ausblick auf mögliche Verbesserungsansätze.

Kapitel 2

Ansatz

Für unser Projekt konzentrieren wir uns auf das in Kapitel 1 eingeführte Szenario und die damit verknüpften Herausforderungen. Wir setzen uns als Ziel, ein Protokoll zu entwerfen und zu implementieren, welches kabellosen Geräten die Möglichkeit gibt, frei miteinander zu kommunizieren. Zusätzlich sollen vermeidbare Kollisionen durch Frequency Division Multiplexing verhindert werden und die Suche nach einem bestimmten Node im Frequenzband effizient verlaufen. Wir erforschen dadurch den Prototyp eines Dienstes, der mobile Kommunikation zur Verfügung stellt, hohen Durchsatz und niedrige Latenzen erzielt und dabei die Nodeverteilung und Suche abstrahiert.

In diesem Kapitel führen wir die Ansprüche und Zielsetzung des Projektes ein, wie sie anfangs festgelegt wurden. Dabei muss beachtet werden, dass in manchen Aspekten dann in der Implementierung davon abgewichen wird. Diese Abweichungen und deren Begründungen sind in Kapitel 4 erläutert.

Geplant ist, ein Protokoll zu entwickeln, welches auf Basis einer Baumstruktur Nodes auf verschiedene Frequenzen verteilt. Die Baumstruktur stellt die Beziehung zwischen den Nodegruppen auf jeweils einer Frequenz her und soll ausgenutzt werden, um eine schnellere Suche zu ermöglichen. Genauer gesagt soll bei einer Suche die Anzahl an *Hops*, also Frequenzwechsel, und die Menge an ausgetauschten Nachrichten minimal gehalten werden. Damit dieses Ziel erreicht werden kann, muss das Protokoll in der Lage sein, den Baum nach einer Veränderung der Nodegruppen umzubalancieren. Nodegruppen sollen also abhängig von ihrer Größe die Position im Baum tauschen. Diese Operationen sollen durch Repräsentanten der Nodegruppen koordiniert werden, sogenannte *Leader*. Wir haben außerdem das Ziel, bei jedem Node zusätzlich einen Cache einzubauen, um bei Gelegenheit schnellere Suchen zu ermöglichen.

In einem verteilten System mit vielen Nodes muss man mit einer hohen Ausfallwahrscheinlichkeit rechnen. Dementsprechend muss ein Protokoll auch potenzielle Fehler tolerieren können. Vor allem bei der kabellosen Kommunikation kann es oft zu einer erfolglosen Nachrichtenübertragung kommen. Da die Bearbeitungszeit des Projekts begrenzt ist, haben wir uns dennoch entschieden, Fehlertoleranzmechanismen nur bei Bedarf und Zeit zu implementieren, um den Fokus auf die Funktionalität des Grundkonzepts zu legen.

Die Problemstellung in Kapitel 1 lässt schon erkennen, dass die Funktionalität des Protokolls nur bei einer größeren Menge an Nodes einschätzbar ist. Wir sind uns

bewusst, dass man das Protokoll mit den zwei zur Verfügung gestellten BeagleBone Blacks[1] nicht ausführlich testen kann. Wir haben uns deswegen dafür entschieden, parallel zur Radiokommunikation eine weitere, virtuelle Kommunikation zu ermöglichen, wobei die Frequenzkanäle durch virtuelle Kanäle nachgeahmt werden. Beim Entwurf des Protokolls war der Plan, diese virtuelle Kommunikation auf Basis von Virtual Ethernet Schnittstellen[2] zu implementieren.

Bei dem Konzept des simulierten Nodes ist einzuräumen, dass diese Abstraktion den Nachteil hat, kein keine Kollisionen und Übertragungsfehler abbilden zu können. Wir sehen dies jedoch vor allem als Möglichkeit, beim Implementieren potenzielle Fehlerquellen voneinander zu trennen.

Neben der virtuellen Implementierung soll auch eine kabellose Kommunikation ermöglicht werden, damit das Protokoll auch seinen eigentlichen Zweck erfüllen kann. Angesichts des Szenarios und der in Kapitel 1 erwähnten Vorteile, soll auf den einzelnen Kanälen eine asynchrone Kommunikation mit *random access* stattfinden. Obwohl das Protokoll durch das Verteilen von Nodes auf verschiedenen Frequenzen das Problem von Kollisionen mindern soll, kann man diese gleichwohl nicht ausschließen. Unser Protokoll soll also auch einen Mechanismus bereitstellen, um Kollisionen zu behandeln. Dafür sollen durch Abhören des RSSI Kollisionen erkannt und mit einem rudimentären, nach IEEE 802.3 inspirierten, *binary exponential backoff* im Weiteren verhindert werden.

Um den Umfang des Projektes angemessen der Bearbeitungszeit anzupassen, ist uns bewusst, dass die Komplexität des Projektes soweit wie möglich reduziert werden muss. Wir haben uns deswegen entschieden, dass der algorithmische Teil, auf dem die Baumstruktur basiert, zum Schwerpunkt werden sollte. Das Restliche sollte nur bei vorhandener Zeit behandelt werden. Bei der Entwicklung des Protokolls liegt der Fokus auf Designentscheidungen, die zu einem effizienteren Protokoll führen. Dabei gehen wir jedoch an manchen Stellen Kompromisse ein, die allein angesichts der begrenzten Bearbeitungszeit begründbar sind.

[1] beagleboard.org/black, abgerufen am 20. August 2022.
[2] man7.org/linux/man-pages/man4/veth.4.html, abgerufen am 18. August 2022.

Kapitel 3

Implementierung

In diesem Kapitel beschreiben wird die Implementierung des Protokolls. Dafür führen wir als erstes die Funktionsweise zusammenfassend ein, um einen Überblick zu verschaffen. Danach beschreiben wir die Nachrichten und deren Aufbau, um dann schließlich tiefer auf die verschiedenen Aspekte des Protokolls einzugehen.

3.1 Grobe Beschreibung der Funktionsweise

In unserem Protokoll hat jeder Node eine globale Sicht der Baumstruktur, worin ein Knoten jeweils eine Frequenz des Frequenzbands darstellt. Es gibt eine Wurzelfrequenz *base*, die niedrigste (in unserer jetzigen Implementierung 820 MHz), worin sich Nodes nach ihrem Start befinden. Die Nachbarn der Frequenz f werden von den folgenden Gleichungen bestimmt:

$$\begin{aligned} \text{parent} &= \text{base} + \frac{(f - \text{base} - 1)}{2} \\ \text{child}_\text{l} &= \text{base} + (f - \text{base}) \cdot 2 + 1 \\ \text{child}_\text{r} &= \text{base} + (f - \text{base}) \cdot 2 + 2 \end{aligned}$$

Beim Eintreten einer Frequenz ist das Verhalten eines Nodes in der Regel davon abhängig, ob ein Leader bereits vorhanden ist. Ist dies der Fall, muss sich der neue Node beim Leader registrieren. Bei der Registrierung wird der eintretende Node in die vom Leader geführte Teilnehmerliste hinzugefügt. Sonst wird der Leader durch eine Wahl bestimmt. Ist der neue Node zum Leader geworden, ist er nun dafür verantwortlich, die Baumstruktur zu pflegen. Sind zu viele Nodes auf seiner Frequenz registriert, verteilt er eine Anzahl davon auf die Kindfrequenzen, um die Wahrscheinlichkeit von Kollisionen zu reduzieren.

Dadurch bilden sich, je weiter man vom Wurzelknoten abweicht, kleinere Nodegruppen. Damit diese Eigenschaft dann von der Suche ausgenutzt werden kann, müssen jedoch die Nodegruppen bei Änderung ihrer Größe ggf. die Anordnung im Baum wechseln. Diese Tauschverfahren finden paarweise zwischen Frequenzgruppen statt und werden durch die jeweiligen Leader der Frequenzen organisiert. Im Erfolgsfall wechseln Nodes beider Frequenzen in die jeweils andere Frequenz über.

Diese beiden beschriebenen Baumoperationen führen dazu, dass Nodegruppen größer sind, je näher sie an der Wurzelfrequenz liegen, was effektiv einen *Max Heap*

bildet. Das Suchverfahren kann dann durch eine Breitensuche mit Priorität auf Elternfrequenzen erst die größeren Gruppen abfragen, wodurch die Wahrscheinlichkeit steigt, dass der gesuchte Node früh gefunden wird. Zusätzlich sorgt ein lokaler Cache bei jedem Node dafür, dass bei den meisten Suchen nur die Frequenzen besucht werden müssen, wo der gesuchte Node sich auch zuletzt aufgehalten hat.

Die direkte Kommunikation innerhalb der Frequenzen findet asynchron statt. Die Nodes können jederzeit versuchen, auf den Kanal zuzugreifen. Wenn eine Kollision erkannt wird, findet ein Exponential Backoff statt. Außerdem bietet das System die Möglichkeit zu einer virtualisierten Kommunikation durch IP-basierten Multicast.

Es wird zusätzlich eine Schnittstelle zur Verfügung gestellt, mit der man das Verhalten der Nodes steuern kann. Ein Benutzer hat dadurch die Möglichkeit, Informationen zum Zustand eines Nodes auszugeben, Suchen nach Nodes zu betätigen, Frequenzgruppen zu wechseln und registrierte Nodes auf Kindfrequenzen zu verteilen, falls der angesteuerte Node der Leader ist.

Das Verhalten eines Nodes kann also mit den folgenden, sich wiederholenden, Schritten beschrieben werden:

1. Es wird geprüft, ob Befehle durch die Schnittstelle empfangen wurden. Diese werden gegebenenfalls ausgeführt.

2. Es wird geprüft, ob Nachrichten über den Kommunikationskanal empfangen werden können. Falls eine Nachricht empfangen wurde, wird sie verarbeitet und darauf reagiert.

3. Schließlich muss der Leadernode prüfen, ob Baumoperationen erforderlich sind und diese dann ausführen.

3.2 Struktur von Nachrichten im Protokoll

Der Header jeder Nachricht besteht aus einem Nachrichtentyp, sowie einer Sender ID und einer Empfänger ID, um Nachrichten im Netzwerk granular zustellen zu können.

Der Nachrichtentyp ist zusammengesetzt aus einer Aktion, die bestimmt, wie der eigentliche Typ interpretiert wird, und dem Typ selbst (Abb. 3.1).

Mögliche Werte für Aktionen sind *DO*, *DONT*, *WILL* und *WONT*. Die ersten beiden Aktionen zeichnen die Nachricht als einen Befehl aus, während die letzten beiden die Nachricht als Information oder Anfrage ausweisen. Ein *DO* / *DONT* kann auch als Reaktion auf ein *WILL* / *WONT* geschickt werden, um eine Anfrage zu beantworten.

Mögliche Werte für den Typ sind *TRANSFER*, *SWAP*, *SPLIT*, *MIGRATE*, *FIND* und *HINT*. Der Zweck der verschiedenen Nachrichtentypen wird in Abschnitt 3.3 näher erläutert.

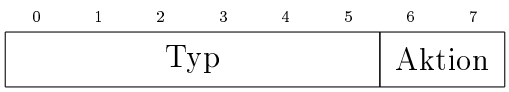

Abbildung 3.1: Der erste Byte einer Nachricht.

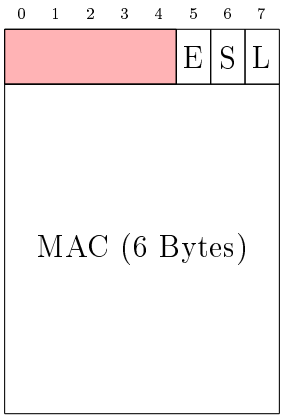

Abbildung 3.2: Eine ID im Protokoll.

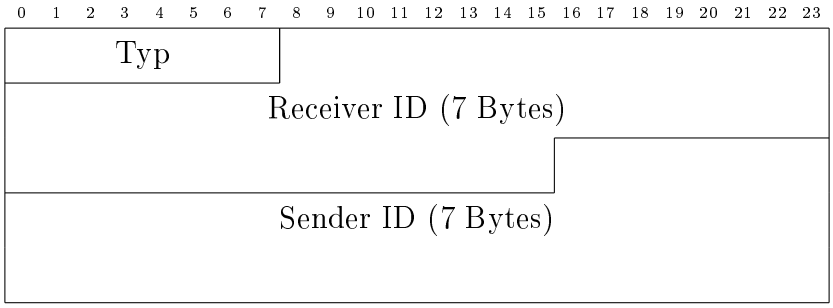

Abbildung 3.3: Der Header von Protokollnachrichten.

Um Nachrichten im Netzwerk adressieren zu können, wird jedem Node eine individuelle ID — die MAC Adresse der Ethernet Schnittstelle des Gerätes[1] — zugewiesen. Diese IDs können dann entweder als Sender oder Empfänger im Paket verwendet werden. Je nachdem, ob diese ID im Sender- oder Empfängerfeld einer Nachricht verwendet wird, enthält der erste Byte Informationen über den sendenden Node oder darüber, an wen Nachrichten mit dieser ID zugestellt werden sollen:

- **E**veryone: Eine Nachricht mit dieser ID als Empfänger wird an alle Nodes zugestellt. Wird nicht in Sender IDs verwendet.
- **S**pecific: Eine Nachricht mit dieser ID als Empfänger wird nur an den Node zugestellt, dessen MAC mit der MAC dieser ID übereinstimmt. Ist in der Sender ID immer gesetzt.
- **L**eader: Eine Nachricht mit dieser ID als Empfänger wird an beliebige Leader zugestellt, außer **S** ist gesetzt. Wenn **S** gesetzt ist, zeichnet es diesen bestimmten Node als Leader aus. In diesem Fall wird dieses Bit in Vergleichen mit anderen IDs ignoriert.

Außerdem tragen Pakete abhängig von ihren Typ eine *Payload*. Die jeweiligen Inhalte der Payloads sind in Abb. 3.4, Abb. 3.5, Abb. 3.6, Abb. 3.7 und Abb. 3.8 dargestellt.

[1] Bei der virtuellen Kommunikation handelt es sich hierbei um die Prozess ID.

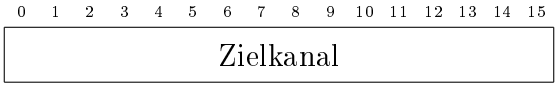

Abbildung 3.4: Eine *TRANSFER* Payload. Wird gleichzeitig auch für *MIGRATE* benutzt.

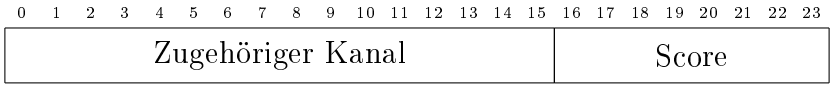

Abbildung 3.5: Eine *SWAP* Payload.

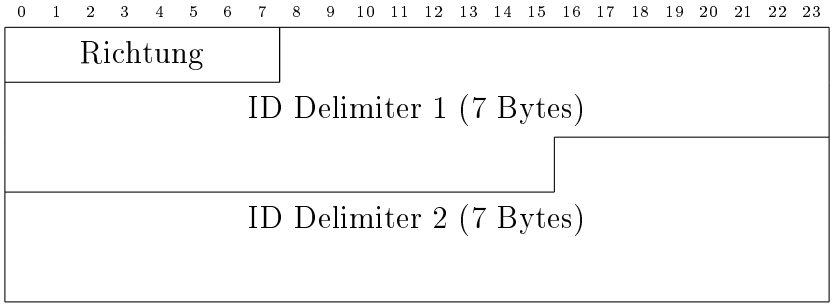

Abbildung 3.6: Eine *SPLIT* Payload.

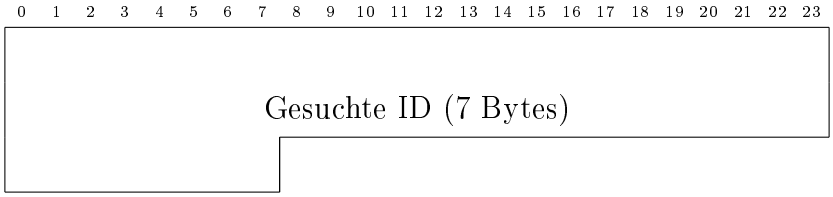

Abbildung 3.7: Eine *FIND* Payload.

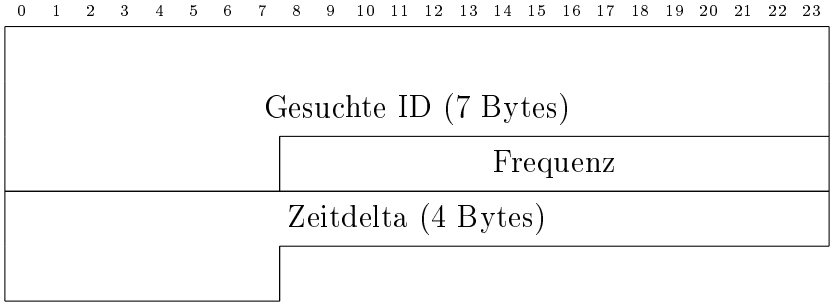

Abbildung 3.8: Eine *HINT* Payload.

3.3 Wichtige Module und deren Funktionsweise

In diesem Abschnitt gehen wir tiefer in die Aspekte des Protokolls ein, die in Abschnitt 3.1 bereits aufgeführt wurden. Dafür lohnt es sich, die eigentliche Implementierung des Protokolls zu betrachten, worin die Aufgaben auf verschiedene Module aufgeteilt werden. Wir beschreiben im Folgenden die Funktionsweise der wichtigsten Module.

3.3.1 Transport

Das Transportmodul bietet eine Abstraktion der Kommunikation und der Kanalwechsel. Die restlichen Module können auf die Transportfunktionen zugreifen, ohne unterscheiden zu müssen, ob die Kommunikation über Radio oder über virtuelle Kanäle stattfindet. Dafür stellt das Modul Funktionen zum Empfangen und Senden von Nachrichten, zum Prüfen der Kanalaktivität und zur Generierung individueller IDs im Netzwerk.

Radio

Die kabellose Kommunikation ist der eigentliche Zweck des Protokolls. Diese Aufgabe wird durch das Radio Transport Modul erfüllt, welches wiederum auf BeagleBone Modulen aufbaut. Diese stellen die Brücke zwischen der Anwendung und der Hardware dar.

Um die Kommunikation über Radio zu ermöglichen, muss bei der Initialisierung eines Nodes eine RSSI Grenze bestimmt werden, damit geprüft werden kann, ob auf einem Kanal gerade gesendet wird. Dafür werden eine Zeit lang RSSI Werte gesammelt und ein gleitender Durchschnitt gebildet. Auf diesen Durchschnitt wird am Ende ein Offset addiert, um die Grenze zu bestimmen.

Beim Senden werden Kollisionen durch Exponential Backoffs im Millisekundenbereich verhindert. Der Backoff wird zufällig aus dem Intervall $[0, 2^k - 1]$ gewählt, wobei k die Anzahl an bisher festgestellten Kollisionen für die aktuelle Nachricht ist. Eine Kollision wird erkannt, indem ein Node unverzüglich vor und nach dem Senden überprüft, ob der gemessene RSSI die festgelegte Grenze überschreitet. Sollte dies der Fall sein, wird k vergrößert und ein neuer Backoff Intervall berechnet. Dauert die gesamte Sendeoperation wegen häufigen Kollisionen zu lange, wird die Nachricht verworfen.

Beim Empfangen prüft der Node eine Zeit lang, ob der gemessene RSSI die festgelegte Grenze überschreitet. In diesem Fall wird danach versucht, die Nachricht zu empfangen. Mögliche Fehler wie das unvollständige Übertragen einer Nachricht werden durch Timeouts toleriert.

Für individuelle IDs wird die MAC Adresse der Ethernetschnittstelle des Beaglebones verwendet.

Virtual

Im virtuellen Modus erfolgt die Kommunikation lokal über die *localhost* Schnittstelle. Multicast Gruppen bilden die Kanäle, wobei jede Frequenz durch eine Multicast Adresse vertreten wird. Die Kanalwechsel finden statt, indem der alte Socket geschlossen wird und ein neuer geöffnet und an die Multicastadresse gebunden wird, die der Zielfrequenz entspricht. Die Kommunikation kann dann über diese Socket stattfinden. Für individuelle IDs wird die Prozess ID verwendet.

3.3.2 Transfer

Das Transfermodul ist zuständig für Bewegungen einzelner Nodes von einer Frequenz zu einer anderen. Ein wichtiger Bestandteil ist dabei der Registrierungsprozess. Hier

muss bedacht werden, dass bei Abwesenheit eines Leaders ein neuer gewählt werden muss, was bei gleichzeitigem Eintreten mehrerer Nodes in die Frequenz herausfordernd sein kann. Um dieses Problem zu lösen, haben wir folgendes Leader Wahlverfahren entworfen:

1. Nach einem Frequenzwechsel lauscht der wechselnde Node für $n \in [0, 50]$ms auf Kanalaktivität. Sollte der Kanal aktiv sein, scheidet der Node aus dem Wahlverfahren aus, wartet 500ms und schickt dann eine *WILL TRANSFER* Nachricht, um sich zu registrieren. Wenn diese Nachricht nicht von einem Leader beantwortet wird, startet der Node ein neues Wahlverfahren.

2. Wenn keine Kanalaktivität festgestellt werden konnte, wird der Node zum Kandidat und schickt sofort eine *WILL TRANSFER* Nachricht. Das führt dazu, dass im vorherigen Schritt Nodes mit längeren Wartezeiten aus dem Wahlverfahren ausscheiden.

3. Alle Kandidaten prüfen den Kanal noch einmal auf Aktivität und versuchen, eine Antwort des Leaders zu empfangen. Wenn ein Leader geantwortet hat, sind sie registriert und scheiden aus dem Wahlverfahren aus. Ansonsten starten die Kandidaten das Wahlverfahren von vorne.

4. Wenn ein Kandidat feststellt, dass der Kanal inaktiv ist, ist nur noch er im Wahlverfahren übrig. Dieser Kandidat wird zum Leader.

Der Leader ist dafür zuständig, eingehende *WILL TRANSFER* Nachrichten zu bearbeiten. Die Payload der Nachricht (Abb. 3.4) enthält die Frequenz, auf die der Node wechseln will. Hier muss also zwischen zwei Fällen unterschieden werden:

1. Die Frequenz ist die des Leaders. In diesem Fall will der Node sich auf der Frequenz des Leaders registrieren. Der Leader antwortet dem Node mit einer *DO TRANSFER* Nachricht und fügt ihn in seine Teilnehmerliste ein.

2. Die Frequenz ist nicht die des Leaders. In diesem Fall will der Node die Frequenz verlassen. Der Leader entfernt ihn also aus seiner Teilnehmerliste. Es wird keine Antwort geschickt, da wir es Nodes erlauben wollen, im Fall eines inaktiven Leaders die Frequenz zu verlassen.

WILL TRANSFER Nachrichten werden immer an alle Nodes einer Frequenz addressiert. Dadurch können bestehende Cache Einträge (Abschnitt 3.3.4), die den wechselnden Node enthalten, aktualisiert werden. Außerdem können dadurch die Nodes einer Frequenz automatisch ein neues Wahlverfahren starten, wenn der aktuelle Leader auf eine andere Frequenz wechseln will.

Das Transfermodul ist außerdem auch für die *Split* Operation zuständig, durch die eine Menge an Nodes auf die beiden Kindfrequenzen verteilt werden. Dafür sortiert der Leader die Teilnehmerliste mit Größe N und schickt die IDs an Index $\lfloor \frac{N}{4} \rfloor$ und $\lfloor \frac{N}{2} \rfloor$ mit einer *DO SPLIT* Nachricht an alle Nodes der Frequenz. Die Empfänger vergleichen ihre eigene ID mit denen in der Nachricht und können dadurch entscheiden, auf welche Frequenz sie wechseln müssen:

1. eigene ID $<$ erste ID: Wechsel auf linke Kindfrequenz.

2. eigene ID < zweite ID: Wechsel auf rechte Kindfrequenz.

3. ansonsten: Bleibe auf aktueller Frequenz.

Abbildung 3.6 entnimmt man, dass man in der *Split* Nachricht auch eine Richtung angeben kann. Dadurch könnten Nodes in die Elternfrequenz geschickt werden, um potenzielle Lücken im Baum zu füllen. Diese Option wird jedoch nicht mehr benutzt und ist daher obsolet.

3.3.3 Swap

Das Swapmodul übernimmt die Aufgabe, *Swap* Operationen auszuführen. Diese werden von den Leader Nodes ausgeführt, damit Gruppen von Nodes abhängig von ihrer Größe Plätze im Frequenzbaum tauschen können.

Swaps funktionieren auf Basis eines Timeouts. Der Leader überprüft nach einem bestimmten Intervall, ob sich die Anzahl an registrierten Nodes, hier *Score* genannt, seit dem letzten *Swap* verändert hat. Ist dies nicht der Fall, wird der Timeout zurückgesetzt und die Operation abgebrochen.

Hat sich die Anzahl an Nodes vergrößert, so wechselt der Leader in seine Elternfrequenz und schickt eine *DO SWAP* Anfrage mit dem aktuellen *Score* und seiner Frequenz. Der Leader der Elternfrequenz vergleicht den Score mit dem eigenen. Sollte der Score der Kindfrequenz größer sein, so wird mit einer *WILL SWAP* Nachricht geantwortet, woraufhin beide Leader eine *DO MIGRATE* Nachricht an alle Nodes der eigenen Frequenz schicken. Die Nodes wechseln dann auf den neuen Kanal und brauchen sich nicht neu zu registrieren, da ihr Umfeld gleich geblieben ist.

Diese Prozedur wird so lange wiederholt, bis der Leader den Rand des Frequenzbands erreicht hat, eine *WONT SWAP* Nachricht empfängt, oder keine Antwort bekommt.

Es besteht die Möglichkeit, dass in diesem Prozess ein Node die *DO MIGRATE* Nachricht beider Leader empfängt. Damit nur ein tatsächlicher Wechsel durchgeführt wird, werden nachfolgende *DO MIGRATE* Nachrichten für einen kleinen Zeitraum ignoriert.

Sollte der Score eines Nodes kleiner sein als beim letzten *Swap*, so wird die Prozedur analog mit den Kindfrequenzen ausgeführt. Hierbei wird bei jeder *Swap* Operation zufällig zwischen der linken und rechten Kindfrequenz gewählt, um ein Bias zu einer Seite zu verhindern. Ist der Swap bei einer Kindfrequenz nicht erfolgreich, so wird die andere angefragt.

3.3.4 Cache

Das Cachemodul sorgt dafür, dass Nodes einen Überblick darüber behalten, auf welcher Frequenz sich andere Nodes vermutlich befinden. Der Cache hat eine feste Größe, sodass veraltete Einträge automatisch durch neuere Einträge ersetzt werden. Eine erfolgreiche Suche führt zur Aktualisierung des Caches, in dem die Frequenz abgespeichert wird, auf der der Node gefunden wurde. Ein Leader fügt außerdem Nodes zum Cache hinzu, die seine Frequenz verlassen. Dieser Schritt wird auch von den Nodes ausgeführt, die bereits einen Cacheeintrag zu dem austretenden Node haben.

Der Cache ist auf Basis einer Priority Queue und einer Hashmap implementiert. Die Priority Queue sorgt dafür, dass die Cacheeinträge nach ihrem Alter sortiert sind, sodass beim Einfügen in einen vollen Cache der älteste Eintrag effizient entfernt werden kann. Die Hashmap dagegen speichert auch die Frequenzen ab, sodass ein Zugriff in $O(1)$ ermöglicht wird.

3.3.5 Suche

Durch das Suchmodul wird es einem Node ermöglicht, auch mit Nodes zu kommunizieren, die sich auf einer anderen Frequenz befinden. Die Voraussetzung hierfür ist, dass die ID des zu suchenden Node bekannt ist.

Das Herzstück der Suche ist eine Priority Queue, in der alle zu besuchenden Frequenzen gespeichert werden. Die Reihenfolge der Frequenzen in der Priority Queue werden durch folgende Regeln bestimmt:

- Frequenzen aus Cache Einträgen haben die höchste Priorität. Wenn zwischen zwei Cache Einträgen verglichen wird, hat der neuere die höhere Priorität.
- Je näher eine Frequenz an der Wurzelfrequenz liegt, desto höher ist ihre Priorität.

Am Anfang der Suche fügt der suchende Node seine eigene Frequenz zur Priority Queue hinzu, sowie die Frequenz aus seinem Cache Eintrag des gesuchten Nodes, falls vorhanden. Danach läuft die Suche nach den folgenden Schritten ab:

1. Nehme die als nächstes zu besuchende Frequenz aus der Priority Queue, solange sie nicht leer ist und die Frequenz nicht vorher schon besucht wurde. Wenn die Priority Queue leer ist, dann konnte der gesuchte Node nicht gefunden werden.
2. Wechsel auf die ausgewählte Frequenz und schicke eine *DO FIND* Nachricht mit der ID des gesuchten Nodes an alle Nodes der Frequenz.
3. Empfange alle Antworten auf die Suchanfrage. Wenn der gesuchte Node mit einem *WILL FIND* selbst geantwortet hat, dann ist die Suche beendet.
4. Wenn nicht, dann füge die Frequenzen aller gesendeten Cache Einträge sowie die Nachbarfrequenzen der aktuellen Frequenz in die Priority Queue ein. Markiere außerdem die aktuelle Frequenz als besucht. Fange danach wieder mit Schritt 1 an.

Falls die Suche erfolgreich war, meldet sich der suchende Node bei seiner registrierten Frequenz ab und wechselt auf die Frequenz des gefundenen Nodes.

3.3.6 Interface

Die in Abschnitt 3.1 angesprochene Schnittstelle wird durch das Interfacemodul implementiert. Dieser sammelt auf der Standardeingabe die Befehle vom Nutzer in einem eigenen Thread und verarbeitet diese nebenläufig. Es wird dann darauf gewartet, dass die Hauptanwendung bereit ist, die Befehle auszuführen.

Die zur Verfügung gestellten Befehle sind in Tabelle 3.1 dargestellt.

id	Gibt die ID eines Nodes aus.
freq	Gibt den Kanal aus, auf den sich der Node zur Zeit befindet.
goto [frequency]	Erlaubt einen Kanalwechsel. Der Node meldet sich bei der aktuellen Nodegruppe ab, wechselt zu *frequency*, und beginnt das Registrationsverfahren.
searchfor [node id]	Startet die Suche nach dem Node mit der ID *node id*. Wird der Node gefunden, so meldet sich der Node bei der aktuellen Nodegruppe ab, wechselt zu der Nodegruppe vom gefundenen Node und beginnt das Registrationsverfahren.
list	Falls der Node der Leader ist, wird die Liste von registrierten Nodes auf dem aktuellen Kanal ausgegeben.
split	Falls der Node der Leader ist, wird eine Split Operation gestartet.

Tabelle 3.1: Durch die Schnittstelle zur Verfügung gestellten Befehle

Kapitel 4

Diskussion und Fazit

In diesem Kapitel wollen wir unser Protokoll bewerten, indem wir Schwächen und Stärken hervorheben, und mögliche Lösungen für die Schwächen vorstellen. Insbesondere jedoch betrachten wir, ob wir die in Kapitel 2 gestellten Anforderungen an das Protokoll erfüllen konnten.

4.1 Nachrichten Overhead

Eines der wichtigsten Ziele war, einen geringen Overhead zu erreichen, damit der Durchsatz für den Austausch von Daten so hoch wie möglich bleibt. Um zu prüfen, ob dieses Ziel erfüllt wurde, betrachten wir die Anzahl an Nachrichten, die durch unser Protokoll auf einer Frequenz verschickt werden:

- Betreten und Verlassen einer Frequenz: ein *WILL TRANSFER* und *DO TRANSFER*.
  Diese Nachrichten fallen nicht besonders ins Gewicht, da wir davon ausgehen, dass ein Node nur Frequenzen wechselt, um neue Daten zu erhalten und deswegen dort auch relativ lange bleibt.

- Swap Operationen: 1 *WILL SWAP* + 1 *DO* / *DONT SWAP* + ggf. 1 *DO MIGRATE* alle 5000 ms.
  Wir haben den *SWAP* Timeout deswegen so hoch gesetzt, damit diese maximal drei Nachrichten nur einen kleinen Teil der gesendeten Nachrichten darstellen. Wir haben außerdem anhand von Stresstests der Suchfunktion beobachtet, dass höhere Timeouts zu einer größeren Anzahl an erfolgreichen Swap Operationen — und damit einem korrekteren Baumzustand — führen. Es besteht also noch Bedarf, diesen Zusammenhang weiter zu erforschen. Jedenfalls ist diese Eigenschaft positiv einzuschätzen, da sie zu einem geringeren Overhead führt.

- Neuverteilung von Nodes: ein *DO SPLIT*.
  Obwohl dieser unser größter Nachrichtentyp ist (Abb. 3.6), konnten wir durch unseren Ansatz die Anzahl an IDs minimieren, die verschickt werden müssen. Wir gehen außerdem davon aus, dass die Abstände zwischen notwendigen Split Operationen groß sind. Wir sind also der Meinung, dass dieser Anteil an Protokollnachrichten vernachlässigbar ist.

- Suchanfragen je angefragte Frequenz:
  1 *DO FIND* + x *WILL HINT* + ggf. 1 *WILL FIND*.
  Wir haben mit unserer Suchsimulation herausgefunden, dass eine durchschnittliche Suche vier Hops benötigt. Da sich die in der Simulation insgesamt 50 Nodes bei einer Split Grenze von fünf Nodes auf zehn Frequenzen verteilen, gibt es eine Wahrscheinlichkeit von 40%, dass eine Suchanfrage an eine beliebige aktive Frequenz f gestellt wird[1]. Die Anzahl an gesendeten Antworten x bleibt dadurch gering, dass alte Cache-Einträge periodisch durch neue Einträge ersetzt werden und nur Nodes in Caches eingetragen werden, die oft gesucht werden. Diese Rechnung trifft nur auf den Fall einer erfolgreichen Suche zu. Leider wird der Anteil an nicht erfolgreichen Suchen höher, je mehr Suchen parallel ausgeführt werden. In diesen Fällen stellt der suchende Node Anfragen an alle Frequenzen. Dadurch steigt die Wahrscheinlichkeit, dass eine Suchanfrage an f gestellt wird, erheblich. Trotzdem denken wir, dass dies noch vertretbar ist, da es nicht eines unserer Ziele war, auf viele Suchen und kleine Datentransfers zu optimieren.

Insgesamt können wir also sagen, dass wir unser Ziel eines geringen Nachrichten Overheads erreicht haben.

4.2 Virtualisierung

In unserer Implementierung wurde von dem in Kapitel 2 abgesprochenen Ansatz basierend auf Virtual Ethernet Schnittstellen abgewichen. Im Gegensatz dazu benötigt unsere finale Lösung — die Simulation von Frequenzen durch IP Multicast — keine Setup-Scripts oder andere Abhängigkeiten. Die einzige technische Verbesserung wäre, einen Weg zu finden, nicht bei jedem Frequenzwechsel die Multicast Socket neu erstellen zu müssen. Nicht zu vernachlässigen ist, dass unsere Lösung keine perfekte Abstraktion der Radiokommunikation darstellt. Da die Pakete das Loopback Interface nicht verlassen, gibt es keine Paketverluste. Es wäre aber möglich, abhängig von der Auslastung der Frequenz einen Prozentsatz der Pakete zu verwerfen.

Ein weiterer Vorteil der Unterstützung von zwei verschiedenen Sendemodi ist, dass die Portierung auf unterschiedliche Hardware dadurch vereinfacht wird. Dafür müssten lediglich die Funktionen des Radio Transportmoduls neu implementiert werden.

4.3 Heap

Ursprünglich war geplant, dass jeder Leader eine Liste seiner Nachbarfrequenzen führt, um den Heap darzustellen. Bei einem Swap zwischen Frequenz a und b müssten a und b nur ihre Listen austauschen und mit den Informationen aus der jeweils anderen Liste ihre eigene anpassen. Jedoch müssten dann auch alle Nachbarfrequenzen von a und b über diesen Tausch informiert werden, damit die Baumstruktur konsistent bleibt. Dadurch, dass diese Nachbarfrequenzen auch alle gleichzeitig Swaps

[1] Dabei wurde das Frequenzband auf 30 Frequenzen reduziert und die Splitgrenze auf fünf Knoten gesetzt. Die Suchsimulation wurde 120 Sekunden lang ausgeführt.

durchführen können und natürlich weiterhin die Möglichkeit besteht, dass Nachrichten während der Swaps verloren gehen, steigt die Komplexität für eine korrekte Implementierung ins Unermessliche. Zusätzlich wäre auch der Nachrichten Overhead bei der Suche gestiegen, da ein suchender Node bei jedem Kanalbesuch den jeweiligen Leader nach den Nachbarkanälen hatte anfragen müssen.

Im Vergleich dazu ist unsere aktuelle Lösung, die Nachbarfrequenzen abhängig von der aktuellen Frequenz durch eine Formel deterministisch und dezentralisiert zu bestimmen, einfach und robust.

4.4 Suche

Ein weiteres Ziel war eine performante Suche. Wie schon in Abschnitt 4.1 beschrieben, stellt die Suche mit durchschnittlich vier besuchten Frequenzen einen vernachlässigbaren Nachrichten Overhead dar. Der ungünstigste Fall ist das Problem: Nodes können nicht gefunden werden, wenn sie gerade selbst auf der Suche sind. Im Extremfall kann das dazu führen, dass sich alle Nodes gegenseitig suchen und deswegen niemand gefunden wird. Als Lösung könnte der Leader einer Frequenz auch auf Suchanfragen antworten, wenn ein Node seiner Frequenz nicht selbst darauf antwortet. Bisher haben wir diese Lösung aber nicht implementiert, weil ein Node jederzeit die Frequenz wechseln kann, ohne dass der Leader das mitbekommt. Dann könnten Suchen nach diesem Node fälschlicherweise auf der alten Frequenz beendet werden. Mit einer besseren Verfolgung von aktiven Nodes könnte diese Lösung aber benutzt werden.

4.5 Load Balancing

Es sind auf jeden Fall auch Verbesserungen an der Funktionsweise von Split Operationen möglich. Aktuell wird nur anhand der Anzahl entschieden, ob Nodes auf die Kindfrequenzen aufgeteilt werden sollen. Das Problem daran ist, dass die Anzahl an Nodes nicht unbedingt etwas über die wirkliche Aktivität einer Frequenz aussagt. Eine Frequenz mit lediglich zwei Nodes, die konstant Daten austauschen, ist zum Beispiel aktiver als eine mit fünf Nodes, die nicht miteinander reden. Stattdessen sollte jeder Node anhand der Auslastung seiner Frequenz selbst entscheiden können, ob er auf eine andere Frequenz wechseln will. Diese Änderung würde auch die Suche verbessern, weil diese es erlaubt, dass sich eine größere Anzahl von Nodes auf den obersten Frequenzen versammelt, die nicht unbedingt unablässig Daten austauschen, sondern stattdessen mehrere Suchanfragen beantworten können.

4.6 Fehlertoleranz

Ein Bereich, den wir leider vernachlässigen mussten, ist die Fehlertoleranz, die in verteilten Systemen und besonders in der Radiokommunikation eigentlich einen hohen Stellenwert hat. Wir haben zwar durch unser Leader Wahlverfahren eine Garantie, dass Frequenzen nicht lange ohne Leader bleiben, allerdings haben wir durch unseren Fokus auf niedrigen Overhead das Problem, dass es mehrere Leader auf einer Frequenz geben kann, sobald der aktuelle Leader kurzzeitig die Frequenz verlässt.

Deswegen können wir es nicht erlauben, dass Leader nach anderen Nodes suchen. Sie müssen aber trotzdem regelmäßig auf Nachbarfrequenzen wechseln, um Swap Operationen durchzuführen, wodurch dieses Problem weiterhin auftreten kann. Ein weiteres Problem ist, dass nur der Leader die Anzahl registrierter Nodes auf der Frequenz kennt und diese Anzahl nicht nachgeprüft wird.

Deswegen kann die vom Leader gespeicherte Anzahl oft von der wirklichen Anzahl abweichen:

1. Wenn ein Node abstürzt. Die Anzahl ist dann höher als gedacht, weil der abgestürzte Node weiterhin registriert ist.

2. Wenn der Leader ein *WILL TRANSFER* nicht empfängt. Die Anzahl ist dann höher als gedacht, weil der Leader den Node nicht abmeldet.

3. Wenn der Leader abstürzt. Die Anzahl ist potenziell weit niedriger als gedacht, weil der nächste Leader nicht weiß, wer bereits schon registriert ist.

Darunter leidet die Korrektheit des Heaps, was die Suche langsamer macht.

Aus diesen Gründen wäre es besser gewesen, auf das Konzept eines Leaders zu verzichten und das Protokoll zusätzlich robuster zu gestalten. Es besteht die Möglichkeit unser Cachekonzept anpassen, um die Anzahl der Nodes auf der Frequenz besser zu verfolgen:

1. Jeder Node hat eine Memberliste und fügt kontinuierlich die Absender aller empfangenen Nachrichten hinzu.

2. Wenn der älteste Eintrag älter als N ms ist, entferne ihn aus der Memberliste. Hier sollte N groß genug gewählt werden, sodass kurze Funkpausen durch Suchen, Backoffs und Vearbeitungszeiten nicht zum Entfernen aus der Liste führen.

3. Wenn eine *TRANSFER* Nachricht zu einer anderen Frequenz empfangen wird, entferne den Absender aus der Liste.

Dadurch kennt jeder Node immer die ungefähre Anzahl anderer Nodes auf der Frequenz, ohne dass Overhead durch weitere Nachrichten hinzukommt. Ein kleiner zusätzlicher Overhead ist aber für die Durchführung eines Swaps nötig. Es muss für jeden Swap erst ein Node bestimmt werden, der die Anfrage an die Nachbarfrequenz stellt. Dieser Node sollte schon länger auf der Frequenz sein und aktuelle Einträge in seiner Memberliste haben. Dadurch würde sichergestellt werden, dass der bei *DO SWAP* mitgeschickte Score so genau wie möglich ist. Zur Bestimmung dieses Nodes könnte unser Leader Wahlverfahren benutzt werden.

Es muss hier erwähnt werden, dass durch diesen Ansatz alle Nodes im Netzwerk mehr Arbeit leisten müssen, was insgesamt den Durchsatz an Daten verringern könnte. In Anbetracht der erreichten Vorteile wäre es aber trotzdem ein guter Kompromiss.

4.7 Kommunikationsdienst

Unser letztes großes Ziel war, das Protokoll auch als richtigen Kommunikationsdienst nutzen zu können, das heißt benutzerdefinierte Daten zwischen beliebigen Nodes austauschen zu können. Es war geplant, dass die überliegende Anwendung eine Funktion bereitstellen würde, die entscheidet, ob ein Node für eine Suchanfrage zuständig ist. Diese Funktion und zwei weitere Parserfunktionen für die Suchpayload sollten dann automatisch vom Protokoll verwendet werden, um die Suche durchzuführen. Leider konnten wir wegen einer Kombination aus Zeitdruck und der zu großen Komplexität, Suche, Caches und allgemein das Protokoll dafür umzuschreiben, dieses Ziel nicht erreichen. Deswegen kann nur nach MAC Adressen gesucht werden, eine Lösung die nicht für alle Zweck geeignet ist, vor allem weil wir keinen Mechanismus zur Verfügung stellen, womit Nodes die Adressen von anderen Nodes erlernen können.

Für die Radiokommunikation kommt hinzu, dass unsere Exponential Backoff Implementierung aktuell blockierend ist. Nach einer Kollision ist es also bis zum nächsten Sendezeitpunkt nicht möglich, Nachrichten zu erhalten. Dieses Problem würde sich aber leicht mit einem asynchronen Timer und einer Warteschlange, in die zu verschickende Nachrichten eingefügt werden, lösen lassen.

4.8 Fazit

Wir haben in unserem Bericht die Probleme der kabellosen Kommunikation dargestellt und uns dabei auf ein unvorhersehbares Kommunikationsmodell konzentriert. Außerdem wurde die Implementierung unseres Protokolls beschrieben, welche anhand der Verteilung von Knoten und einer Baumstruktur das Ziel hat, die kabellose Kommunikation im erwähnten Szenario zu verbessern. Insgesamt ist das Ergebnis zufriedenstellend, da die gesetzten Ziele größtenteils erreicht werden konnten. Der Zeitmangel und initiale Fehler haben dazu geführt, dass wir suboptimale Kompromisse eingehen mussten, jedoch haben wir auch mehrere Designentscheidungen getroffen, die zu einem effizienteren, robusteren Protokoll geführt haben. Außerdem haben wir mehrere Verbesserungen genannt, womit man das Protokoll in Zukunft erweitern könnte.