OTTO VON GUERICKE UNIVERSITÄT MAGDEBURG
FACULTY OF COMPUTER SCIENCE
Communication and Networked Systems

Communication and Networked Systems

Bacheloarbeit

B.A.T.M.A.N Routing im Internet of Things

Lukas Gehreke

Betreuer: Prof. Dr. rer. nat. Mesut Güneş
Betreuender Assistent: M.Sc. Kai Kientopf

Institut für Intelligente Kooperierende System, Otto-von-Guericke-Universität Magdeburg

3. August 2020

Zusammenfassung

Zusammenfassung

B.A.T.M.A.N-adv ist ein Routingprotokoll für kabellose Mesh-Netzwerke. Es besteht bereits eine Implementierung des Protokolls als Linux-Kernelmodul. Außerdem wird das Protokoll in bereits bestehenden Netwerken, z.B. dem der Freifunk-Community, eingesetzt. Diese Bachelorarbeit befasst sich mit der Frage, ob sich B.A.T.M.A.N-adv im Internet of Things (IoT)-Sektor einsetzen lässt. Der IoT-Sektor stellt Anforderungen an eine Firmware, wie z.B. beschränkten Speicherplatz, beschränkten Arbeitsspeicher und eingeschränkte Energieversorgung. Um zu evaluieren, ob sich das Protokoll im IoT-Sektor einsetzen lässt, wurde eine Proof-of-Concept-Implementierung des Protokolls geschrieben und in Riot OS integriert. Um die Leistung der Implementierung zu bewerten, wurde sie mit verschiedenen Metriken getestet. Diese sind der Speicherverbrauch, das Packet Delivery Ratio (PDR) und die Latenz. Die Messungen wurden in zwei Netzwerken durchgeführt: dem Prototypensystem und dem MIOT-Testbed. Für den Speicherverbrauch wurde eine obere Schranke hergeleitet. Weiterhin wurde eine maximale Read-only memory (ROM)-Belegung von 95,1 KB gemessen. Die Random-Access Memory (RAM)-Belegung betrug 27,318 KB bei einer Netzwerkgröße von 8 Knoten. Das PDR ließ sich durch Anpassen der Parameter der B.A.T.M.A.N-adv-Implementierung auf 0,83-0,97 im Median anheben. Die Latenz betrug pro Hop 24 ms. Diese lässt sich allerdings durch Verändern der *Physical Layer*-Hardware verbessern. Somit ist ein Einsatz im IoT-Sektor möglich.

Abstract

B.A.T.M.A.N-adv is a routing protocol for wireless mesh networks. There already exists a implementation of the protocol as Linux Kernel module. Furthermore the protocol is used in existing networks, e.g. in the networks of *Freifunk-Community*. This Bachelor-Thesis evaluates if B.A.T.M.A.N-adv is suitable for use in the IoT sector. The IoT sector sets certain limiting conditions to a firmware. These conditions are limited ROM and RAM and limited power consumption. To evaluate the protocol a proof-of-concept-implementation was written and integrated in Riot OS. The protocol was evaluated by performance metrics. These metrics are the memory usage, the PDR and the latency. The measurements were performed on two systems: the Prototypes and the MIOT-Testbed. For the memory usage a upper bound was derived. The ROM usage of the firmware was 95.1 KB. The RAM usage was 27.318 KB at a network size of 8 nodes. By tweaking the configurable parameters of the B.A.T.M.A.N-adv implementation a PDR of 0.83-0.97 in median could be achieved. The latency was 24ms per hop. This value can be altered by changing the *physical layer* hardware. Thus the usage of B.A.T.M.A.N-adv in the IoT is possible.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Quellcodeverzeichnis

Glossar

B.A.T.M.A.N Layer 3 Protokollversion. 3, 4, 11, 17

B.A.T.M.A.N-adv Neue Layer 2 Protokollversion. iii, iv, 1–8, 10–12, 15–19, 21–27, 30–34, 38–40, 43–46

ELP Echo Location Protocol zur Interfaceerkennung. 3, 5–8, 18, 19, 25–27, 30, 33–35, 37–40, 43, 44

MIOT-Lab Magdeburg Internet of Things Testbed mit Fokus auf kabellose Mesh- und Sensornetzwerke. 1, 2, 24, 25, 31, 43

OGMv2 Originator Nachrichten der Version 2 zur Übertragung von Routing Informationen. 3–5, 7, 8, 18, 19, 25–27, 32–35, 37–40, 43, 44

OMNET++ Ein auf C++ basierendes Simulationsframework zur Simulation von Netzwerken. 15

Riot OS Ein IoT Betriebssystem. iii, iv, 1, 17–19, 22–24, 30, 40, 43

Akronyme

GCC GNU Compiler Collection. 23

ICMPv6 Internet Control Message Protocol Version 6. 24, 40, 41

IoT Internet of Things. iii, iv, 1, 2, 15, 22, 23, 42–45

MAC Media Access Control. 18

MTU Maximum Transmission Unit. 31, 43

OLSR Optimized Link State Routing. 4

PDR Packet Delivery Ratio. iii, iv, 22–24, 26, 29–31, 33, 35, 38–40, 43, 44

RAM Random-Access Memory. iii, iv, 23, 24, 29, 32, 33, 43, 44

ROM Read-only memory. iii, iv, 23, 33, 43, 44

RPL Routing Protocol for Low power and Lossy Networks. 16

RSSI Received Signal Strength Indicator. 19

RTT Round Trip Time. 24, 25, 40–42, 44

TTL Time To Live. 23, 38–40

UDP User Datagram Protocol. 22, 23, 26, 30, 31, 34, 35, 38–40

KAPITEL 1

Einführung

Saba Farooq Abbasi et al. [1] beschreibt, dass es kein standardisiertes Routingprotokoll für den IoT Sektor gibt. Laut Andreas Boes et al. [2] entwickelt sich das IoT in zahlreichen Branchen, wie z.B. Gebäudetechnik, Gesundheitsversorgung und Mobilität. Somit gewinnt folglich auch das Routing an Bedeutung. Diese Arbeit evaluiert, ob das Protokoll B.A.T.M.A.N-adv sich für den Einsatz im IoT-Sektor eignet.

B.A.T.M.A.N-adv wird schon produktiv in der Freifunkcommunity eingesetzt [3]. Bhavsar [4] beschreibt, dass der Einsatz im IoT-Sektor Anforderungen wie z.B. begrenzte Energieversorgung an ein IoT-Gerät stellt. Weiterhin sagt er, dass Microcontroller für IoT-Projekte über möglichst wenig Speicher verfügen sollen, da der vorhandene Speicherplatz die Geschwindigkeit des Prozessors beeinflusst. Somit muss auch ein Routingprotokoll mit diesen Einschränkungen umgehen können.

Für diese Arbeit wurde eine Proof-of-Concept-Implementierung des B.A.T.M.A.N-adv Protokolls auf einem Microcontroller geschrieben. Um mit möglichst vielen Microcontrollern und Physical-Layer Technologien kompatibel zu sein, wurde die Proof of Concept Implementierung in Riot OS integriert. Dieses bietet eine Vielzahl an unterstützten Prozessoren, Peripheriegeräten und einen modularen Netzwerkstack [5] [6] [7]. Die Implementierung wurde im MIOT-Lab mit Metriken wie Latenz, Speicherverbrauch und Paketverlust untersucht, um die Leistung des Algorithmus auf einem Microcontrollersystem zu bewerten.

1.1 Ziel der Arbeit

Ziel der Arbeit ist es zu evaluieren, ob B.A.T.M.A.N-adv sich für den Einsatz im IoT-Sektor eignet. Um dies messbar zu machen, werden im Folgenden Hypothesen formuliert, die durch die durchgeführten Experimente angenommen oder verworden werden.

Hypothese 1.1 *B.A.T.M.A.N-adv eignet sich nicht zum Einsatz im IoT-Sektor.*

Hypothese 1.2 *B.A.T.M.A.N-adv eignet sich zum Einsatz im IoT-Sektor.*

Um die Hypothesen überprüfen zu können, wird der Algorithmus mit Metriken für Latenz, Speicherverbrauch und Paketverlust untersucht. Für diese Messungen lassen sich ebenfalls Hypothesen aufstellen. Für die Latenz, den Speicherverbrauch und den Paketverlust werden die folgenden Hypothesen untersucht:

Hypothese 2.1 *B.A.T.M.A.N-adv weist eine zu hohe Latenz für den IoT-Sektor auf.*

Hypothese 2.2 *B.A.T.M.A.N-adv weist keine zu hohe Latenz für den IoT-Sektor auf.*

Hypothese 3.1 *B.A.T.M.A.N-adv ist aufgrund des Speicherverbrauchs für den IoT-Sektor ungeeignet.*

Hypothese 3.2 *B.A.T.M.A.N-adv ist aufgrund des Speicherverbrauchs für den IoT-Sektor geeignet.*

Hypothese 4.1 *B.A.T.M.A.N-adv weist einen zu hohen Paketverlust für den IoT-Sektor auf.*

Hypothese 4.2 *B.A.T.M.A.N-adv weist keinen zu hohen Paketverlust für den IoT-Sektor auf.*

Um diese Hypothesen zu untersuchen, werden Messungen mit der Proof-of-Concept-Implementierung durchgeführt. Dafür wird die Implementierung sowohl auf ein Prototypensystem, das für die Entwicklung der Proof-of-Concept-Implementierung genutzt wurde, als auch auf das MIOT-Lab eingespielt und unter in der Arbeit definierten Gesichtspunkten getestet.

KAPITEL 2

B.A.T.M.A.N

Im folgenden Kapitel wird der B.A.T.M.A.N-adv Algorithmus detailliert beschrieben. Dabei wird die auf die ab B.A.T.M.A.N V verwendeten ELP-Nachrichten zur Bestimmung der Linkqualität, die OGMv2-Nachrichten zur Bestimmung von Routen und die verwendeten Unicast- und Broadcastheader und Algorithmen eingegangen.

2.1 Definitionen

Im Folgenden wird der B.A.T.M.A.N-adv V Routingalgorithmus beschrieben. Dafür sind Definitionen der verwendeten Begriffe nötig. Die folgenden Definitionen enstammen der Protokollbeschreibung des B.A.T.M.A.N-adv Algorithmus [8].

2.1.1 Namen

Name	Beschreibung
Knoten	Ein Gerät mit Netzwerkschnittstelle, welche B.A.T.M.A.N-adv benutzt.
Originator	Ein Knoten, der OGMv2 Nachrichten versendet.
Nachbarknoten	Ein Originator in Entfernung eines Hops.
Router	Ein Nachbarknoten, über den Pakete zu einem bestimmten Originator gesendet werden können.
Metrik	Von B.A.T.M.A.N-adv genutzter Indikator zur Ermittlung der Verbindungsqualität.

Tabelle 2.1: Namen im B.A.T.M.A.N-adv Algorithmus

2.1.2 Werte

Name	Wert
BATADV_SEQNO_MAX_AGE	64
BATADV_EXPECTED_SEQNO_RANGE	65536
OGM_MAX_ORIG_DIFF	5

Tabelle 2.2: Konstanten im B.A.T.M.A.N-adv Algorithmus

2.2 B.A.T.M.A.N

B.A.T.M.A.N ist ein Routingprotokoll für kabellose Mesh-Netzwerke. Laut einer Email der öffentlichen Freifunk Maillingliste [9] wurde B.A.T.M.A.N aktiv von der Freifunk-Community entwickelt. Dort sollte es vorherige genutzte Protokolle, wie z.B. Optimized Link State Routing (OLSR), ablösen. Wie aus der Email hervorgeht, zeichnet sich B.A.T.M.A.N durch eine geringe CPU und Speichernutzung aus.

Wie dem open-mesh Wiki [10] [11] [12] zu entnehmen ist, existiert der B.A.T.M.A.N Algorithmus in mehreren Versionen, welche mit römischen Ziffern benannt sind. Diese reichen von III bis V. Während Version III bis IV noch auf ISO/OSI Layer 3 agieren, nutzt Version V Layer 2. Ab Version V wird das Protokoll mit B.A.T.M.A.N-adv bezeichnet. Das *adv* steht hier für *advanced*. Das ursprüngliche B.A.T.M.A.N Protokoll arbeitet, indem es die Kernel Routingtabellen verändert. B.A.T.M.A.N-adv kapselt jeglichen Datenverkehr in Ethernetframes. Knoten haben keine Informationen über die Netzwerktopologie und sind somit nicht anfällig gegenüber Änderungen im Netzwerk. Dadurch, dass B.A.T.M.A.N-adv auf ISO/OSI Layer 2 umgesetzt ist, sind keine IP Adressen erforderlich und es lassen sich beliebige ISO/OSI Layer 3 Protokolle auf dem Mesh-Netzwerk umsetzen.

Die Protokollübersicht [13] nennt als Grundidee des Algorithmus, dass die Informationen über die besten Ende zu Ende Pfade über das Netzwerk verteilt abgespeichert werden. Diese werden durch sogenannte Originator Nachrichten (OGMv2) im Netzwerk verteilt. OGMv2s enthalten eine Metrik, die die Qualität des Pfades angibt. Jeder Knoten speichert sich eine Liste potentieller Router zu jedem Originator im Netzwerk ab. Hat ein Knoten die Auswahl zwischen mehreren Routern zu einem bestimmten Originator, wird er den auswählen, der über den besten Metrikwert verfügt. OGMv2 werden von jedem Knoten in bestimmten Intervallen als Broadcast versendet. Dadurch lernen andere Knoten über die Anwesenheit des versendenden Knotens. Nun überprüft jeder Knoten, der diese Nachricht empfangen hat, ob der Originator als bester Router angesehen werden kann. Ist dies der Fall, leitet er die empfangene OGMv2 weiter. Somit wird das gesamte Netzwerk mit OGMv2s geflutet. Durch das Fluten werden automatisch Routen benachteiligt, welche einen hohen Paketverlust aufweisen. Um sicherzugehen, dass das Netzwerk nicht mit unnötig vielen OGMv2s geflutet wird, leitet jeder Knoten nur die OGMv2s des Nachbarn weiter, den er als besten Hop ansieht. Außerdem enthalten OGMv2s Sequenznummern, die verhindern, dass veraltete OGMv2s weiter das Netzwerk fluten.

Der B.A.T.M.A.N-adv Algorithmus teilt sich in zwei Teile auf: Erkennung von Nachbarknoten und Finden von Routen. Die Erkennung von Nachbarknoten wird vom Echo Location

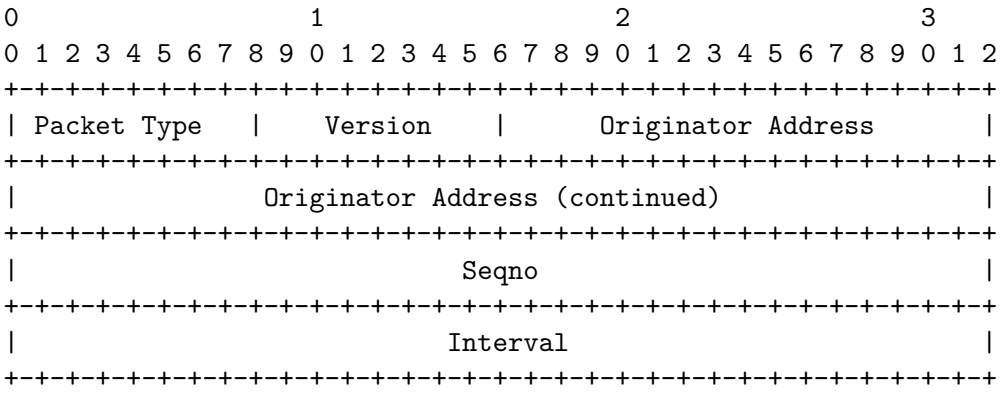


Abbildung 2.1: ELP Nachrichtenformat zum Erkennen von Nachbarinterfaces

Protocol (ELP) geregelt. Zur Festlegung von Routen werden die OGMv2 Nachrichten verwendet.

2.2.1 ELP

Im Folgenden wird der ELP-Algorithmus wie in der Dokumentation [14] und der aktuellen Linux-Kernelmodul Implementierung [15] beschrieben. Durch das ELP werden Netzwerkinterfaces von Nachbarknoten entdeckt. Für jedes Nachbarinterface wird der Wert der Metrik abgespeichert, der die Verbindungsqualität angibt. Zum Bestimmen dieses Metrikwerts werden vom Algorithmus periodisch ELP Pakete als Broadcast versendet.

In Abbildung 2.1 ist das Nachrichtenformat der ELP Nachrichten abgebildet, wie es in der aktuellen Linux-Kernelmodul Implementierung verwendet wird [16]. Der `Packet Type` gibt den Typ der Nachricht an. Dieser muss auf den Typ ELP-Nachricht gesetzt werden. Durch das Feld `Version` wird sichergestellt, dass der Originator der Nachricht über eine kompatible Version des B.A.T.M.A.N-adv Algorithmus verfügt. Die `Originator Address` ist die Adresse des Originators, der die ELP-Nachricht gesendet hat. Die `Seqno` ist eine Sequenznummer und stellt sicher, dass keine veralteten Nachrichten (mit geringerer `Seqno`) verarbeitet werden. Das Feld `Interval` enthält das Intervall in Millisekunden, in denen der Originator ELP-Nachrichten versendet.

Empfängt ein Knoten eine ELP-Nachricht, muss der Knoten feststellen, ob es sich um eine valide ELP-Nachricht handelt. Dazu wird Algorithmus 1 durchgeführt. Die `src_addr` und `dst_addr` stammen aus bestehenden Layer 2 Protokollen. Weiterhin wird die zuletzt empfangene Sequenznummer (*LastReceivedSeqno*) für jedes Nachbarinterface abgespeichert. Jeder Knoten verfügt über mindestens ein Netzwerkinterface. Der Knoten, welcher die ELP-Nachricht empfängt, speichert sich für jeden Nachbarknoten die Interfaces ab, von denen er die ELP Nachrichten empfangen hat. Hat die ELP-Nachricht diese Checks überstanden, können die Informationen über das Interface des Nachbarn aktualisiert werden. Dazu werden der Zeitstempel, zu dem die ELP-Nachricht empfangen wurde, das Intervall und die Sequenznummer der ELP-Nachricht für das entsprechende Interface abgespeichert. Außerdem

Algorithmus 1 Prelimary Checks ELP

function PRELIMARY($elp\_packet, src\_addr, dst\_addr$)
  if $elp\_packet \rightarrow Version \neq COMPAT\_VERSION$ **then** Drop packet
  else if $src\_addr = BROADCAST\_ADDRESS$ **then** Drop packet
  else if $dst\_addr = UNICAST\_ADDRESS$ **then** Drop packet
  else if $elp\_packet \rightarrow OriginatorAddress = OWN\_ADDRESS$ **then** Drop packet
  else if $elp\_packet \rightarrow Seqno < LastReceivedSeqno$ **then** Drop packet
  end if
end function

wird die Metrik für das Nachbarinterface aktualisiert. Als Metrik wird von B.A.T.M.A.N-adv der Durchsatz verwendet. Der Durchsatz wird in 100 kbit/s angegeben. Beim Empfang jeder ELP-Nachricht wird vom Interface, über das die Nachricht empfangen wurde, der Durchsatz zum Nachbarinterface abgefragt. Dieser spiegelt die Qualität der Verbindung wieder. Auf die Metrik wird ein gleitender Mittelwertfilter angewendet. Der gefilterte Wert wird für jedes Nachbarinterface abgespeichert, damit er von anderen Teilen des Algorithmus verwendet werden kann.

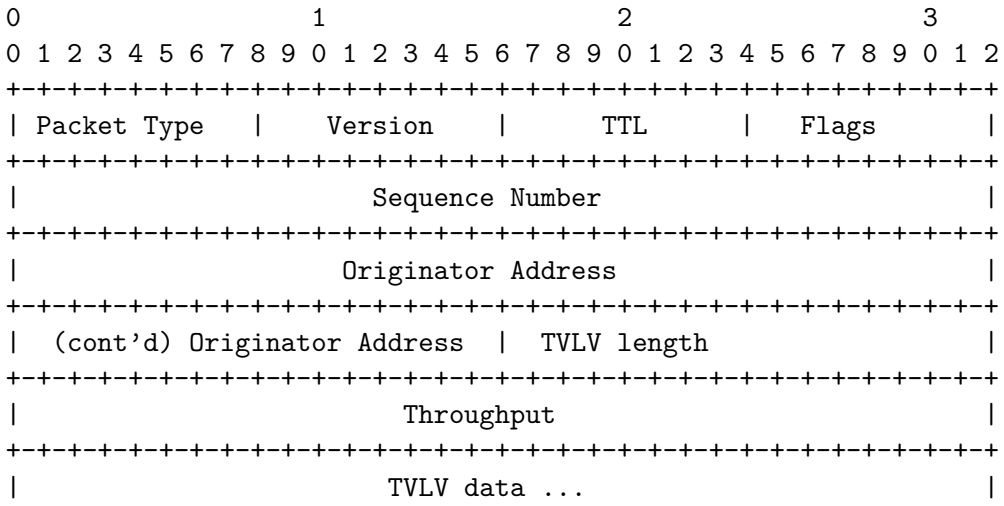


Abbildung 2.2: Originator Nachricht Version 2 zur Findung von Routen im Netzwerk

2.2.2 OGMv2

Das Finden von Routen durch das Mesh-Netzwerk wird im B.A.T.M.A.N-adv Algorithmus von den Originatornachrichten OGMv2 übernommen. Dies wird in der Dokumentation der OGMv2 Nachrichten [8] beschrieben. Die OGMv2 Nachrichten werden in einem bestimmten Intervall von jedem Knoten als Broadcast versendet und nach den im Folgenden beschriebenen Regeln weitergeleitet.

In Abbildung 2.2 ist das Format der Originatornachrichten dargestellt. Der `Paket Type` gibt die Art der Nachricht an. Dieser muss auf den Typ OGMv2-Nachricht gesetzt werden. Die `Version` stellt sicher, dass der Originator der Nachricht eine kompatible Protokollversion benutzt. Die `Time to live (TTL)` wird vom Originator auf einen fixen Wert gesetzt. Jeder Knoten, der das Paket weiterleitet, dekrementiert sie um eins. Erreicht die `TTL` des Paketes 0, so muss es verworfen werden. Das `Flags` Feld ist zwar noch im Header vorhanden, wird allerdings in dieser Version nicht mehr genutzt. Die `Sequence Number` wird während der Initialisierung auf einen beliebigen Wert gesetzt und nach jeder gesendeten OGMv2 Nachricht um 1 erhöht. Das Feld `Originator Address` enthält die Layer 2 Adresse des Originators der Nachricht. Die `TVLV length` gibt die Länge der `TVLV data` an, die der Nachricht folgt. Der `Throughput` ist die Metrik, nach der die Qualität einer Route gemessen wird. Dieser wird, wie in Sektion 2.2.1 beschrieben, über die ELP-Nachrichten ermittelt. Jeder Originator setzt dieses Feld initial auf `0xFFFFFFFF`.

Empfängt ein Knoten eine OGMv2-Nachricht, muss er überprüfen, ob die Nachricht das korrekte Format besitzt und ob sie korrekt versendet wurde. Dies wird durch Algorithmus 2 sichergestellt.

Wenn die Nachricht die vorangegangenen Checks überstanden hat, wird überprüft, ob es sich bei dem Nachbar, von dem die Nachricht empfangen wurde, um einen potentiellen

Algorithmus 2 Prelimary Checks OGMv2

function PRELIMARY($ogmv2\_packet, src\_addr, dst\_addr$)
if $ogmv2\_packet \rightarrow Version \neq COMPAT\_VERSION$ **then** Drop packet
else if $src\_addr = BROADCAST\_ADDRESS$ **then** Drop packet
else if $dst\_addr = UNICAST\_ADDRESS$ **then** Drop packet
else if $ogmv2\_packet \rightarrow OriginatorAddress = OWN\_ADDRESS$ **then** Drop packet
end if
end function

Router handelt. Diese Schritt nennt sich *Metric Update* und ist in Algorithmus 3 dargestellt.

Wurde die Nachricht im Schritt *Metric Update* nicht verworfen, wird der Nachbar, von dem die Nachricht empfangen wurde, als potentieller Router angesehen. Nun wird überprüft, ob die bestehenden Routen geändert werden müssen. Dieser Schritt nennt sich *Route Update* und ist in Algorithmus 4 dargestellt.

Wenn die OGMv2-Nachricht von dem Router empfangen wurde, der nun der ausgewählte Router ist, so wird die Nachricht weitergeleitet. Allerdings wird die OGMv2-Nachricht nicht weitergeleitet, wenn für diesen Originator bereits eine OGMv2-Nachricht mit der selben Sequenznummer weitergeleitet wurde. Wenn die OGMv2-Nachricht weitergeleitet wird, müssen flogende Felder verändert werden:

- Die `TTL` wird um 1 dekrementiert.
- Erreicht die `TTL` 0 nach dem Dekrementieren, wird die OGMv2-Nachricht nicht weitergeleitet.
- Der `Throughput` des Interfaces, über das die Nachricht weitergeleitet wird, wird übernommen.

2.2.3 Subheader

Es ist zu beachten, dass alle von B.A.T.M.A.N-adv genutzten Header über einen gemeinsamen Subheader verfügen. Dies bezieht sich sowohl auf die in Abbildung 2.1 und Abbildung 2.2 dargestellten ELP- und OGMv2-Header, sowie auf die im folgenden vorgestellten Unicast- und Broadcastheader.

Der Subheader ist in Abbildung 2.3 dargestellt. Er besteht aus 16 Bits und enthält den `Packet Type` und die `Version`. Bei jedem von B.A.T.M.A.N-adv verwendeten Header sind die ersten 16 Bit der Subheader. Anhand des `Packet Type` wird ermittelt um welche Art von Paket es sich handelt, z.B. Unicast oder OGMv2. Somit kann am ersten Byte bestimmt werden, wie mit dem Paket weiter verfahren wird. Das `Version` Feld wird vom Sender auf die verwendete B.A.T.M.A.N Version gesetzt. Dadurch wird sichergestellt, dass nur Pakete von kompatiblen B.A.T.M.A.N Versionen verarbeitet werden.

Algorithmus 3 Metric Update

```
function MetricUpdate(ogmv2_packet, originator, router, ifnterface_incoming)
    seqno := ogmv2_packet.Seqno
    last_seqno := originator.seqno
    sqeno_diff := seqno − last_seqno
    age_check := seqno_diff ≤ −BATADV_SEQNO_MAX_AGE
    range_check := seqno_diff ≥ BATADV_EXPECTED_SEQNO_RANGE
    protection_started := false

    if age_check || range_check then
        if Protection Window is active then
            Drop packet
            return
        end if
        if age_check & range_check & Protection Window not active then
            Activate Protection Window
            protection_started := true
        end if
    end if
    if seqno_diff < 0 & ¬protection_started then
        Drop packet
        return
    end if

    router.last_seen := time_now
    originator.last_seen := time_now
    router.seqno := ogmv2_packet.seqno
    originator.seqno := ogmv2_packet.seqno
    router.ttl := ogmv2_packet.ttl
    originator.ttl := ogmv2_packet.ttl

    througput := min(ogmv2_packet.througput, router.througput)

    if is_default_if(ifnterface_incoming) then
        througput := througput
    else if is_halduplex(ifnterface_incoming) & througput > 1MBit/s then
        througput := througput / 2
    else
        througput := throughput - (throughput * penalty)
    end if

    router.througput := throughput
end function
```

Algorithmus 4 Route Update

function RouteUpdate($originator, router, src\_addr$)
  $neigh\_orig := get\_orig(src\_addr)$ ▷ Only accept Packets from known Originators

  if $\neg neigh\_orig$ **then**
    return
  end if

  if $originator.router = router$ **then**
    return
  else if $originator.router = NULL$ **then**
    $originator.router := router$
    return
  else if $originator.router.througput < router.througput$ **then**
    $originator.router := router$
    return
  else if $router.seqno - originator.router.seqno \geq OGM\_MAX\_ORIG\_DIFF$ **then**
    $originator.router := router$
    return
  end if
end function

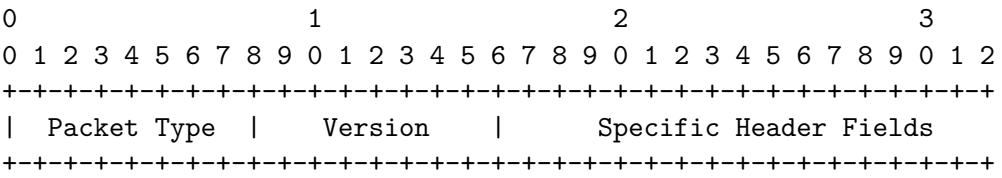


Abbildung 2.3: B.A.T.M.A.N-adv Subheader

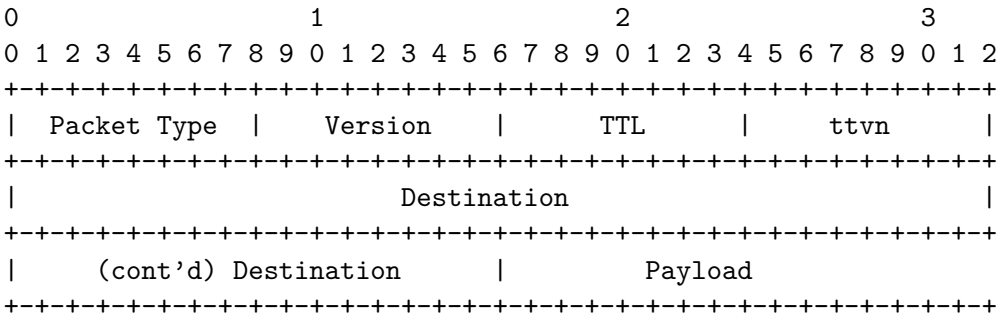


Abbildung 2.4: B.A.T.M.A.N-adv Unicast Header

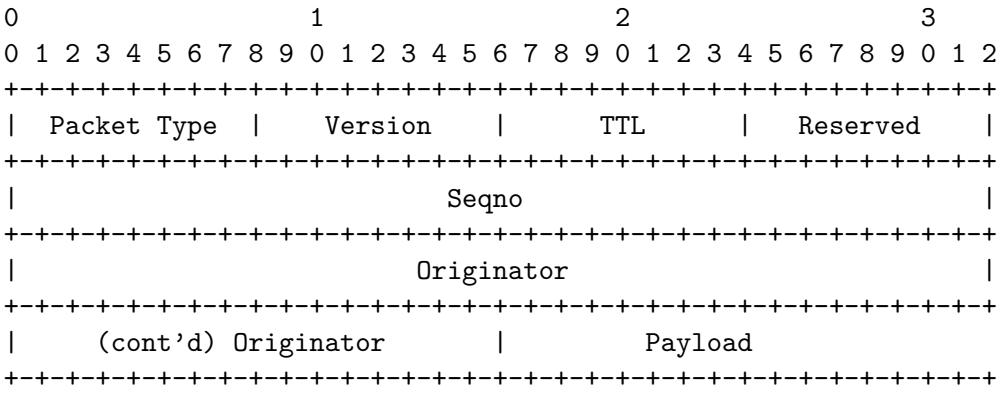


Abbildung 2.5: B.A.T.M.A.N-adv Broadcast Header

2.2.4 Routing

Neben dem Finden von Routen ist die andere Aufgabe von B.A.T.M.A.N-adv das Versenden und Weiterleiten von Paketen durch das Netzwerk. Um dies zu bewerkstelligen, verfügt B.A.T.M.A.N-adv über Unicast und Broadcast Header. Die Routingalgorithmen sind in vereinfachter Form dem B.A.T.M.A.N-adv Linux-Kernelmodul [17] entnommen. Die Beschreibung der Paketheader enstammt ebenfalls dem Linux-Kernelmodul [16].

In Abbildung 2.4 ist der B.A.T.M.A.N-adv Unicast Header dargestellt. Der `Packet Type` wird auf Unicast Header gesetzt und gibt den Typ des Pakets an. Die `Version` stellt sicher, dass eine kompatible Protokollversion benutzt wird. Die `TTL` wird auf einen fixen Wert gesetzt und von jedem Router, der das Paket weiterleitet, dekrementiert. Fällt sie auf 0, wird das Paket verworfen. Die `ttvn` gibt die Version der Translation Table des Originators des Pakets an. Die `Destination` gibt die Zieladresse des Pakets an. Dem Header folgt die `Payload` des Pakets.

In Abbildung 2.5 ist der B.A.T.M.A.N-adv Broadcast Header dargestellt. Der `Packet Type` wird auf Unicast Header gesetzt und gibt den Typ des Pakets an. Die `Version` stellt sicher, dass eine kompatible Protokollversion benutzt wird. Die `Seqno` wird zur Flusskontrolle von

Broadcastpaketen genutzt. Das Feld `Originator` enthält die Adresse des Originators der Nachricht. Dem Header folgt die `Payload` des Pakets.

Algorithmus 5 Unicast Routing

```
function RouteUnicast(unicast_packet)
    if unicast_packet.version ≠ BATADV_COMPAT_VERSION then
        Drop Packet
    else if unicast_packet.type ≠ Unicast then
        Drop Packet
    end if
    if is_my_mac(unicast_packet.destination) then
        to_layer3(unicast_packet)
    else
        orig := get_originator(unicast_packet.destination)
        router := orig.router
        if unicast_packet.ttl < 2 then
            Drop Packet
        end if
        unicast_packet.ttl := unicast_packet.ttl - 1
        if router ≠ NULL then
            send_to(unicast_packet, router)
        else
            Drop Packet
        end if
    end if
end function
```

Algorithmus 5 stellt das Routing von Unicast Paketen dar. Jedes empfangene Paket wird durch die RouteUnicast Funktion behandelt. Die Funktion *to_layer3* entfernt den B.A.T.M.A.N-adv Unicast Header vom Paket und leitet es an den nächsthöheren Layer im Stack weiter. Die Funktion *send_to* leitet das Paket zurück zu Layer 1 und dieser sendet es dann zum neuen Ziel.

Algorithmus 6 stellt das Routing von Broadcastpaketen dar. Jedes empfangene Paket wird durch die `RouteBroadcast` Funktion behandelt. Anders als beim Unicast Routing wird hier jedes Paket an Layer 3 weitergegeben und erneut als Broadcast versendet. Durch das erneute Versenden von Broadcast kann es zu *Rebroadcasting Echos* kommen. Diese entstehen, wenn ein Router seinen eigenen Broadcast empfängt, der durch einen benachbarten Router erneut gesendet wurde. Um diese *Echos* zu filtern, enthalten Broadcast Header eine Sequenznummer. Diese wird vom Originator des Broadcasts gesetzt. Jeder Router speichert sich für jeden Originator die letzte empfangene Broadcast Sequenznummer ab. Empfängt er nun einen Broadcast von einem Originator, der eine kleinere oder gleiche Sequenznummer hat als die zuletzt Empfangene, handelt es sich um ein *Echo* und das Paket wird verworfen.

Algorithmus 6 Broadcast Routing

```
function RouteBroadcast(bcast_packet)
    if bcast_packet.version ≠ BATADV_COMPAT_VERSION then
        Drop Packet
    else if bcast_packet.type ≠ Broadcast then
        Drop Packet
    end if

    orig := get_originator(bcast_packet.originator)
    if orig = NULL then
        Drop packet
    else if bcast_packet.seqno ≤ orig.last_bcast_seqno then
        Drop packet
    end if
    orig.last_bcast_seqno := bcast_packet.seqno
    to_layer3(bcast_packet)
    if bcast_packet.ttl > 1 then
        bcast_packet.ttl := bcast_packet.ttl - 1
        send_broadcast(bcast_packet)
    end if
end function
```

KAPITEL 3

Related Work

3.1 B.A.T.M.A.N LE

Shafaq Malik et al. [18] entwickelten eine B.A.T.M.A.N-adv Variante, welche besonders im Bereich IoT eingesetzt werden soll. Sie wählten B.A.T.M.A.N-adv als Protokoll, da es als Layer 2 Protokoll nicht auf IP Adressen basiert. Es ist also möglich, IoT Geräte in das Mesh-Netzwerk einzubinden, die über keine IP Adresse verfügen. Weiterhin lässt sich B.A.T.M.A.N-adv auf mehreren Funkschnittstellen umsetzen, z.B. WiFi oder Bluetooth. Als Netzwerkstack verwendeten sie einen erweiterten TCP/IP Stack, bei dem B.A.T.M.A.N-adv zwischen Data Link Layer und Network Layer in einem Adaption Layer implementiert wurde. Sie verwendeten die Version IV des B.A.T.M.A.N-adv Algorithmus.
Die Autoren sehen als Erweiterung des B.A.T.M.A.N-adv Algorithmus vor, dass ein Knoten für eine bestimmte Zeit in einen Schlafmodus wechselt, in dem er keine Nachrichten empfängt und sendet. Das soll vor allem im IoT-Sektor zum Einsatz kommen, wenn Knoten lange Zeit von einer Batterie betrieben werden oder generell ein geringer Energieverbrauch vorgesehen ist. Hierzu führen sie das Konzept von *Time Originator Messages* ein. Über diese informiert ein Knoten andere Knoten, dass er für eine bestimmte Zeit in den Schlafmodus übergeht. Weiterhin gibt der Knoten an, für welches Zeitfenster er im Schlafmodus bleibt. Wenn ein Knoten ein Paket empfängt, welches für einen Knoten bestimmt ist, der sich im Schlafmodus befindet, speichert dieser das Paket, bis der schlafende Knoten den Schlafmodus verlässt. Ist der Knoten wieder aktiv, wird das gespeicherte Paket weitergeleitet.

Zur Erhebung von Ergebnissen passten Shafaq Malik et al. eine existierende OMNET++ Implementation von B.A.T.M.A.N-adv IV dem *Time Originator Message* Algorithmus an. Sie verglichen B.A.T.M.A.N-adv und B.A.T.M.A.N-adv LE miteinander. Dabei erheben sie folgende Statistiken:

- Delivery Ratio: $DR = \frac{\text{Anzahl von der Quelle gesendeter Pakete}}{\text{Anzahl von der Senke empfangener Pakete}}$
- Message Overhead
- Batterielaufzeit
- Latenz

Nach der erhobenen Statistik hat B.A.T.M.A.N-adv LE eine höhere Delivery Ratio, einen geringeren Overhead und eine längere Batterielaufzeit als B.A.T.M.A.N-adv IV. Die Autoren befürchten allerdings einen Anstieg der Latenz aufgrund des Puffern von Paketen für schlafende Knoten.

3.2 B.A.T.M.A.N v.s. RPL

Saba Farooq Abbasi et al. [1] haben B.A.T.M.A.N-adv mit dem Routingprotokoll Routing Protocol for Low power and Lossy Networks (RPL) verglichen. Sie benutzen den B.A.T.M.A.N-adv IV Algorithmus.

RPL lässt sich wie B.A.T.M.A.N-adv auf Geräten mit verschiedener Physical-Layer-Hardware umsetzen. Anders als B.A.T.M.A.N-adv agiert RPL auf Layer 3 des ISO/OSI Modells und ist nur für Geräte spezifiziert, welche IPv6 Adressen verwenden. RPL baut über Distanzvektoren einen zielorientierten, gerichteten azyklischen Graphen (DODAG) auf. Dieser Graph kann nur einen Wurzelknoten haben (den Root-Knoten). Der DODAG wird aufgebaut, sobald der Root-Knoten aktiv wird. Jeder Knoten sendet in bestimmten Abständen ein *DODAG Information Object* (*DIO*). Mit dem *DIO* informiert der Knoten andere Knoten über die Anwesenheit des DODAGs. Ein neuer Knoten kann explizit nachfragen, ob ein DODAG existiert, indem er eine *DODAG Information Solicitation* (*DIS*) sendet. Empfängt ein Knoten eine *DIO* Nachricht, speichert er sich den Elternknoten ab, über den er die Nachricht empfangen hat. Ein Knoten kann mehrere Elternknoten haben. Außerdem sendet der Knoten als Antwort ein *DODAG Advertisement Object* (*DAO*) um den Root-Knoten über seine Anwesenheit zu informieren. RPL betreibt *Loop Avoidance* basierend auf dem Rang eines Knoten im Graphen.

Saba Farooq Abbasi et al. nutzten den *Network Simulator 3* (*NS-3*), um die beiden Protokolle zu vergleichen. Hierbei implementierten sie die *DAO* Funktion von RPL nicht, um die Komplexität zu reduzieren. Dabei betrachteten sie drei verschiedenen Metriken:

- Packet Delivery Ratio: $PDR = \frac{\text{Anzahl von der Quelle gesendeter Pakete}}{\text{Anzahl von der Senke empfangener Pakete}}$
- Routing Overhead: Alle Pakete, die zum Routing gesendet werden
- End-to-End Delay: Zeit, die ein Paket zum Erreichen seines Ziels benötigt

Die Autoren kamen zu dem Ergebnis, dass B.A.T.M.A.N-adv im Gegensatz zu RPL eine geringeres End-to-End Delay bei jeglicher Knotendichte aufweist. Dies schreiben sie der vergleichsweise langsamen *DODAG* Konstruktion von RPL zu. Weiterhin stellten sie fest, dass bei beiden Protokollen mit zunehmendem Abstand der Knoten das *Packet Delivery Ratio* schlechter wird. B.A.T.M.A.N-adv weist allerdings in jedem Fall eine bessere *Packet Delivery Ratio* auf. Die Ursache dafür sehen die Autoren darin, dass bei einer Routenänderung bei RPL der gesamte *DODAG* neu gebaut werden muss. Dies erzeugt eine große Menge an Paketen und dadurch werden weniger Datenpakete weitergeleitet. Der Routing Overhead ist bei B.A.T.M.A.N-adv über die Zeit größer als bei RPL. Dies liege daran, dass RPL nachdem es den *DODAG* aufgebaut hat, die Anzahl der Kontrollpakete vermindert. B.A.T.M.A.N-adv hingegen sendet während der gesamten Zeit periodisch weiter Originator-Nachrichten.

KAPITEL 4

Thesis-Beitrag

In Kapitel 1 wurde die Entwicklung des B.A.T.M.A.N Routingalgorithmus beschrieben. Im Detail wurde die Funktionsweise der neuen B.A.T.M.A.N-adv Version V beschrieben. Der Algorithmus wird schon in der Realität von z.B. Freifunk verwendet und existiert bereits als Linux-Kernelmodul [3] [19].

Ziel dieser Arbeit ist es zu evaluieren, ob sich der B.A.T.M.A.N-adv Algorithmus für den Einsatz im IoT-Sektor eignet. Dafür wurde eine proof-of-concept-Implementierung des Algorithmus vorgenommen. Bhavsar [4] beschreibt, dass die Arbeit auf eingebetteten Systemen Einschränkungen mit sich bringt, wie z.B. beschränkten Speicherplatz und beschränkten Arbeitsspeicher. Die für die Implementierung verwendete Hard- und Software wird im Folgenden vorgestellt. Weiterhin wird auf die vorgenommenen Anpassungen des B.A.T.M.A.N-adv Algorithmus eingegangen, welche nötig waren, um ihn auf der gegebenen Hardware zu implementieren.

4.1 Implementation

Im Folgenden wird die verwendete Hardware und Software vorgestellt. Weiterhin wird beschrieben, welche Anpassungen an Riot OS und dem B.A.T.M.A.N-adv Algorithmus vorgenommen wurden, um den Algorithmus auf der gegebenen Hardware implementieren zu können.

4.1.1 Verwendete Hard-und Software

Für die Proof of Concept Implementierung wird folgende Hardware verwendet:

- STM32 Nucleo-144 Board [20]:
  - Prozessor: Nucleo-F767ZI (32bit Cortex M7)
  - Flash: 2MB
  - RAM: 512KB
- CC1101 Tranceiver [21]:
  - Ultra low power.

  - 433 MHz Frequenzband

Verwendete Software:

- Riot OS [22]:
  - Betriebssystem für IoT Applikationen
  - Bestehender Netzwerkstack (gnrc)
  - Treibersupport für den CC1101

4.1.2 Algorithmusanpassungen für RIOT

Damit der B.A.T.M.A.N-adv Algorithmus in Riot OS implementiert werden kann, werden folgende Änderungen vorgenommen:

- Der CC1101 Tranceiver nutzt keine herkömmlichen Media Access Control (MAC)-Adressen. Die Hardwareadressen des CC1101 Tranceivers haben eine Länge von einem Byte. Daher werden diese mit Nullen auf sechs Byte *gepadded*, um dem Format der Originatornachricht gerecht zu werden.
- Da die verwendeten Microcontroller nur über ein Netzwerkinterface verfügen, entfallen die Überprüfungen auf Default Interface und Half Duplex im Algorithmus 3.

```
1  static void _bat_v_init_netif(struct bat_v_netif *netif, gnrc_netif_t *gnetif,
       char *dev)
2  {
3      netif->dev = dev;
4      netif->if_num = 0;
5      netif->bcast_seqno = 0;
6      netif->use_fixed_route = 0;
7      memset(netif->fix_route_addr, 0, BAT_V_ALEN);
8
9      for (int i = 0; (i < gnetif->l2addr_len) && (i < BAT_V_ALEN); i++)
10         netif->hw_addr[BAT_V_ALEN - 1 - i] = gnetif->l2addr[i];
11
12     INIT_LIST_HEAD_FIRST(netif->netif_neigh_list);
13
14     _bat_v_init_ogmv2(&(netif->out), netif->hw_addr);
15
16     _bat_v_init_elp(&(netif->elp), netif->hw_addr);
17 }
```

Quellcode 4.1: Padding der Layer 2 Adresse mit Nullen

In Quellcode 4.1 ist das Auffüllen der Layer 2 Adresse mit Nullen im Sourcecode dargestellt. Dieser Code würde Adressen mit einer Länge von über einem Byte in der *Byteorder* umkehren. Das ist beim CC1101 Tranceiver aufgrund der 1 Byte Adresslänge nicht der Fall. Die Funktionen `_bat_v_init_ogmv2` und `_bat_v_init_elp` initialisieren die OGMv2 und ELP Pakete, die von dem Originator versendet werden, mit der aufgefüllten Adresse.

Weiterhin ist es nicht möglich, den *Throughput* vom CC1101 Treiber abzufragen. Da der *Throughput* allerdings die Metrik ist, nach der die Routen vom B.A.T.M.A.N-adv Algorithmus ausgewählt werden, muss diese auf den CC1101 Treiber angepasst werden. Der CC1101

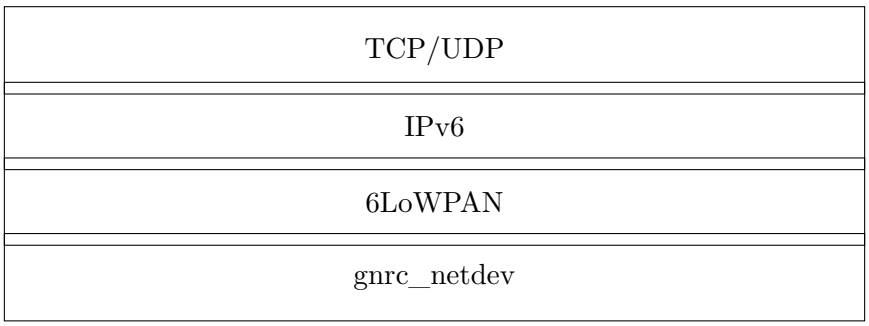


Abbildung 4.1: Standard *GNRC* Netwerkstack

Treiber bietet einen Wert, der die Signalstärke des empfangenen Signals beschreibt. Dieser Wert ist der Received Signal Strength Indicator (RSSI). Der RSSI ist ein Wert zwischen 0 und -138, wobei 0 eine perfekte Verbindung und -138 eine sehr schlechte Verbindung darstellt. Bei einem RSSI von -138 läuft der Paketverlust gegen 100%. Der RSSI wird an folgenden Stellen im Protokoll eingebracht:

- Der *Throughput* wird als Metrik durch RSSI ersetzt.
- Der *Throughput* in der OGMv2 wird durch den RSSI ersetzt.
- Der OGMv2 Algorithmus legt seine Routen anhand des RSSI fest.
- ELP-Nachrichten erzeugen ein Update des gleitenden Mittelwerts über den RSSI des Nachbarinterfaces.

Damit der RSSI mit dem B.A.T.M.A.N-adv Algorithmus verwendet werden kann, muss er transformiert werden. Der B.A.T.M.A.N-adv Algorithmus geht davon aus, dass höhere Werte in der gewählten Routing-Metrik besser sind als niedrigere Werte. Der B.A.T.M.A.N-adv kompatible RSSI ergibt sich aus $(-138 - rssi) * (-1)$. Im Folgenden wird der Metrikwert RSSI aufgrund des gleichnamigen OGMv2 Header Feldes weiterhin als *Throughput* bezeichnet.

In Quellcode 4.2 ist die Ersetzung des *Throughput* durch den RSSI im Sourcecode dargestellt. Der RSSI wird wie zuvor beschrieben skaliert, um mit B.A.T.M.A.N-adv kompatibel zu sein. Ber der Funktion `ewma_add` handelt es sich um den gleitenden Mittelwertfilter, der auf die Metrikwerte angewendet wird.

Da B.A.T.M.A.N-adv auf Layer 2 agiert, muss der *GNRC* Netzwerkstack von Riot OS angepasst werden. Der abstrahierte *GNRC* Netzwerkstacks in Abbildung 4.1 enstammt der Beschreibung des Stacks on Riot OS [7]. Wie der Dokumentation der *Network Device Driver API* [23] zu entnehmen ist, implementiert das *netdev* die Treiber für Netzwerkgeräte, wie z.B. IEEE802.15.4 oder den CC1101 Tranceiver und stellt diese einem beliebigen, darüberliegenden Netzwerkstack über ein einheitliches *Interface* zur Verfügung.

```
1  #define CC1101_RSSI_MIN 138
2
3  /**
4   *  updates link metric
5   *  @param: netif_neigh link information structure
6   *  @param: netif_hdr link information header
7   */
8  static void _elp_update_metric(struct bat_v_netif_neigh *netif_neigh,
       gnrc_netif_hdr_t *netif_hdr)
9  {
10     //char str1[ETH_STR_LEN];
11     int32_t rssi, scaled_rssi;
12
13     assert(netif_neigh != NULL);
14     assert(netif_hdr != NULL);
15
16     /*
17      * rssi scales much better than lqi. therefore it should be used for judging
            the
18      * quality of a link. in order to work correct batman v expects a metric
            where
19      * 0 means unusable and METRIC_MAX > 0 means perfect. the rssi ranges from
20      * -RSSI_MIN < 0 to 0 where 0 means perfect and -RSSI_MIN means very bad.
21      * The following code transforms the rssi reported by the used CC1101
            tranceiver
22      * into a batman v usable metric. IT ONLY WORKS SAFE WITH THE CC1101
            TRANCEIVER.
23      * IF OTHER TRANCEIVERS ARE USED THIS TRANSFORMATION MAY BE CHANGED.
24      */
25
26     rssi = (int32_t) netif_hdr->rssi;
27     scaled_rssi = (-CC1101_RSSI_MIN - rssi) * -1;
28
29     ewma_add((uint32_t) scaled_rssi, &(netif_neigh->metric.ewma_throughput));
30
31     //addr_to_string(str1, netif_neigh->addr);
32     //debug_log(LOG_LVL_ALL, "[BATMAN] _elp_update_metric: updating metric for %s
            -> %lu\n", str1,
33     //

           ewma_get(&(netif_neigh->metric.ewma_throughput)));
34 }
```

Quellcode 4.2: Ersetzung des Througputs durch den RSSI

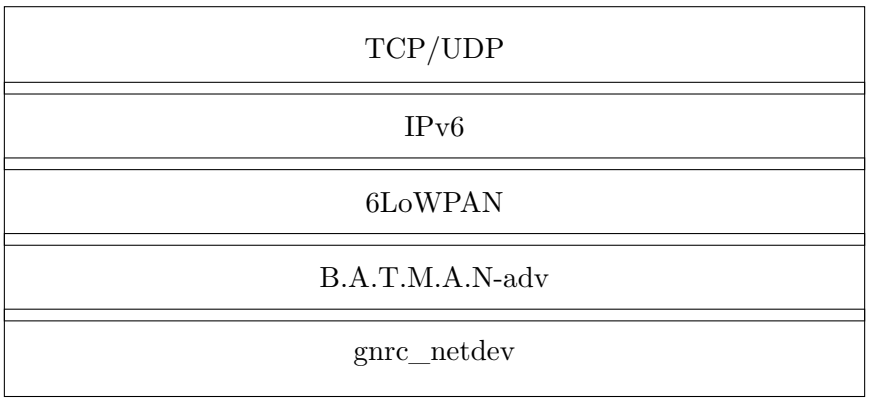


Abbildung 4.2: Angepasster *GNRC* Netwerkstack

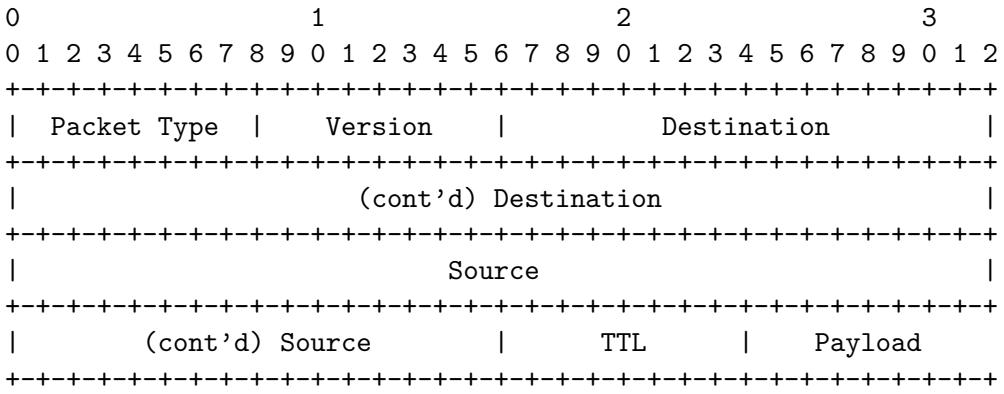


Abbildung 4.3: Unicast Header der Proof of Concept Implementierung

Um B.A.T.M.A.N-adv implementieren zu können, wurde zwischen dem *gnrc_netdev* und *6LoWPAN* ein weiterer Layer eingezogen. Dieser empfängt Packete vom *gnrc_netdev* und agiert somit als Layer 2. Dies ist in Abbildung 4.2 dargestellt.

Außerdem wurde der B.A.T.M.A.N-adv Unicast Header um ein Feld *src* erweitert, das die Adresse des Originators des Unicasts enthält. Diese wird genutzt um das *Packet Delivery Ratio* zu messen.

In Abbildung 4.3 ist der verwendete Unicast Header der Proof of Concept Implementierung abgebildet. Im Gegensatz zum originalen Unicast Header von B.A.T.M.A.N-adv, der in Abbildung 2.4 dargestellt ist, wurde dieser um ein `Source` Feld erweitert, das die Adresse des Knotens speichert, der das Paket gesendet hat. Weiterhin entfällt das `ttvn` Feld des Headers, da die Proof of Concept Implementierung keine Klienten im Netzwerk unterstützt und somit keine Translation Table führt.

4.2 Experimente

In Sektion 4.1 wurden die Anpassungen des B.A.T.M.A.N-adv Algorithmus für Riot OS beschrieben. Die folgende Sektion beschreibt den Aufbau der Experimente und der Testnetzwerke. Es wird darauf eingegangen, welche Metriken verwendet werden, um die Qualität des Algorithmus zu bewerten und wie diese gemessen werden.

4.2.1 Metriken

Für jegliche Netzwerkanwendungen gibt es Charakteristika, welche die Qualität des Netzwerks beschreiben. Für konventionelle Netzwerke, wie z.B. das Internet, umfassen diese Charakteristika unter anderem den Paketverlust, die Lazenz und die Bandbreite. Im IoT Sektor und auf kleinen Embedded Prozessoren verschieben sich diese Charakteristika. Für Sensornetzwerke, bei denen ein Sensor z.B. jede Stunde einen Temperaturwert sendet, ist die Bandbreite des Netzwerks nicht relevant, da nur sehr geringe Datenmengen übertragen werden. Weiterhin spielt die Latenz in diesem Beispiel auch keine Rolle, da nur stündlich Pakete gesendet werden. Bei anderen Anwendungzwecken, wie z.B. einem Netzerk, bei dem ein zeitkritischer Aktuator gesteuert wird, kann die Latenz allerdings von Bedeutung sein. Bei diesem Beispiel ist der Paketverlust allerdings von sehr hoher Relevanz. Wenn nur stündlich Werte übertragen werden, wäre es fatal, wenn das Netzwerk einen hohen Paketverlust aufweist, da sonst viele Werte verloren gehen. Dieses Problem könnte durch Neuübertragung der Werte gelöst werden. Dies ist allerdings bei batteriebetriebenen Geräten, die eine lange Zeit mit einer Batterie auskommen müssen, nicht erwünscht, da jeder erneute Sendevorgang Energie verbraucht.

Eine Charakteristik, welche gerade bei Embedded Prozessoren an Wichtigkeit gewinnt, ist der Speicherverbrauch. Gerade schwächere Prozessoren verfügen über wenig Arbeitsspeicher. Daher müssen Protokolle, die auf diesem Prozessor laufen sollen, möglichst speicherschonend arbeiten. Um nun B.A.T.M.A.N-adv zu bewerten, werden folgende Charakteristika betrachtet:

- Paketverlust
- Speicherverbrauch
- Latenz

Um die genannten Charakteristiken zu messen, sind Metriken von Nöten. Die verwendeten Metriken werden im Folgenden detailliert beschrieben.

Paketverlust

Der Paketverlust wird über das PDR abgebildet. Das PDR ergibt sich aus der Anzahl der gesendeten und empfangenen Pakete: $PDR = \frac{Anzahl empfangener Pakete}{Anzahl gesendeter Pakete}$. Um das PDR zu messen, wird ein Knoten als Quelle ausgewählt und ein weiterer als Senke. Daraufhin wird abgewartet, bis eine Route von der Quelle zur Senke aufgebaut wurde. Ist dies geschehen, werden eine festgelegte Anzahl an Paketen von der Quelle zur Senke gesendet. Da Riot OS über einen IP-Stack verfügt, werden dazu User Datagram Protocol (UDP) Pakete verwendet, da UDP keine verlorenen Pakete erneut überträgt. Der Quellknoten erzeugt für jedes

gesendete Paket eine serielle Ausgabe. Ebenso erzeugt die Senke für jedes empfangene Paket eine serielle Ausgabe. Ist die Messung abgeschlossen, wird die Anzahl der Ausgaben von Quelle und Senke gezählt und daraus das PDR berechnet. Bei dem PDR handelt es sich um eine *"höher ist besser"* Metrik. Der Maximalwert des PDR beträgt 1. Dies bedeutet, dass kein Paket verloren wurde. Der Minimalwert des PDR beträgt 0. Das bedeutet, dass jedes Paket verloren wurde. Bei den gesendeten UDP Paketen wird zusätzlich beim Empfang, die Time To Live (TTL) erfasst, die dem B.A.T.M.A.N-adv Unicast Header entnommen wird. Mit der ist es möglich, die Anzahl der Hops zu bestimmen, die ein Paket auf einer Route genommen hat.

Speicherverbrauch

Der Speicherverbrauch ist bei Embedded Prozessoren von besonderem Interesse. Er teilt sich in zwei Bereiche auf: 1. die Größe der Firmware, die im Flashspeicher des Controllers gespeichert wird und 2. die Nutzung des RAM. Bei dem Flashspeicher des Microcontrollers handelt es sich um ROM. Durch die Verwendung der GNU Compiler Collection (GCC) lässt sich die Größe des statisch zugewiesenen Speichers der Firmware durch das Tool *size* bestimmen. Das Tool *size* gibt die Speichernutzung in bestimmten Segmenten an. Diese Sektionen umfassen *text*, *data* und *bss*. Der Beschreibung des C Startups [24] ist zu entnehmen, dass diese Speichersegmente entweder im ROM, RAM oder in beiden liegen. Das *text* Segment liegt komplett im Flash (ROM). Das *data* Segment umfasst initialisierte, globale Variablen. Diese liegen zu Beginn der Ausführung im Flash, werden dann aber in den RAM kopiert. Sie liegen also in beiden Speicherbereichen. Das *bss* Segment umfasst nicht initialisierte, globale Variablen. Sie werden laut C Standard mit 0 initialisiert und liegen daher komplett im RAM. Laut Übersicht über die Standard Linker Sections [25] enthält das *text* Segment den ausführbaren Programmcode. Allerdings wird für Programmbestandteile, wie z.B. die Routingtabelle, dynamisch zugewiesener Speicher verwendet. Die von Riot OS verwendete *malloc* Implementation teilt keine *bookkeeping* Informationen mit dem Nutzer. Das bedeutet, dass es keine Möglichkeit gibt, den dynamisch angeforderten Speicher von Riot OS abzufragen. Daher wurde ein Wrapper um die *malloc* Implementation geschrieben, welcher unter bestimmten Labels den angeforderten und freigegebenen Speicher verfolgt. Die Label umfassen folgende Speichersegmente:

- Datenstrukturen
- Originator- und Nachbarknoten
- Nachbarinterfaces

Zum Auswerten des Speicherverbrauchs wird das Netzwerk für einen bestimmten Zeitraum laufen gelassen. Dies hat zur Folge, dass sich die Routen stabilisieren und der Speicherverbrauch gegen seinen Maximalwert konvergiert. Anschließend wird auf jedem Knoten der Speicherverbrauch ausgelesen. Der maximale Speicherverbrauch ist von Interesse. Für den Speicherverbrauch gilt, dass niedrigere Werte besser sind als höhere.

Zur Klassifikation von Microcontrollern im IoT-Sektor wird sich im Folgenden auf RFC 7228 [26] berufen. Dieses klassifiziert *Constrained-Nodes*, Knoten, welche sich unter

Klasse	RAM	Flash
C0	« 10 KiB	« 100 KiB
C1	~ 10 KiB	~ 100 KiB
C2	~ 50 KiB	~ 250 KiB

Tabelle 4.1: Klassen von *Constrained-Devices* KiB = 1024 bytes)

Anderem durch limitierten Speicher, limitierte Energieversorgung, limitierte Rechenleistung und andere physische Eigenschaften auszeichnen, in drei verschiedene Speicherklassen.

In Tabelle 4.1 sind die drei Speicherklassen dargestellt. Geräte der Klasse C0 verfügen nicht über genug Ressourcen, um selbst sicher über das Internet kommunizieren zu können. Klasse C1 umfasst Geräte, welche durch angepasste Protokollstacks selbst über das Internet kommunizieren können. Geräte der Klasse C2 können die meisten Protokolle, die auch auf Laptops und Servern verwendet werden, nutzen.

Latenz

Zum Messen der Latenz sind präzise und synchronisierte Uhren vonnöten. Da diese in dem Setup nicht zur Verfügung stehen, wird als Maß der Latenz die Round Trip Time (RTT) verwendet. Dieses Maß bietet zwar einen Überblick über die Netzwerklatenz, ist allerdings mit Vorsicht zu betrachten, da es gerade in kabellosen Meshnetzwerken zur Bildung von assymetrischen Routen kommen kann. Das bedeutet, wenn ein Knoten A in einem Netzwerk eine Route zu Knoten B hat und umgekehrt, dann sind beide Routen nicht zwangsläufig identisch. Deshalb ist es möglich, dass die Latenz von Knoten A zu Knoten B geringer ist, als die von Knoten B zu Knoten A. Zur Messung der RTT wird auf die Internet Control Message Protocol Version 6 (ICMPv6) Implementierung von Riot OS zurückgegriffen. Es wird eine bestimmte Anzahl an ICMPv6-Echos von einem Knoten zu einem Anderen gesendet. Die ICMPv6 Implementierung berechnet dann automatisch den Minimal- und Maximalwert und den Mittelwert der RTT. Bei dieser Metrik sind kleinere Werte besser als größere Werte.

4.2.2 Messaufbau

Die Messungen wurden in zwei verschiedenen Systemen durchgeführt. Das erste System stellten die für die Entwicklung der B.A.T.M.A.N-adv Implementierung genutzten Microcontroller dar. Es wird im Folgenden als Prototyp bezeichnet. Beim zweiten System handelt es sich um das MIOT-Lab. Im Folgenden wird beschrieben, welche Daten gemessen wurden und wie die beiden Systeme aufgebaut sind.

Messdaten

Um das PDR, die Latenz und den Speicherverbrauch zu messen, wird die serielle Schnittstelle der Microcontroller genutzt. Über diese wird von Riot OS eine Shellumgebung bereitgestellt. Sie wird von Riot OS als Standardausgabe verwendet. Jegliche Ausgaben der Firmware werden über die Schnittstelle ausgegeben. Weiterhin ermöglicht diese eine Kommunikation von einem Hostsystem mit dem Microcontroller über Shellkommandos. Die

Messung des Speicherverbrauchs wird mittels Shellkommando umgesetzt. Es erzeugt eine einfache Ausgabe des zugewiesenen Speichers unter den verschiedenen Labels. Der Paketverlust wird ebenfalls über serielle Ausgaben auf der Shell gemessen. Jeder Knoten erzeugt Ausgaben, an welche Knoten Pakete gesendet und von welchem Knoten Pakete empfangen werden. Außerdem werden protokollspezifische Ausgaben erzeugt, wenn Pakete verworfen werden. Gründe dafür können sein, dass:

- keine Route zum Ziel existiert,
- der Througput eines OGMv2 Pakets 0 ist,
- ein OGMv2 Paket von einem Knoten empfangen wurde, von dem zuvor keine ELP Pakete empfangen wurden oder
- das protection Window für einen Knoten aktiviert wurde.

Für die Messung der RTT wurde die Ausgabe des *ping6* Shellkommandos geparsed. Die 6 bedeutet, dass IPv6 Adressen verwendet werden.

Prototypen

Das Prototypensetup besteht aus vier Microcontrollern. Sie werden über eine auf den Controllern vorhandene USB-Buchse mit einem Raspberry PI verbunden. Dieser steuert die Messungen. Für die Kommunikation über die serielle Schnittstelle wird Python mit der Bibliothek pyserial verwendet. Zu Beginn einer Messreihe wird die B.A.T.M.A.N-adv Binary per Hand auf jedem Controller eingespielt. Danach werden die Messungen von einem Pythontool durchgeführt. Es wird eine Verbindung zu jedem Microcontroller geöffnet und die Messungen werden, wie in Sektion 4.2.1 beschrieben, durchgeführt. Um die Controller vor jeder Wiederholung in einen definierten Zustand zu bringen, werden sie zu Beginn jeder Messung über das Shellkommando *reboot* neu gestartet. Nach jedem Messdurchgang wird eine Logdatei mit den Ausgaben aller Microcontroller geschrieben. Aus dieser Datei können anschließend die Metrikwerte berechnet werden.

MIOT-Lab

Das Testbed besteht aus einer Menge von Testbedknoten. Auf diesen Testbedknoten befinden sich u.A die Nucleo Microcontroller, die mit einem auf dem Testbedknoten befindlichen Computer verbunden sind. Auf diesem Computer läuft eine Linux Distribution. Die Messungen auf dem Testbed werden von einem zentralisierten Managementsystem ausgeführt. Auf dem System kann angegeben werden, welche der einzelnen Knoten für eine Messung verwendet werden sollen. Die Messungen werden über eigens angefertigte Python und Bash Skripte gesteuert. Die für die Messungen benötigten Skripte und Dateien werden für jede Messung in das Managementsystem geladen. Es sorgt im Laufe der Messung dafür, dass diese Dateien auf allen Knoten verfügbar sind und die Skripte auf allen Knoten ausgeführt werden.

Pro Messung wurden 4 Dateien auf das Testbed kopiert: 1. die B.A.T.M.A.N-adv Firmware, 2. ein Skript zum Flashen der Firmware, 3. ein Skript zum Steuern der Messung, 4. ein

Skript zum Starten des Flashvorgangs und zum Starten des Messskripts.

Eine Messung auf dem Testbed läuft in mehreren Schritten ab. Zuerst werden alle Knoten in einen definierten Zustand gebracht. Um dies zu erreichen, wird zu Beginn jeder Messwiederholung die B.A.T.M.A.N-adv Implementierung mittels *openOCD* erneut auf die Microcontroller *geflasht*. Dies stellt sicher, dass alle Knoten über die selbe Protokollversion verfügen. Außerdem löst dies einen CPU Reset aus und bringt die Microcontroller in einen definierten Zustand.

Im zweiten Schritt wird ein Pythonskript ausgeführt, das mit der Bibliothek *pyserial* eine Verbindung zu der seriellen Schnittstelle des Microcontrollers aufbaut. Nun können die Messungen, wie in Sektion 4.2.1 beschrieben, durchgeführt werden. Die Testbedknoten erfasssen jegliche seriellen Ausgaben der Microcontroller. Nach Abschluss der Messung werden diese Ausgaben in Form eines Archivs gesammelt zur Verfügung gestellt. Aus diesen werden die Metrikwerte berechnet.

Messparameter

Der B.A.T.M.A.N-adv Algorithmus verfügt über einstellbare Parameter, die für die Performance des Algorithmus angepasst werden können. Bei den Parametern handelt es sich um:

- OGMv2 Intervall: Intervall, in dem OGMv2 Nachrichten von jedem Knoten gesendet werden.
- ELP Intervall: Intervall, in dem ELP Nachrichten von jedem Knoten gesendet werden.
- *HOP PENALTY*: Strafe, die auf die Performancemetrik bei jedem Hop gerechnet wird.

Diese Werte werden bei den verschiedenen Messreihen geändert und deren Auswirkung auf die Metriken untersucht. Dabei wird untersucht, inwiefern die *HOP PENALTY* den Aufbau von Routen beeinflusst und wie sich eine Veränderung der Intervalle für OGMv2 und ELP Nachrichten auf das PDR auswirkt.

Damit der B.A.T.M.A.N-adv Algorithmus funktioniert, muss, bevor eine OGMv2 Nachricht verarbeitet werden kann, mindestens ein ELP Paket von demselben Originator empfangen werden. Außerdem wird über die ELP Nachrichten die Qualität des Links bestimmt. Daher wurde für die Messreihen das ELP Intervall auf $\frac{OGMv2-Intervall}{2}$ gesetzt.

Wie in Sektion 4.2.1 beschrieben, werden zur Messung des PDR UDP Pakete versendet. Ein weiterer Parameter, der untersucht wird, ist die Größe der Payload. UDP Pakete besitzen eine bestimmte Größe, welche aus den vorangestellten Protokollheadern und dem UDP Header selbst resultiert. Die Größe der Payload des UDP Paket ist allerdings variabel. Daher wird evaluiert, wie sich die Größe der UDP Payload auf das PDR auswirkt.

KAPITEL 5

Thesis-Ergebnis

In Kapitel 4 wurden die durchgeführten Messungen, Metriken und Systeme vorgestellt. Dieses Kapitel befasst sich mit der Beschreibung, Darstellung und Interpretation der Messergebnisse. Die Ergebnisse teilen sich in zwei Bereiche auf: Die auf dem Prototypensystem umgesetzten Messungen und die auf dem Testbed umgesetzten Messungen. Obwohl der Großteil der Messungen auf dem Testbed umgesetzt wurde, wurden bestimmte Messreihen, die auf dem Prototypensystem durchgeführt wurden, nicht auf dem Testbed wiederholt, da die beobachteten Charakteristika hardwareabhängig sind und eine Vergrößerung des Netzwerks keinen Einfluss auf sie hat.

5.1 Prototyp

Zuerst werden die Ergebnisse der Messungen auf dem Prototypensystem vorgestellt. Im Anschluss werden die Messungen auf dem Testbed betrachtet.

5.1.1 Speicherverbrauch

In Sektion 4.2.1 wurde bereits beschrieben, wie der Speicherverbrauch gemessen wird. Um eine Entwicklung des Speicherverbrauchs darzustellen, wurde die Messung zuerst auf einem einzelnen Knoten durchgeführt. Danach wurde dem Netzwerk je ein weiterer Knoten hinzugefügt und die Messung wiederholt.

In Abbildung 5.1 ist die Speichernutzung der B.A.T.M.A.N-adv Implementierung zu sehen. Auf der X-Achse ist die Anzahl an Knoten eingetragen, die während der Messung aktiv waren. Auf der Y-Achse wird der dynamisch angeforderte Speicher in Bytes angegeben. Bei dem Speicher handelt es sich um dynamisch angeforderten Speicher, der über dem *malloc* Wrapper gemessen wird. In die Kategorie *Datenstrukturen* fallen alle Datenstrukturen, welche in der B.A.T.M.A.N-adv Implementierung verwendet werden. Dies beinhaltet eine Hash-Tabelle als Routingtabelle und eine Workqueue, die periodische Aufgaben übernimmt, wie das Senden von ELP und OGMv2 Nachrichten. *Originatoren* enthält alle Originatoreinträge in der Routingtabelle. *Nachbarknoten* enthält Nachbarknoteneinträge über welche ein Originator erreicht werden kann. *Nachbarinterfaces* enthält alle sichtbaren Hardwareinterfaces von Nachbarknoten.

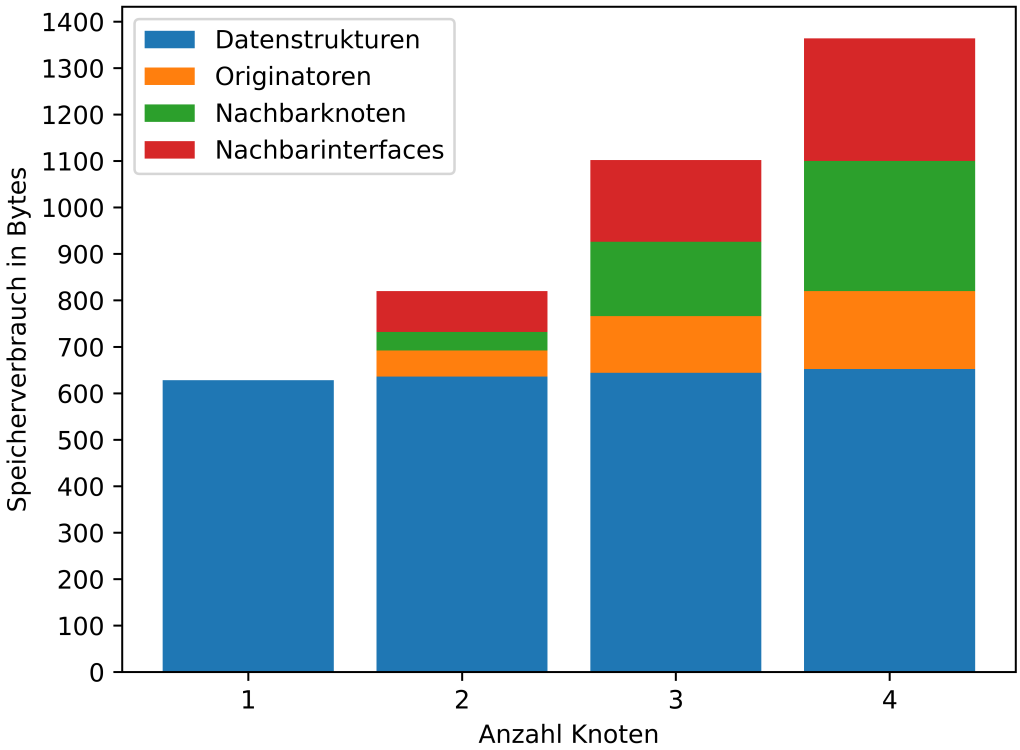


Abbildung 5.1: Speicherverbrauch bei wachsender Knotenzahl

Die Messungen des Speicherverbrauchs beziehen sich hier auf den RAM. Der Speicherverbrauch für Datenstrukturen bei einem Knoten beträgt 628 Bytes. Dies ist eine Größe, welche bei der Initialisierung des Algorithmus angefordert wird. Für jeden hinzugefügten Knoten steigt der Verbrauch um 8 Bytes. Somit verhält sich der Speicherverbrauch der Datenstrukturen linear zur Anzahl der Knoten. Auch in dem Speicherverbrauch der Originatorknoten ist eine Linearität zu erkennen. Für jeden hinzugefügten Knoten steigt der Speicherverbrauch für Originatoren um 56 Bytes an. Weiterhin weist der Speicherverbrauch für Nachbarinterfaces ein lineares Verhalten auf. Er steigt pro neuem Knoten um 88 Bytes an. Der Speicherverbrauch für Nachbarknoten weist quadratische Tendenzen auf. Er steigt von 40 Bytes bei zwei Knoten auf 160 Bytes bei drei Knoten und 280 Bytes bei vier Knoten an. Diese Entwicklung kommt daher, dass in einem Netzwerk, in dem alle Knoten jeden anderen Knoten direkt, also in 1 Hop Distanz, erreichen kann ein Paket über jeden der anderen Knoten zum Zielknoten gesendet werden. Wenn ein Netzwerkgraph konstruiert wird, in dem für jeden Knoten alle potentiellen Router eingezeichnet sind, handelt es sich dabei um einen vollständigen Graphen. In einem Netzwerk welches aus drei Knoten (A, B und C) besteht, könnte Knoten A Knoten C direkt erreichen oder über Knoten B. Für so ein Netzwek lässt sich eine obere Schranke für den Speicherverbrauch der Nachbarknoten durch die Formel

$$Speicherverbrauch = 40Bytes * (N-1)^2$$

beschreiben, wobei 40 Bytes der Speicherverbrauch für einen Nachbarknoten sind und N die Anzahl der aktiven Knoten im Netzwerk. Der Speicherverbrauch für Nachbarknoten weist also im Maximum einen quadratischen Anstieg auf. Daraus ergibt sich als obere Schranke für den dynamisch angeforderten Speicher

$$40Bytes * (N-1)^2 + N * (8Bytes + 56Bytes + 88Bytes) + 628Bytes$$

wobei N die Anzahl der Knoten ist. Zusammenfassend ist festzustellen, dass im Experiment eine maximaler Speicherverbrauch von 1364 Bytes gemessen wurde.

5.1.2 Packet Delivery Ratio

Auf dem Prototypensystem wurden die Messungen für das PDR, wie in Sektion 4.2.1 beschrieben, durchgeführt. Zum Messen des PDR auf dem Prototypensystem wurden die Knoten in einem festgelegten Abstand zueinander plaziert. Ein Knoten wurde als Quelle festgelegt und ein anderer Knoten als Senke. Um den Routingalgorithmus testen zu können, sollte der triviale Fall, dass Quelle und Senke in direkter Nachbarschaft stehen, also in ein Hop Distanz sind, ausgeschlossen werden. Dazu wurden Quelle und Senke so gewählt, dass sie zwischen allen Knoten die größte räumliche Distanz aufweisen.

Das Prototypensystem brachte folgende Einschränkungen bei den Messungen:

- Für die Messung wurde zwischen den Knoten eine Distanz von drei Metern festgesetzt. Außerdem wurden die Knoten räumlich getrennt. Dadurch ergab sich eine Beschränkung auf 3 Knoten.
- Die CC1101-Tranceiver waren mit den mitgelieferten Antennen ausgestattet, welche je nach Orientierung entweder sehr gute oder sehr schlechte Sende- und Empfangsei-

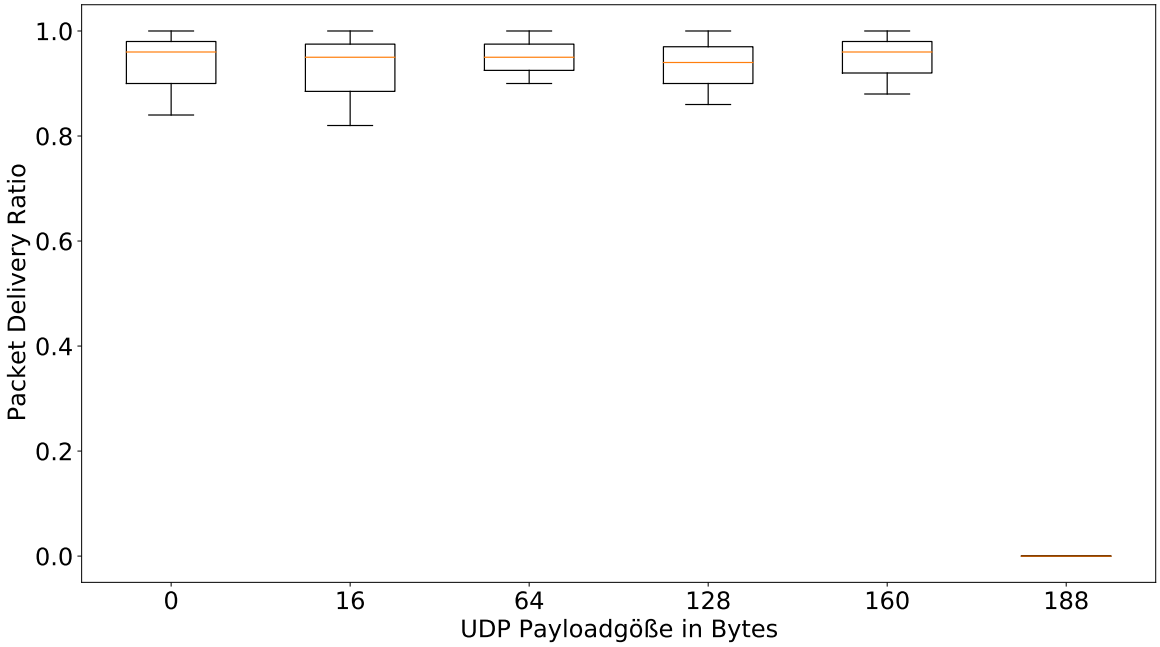


Abbildung 5.2: PDR bei wachsender Payloadgröße

genschaften aufweisen. Da diese nur über Jumper-Kabel verbunden waren, konnte die Orientierung nicht über mehrere Messungen gleich gehalten werden.

Zuerst wurde der Einfluss der Payloadgröße auf das PDR untersucht. Dazu wurde die Sendestärke des CC1101-Tranceivers konstant auf 0 dBm gehalten. Die *HOP_PENALTY* wurde auf $\frac{1}{255}$ gesetzt, damit sich wahrscheinlicher Routen über mehrere Hops bilden. Das OGMv2 Intervall wurde auf 1000 ms gesetzt und das ELP Intervall auf 500 ms. Die Payloadgröße wurde pro Messung vergrößert. Pro Messung wurden 50 UDP Pakete von der Quelle zur Senke gesendet. Jede Messung wurde 30 Mal wiederholt, um eine signifikante statistische Aussagekräftigkeit der Ergebnisse sicherzustellen. Zu beachten ist, dass die Payloadgröße nicht der Paketgröße entsprach. Zu der UDP Payload wurden noch 8 Bytes UDP-Header, 40 Bytes IPv6-Header und 15 Bytes B.A.T.M.A.N-adv-Unicast-Header hinzugefügt. Außerdem war der CC1101 Dokumentation von Riot OS [27] zu entnehmen, dass der CC1101 Treiber die Payload in ein eigenes Paketformat einbettete, welches dem Paket nochmal 13 Bytes hinzufügte.

In Abbildung 5.2 ist das PDR bei wachsender Payloadgröße zu sehen. Auf der X-Achse ist die Payloadgröße des versendeten UDP Pakets in Bytes zu sehen. Auf der Y-Achse ist das PDR dargestellt. Ein PDR Wert von 0 beschreibt, dass keines der versendeten Pakete sein Ziel erreicht hat. Ein PDR Wert von 1 beschreibt, dass jedes der versendeten Pakete sein Ziel erreicht hat.

Es ist deutlich zu erkennen, dass unterhalb von 188 Bytes eine Änderung in der Payloadgröße keine Änderung im PDR bewirkt. Im Mittel liegt das PDR bei 0,95. Dieser Wert ist sehr gut und für einen späteren Einsatz des Protokolls geeignet. Ferner gibt es keine Ausreißer im

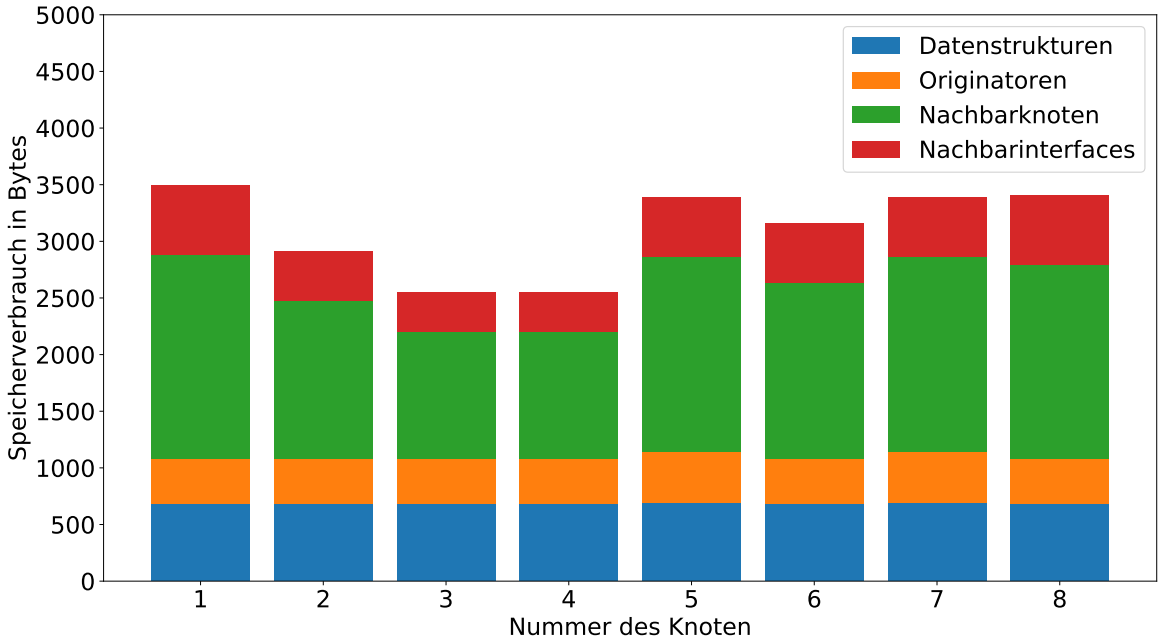


Abbildung 5.3: Speicherverbrauch der Testbedknoten

Interquartilabstand, was darauf schließen lässt, dass die Routen stabil sind. Ab 188 Bytes ist ein Sprung in den Daten erkennbar. Hier fällt das PDR auf 0. Dieses Ergebnis liegt an dem verwendeten Tranceiver. Dem Datenblatt [21] ist zu entnehmen, dass dieser eine Maximum Transmission Unit (MTU) von 256 Bytes aufweist. Addiert man zu den 188 Bytes Payloadgröße noch die 8 Bytes UDP-Header, 40 Bytes IPv6-Header, 15 Bytes B.A.T.M.A.N-adv-Unicast-Header und 13 Bytes CC1101-Header und Trailer, liegt die Paketgröße bei $188 + 8 + 40 + 15 + 13 = 264$ Bytes. Das ist bei Netzwerken, die keine großen Datenmengen versenden, nicht von großer Relevanz.

5.2 MIOT-Lab

Die folgenden Messungen wurden auf dem Testbed durchgeführt. Da die vorangegangenen Messungen auf dem Prototypensystem ergeben haben, dass die Payloadgröße keinen Einfluss auf das PDR hat, konnte bei den Messungen auf dem Testbed die Payloadgröße konstant gehalten werden. Es wurde eine Payloadgröße von 4 Byte festgelegt, um Sensorwerte eines potentiellen Sensors zu simulieren. Das Testbed bestand aus 8 Knoten, die für jegliche Messungen verwendet wurden. Die Sendestärke (*TX POWER*) des CC1101 Tranceivers wurde konstant auf 10 dBm gehalten, um eine möglichst große Reichweite der Knoten zu erlangen.

5.2.1 Speicherverbrauch

Die Messung des Speicherverbrauchs wurde, wie in Sektion 4.2.1 beschrieben, durchgeführt. In Abbildung 5.3 ist der Speicherverbrauch aller Knoten des Testbeds zu sehen, nachdem die Testbedknoten für einen Zeitraum von über 8 Stunden liefen. Dies stellt sicher, dass der

```
text    data     bss     dec     hex filename
94884     184   23328  118396   1ce7c batman_routing.elf
```

Abbildung 5.4: Ausgabe des *size* Tools für die B.A.T.M.A.N-adv Firmware

Speicherverbrauch der Knoten gegen seinen Maximalwert konvergiert. Auf der X-Achse ist die Nummer des Testbedknotens eingetragen. Auf der Y-Achse ist der Speicherverbrauch der Knoten in Bytes angegeben. Die Speichersegmente teilen sich in Originatoren, Nachbarknoten, Nachbarinterfaces und Datenstrukturen auf.

Die Menge des für Datenstrukturen angeforderten Speichers bleibt bei allen Knoten konstant. Ebenso bleibt der Speicher für Originatoren bei den meisten Knoten konstant. Dies entspricht den Erwartungen, da im Testbed nur 8 Originatoren existieren und jeder Knoten jeden Originator erreichen kann. Der höhere Speicherverbrauch von Knoten 5 und 7 erklärt sich dadurch, dass durch einen Übertragungsfehler eine inkorrekte Originatoradresse in einer OGMv2 Nachricht steht. Dadurch kann ein Originator angelegt werden, der nicht im Netzwerk existiert. Das schmälert allerdings nicht die Performance des Algorithmus, da Pakete mit einer unbekannten Zieladresse verworfen werden. Der Speicher für Nachbarknoten und Nachbarinterfaces variiert stärker. Das lässt sich dadurch erklären, dass Knoten, die isolierter stehen, weniger andere Knoten in Funkreichweite haben und somit weniger Nachbarinterfaces sehen und weniger Knoten zur Auswahl als nächsten Hop haben. Hier ist zu sehen, dass Knoten, die weniger Nachbarinterfaces sehen, auch weniger Speicher für diese anlegen und weniger Speicher für Nachbarknoten anfordern.

Weiterhin lässt sich feststellen, dass kein Knoten die mögliche obere Schranke des Speicherverbrauchs für Nachbarknoten aus Sektion 5.1.1 von $40 * (8 - 1)^2 = 1960$ Bytes erreicht. Der höchste für Nachbarknoten gemessene Speicherverbrauch beträgt 1800 Bytes und der niedrigste 1120 Bytes. Daraus lässt sich deutlich erkennen, dass nicht alle Knoten in 1 Hop Distanz stehen und somit einige Routen mindestens die Länge 2 aufweisen.

Zusätzlich wurde wie in Sektion 4.2.1 die Größe des statistische Speichers und die Firmwaregröße ausgelesen. Die Ausgabe des *size* Tools ist in Abbildung 5.4 zu sehen. In der ersten Zeile stehen die Sektionsnamen, der aufsummierte Speicher in Bytes als Dezimal- und Hexadezimalzahl und der Dateiname. Die zweite Zeile enthält die Größe des genutzten Speichers in Bytes und den Dateinamen. Um den belegten Speicherplatz des Flashspeichers zu erlangen, müssen das *text* und *data* Segment aufsummiert werden. Daraus ergibt sich, dass die Firmware 95068 Bytes (95,1 KB) Flashspeicher belegt. Für den statisch belegten RAM muss das *data* Segment mit dem *bss* Segment addiert werden. Daraus ergibt sich eine statische RAM Nutzung der Firmware von 23512 Bytes (23,512 KB).

Zusammenfassend lässt sich feststellen, dass bei jedem Knoten 95,1 KB Flashspeicher belegt werden. Die Implementierung enthält allerdings auch Datenstrukturen und Variablen, die zum Messen der Metriken angelegt wurden und in einer realen Anwendung aus der Firmware entfernt werden. Außerdem werden auf jedem Knoten statisch 23,512 KB

RAM belegt. Wird der dynamisch angeforderte Speicher hinzugezogen, werden maximal 25312 Bytes (25,312 KB) RAM von der Firmware belegt. Der ROM und RAM für den Microcontroller muss in einer realen Anwendung etwas größer als die gemessenen Werte gewählt werden, um ausreichend Platz auf dem Stack zu haben und um eine eigene Firmware auf B.A.T.M.A.N-adv aufbauen zu können.

Anhand dieser Zahlen kann einen Einordnung in die in Sektion 4.2.1 vorgestellten Speicherklassen von RFC 7228 getroffen werden. Der verwendete Nucleo F767ZI kann aufgrund seiner 2MB Flashspeicher und 512KB RAM nicht in eine der Klassen eingeordnet werden und zählt somit nicht als *Constrained-Device*. Allerdings liegt die Implementierung mit einer Speichernutzung von 95,1 KB Flashspeicher und 25,312 KB RAM in Klasse C2 und lässt sich somit auch auf *Constrained-Devices* ausführen.

5.2.2 Hop Penalty

Für die Messung des PDR und der Latenz ist es wichtig, einen guten Wert für die *HOP PENALTY* eingestellt zu haben. Der von B.A.T.M.A.N-adv vorgeschlagene Standardwert [8] beträgt $\frac{15}{255}$. Um die Auswirkung der *HOP PENALTY* auf das Netzwerk zu untersuchen wurden wurde die *HOP PENALTY* auf einen festen Wert gesetzt. Dann wurden 100 UDP Pakete an einen Knoten des Netzwerks gesendet. Dieser wurde so gewählt, dass der Knoten immer direkt von der Quelle aus erreichbar war, da mit zunehmender *HOP PENALTY* nicht mehr sichergestellt werden kann, dass alle Knoten des Netzwerks erreichbar sind. Während dieser Zeit wurden die Anzahl der Pakete gezählt, die vom B.A.T.M.A.N-adv Algorithmus verworfen wurden. Der Algorithmus kann aus vier Gründen Pakete verwerfen:

1. Wenn der *Throughput* eines OGMv2 Pakets 0 beträgt,
2. Wenn von einem Nachbarinterfaces ein OGMv2 Paket empfangen wurde, ohne dass zuvor ein ELP Paket von diesem Interface empfangen wurde,
3. Wenn für einen Originator das *Protection Window* aktiviert wurde,
4. Wenn keine Route zu einem Originator besteht.

Falsch formatierte Pakete wurden dabei nicht mitgezählt.

Für die Messungen wurden die Intervalle für OGMv2 und ELP Nachrichten mit 1000 ms und 500 ms konstant gehalten.

In Abbildung 5.5 sind die OGMv2 Pakete dargestellt, die verworfen wurden, da ihr *Throughput* 0 war. Auf der X-Achse ist die eingestellte *Hop Penalty* angezeigt. Der Wert, um den der *Throughput* verringert wird, ergibt sich aus $\frac{X}{255}$. Wenn auf der X Achse ein Wert von 15 vermerkt ist, wird der *Throughput* um $Throughput * \frac{15}{255}$ verringert. Auf der Y-Achse ist die summierte Anzahl der verworfenen Pakete aller Knoten dargestellt.

Es ist zu erkennen, dass bei *Hop Penalties* zwischen 0 und 108 und der Testbedgröße von 8 Knoten keine Pakete aufgrund der Routingmetrik verworfen werden. Für *Hop Penalties* ab 144 steigt die Anzahl verworfener Pakete beinahe exponentiell an. Diese Werte eignen sich

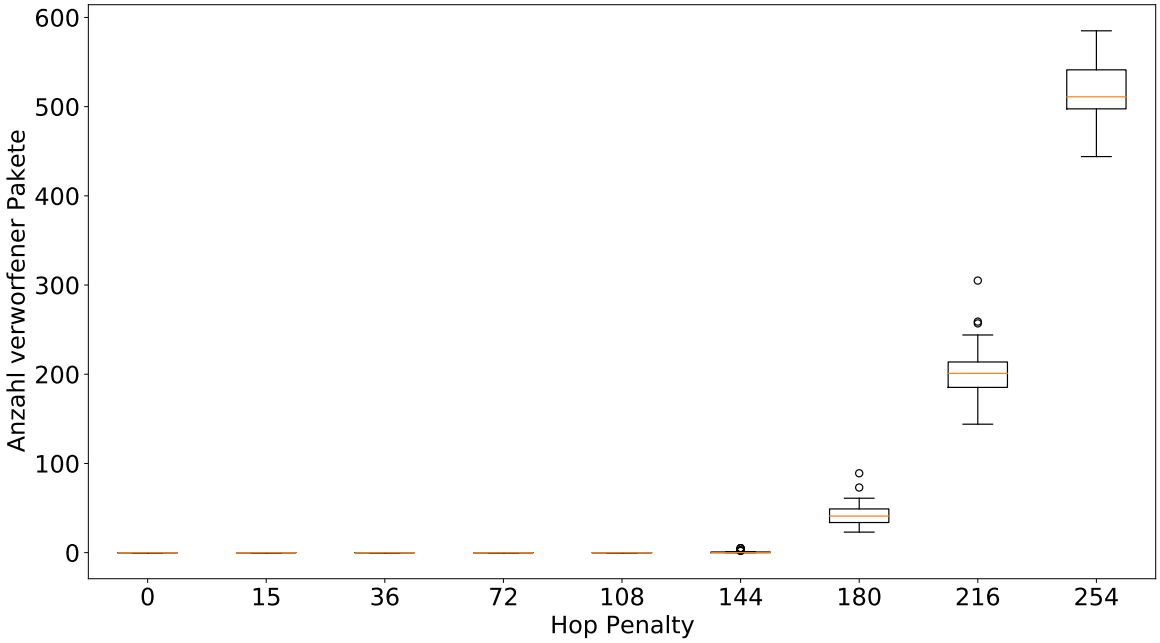


Abbildung 5.5: Anzahl verworfener Pakete aufgrund des Throughputs

also nicht für einen produktiven Einsatz im Testbed. Für die weiteren Messungen wurde der vom B.A.T.M.A.N-adv vorgeschlagene Standardwert von $\frac{15}{255}$ angewandt.

In Abbildung 5.6 sind die OGMv2 Pakete dargestellt, die verworfen wurden, da zuvor kein ELP Paket vom selben Interface empfangen wurde. Auf der X-Achse ist die eingestellte *Hop Penalty* angezeigt. Auf der Y-Achse ist die summierte Anzahl der verworfenen Pakete aller Knoten dargestellt. Hier ist zu erkennen, dass die *Hop Penalty* keinen erkennbaren Einfluss auf die verworfen Pakete aufweist.

Während der Messung wurden keine Pakete aufgrund des aktivierten *Protection Windows* oder aufgrund fehlender Route zum Ziel verworfen. Daraus lässt sich ableiten, dass kein Knoten während einer Messung neu gestartet wurde, sei es durch einen manuellen Reset oder ein fehlerhafter Softwarezustand.

5.2.3 Paketverlust

Die Messung des Paketverlusts wurde, wie in Sektion 4.2.1 und Sektion 4.2.2 beschrieben, durchgeführt. Das Testbed bestand zum Zeitpunkt der Messung aus acht Knoten. Es wurde einer der Knoten als Quelle ausgewählt. Ein anderer Knoten wurde als Senke genommen. Der Quellknoten sendete 100 UDP Pakete zur Senke. Die Messung wurde so oft wiederholt, dass der Quellknoten zu jedem anderen Knoten des Testbeds Pakete gesendet hat. Dies ergab 7 Messungen pro gewählter Einstellung.

Die Messung wurde wie in Sektion 4.2.2 durch ein Python Script gesteuert, welches auf jedem Testbedknoten ausgeführt wurde. Zu Beginn jeder Messung wurde die Firmware neu

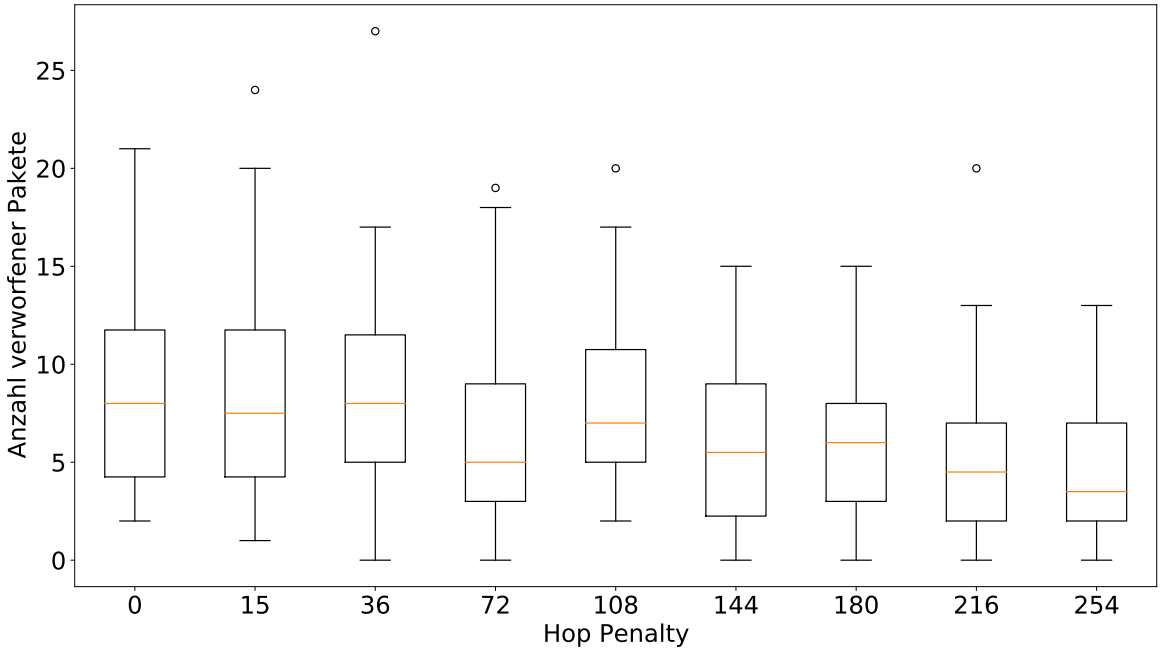


Abbildung 5.6: Anzahl verworfener Pakete aufgrund fehlendem ELP Paket

auf die Nucleos aufgespielt. Das versetzt die Knoten in einen definierten Anfangszustand. Mit dem Senden der Pakete wurde begonnen, sobald der Quellknoten eine Route zu Senkenknoten aufgebaut hatte. Dies wurde über die serielle Schnittstelle des Microcontrollers sichergestellt. Jeder Knoten erzeugt eine Ausgabe, wenn der Knoten eine neue Route findet oder sich eine Route ändert. In den Messreihen sendete Knoten 2 (Layer 2-Adresse: 0x05) Pakete an alle anderen Knoten. Die *Hop Penalty* wurde auf den Standardwert von $\frac{15}{255}$ gesetzt. Für die Messreihen wurde das Intervall der OGMv2 und ELP Nachrichten zuerst auf einen festgelegten Minimalwert gesetzt und schrittweise gesteigert.

In Abbildung 5.7 sind die Routen zu sehen, die sich mit $HOP\ PENALTY = \frac{15}{255}$ aufbauen. Die Zahlen in den Knoten stellen die Layer 2-Adressen der Knoten dar. Die Kanten stellen die Routen dar, die jeder Knoten als besten nächsten Hop zu einem anderen Originator sieht. Zu beachten ist, dass hier auch assymetrische Routen aufgebaut wurden: Knoten 4a baut eine direkte Route zu Knoten 7a auf, 7a aber nicht zu 4a. Dies ist auch bei Knoten 68, 05 und 40 zu beobachten.

In Abbildung 5.8a ist das PDR von Knoten 0x05 zu jedem anderen Knoten des Testbeds dargestellt. Das OGMv2 Intervall ist bei dieser Messung auf 1000 ms eingestellt. Daraus ergibt sich, wie in Sektion 4.2.2 beschrieben, ein ELP Intervall von 500 ms. Auf der X-Achse ist die Layer 2-Adresse des Knotens, zu dem die UDP Pakete gesendet wurden, abgebildet. Auf der Y-Achse ist das PDR abgebildet.

Das maximale PDR beträgt 0,92 bei Knoten 0x7a. Das minimale PDR weist Knoten 0x77 mit 0,5 auf. Im Median bewegt sich das PDR zwischen 0,58 und 0,83. Hier ist ein

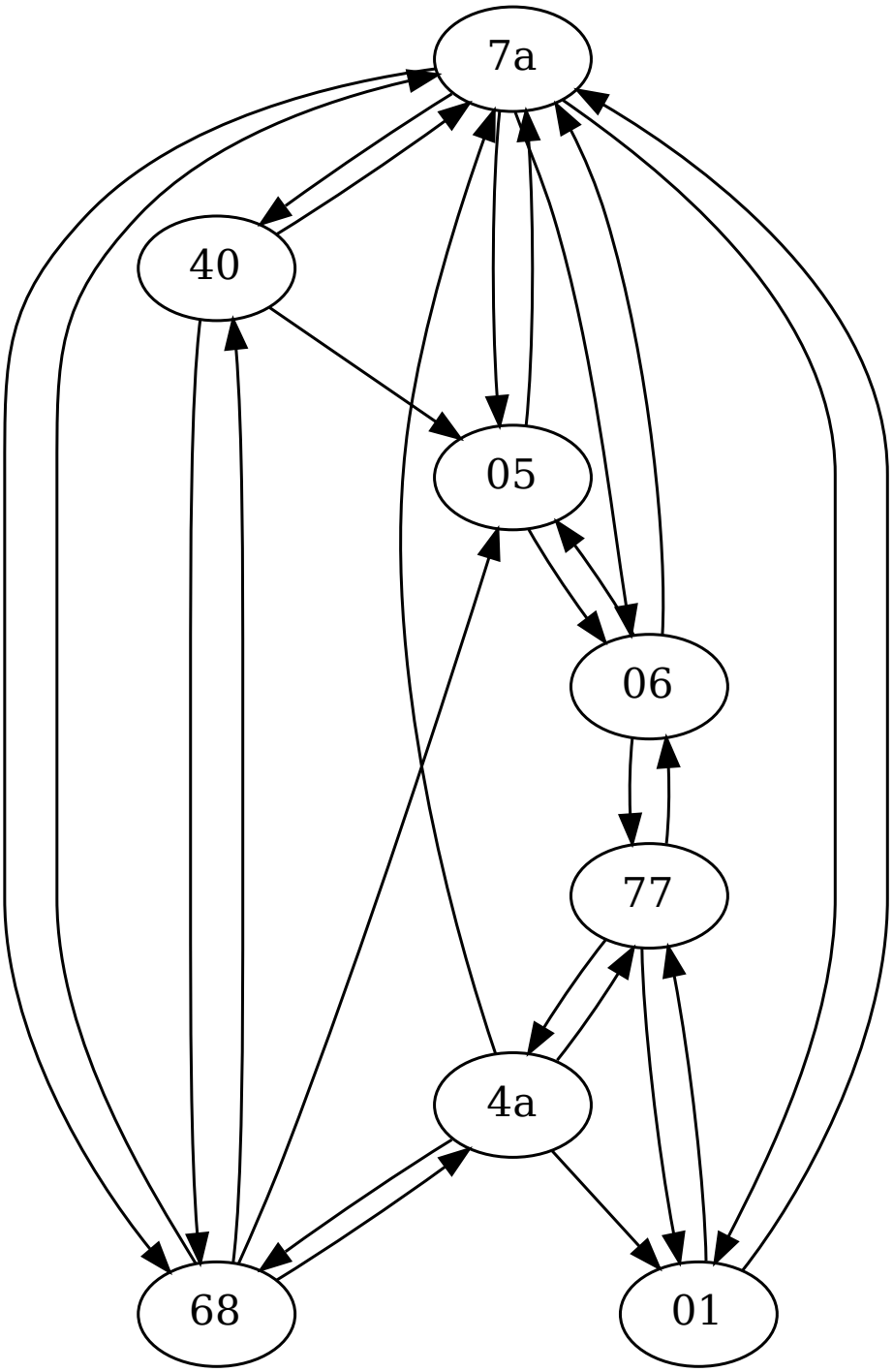
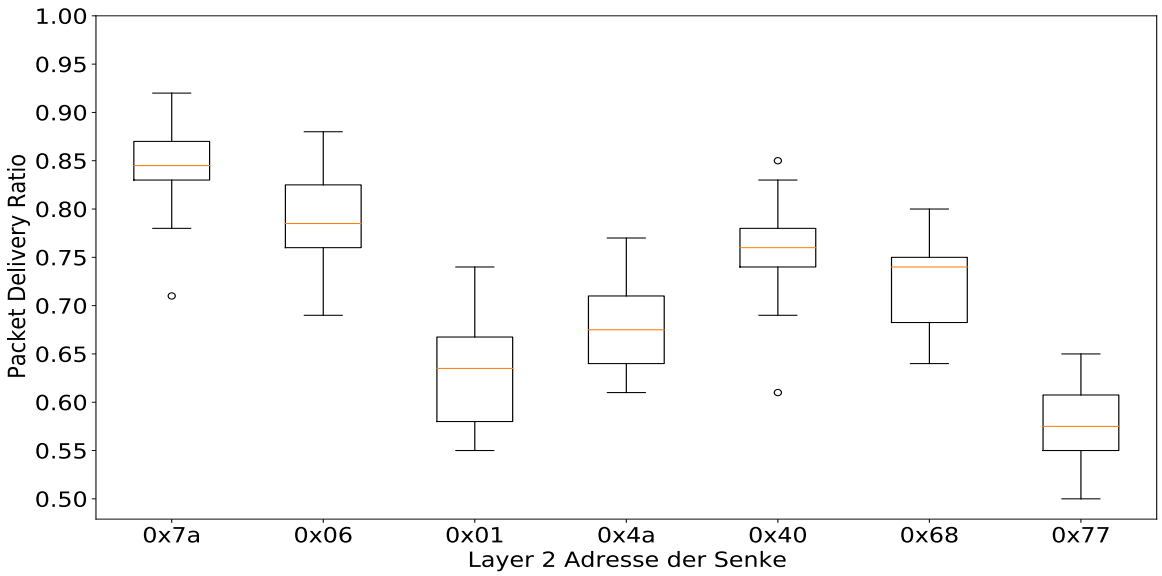
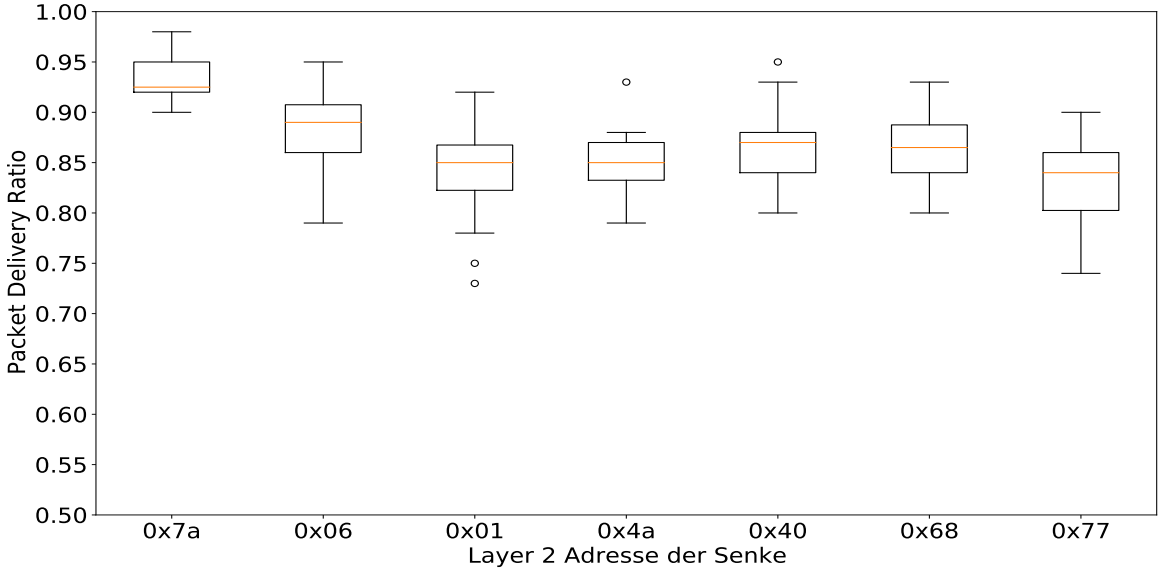


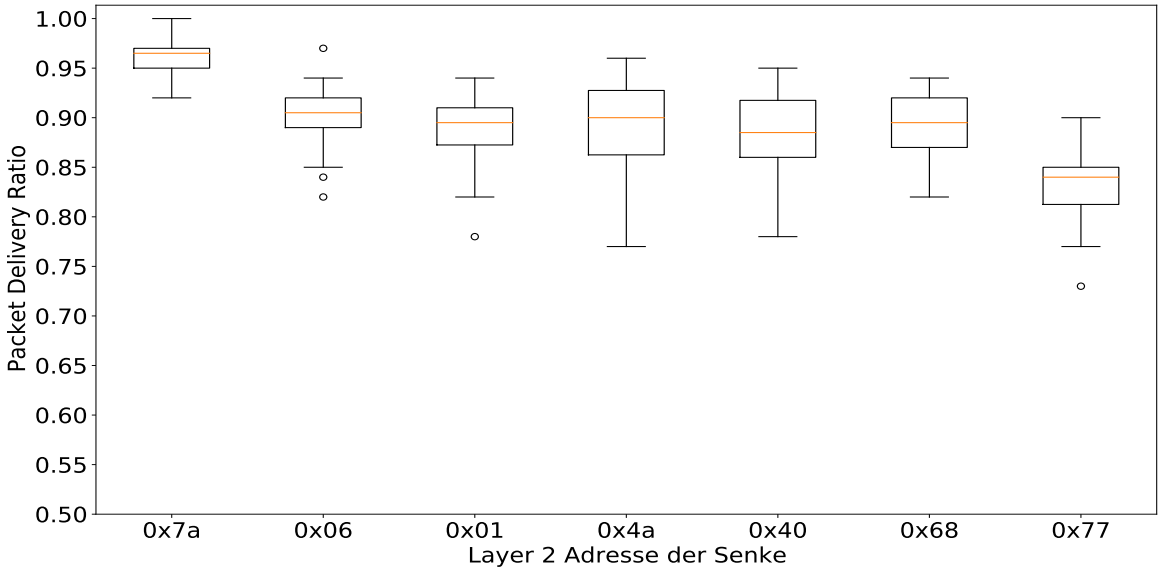
Abbildung 5.7: Beispielrouten aus der PDR Messung



a) PDR zur Senke bei einem OGMv2/ELP Intervall von 1000 ms/500 ms



b) PDR zur Senke bei einem OGMv2/ELP Intervall von 3000 ms/1500 ms



c) PDR zur Senke bei einem OGMv2/ELP Intervall von 5000 ms/2500 ms

Abbildung 5.8: PDR zur Senke bei variierenden OGMv2/ELP Intervallen

Layer 2 Adresse der Senke	Maximale TTL	Minimale TTL
0x7a	50	49
0x06	50	49
0x01	49	47
0x4a	49	47
0x40	50	49
0x68	50	48
0x77	50	48

(a) TTL empfangener Pakete mit OGMv2/ELP Intervall von 1000 ms/500 ms

Layer 2 Adresse der Senke	Maximale TTL	Minimale TTL
0x7a	50	50
0x06	50	47
0x01	49	47
0x4a	49	47
0x40	50	46
0x68	50	48
0x77	49	48

(b) TTL empfangener Pakete mit OGMv2/ELP Intervall von 3000 ms/1500 ms

Layer 2 Adresse der Senke	Maximale TTL	Minimale TTL
0x7a	50	50
0x06	50	50
0x01	49	47
0x4a	49	47
0x40	50	48
0x68	50	48
0x77	49	47

(c) TTL empfangener Pakete mit OGMv2/ELP Intervall von 5000 ms/2500 ms

Tabelle 5.1: TTL empfangener Pakete mit variierenden OGMv2/ELP Intervallen

deutlicher Unterschied im PDR zu erkennen. Knoten 0x7a weist das beste PDR auf. Dies entspricht den Erwartungen, da, wie aus Abbildung 5.7 ersichtlich, Knoten 0x7a direkt von Knoten 0x05 aus erreichbar ist. Außerdem weisen diese Testbedknoten die geringste räumliche Distanz auf. Knoten 0x06, der auch direkt über Knoten 0x05 erreichbar ist, weist ein geringeres PDR als Knoten 0x7a auf. Dies lässt sich über die erhöhte räumliche Distanz der Knoten erklären.

Die Routen zu den einzelnen Knoten weisen verschiedene Längen auf. Diese wurden in der Messung über die TTL der empfangenen Messpakete erfasst. Die Routenlänge variierte im Laufe der Messungen. In Tabelle 5.1a ist die minimale und maximale TTL der empfangenen UDP Pakete dargestellt. Diese wurden aus dem B.A.T.M.A.N-adv Unicast Header entnommen. Die B.A.T.M.A.N-adv Implementierung setzt die TTL eines neuen Pakets auf 50. Somit wurden Pakete, welche mit einer TTL von 50 bei der Senke ankommen, direkt von der Quelle empfangen. Jeder Knoten, der ein Paket in Richtung

Ziel weiterleitet, dekrementiert die TTL um 1. Eine TTL von 47 bedeutet also, dass ein Paket von drei Knoten zur Senke weitergeleitet wurde. Die Route hat also eine Länge von 4. Die kürzeste Route, die bei allen Knoten außer 0x01 und 0x40 zu beobachten ist, hat die Länge 1. Die längste Route, die bei Knoten 0x01 und 0x4a zu sehen ist, hat die Länge 4.

Vergleicht man die Werte aus Tabelle 5.1a mit den Werten aus Abbildung 5.8a, erkennt man, dass auf kürzeren Routen, die eine geringere Differenz in der Routenlänge zeigen, wie z.B. 0x7a, 0x06 und 0x40, das PDR höher ist als auf Routen mit einer größere Differenz. Bemerkenswert ist, dass das PDR auf Routen, die teilweise mehr Hops als andere Routen aufweisen, besser ist. Dies ist bei Knoten 0x77 und 0x4a der Fall. B.A.T.M.A.N-adv versucht für jeden Originator die beste Route zu diesem zu finden. Wenn allerdings der Link der besten Route trotzdem einen hohen Paketverlust aufweist, kann dieses Phänomen auftreten. Für Netzwerke, bei denen eine geringe Neuübertragung der Pakete relevant ist, ist ein Setup mit diesem beobachteten Paketverlust nicht geeignet.

Für die nächste Iteration der Messung wurde das OGMv2 Intervall um 2000 ms auf 3000 ms angehoben. Daraus ergibt sich ein Intervall für die ELP Pakete von 1500 ms. Die übrigen Parameter wurden nicht verändert. Die *Hop Penalty* beträgt $\frac{15}{255}$. Knoten 0x05 ist weiterhin der Quellknoten und sendet 100 UDP Pakete zu allen anderen Knoten des Testbeds.

In Abbildung 5.8b ist das PDR von Knoten 0x05 zu jedem anderen Knoten des Testbeds zu sehen. Das OGMv2 und ELP Intervall wurde für diese Messung angepasst. Auf der X-Achse ist wieder die Layer 2 Adresse der Senke abgebildet. Auf der Y-Achse ist das PDR dargestellt.

Hier ist eine deutliche Verbesseung des PDR zu erkennen. Das minimale PDR befindet sich bei Knoten 0x01 mit 0,73. Das maximale PDR befindet sich bei Knoten 0x7a mit 0,97. Der Median bewegt sich zwischen 0,93 und 0,84.

In Tabelle 5.1b ist die minimale und maximale TTL der empfangenen UDP Pakete dargestellt. Diese wurden, wie zuvor, dem B.A.T.M.A.N-adv Unicast Header entnommen. Die Routenlängen zu Knoten 0x01, 0x4a und 0x68 haben sich ausgehend von Tabelle 5.1a nicht verändert. Die Route zu Knoten 0x7a ist in dieser Messung direkt. Die Routen zu Knoten 0x06 und 0x40 sind für Teile der Messung länger geworden. Die kürzeste Route stellte eine direkte Verbindung zu Knoten 0x7a dar. Die längste Route mit einer Länge von 5 weist Knoten 0x40 auf. Dieser Wert trat allerdings nur bei einer Messung auf und ist daher als Ausreißer zu betrachten. Das ist darauf zurückzuführen, dass sich nach einem Neustart die Routen erst einmal stabilisieren müssen.

Auch wenn sich bei Knoten 0x06 und 0x40 die Routen teilweise verlängert haben, ist das PDR bei allen Knoten signifikant angestiegen. Die größte Verbesserung zeigt sich bei Knoten 0x77. In Abbildung 5.8a ist zu sehen, dass Knoten 0x77 ein niedrigeres PDR als alle anderen Knoten aufweist. In Abbildung 5.8b weisen alle Knoten, bis auf Knoten 0x7a, ein ähnliches PDR auf. Knoten 0x7a schneidet hier besser als der Durchschnitt ab. Vor allem bei Knoten 0x01 und 0x68 haben sich im Gegensatz zu Abbildung 5.8a die Interquantilabstände verringert. Das weist darauf hin, dass die Routen an Stabilität gewonnen haben.

In der nächsten Iteration der Messung wurden OGMv2 und ELP Intervall erneut erhöht. Die *Hop Penalty* wurde wieder auf dem Standardwert von $\frac{15}{255}$ belassen. Das OGMv2 Intervall wurde auf 5000 ms angehoben. Daraus resultierte, wie in Sektion 4.2.2 beschrieben, ein ELP Intervall von 2500 ms. Knoten 0x05 wurde erneut als Quelle gewählt und es wurden 100 UDP Pakete zu jedem anderen Knoten im Netzwerk gesendet.

In Abbildung 5.8c ist das PDR von Knoten 0x05 zu jedem anderen Knoten des Testbeds zu sehen. Auf der X-Achse ist die Layer 2 Adresse der Senke dargestellt. Auf der Y-Achse ist das PDR abgebildet. Das maximale PDR weist Knoten 0x7a mit einem Wert von 1,0 auf. Das minimale PDR mit einem Wert von 0,73 befindet sich bei Knoten 0x77. Die Mediane bewegen sich in einem Bereich von 0,84 bei Knoten 0x77 bis 0,97 bei Knoten 0x7a. Die Anhebung der Intervalle brachte bei allen Knoten, außer Knoten 0x77, im Gegensatz zu den vorangegangenen Messungen eine Verbesserung im PDR.

In Tabelle 5.1c ist die minimale und maximale TTL der empfangenen UDP Pakete dargestellt. Die maximale Routenlänge beträgt 4 bei Knoten 0x01, 0x4a und 0x77. Die minimale Routenlänge beträgt 1 bei Knoten 0x7a und 0x06. Die Routen zu Knoten 0x7a und 0x06 haben sich in dieser Iteration stabil auf Länge 1 gehalten. Die Länge der Routen variiert, im Vergleich zu den vorangegangenen Iterationen, allerdings in keinem signifikanten Ausmaß. Die maximale Routenlänge wird also nicht durch die OGMv2 und ELP Intervalle beeinflusst.

Zusammenfassend lässt sich festhalten, dass das Intervall der OGMv2 und ELP Nachrichten einen erheblichen Einfluss auf das PDR hat. Auf dem Testbed hat eine Erhöhung dieser Intervalle zu einer signifikanten Steigerung des PDR bei allen Knoten geführt. Mit einem OGMv2 Intervall von 5000 ms und einem ELP Intervall von 2500 ms lag der Median des PDR bei 0,84-0,97. Zudem nimmt mit zunehmender Intervalllänge der Interquantilabstand der einzelnen PDR Messungen ab. Dies lässt darauf schließen, dass die Routen an Stabilität gewinnen. Ein Einfluss der OGMv2 und ELP Intervalle auf die Länge der Routen lässt sich nicht erkennen. Durch die höheren Sendeintervalle der OGMv2 und ELP Pakete sind die Links zwischen den Knoten weniger stark ausgelastet. Daher gehen mit höheren Intervallen weniger gesendete UDP Pakete verloren. Zu beachten ist allerdings, dass eine Erhöhung der Intervalle das Netzwerk weniger flexibel gestaltet. In einem örtlich gebundenen Netzwerk wie dem Testbed führt dies zu keinen Nachteilen. Für ein Netzwerk, bei dem die einzelnen Knoten ständig in Bewegung sind, wie z.B. beim autonomen Fahren, kann ein zu hohes Intervall eine merkliche Verschlechterung des PDR bedeuten, da neue Routen zu spät erkannt werden. Daher müssen diese Parameter individuell an das Netzwerk angepasst werden.

5.2.4 Latenz

Die Messung der Latenz wurde, wie in Sektion 4.2.1 beschrieben, durchgeführt. Wie in der Messung des PDR wurde Knoten 0x05 als Quellknoten festgelegt. Dieser sendete zu jedem anderen Knoten des Testbeds 50 ICMPv6 Echo Requests. Die Implementierung des ICMPv6 Protokolls von Riot OS maß die RTT der Pakete und ermittelte daraus Minimum, Maximum und Durchschnitt. Die Messung der RTT wurde mit den folgenden B.A.T.M.A.N-adv Parametern durchgeführt

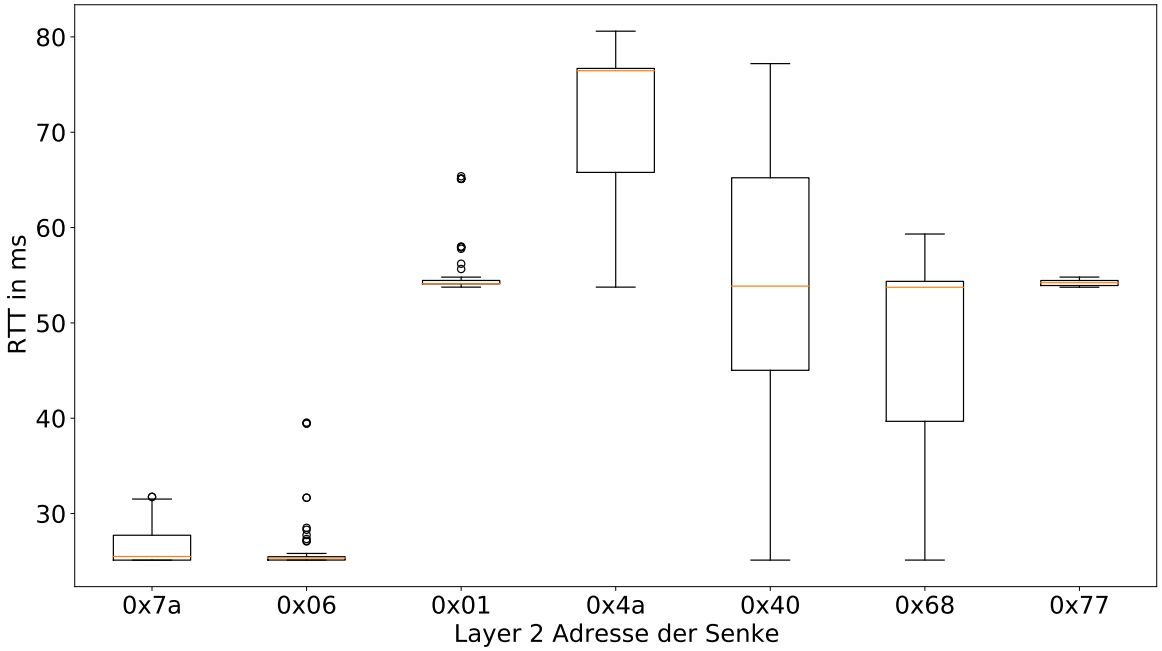


Abbildung 5.9: RTT zu allen Knoten des Testbeds

- *HOP PENALTY*: $\frac{15}{255}$
- *OGMv2 INTERVALL*: 1000 ms
- *ELP INTERVALL*: 500 ms

In Abbildung 5.9 ist die RTT der versendeten ICMPv6 Echo Requests zu sehen. Auf der X-Achse sind die Layer 2 Adressen der Knoten dargestellt, zu denen die ICMPv6 Echo Requests gesendet wurden. Auf der Y-Achse ist die RTT der ICMPv6 Echos abgebildet.

Es sind deutliche Unterschiede in den RTTs der einzelnen Knoten zu erkennen. Die minimale RTT weisen Knoten 0x7a, 0x06, 0x40 und 0x68 mit 25 ms auf. Diese RTT kommt auf einer Route der Länge 1 zustande. Dadurch, dass sie bei allen 4 Knoten konstant 25 ms beträgt, zeigt sich in der Implementierung eine gewisse Stabilität. Die maximale RTT weist Knoten 0x4a mit 80 ms auf. Knoten 0x7a, 0x06, 0x01 und 0x77 weisen eine sehr geringe Diskrepanz zwischen minimaler und maximaler RTT auf. Bei Knoten 0x4a, 0x40 und 0x68 ist die Diskrepanz zwischen minimaler und maximaler RTT mit bis zu 52 ms sehr hoch. Die Diskrepanzen lassen sich mit den Werten aus Tabelle 5.1a erklären. Wie dort abzulesen ist, variieren die Routen während einer Messung in ihrer Länge. Bei Routen, die eine geringe Varianz in ihrer Länge aufweisen, wie z.B. die Route zu Knoten 0x7a und 0x06, fällt die Diskrepanz der RTT Maxima gering aus. Bei anderen Routen, die eine hohe Varianz in ihrer Länge aufweisen, wie z.B. die Route zu Knoten 0x4a, fällt die Diskrepanz höher aus. Aus den Werten von Knoten 0x7a und 0x06 lässt sich ablesen, dass eine Route der Länge 1 im Mittel eine RTT von 25 ms aufweist. Jede Verlängerung der Route fügt der RTT im Mittel weitere 24 ms hinzu. Zu beachten ist, es sich bei den Latenzen um *Processing Delay* handelt, da sich Funkwellen mit Lichtgeschwindigkeit ausbreiten.

Zu beachten ist allerdings, dass die Latenz einerseits vom Routing und der Länge der Route abhängig ist, andererseits die verwendete Technologie auf dem *Physical Layer* eine ausschlaggebende Rolle spielt. Für alle Messungen wurde eine Datenrate von 38 kbps verwendet. Eine Änderung der Datenrate des Tranceivers auf 250 kbps führt auf einer Route der Länge 1 zu einer Verbesserung der RTT. Mit diesen Einstellungen wurde eine RTT von im Minimum 9,874 ms, im Maximum 10,384 ms und im Mittel 10,167 ms gemessen. Dies stellt im Mittel eine Verbesserung von 59,33% dar. Für den Einsatz im Testbed war diese Datenrate nicht geeignet, da laut Datenblatt [21], mit zunehmender Datenrate die Sensitivität des Tranceivers sinkt. Somit ist es mit höheren Datenraten nicht mehr gewährleistet, dass alle Knoten in das Netzwerk eingebunden werden. Somit ist ein Kompromiss zwischen Datenrate und Sensitivität nötig. Zu beachten ist dabei, dass eine höhere Datenrate die *On-Air-Time* des Tranceivers pro Übertragung verringert. Dadurch verringert sich in der Theorie der Stromverbrauch des Tranceivers. Das ist bei IoT-Anwendungen von besonderem Interesse.

KAPITEL 6

Fazit

6.1 Zusammenfassung

Ziel der Arbeit war es zu evaluieren, ob sich B.A.T.M.A.N-adv als Routingprotokoll im IoT-Sektor eignet. Dafür wurde eine Proof-of-Concept-Implementation des B.A.T.M.A.N-adv V Algrithmus geschrieben und in Riot OS integriert. Um die Performance des Algorithmus zu bewerten, wurden Performancemetriken definiert. Bei diesen handelte es sich um das PDR, die Latenz und den Speicherverbrauch. Um diese zu messen, wurden zwei Netzwerke genutzt: das Prototypensystem und das MIOT-Lab. Gesteuert wurden die Messungen über das TBMS System des Testbeds und Python Skripte. Die einzelnen Microcontroller erzeugten während der Messungen Ausgaben über eine serielle Schnittstelle. Aus diesen Ausgaben wurden die Metrikwerte berechnet. Die B.A.T.M.A.N-adv Implementation wies einstellbare Parameter auf. Diese waren die *Hop Penalty* und das Intervall der OGMv2 und ELP Nachrichten. Weiterhin war die Größe der Pakete, welche im Netzwerk versendet wurden, variabel. Es wurde untersucht, wie sich eine Veränderung der Parameter und Pakegröße auf das Netzwerk und die Performancemetriken auswirkte.

Als Ergebnis der Messungen lies sich festhalten, dass die Größe der versendeten Payload keinen Einfluss auf das PDR hatte. Allerdings konnte der verwendete CC1101-Tranceiver nur Pakete einer maximalen Größe von 256 Bytes versenden. Dies ist im IoT bereich allerdings kein Ausschlusskriterium, wenn es z.B. nur darum geht, Sensorwerte zu versenden. Sind größere Datenmengen vonnöten, kann mit Fragmentierung gearbeitet werden oder ein Tranceiver verwendet werden, der eine größere MTU bietet.

Der Speicherverbrauch teilte sich in zwei Bereiche auf: Den Flashspeicher (ROM) und den Arbeitsspeicher (RAM). Weiterhin teilte sich die Nutzung des Arbeitsspeichers in statisch angeforderten und dynamisch angeforderten Speicher auf. Es ließ sich beobachten, dass die Firmware 95,1 KB Flashspeicher belegt. Dies umfasste allerdings auch Riot OS. Der statisch belegte RAM belief sich auf 23,512 KB. Für den dynamisch angeforderten RAM wurde eine obere Schranke hergeleitet: $40Bytes*(N-1)^2+N*(8Bytes+56Bytes+88Bytes)+628Bytes$, wobei N die Anzahl der Knoten ist. Damit ergab sich für ein Netzwerk aus 8 Knoten eine maximale RAM Auslastung von $3,804KB + 23,512KB = 27.316KB$. Diese Schranke wies quadratische Bestandteile auf. Im Testbed erreichte allerdings keiner der Knoten

diese obere Schranke. ROM und RAM mussten größer als diese Speicherbelegung gewählt werden, um ausreichend Platz auf dem Stack zu haben, um die Firmware auszuführen und um eigene Firmware auf B.A.T.M.A.N-adv aufzubauen. Der verwendete Nucleo bot mehr als genug Speicher für diese Implementierung. Hier kann in Betracht gezogen werden, einen Controller mit signifikant weniger ROM und RAM auszuwählen. Dies könnte z.B. ein Cortex-M0+ Microcontroller, wie der LPC51U68 mit 265 KB Flash und 96 KB RAM sein [28]. Beim LPC51U68 ist zu beachten, dass sich der RAM aus zwei Speicherbänken zusammensetzt: einer 64 KB und einer 32 KB Speicherbank. Die 32 KB Speicherbank ist standardmäßig abgeschaltet und kann bei Bedarf aktiviert werden. In Sektion 4.2.1 wurden die Speicherklassen von *Constrained-Devices* nach RFC 7228 vorgestellt. Der LPC51u68 kann also in Klasse C2 eingeordnet werden und zählt somit als *Constrained-Device*. Dadurch, dass es möglich ist, das Protokoll auf einem Microcontroller der Klasse C2 umzusetzen, lässt sich Hypothese 3.1 verwerfen und Hypothese 3.2 annehmen. Somit ist in Bezug auf den Speicherverbrauch, die Implementierung für ein IoT Szenario geeignet.

Weiterhin wurde die *Hop Penalty* betrachtet, der Wert, um den die Routingmetrik pro Hop verringert wurde. Hier ergab sich, dass sich der von B.A.T.M.A.N-adv empfohlene Wert von $\frac{15}{255}$ für das MIOT Testbed eignete. Wurde dieser Wert zu hoch gewählt, war nicht mehr sichergestellt, dass alle Knoten des Netzwerks erreichbar waren. Ebenfalls ließ sich im Testbed durch Anpassen der Intervalle für OGMv2 und ELP Pakete auf 5000 ms und 25000 ms ein PDR von im Median 0,84-0,97 erreichen. Für Netzwerke, bei denen ein geringer Paketverlust essentiell ist, ist ein Wert von 0,97 akzeptabel. Bei 0,84 muss getestet werden, ob sich durch weitere Knoten oder veränderte Physical Layer-Hardware, wie z.B. verbesserte Antennen, eine Verbesserung ergibt. Mit den gewählten Einstellungen wies allerdings kein Knoten ein signifikant schlechteres PDR als alle anderen Knoten auf. Die Einstellungen für die Intervalle müssen für jedes Netzwerk individuell angepasst werden. Für das Testbed waren lange Intervalle der Pakete möglich, da es sich um ein statisches Netzwerk handelte. Enthält das Netzwerk allerdings mobile Knoten, sollten die Intervalle möglichst klein gehalten werden, um eine Veränderung der Routen schnell zu propagieren. Durch Optimierungen der Parameter ließ sich für das Testbed ein akzeptables PDR erreichen. Für diesen Anwendungszweck lässt sich Hypothese 4.1 verwerfen und Hypothese 4.2 annehmen. Werden die Intervalle also individuell auf das Netzwerk angepasst, lässt sich das Protokoll ebenfalls im IoT-Sektor einsetzen.

Für die Latenz ließ sich festhalten, dass die RTT linear mit der Routenlänge zunahm. Im Testbed wurde bei einer Datenrate von 38 kbps eine minimale RTT von 25 ms und eine maximale RTT von 80 ms gemessen. Bei dieser Datenrate fügte jeder Hop der RTT im Mittel 24 ms hinzu. Dies setzt vorraus, dass die Routen symmetrisch sind. Im realen Netzwerk war dies, wie in Abbildung 5.7 zu sehen, nicht immer der Fall. Weiterhin ist die RTT stark von den Einstellungen und Fähigkeiten der verwendeten Physical Layer Hardware abhängig. So lies sich z.B. die RTT auf einen Hop auf im Mittel 10,167 ms verbessern, wenn die Datenrate des Tranceivers auf 250 kbps angehoben wurde. Bei Sensornetzwerken, die stündlich Werte übertragen, fällt eine RTT von 80 ms nicht ins Gewicht. Für zeitkritische Netzwerke, wie z.B. Aktuatornetzwerke, muss möglicherweise Hardware, die eine höhere Datenrate bietet, ausgewählt werden. Dies bezieht sich allerdings nicht mehr auf B.A.T.M.A.N-adv als Protokoll. Hier bleibt nur festzuhalten, dass B.A.T.M.A.N-adv eine lineare Steigerung der RTT zur Routenlänge aufweist.

B.A.T.M.A.N-adv versucht durch die *HOP_PENALTY* je nach Einstellung die kürzesten, aber trozdem stabilsten Routen durch das Netzwerk zu finden. Den größten Faktor in der Latenz macht allerdings die verwendete Physical-Layer-Technologie aus. Somit lassen sich Hypothese 2.1 und 2.2 weder verwerfen, noch annehmen. Wird für den Anwendungsfall die passende Physical-Layer-Technologie gewählt, ist ein Einsatz im IoT-Sektor möglich.

Da zuvor Hypothese 3.2 und Hypothese 4.2 angenommen wurden und Hypothese 2.1 und 2.2 maßgeblich von der verwendeten Physical-Layer-Technologie abhängig sind, statt von B.A.T.M.A.N-adv als Protokoll, lässt sich Hypothese 1.1 verwerfen. Somit wird Hypothese 1.2 angenommen und B.A.T.M.A.N-adv eignet sich zum Einsatz im IoT-Sektor.

6.2 Ausblick

Der B.A.T.M.A.N-adv Algorithmus sieht noch folgende Eigenschaften und Optimierungen vor [29]:

- Unterstützung für mehrere Netwerkinterfaces
- Gatewayknoten [30]
- Layer 2-Fragmentierung [31]
- Network Coding [32]

Durch die Unterstützung mehrerer Netzwerkinterfaces können neue Eigenschaften implementiert werden. Zum Einen können verschiedene Physical Layer-Technologien zur Übertragung verwendet werden, zum Anderen kann ein Router, der mehrere Interfaces und Technologien unterstützt, mehrere Mesh-Netzwerke zu einem großen Mesh-Netzwerk verbinden. Hier kann untersucht werden, wie sich B.A.T.M.A.N-adv auf verschiedenen Physical Layer-Technologien verhält. Wenn alle Router verschiedene Interfaces unterstützen, können Pakete abwechselnd über die verschiedenen Interfaces versendet werden, damit sich die einzelnen Interfaces nicht gegenseitig stören.

Die Einführung von Gatewayknoten ermöglicht es einem B.A.T.M.A.N-adv Router anzukündigen, dass er über eine Verbindung zum Internet verfügt. Somit kann, sobald ein Knoten als Gateway im Netzwerk aktiv ist, für das gesamte Meshnetzwerk eine Internetverbindung verfügbar gemacht werden. Dadurch kann über eine große Fläche ein Internetzugang gewährleistet werden.

Layer 2-Fragmentierung ist für IoT Netwerke in der Regel nicht relevant, da die Größe der versendeten Pakete in der Regel klein ist. Für spezielle Anwendungsfälle, bei denen größere Datenmengen transportiert werden müssen, die die MTU des Netwerkgerätes überschreiten, kann eine Fragmentierung der Pakete auf Layer 2 implementiert werden. B.A.T.M.A.N-adv stellt dann das korrekte Fragmentieren und Zusammensetzen der Pakete sicher.

Es kann weiterhin untersucht werden, ob sich B.A.T.M.A.N-adv durch Network Coding verbessen lässt. Die Grundidee von Network Coding ist es, dass ein Knoten zwei Pakete zu

einem Paket zusammenfasst, um die aktive Sendezeit zu verkürzen. Damit dies umgesetzt werden kann, muss ein Knoten empfangene und gesendete Pakete puffern, um zu berechnen, ob seine Nachbarn ein codiertes Packet decodieren können. Dies erhöht zwar den Speicherverbrauch und die Latenz, kann aber in der Theorie den Throughput verbessern.

Abgesehen von diesen Optimierungen sollte B.A.T.M.A.N-adv auch in produktiven Netzwerken eingesetzt werden. Dies kann durch den Einsatz im ausgebauten Testbed umgesetzt werden. Eine weitere Möglichkeit wäre es, B.A.T.M.A.N-adv in einem IoT-Sensornetzwerk einzusetzen.

Literatur

[1] Saba Farooq Abbasi u. a. „Can BATMAN Replace RPL for IoT Applications?“ In: *International Journal of Computer Applications* 975 (), S. 8887.

[2] Andreas Boes/Katrin Gül/Tobias Kämpf/Barbara Langes/Thomas Lühr/Kira Marrs/Elisabeth Vogl/Alexander Ziegler. *Der Aufstieg des Internet of Things*. https://www.isf-muenchen.de/wp-content/uploads/2019/01/ISF-Report-IoT-180612r.pdf. Letzter Zugriff 7 Juni 2020. 2018.

[3] freifunk.net. *Batman-roaming*. https://wiki.freifunk.net/Batman-roaming. Letzter Zugriff 7 Juni 2020. 2018.

[4] Atit Bhavsar. *A Guide for selecting the right microcontroller for your IoT project*. https://iiot-world.com/industrial-iot/connected-industry/a-guide-for-selecting-the-right-microcontroller-for-your-iot-project/. Letzter Zugriff 26 Juni 2020. 2020.

[5] riot-os.org. *CPU*. https://doc.riot-os.org/group__cpu.html. Letzter Zugriff 8 Juni 2020. 2020.

[6] riot-os.org. *Drivers*. https://doc.riot-os.org/group__drivers.html. Letzter Zugriff 8 Juni 2020. 2020.

[7] riot-os.org. *Generic (GNRC) network stack*. https://riot-os.org/api/group__net__gnrc.html. Letzter Zugriff 8 Juni 2020. 2020.

[8] open-mesh. *Originator Message (OGMv2)*. https://www.open-mesh.org/projects/batman-adv/wiki/OGMv2. Letzter Zugriff 2 März 2020. 2020.

[9] onelektra. *[WLANware] [WLANnews] B.A.T.M.A.N - Version 0.0.6 und Subversion*. https://lists.freifunk.net/pipermail/wlanware-freifunk.net/2006-March/000038.html. Letzter Zugriff 05 Februar 2020. 2006.

[10] open-mesh. *B.A.T.M.A.N. IV*. https://www.open-mesh.org/projects/batman-adv/wiki/BATMAN_IV. Letzter Zugriff 26'Juni 2020. 2020.

[11] open-mesh. *B.A.T.M.A.N. V*. https://www.open-mesh.org/projects/batman-adv/wiki/BATMAN_V. Letzter Zugriff 26 Juni 2020. 2020.

[12] open-mesh. *B.A.T.M.A.N. advanced*. https://www.open-mesh.org/projects/batman-adv/wiki/Wiki. Letzter Zugriff 05 Februar 2020. 2020.

[13] open-mesh. *B.A.T.M.A.N. protocol concept*. https://www.open-mesh.org/projects/open-mesh/wiki/BATMANConcept. Letzter Zugriff 05 Februar 2020. 2020.

[14] open-mesh. *Originator Message (ELP)*. https://www.open-mesh.org/projects/batman-adv/wiki/ELP. Letzter Zugriff 2 März 2020. 2020.

[15] batman-adv. *bat_v_elp.c*. https://github.com/open-mesh-mirror/batman-adv/blob/master/net/batman-adv/bat_v_elp.c. Letzter Zugriff 14 Juli 2020. 2020.

[16] batman-adv. *batadv_packet.h*. https://github.com/open-mesh-mirror/batman-adv/blob/master/include/uapi/linux/batadv_packet.h. Letzter Zugriff 14 Juni 2020. 2020.

[17] batman-adv. *routing.c*. https://github.com/open-mesh-mirror/batman-adv/blob/master/net/batman-adv/routing.c. Letzter Zugriff 14 Juni 2020. 2020.

[18] Shafaq Malik, Ghulam Shabbir und Adeel Akram. „BEIOX: The BATMAN Enabled Internet of X“. In: *International Journal of Computer Applications* 975 (), S. 8887.

[19] kernel.org. *B.A.T.M.A.N-adv Kernel Modul*. https://www.kernel.org/doc/html/v4.15/networking/batman-adv.html. Letzter Zugriff 5 Juni 2020. 2020.

[20] mbed.com. *NUCLEO-F767ZI*. https://os.mbed.com/platforms/ST-Nucleo-F767ZI/. Letzter Zugriff 14 Juni 2020. 2020.

[21] Texas Instruments. *Low-Power Sub-1GHz RF Transceiver*. http://www.ti.com/lit/ds/symlink/cc1101.pdf?ts=1591538336036&ref_url=https://www.google.com/. Letzter Zugriff 7 Juni 2020. 2020.

[22] riot-os.org. *RIOT - The friendly Operating System for the Internet of Things*. https://www.riot-os.org/. Letzter Zugriff 14 Juni 2020. 2020.

[23] riot-os.org. *Netdev - Network Device Driver API*. https://riot-os.org/api/group__drivers__netdev__api.html. Letzter Zugriff 15 Juli 2020. 2020.

[24] bravegnu.org. *Übersicht über Standard Linker Sections*. http://www.bravegnu.org/gnu-eprog/c-startup.html. Letzter Zugriff 11 Juni 2020. 2020.

[25] andinet.de. *10. C Startup*. https://www.andinet.de/technik/programmierung/standard_linker_sections.html. Letzter Zugriff 11 Juni 2020. 2020.

[26] C. Bormann, M. Ersue und A. Keranen. *Terminology for Constrained-Node Networks*. RFC 7228. Letzter Zugriff 29 Juli 2020. RFC Editor, Mai 2014. URL: http://www.rfc-editor.org/rfc/rfc7228.txt.

[27] riot-os.org. *CC1100/CC1100e/CC1101 Sub-GHz transceiver driver*. https://riot-os.org/api/group__drivers__cc110x.html. Letzter Zugriff 27 Juli 2020. 2020.

[28] NXP. *LPC51U68*. https://www.nxp.com/docs/en/data-sheet/LPC51U68.pdf. Letzter Zugriff 13 Juni 2020. 2019.

[29] open-mesh.org. *B.A.T.M.A.N. Advanced Documentation Overview*. https://www.open-mesh.org/projects/batman-adv/wiki. Letzter Zugriff 15 Juni 2020. 2020.

[30] open-mesh.org. *Batman-adv gateways*. https://www.open-mesh.org/projects/batman-adv/wiki/Gateways. Letzter Zugriff 29 Juli 2020. 2020.

[31] open-mesh.org. *Layer 2 fragmentation*. https://www.open-mesh.org/news/43. Letzter Zugriff 29 Juli 2020. 2020.

[32] open-mesh.org. *NetworkCoding*. https://www.open-mesh.org/projects/batman-adv/wiki/NetworkCoding. Letzter Zugriff 29 Juli 2020. 2020.

Hiermit versichere ich, dass die vorliegende Arbeit mit dem Titel *B.A.T.M.A.N Routing im Internet of Things* selbstständig verfasst worden ist, dass keine anderen Quellen und Hilfsmittel als die angegebenen benutzt worden sind und dass die Stellen der Arbeit, die anderen Werken – auch elektronischen Medien – dem Wortlaut oder Sinn nach entnommenen wurden, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht worden sind.

Magdeburg, 5. Februar 2021

(Lukas Gehreke)