Cross-mesh communication in Contiki OS

Leendert van Doorn
Ahmet Güdek
Daan van der Valk

TU Delft

SOWNet technologies

Cover photo by Pexel on Pixabay
https://pixabay.com/en/architecture-blue-blur-bridge-2178736/

Cross-mesh communication in Contiki OS

Final Report Bachelor Project

by

Leendert van Doorn
Ahmet Güdek
Daan van der Valk

"Let there be light"

Project duration:	April 24, 2017 – June 26, 2017	
Supervisors:	K.G. Langendoen	TU Delft, supervisor
	O.W. Visser	TU Delft, coordinator
	H. Wang	TU Delft, coordinator
	W. Kavelaars	SOWNet Technologies

TU Delft

Preface

This document is written as a draft version of the final report for the bachelor project. The project, started on April 24, 2017 and set to end on June 26, 2017, was performed by Leendert van Doorn, Ahmet Güdek and Daan van der Valk in cooperation with SOWNet and Delft University of Technology.

We thank Koen Langendoen, Otto Visser and Winelis Kavelaars for the opportunity to carry out our bachelor project in cooperation with them, and for their support during the project.

Leendert van Doorn
Ahmet Güdek
Daan van der Valk

Pijnacker, June 2017

Contents

1 Introduction

The *Internet of Things* (IoT) is a contemporary paradigm concerning the inter-networking of all kind of *things*, such as mobile phones, sensors, actuators and vehicles [2]. *Smart Cities* use IoT technologies to manage their assets by connecting all kinds of devices, striving for safe, green, efficient and sustainable urban life [6].

Intelligent street lighting is a Smart City application aiming to reduce energy consumption. Passing traffic is measured and used to control lighting intensity. Street lights can be made intelligent by adding sensors to detect passing road users. From the perspective of the road user, a surrounding area of light is required for a safe journey. This can be achieved by extending lampposts with smart devices, called nodes, that communicate with nearby lampposts, signalling road activity to each other. Numerous governments and companies have been involved in the developments of such systems. The communication between nodes could be done using existing power lines (PLC) [13], or wireless [5] [14].

FutUrlight is a Smart City initiative aiming to implement intelligent street lights, starting in Zoetermeer, The Netherlands. Several parties are involved in the development, including the Zoetermeer administration, TNO, DCD, Sense-OS and SOWNet Technologies. The system should be an autonomous, plug-and-play city lighting system using wireless communication between lampposts [9].

When a car drives along a street lit by smart street lights, there should be an area of light around the car to provide the driver with a comfortable range of sight. To achieve this, lampposts communicate information about passing road users to each other.

A city with many lampposts will have a large network of nodes. To prevent problems that can occur in populous wireless networks, such as congestion and long delays, they are often split up into smaller networks with a mesh topology, called Personal Area Networks (PANs) (see Figure 1.1). Nodes are able to communicate within their PAN efficiently. However, communication with nodes from other PANs is sent over the Internet; since it may take multiple hops to the Internet router (the *edge router* of the PAN) this typically causes delays. The transmission over the Internet is usually performed over a fast medium, not adding considerable delay. This is particularly inefficient when nodes in different PANs are only a few meters apart. Figure 1.2 illustrates this problem, which we aim to solve as efficiently as possible.

Problem

Direct communication between nodes in different PANs is currently not possible, for reasons explained later in this chapter. Figure 1.3a illustrates this problem and the desired solution: when a node (N1) in a subnetwork produces a message for a node (N2) in another subnetwork, the following steps are taken:

1. N1 sends the packets to the edge router of network 1 (ER1) via nodes in PAN 1. This may include hopping between several other nodes.

2. ER1 sends the packets to the edge router of the second network (ER2) via the Internet, usually using faster communication technology (e.g. a 4G network).

3. ER2 sends the packets to N2 via nodes in PAN 2. Again, this may include hopping between several other nodes.

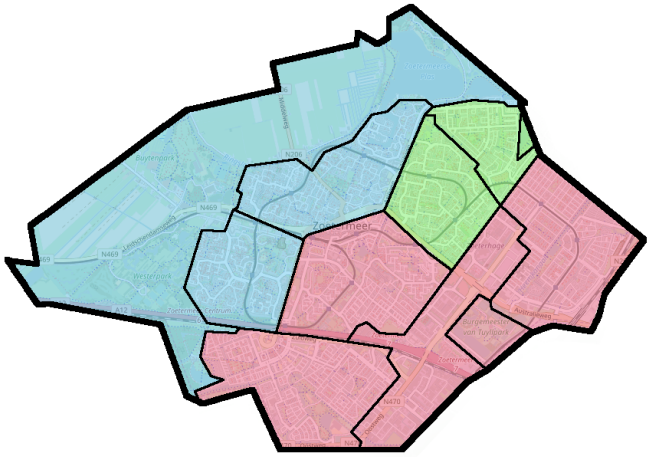


Figure 1.1: An example network configuration in a city. To alleviate network stress, multiple PANs can be utilised. In this example, separate PANs are used for three residential areas, coloured blue, green and pink.

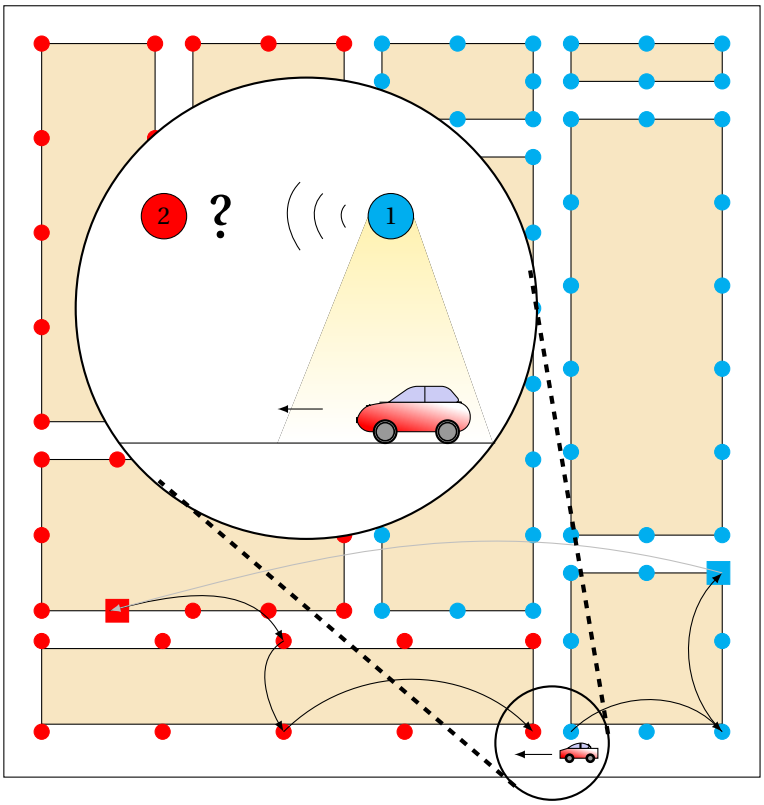
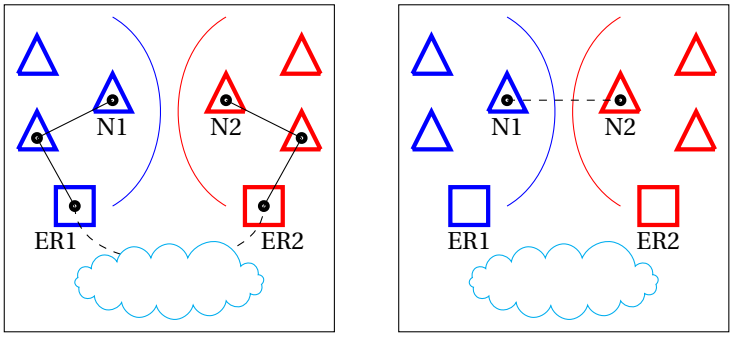


Figure 1.2: Schematic diagram illustrating a problematic situation. A coloured circle denotes a street light, with an integrated sensor and wireless communication node. Each PAN is indicated by a single colour, with one coordinating edge router depicted as a square. When a car is detected by node 1, this node tries to signal node 2. However, because the nodes are in different PANs, direct communication is currently not possible. The only alternative is to hop over multiple other nodes to the edge router that is connected to the Internet, which can then relay the information to the edge router of the other PAN. The black curved arrows indicate hops over nodes using RF, while the grey arrow shows communication over the Internet.



(a) Communication via edge routers (b) Direct communication

Figure 1.3: Schematic depiction of the two communication methods between nodes N1 and N2 of two distinct PANs. The full lines show communication within a PAN. The dashed lines show cross-mesh communication, which is currently only possible via the edge routers (a). The aim of this project is to enable direct cross-mesh communication between the neighbouring nodes (b).

The desired, direct cross-mesh communication is shown in Figure 1.3b.

This problem can be extended to nodes close enough to be interested in each other's sensor data, but not directly within RF range. If the route to the target node is efficient enough to be traversed by direct routing, then this route is preferred to the standard route via the edge router.

Requirements

We now describe the functional and non-functional requirements that must be met before the problem can be considered solved, and we explain why the problem exists.

The functional requirements are fairly straightforward:

1. Nodes should be able to send a message to a node in a neighbouring PAN, without using the Internet connection of the edge routers;
2. The solution must not introduce considerable network traffic and delays;
3. Each sent message should arrive only once at the destination application; duplicate messages should be filtered out.

The definition for an appropriate packet structure for such cross-mesh communication is part of the assignment, as is the implementation of handling such incoming packets. However, outgoing packet creation — including setting the values of parameters defined by our solution — is assumed to be done on the application level, and is not part of our project.

The client company, SOWNet Technologies, an IoT company based in Pijnacker, The Netherlands, has determined several non-functional requirements. Because the project is part of a larger system, there are certain constraints regarding the technologies to be used. First of all, our solution must be integrated into Contiki, an operating system for IoT devices. Next, our solution must work in a network stack consisting of the IEEE 802.15.4, 6LoWPAN, and IPv6 protocols (see Chapter 2 for more information on Contiki and these network protocols).

In the existing Contiki network stack, it is not possible to send a unicast message to another PAN, which in general is not problematic. However, the use case of smart city lighting introduces PANs that can stretch to distances of multiple kilometres, combined with a need for low-latency communication between PANs, making it necessary to mitigate delays caused by relaying messages through edge routers.

Methodology

Scrum-based Development

The Bachelor Project took 10 weeks, which we spent full time in the SOWNet office in Pijnacker. During the project, we have used parts of the Scrum methodology of software development to steer our process, because we wanted to remain flexible and did not know what exactly the end product would include. Because our team comprised just three members, there are parts of scrum that we have not used, such as the roles of product owner and scrum master, and sprint review meetings. We have divided the ten weeks in which the

project took place into sprints of one week each. At the start of every sprint, we decided on the tasks to be completed in that sprint. Instead of scheduling daily “stand up” meetings, we deliberated on problems when they came up, which was efficient because we were working in the same room. For a reflection on our Scrum usage, see Chapter 7.

Tooling and Setup

While developing our implementation, we have used Contiki’s Cooja simulator to run and test our code. The Cooja simulator is part of the Contiki operating system and makes it possible to simulate a network of Contiki nodes. We have only started testing on real hardware from week 7 onward, because the hardware modules were not ready before then. Our implementation and the used Cooja version is based on Contiki 3.1. Unit tests are written with version 1.1.1 of the Cmocka unit testing framework, of which the code coverage was calculated with Gcov. Git was used to keep track of code versions.

Contributions

We have extended Contiki 3.1 with cross-mesh communication functionality. When it is desirable to send a message between nodes of neighbouring PANs without using the Internet connection of edge routers, we offer four different ways to do so (outlined in Chapter 3). With all solutions, at least 98% of packets arrive at the destination node in our simulations, in situations where the radio is always on. Furthermore, we have tested our solution extensively with unit tests, attaining 98.85% line coverage, and on hardware, as a proof of concept. These tests have demonstrated that cross-mesh communication is possible with the implemented solutions.

2

Research

During the first two weeks of the project, we have researched the technologies involved in the project, and identified work that has already been done to enable cross-mesh communication. We have mainly looked at scientific papers and protocol specifications, but we have also obtained information from within SOWNet and from various online resources. In this chapter we present our findings about the hardware used, the Contiki operating system, its network stack, and related work.

Hardware

SOWNet uses smart devices that communicate with each other by relaying their sensor data to a custom-built ARM-based device running a Linux operating system. This device has a ST-6LP01 radio module attached, which is developed by SOWNet Technologies. Each module is connected over USB into the USB port, or over UART onto the GPIO headers available on the ST-6LP01. The pin-out of the ST-6LP01 radio board can found in Appendix A.

Network

The nodes run a 6LoWPAN network stack. In this network stack, the 6LoWPAN protocol is in the top of the Data Link Layer within the Open Systems Interconnection (OSI) model. It receives IEEE 802.15.4 frames and assembles them into IPv6 packets. When sending messages, IPv6 packets are placed into IEEE 802.15.4 frames, with optional fragmentation and header compression. Figure 2.1 shows a possible full network stack for a 6LoWPAN network. While there are other solutions available for low-power networks, 6LoWPAN is the only non-proprietary architecture that binds the standardised non-proprietary protocols IEEE 802.15.4 and IPv6 together. We refer to Appendix B for more information on the 6LoWPAN protocol.

The IEEE 802.15.4 protocol is designed for low-power and constrained devices; it also stands out for supporting a mesh network topology, and has been revised multiple times to increase robustness. IPv6 is particularly useful because of its gigantic address space (2^{128}) and it allows easy integration with the Internet. To be able to utilise this large address space in low-power communication with relatively small frames, IEEE 802.15.4 defines 16-bit PAN IDs, 16-bit short identifiers and EUI-64 identifiers that can be used for addressing nodes. On the MAC layer, IEEE 802.15.4 headers contain a sequence number and the optional fields of destination PAN ID and address, and source PAN ID and address, where the address fields can be short identifiers or EUI-64 identifiers. The protocol also permits broadcasting by using the broadcast address `0xFFFF` for both the short address and the PAN ID. Broadcast address can also refer to the 64-bit broadcast MAC address which is defined in IEEE Std 802-2014 (`0xFFFF FFFF FFFF FFFF`). For more information about IEEE 802.15.4, see Appendix C.

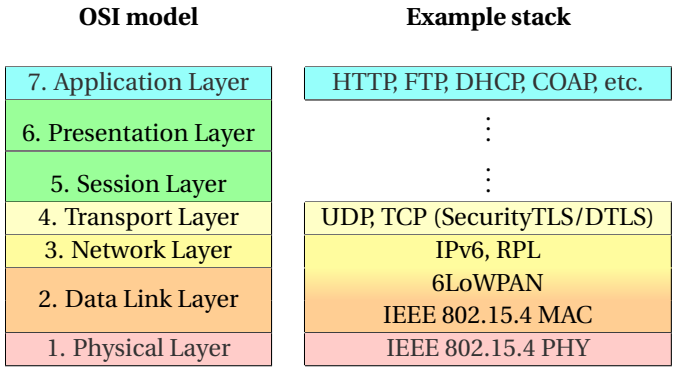


Figure 2.1: The Open Systems Interconnection (OSI) model layers, and an example stack of protocols in a network using 6LoWPAN.

IPv6

Compared to its predecessor IPv4, Internet Protocol version 6 (IPv6) increases the available address space from 2^{32} to 2^{128} addresses, by changing the addressing format from 4×8 bits to 8×16 bits. The protocol also has a larger header type field, allowing for more applications, while there is no strong limit on lengths of headers. This provides greater flexibility for introducing new options later on.

The IPv6 Header Format

An IPv6 packet starts with an IPv6 header, containing the following fields: version, traffic class, flow label, payload length, next header type, hop limit and source and destination addresses (see Figure 2.2). The version field is always set to 6, and traffic class and flow label are normally all zeroes. The payload length indicates the length of the data in bytes. The next header type is an 8-bit selector that describes the type of the header that starts directly after the address fields. This can be an IPv6 extension header, or the header of a datagram — for example, UDP or TCP. Multiple headers can be chained in this format before the payload is reached. The hop-limit field is decremented at each intermediate router, and the packet is dropped when it reaches 0. [4]

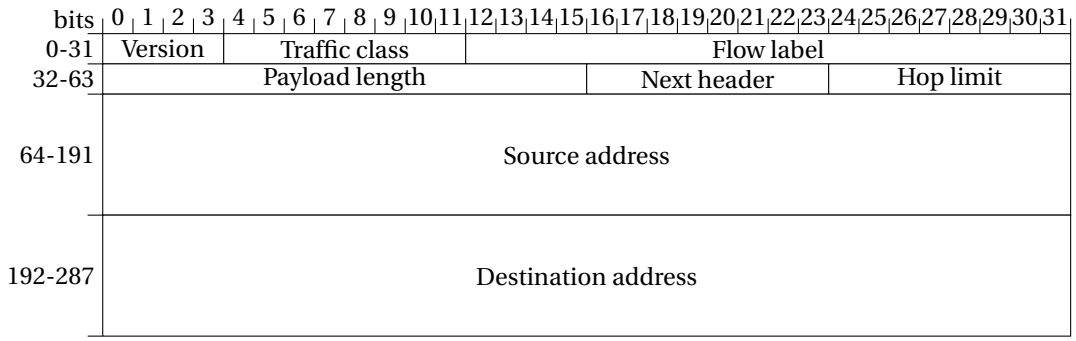


Figure 2.2: The IPv6 header format

Extension Headers

Extension headers help IPv6 provide flexibility and extensibility. Extension headers are placed after the standard IPv6 header, and the next header field in the standard header should be set depending on the first extension header.

The following extension headers are specified: hop-by-hop options header, destination options header, routing header, fragment header, authentication header, and encapsulating security header. As the hop-by-hop extension header is defined for setting routing behaviour, which is related to the problem of this project, this extension header will be described in detail.

The hop-by-hop options extension header is processed by all nodes through which an IPv6 packet passes. To use this header, the next header field in the normal IPv6 header needs to be set to `0x00`. This extension header, when present, is always placed directly after the IPv6 header, and starts with an 8-bit field to indicate the header *after* the extension header. Next is an 8-bit field that contains the length of the hop-by-hop options

header in 8-octet units, excluding the first 8 octets. Next, any number of type-length-value (TLV) encoded options may follow, but the total length of the extension header in octets must be a multiple of eight. See Figure 2.3.

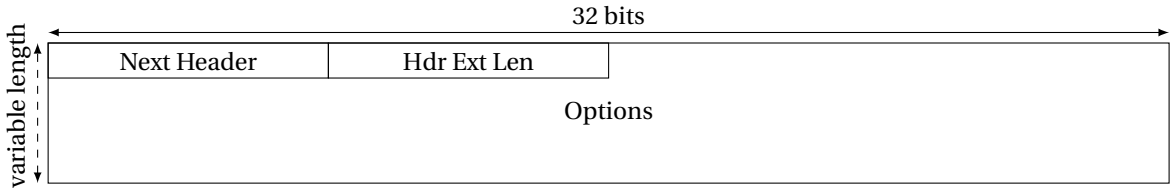


Figure 2.3: The hop-by-hop extension header format

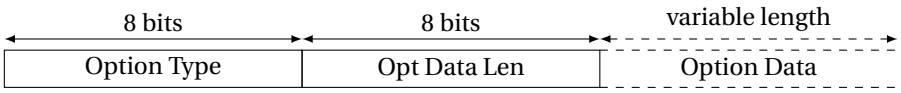


Figure 2.4: The Type-Length-Value encoding format

TLV encoding uses one octet to carry the option type and one octet to carry the option data length (see Figure 2.4). If the total length of the extension header in octets is not a multiple of eight, padding options are used to align the header.

The IPv6 standard further defines the format of the type field by using the 3 highest-order bits for several options. When a packet is received with a hop-by-hop option that is unknown by the system, the following steps should be taken depending on the the first 2 bits of the type field:

- **00** – Skip over the option and continue processing the header.
- **01** – Discard the packet.
- **10** – Discard the packet and send an ICMP Parameter Problem message to the packet's Source Address, pointing to the unrecognised Option Type.
- **11** – Discard the packet and, only if the packet's Destination Address was not a multicast address, send an ICMP Parameter Problem message to the packet's Source Address, pointing to the unrecognised Option Type.

The third bit specifies whether or not the values inside the option can change en-route to its destination:

- **0** – Option values do not change en-route.
- **1** – Option values can change en-route.

Contiki

The nodes run Contiki OS, an open-source operating system designed for IoT[1]. Contiki implements the complete 6LoWPAN network stack utilizing the NETSTACK drivers. Figure 2.5 gives an overview of NETSTACK, in which we worked most of the time during our project. For more information on the inner workings of Contiki, see Appendix D.

[1] `http://www.contiki-os.org/`

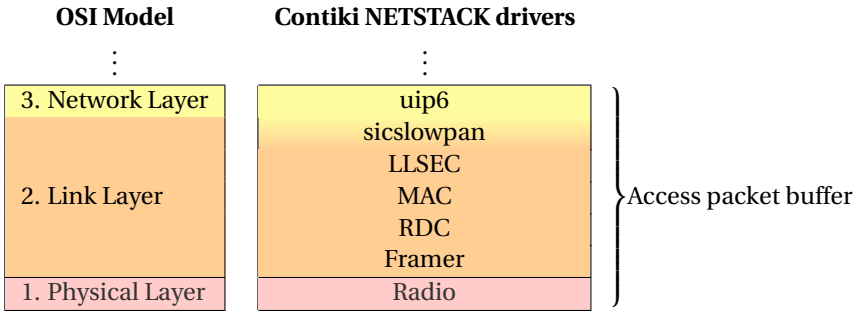


Figure 2.5: Overview of the Contiki network stack (NETSTACK)

Cooja

Contiki OS provides a simulation environment, called Cooja, in which networks of Contiki nodes can be created. The simulation can be done on an abstract level or with hardware emulation. This allows developers to test the code in simulation before running on hardware.

Related Work

As the usage of large-scale sensor networks in urban projects is newly emerging, research on our specific problem is scarce. Many sensor network projects either use a single PAN for the entire system, choose proprietary solutions such as ZigBee, or use more capable and more expensive technologies. The IEEE 802.15.4 and 6LoWPAN specifications do not offer any solutions other than providing support for broadcasting within the protocols.

In ZigBee, PAN Merge and PAN Bridge [8] attempt to solve the issue of cross-mesh communication. However, both methods inflict performance penalties in the form of higher latencies, packet loss and memory utilisation. The merging approach combines multiple PANs and creates a routing tree for the newly created combined PAN. This can be resource intensive and can take a long time to complete for larger networks, which makes it less efficient than communication via the edge router. Also, this reintroduces the problem of large unwieldy networks, which was the initial reason to split the network into multiple PANs. With bridging, only a single node per PAN needs to switch between networks. While this approach reduces the reconfiguration time, it does not provide the same functionalities as PAN Merge. During the time that the bridge is connected to one PAN, it cannot receive any packets from the other and essentially creates a one-way communication path [10].

Another solution is to make the PAN bridge take part in multiple networks by providing it with multiple transceivers, but this increases production costs, requires additional logistics and raises the network setup complexity. Neighbour discovery is initialised by the bridge nodes and the network is subdivided by optimizing the bridge, node and edge router ratios. To also limit the bottleneck effects of singular bridges between PANs, the system needs to be subdivided into smaller segments with alternative paths to other nodes over multiple bridges [3].

Conclusion

Based on our research of the related work, we conclude that there is currently no solution to the problem of direct communication between two PANs in Contiki. Furthermore, the solutions used in other protocols have too many disadvantages to apply them analogously to our situation. This means that we have to create a new mechanism to enable cross-mesh communication in Contiki.

3

Solutions

Assumptions

We have devised several approaches that enable cross-mesh communication between nodes, each approach tailored to specific use cases (see Section 3.3) and requirements. Because PAN management and coordination is a complex issue, we have made some simplifying assumptions to create a more manageable scenario.

1. For certain parts of our solution, the sending node requires *additional information* such as PAN IDs, or short IDs of nodes. We assume that all relevant information is known by the sending node. This could, for example, have been provided by the network coordinator or a cloud service.

2. *Link layer encryption is turned off, or done using the same AES-128 key for the concerning PANs.* Encryption at the link layer complicates packet inspection, so turning it off simplifies development.

3. *The nodes all communicate over the same IEEE 802.15.4 channel, i.e. use the same radio frequency.* Within one PAN, a single radio channel is used, as defined in the IEEE 802.15.4 standard. As some channels may turn out unsuitable for communication due to noise or interference of other networks (such as Wi-Fi), it is preferable to pick a fitting channel. When two PANs use different channels, this complicates direct cross-mesh communication.

Solving the Problem

Our solution to allow cross-mesh communication consists of four mechanisms. These are, in order of increasing sophistication: Flooding, 2-PAN Flooding, Hybrid and Routing-Twice. None of these mechanisms, including Flooding, were previously implemented in Contiki. As situations may arise with multiple hops between the source and destination nodes, a method is required to inform the nodes on the communication path about the message type, in order to relay the packets according to the corresponding mechanism. For this, we employ the hop-by-hop extension header defined in the IPv6 standard, and define our own hop-by-hop extension header option with a fixed format. This format can be found in Figure 3.1, with per-solution specifics in Table 3.1. The values used for the Option Type field are not from standards; we have chosen these values ourselves and in accordance with the IPv6 standard for hop-by-hop option types (see Section 2.3.2). We compare the four approaches and their network impact in Chapter 5. The next sections describe the mechanisms mentioned earlier.

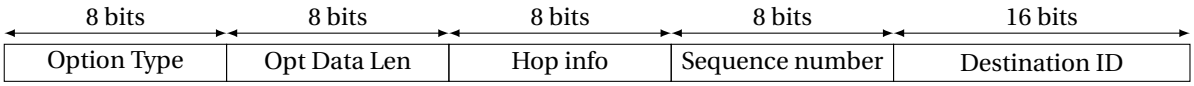


Figure 3.1: Cross-Mesh IPv6 Hop-by-Hop Extension Header Options.

Algorithm	***Type*** **8 bits**	***Length*** **8 bits**	***Hop Info*** **8 bits**	***Sequence Number*** **8 bits**	***Destination ID*** **16 bits**
Flooding	0x21	0x04	0x00	*variable*[4]	0x0000
2-PAN Flooding	0x22		*variable*[1]		*variable*[5]
Hybrid	0x23		*variable*[2]		0x0000
Routing-Twice	0x24		*variable*[3]		*variable*[6]

[1] Time-to-live in the PAN of the destination node.
[2] Time-to-live when flooding. Ignored in the PAN of the destination node, in which is routed normally.
[3] 0x00 initially, changed to 0x01 by the bridge node, and changed to 0x02 when PAN of destination node is reached.
[4] Sequence number set by the sending node, in the range 0x00–0xFF. After a cross-mesh packet is sent, the next cross-mesh packet from the same node should have the subsequent number (modulo 255).
[5] The PAN ID of the destination node's PAN.
[6] Short ID of node on the border. Because SOWNet's current implementation does not use short ID's, for now this field contains the last 16 bits of the bridge node's IPv6 address.

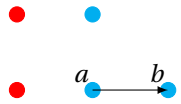
Table 3.1: Cross-Mesh IPv6 Hop-by-Hop Options Extension Header.

Flooding

Sending broadcast messages that are received by every node is the most straightforward cross-mesh communication technique, and flooding is the simplest way of implementing this. With flooding, every node retransmits a packet after the first time it has received it. A packet's source address and sequence number are stored to keep track of previous messages. To propagate over any node in range, the broadcast PAN ID 0xFFFF is used on the link layer. This approach makes no distinction between PANs, as a packet is solely directed at the broadcasting address. This makes for a simple way of crossing mesh boundaries. However, the echoing of broadcast packets may lead to network congestion. To avoid unnecessary spreading throughout the network, the IPv6 hop-limit field is used. Figure 3.2 shows an example transmission with flooding.

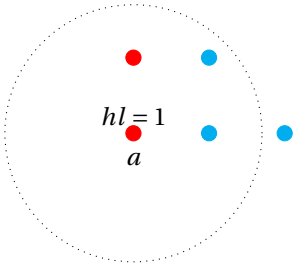
Explanation of symbols used in diagrams

Unicast



Nodes in the same PAN have the same colour. Node `a` sends a unicast message to node `b`, which is in its own PAN. Other nodes do not process this message.

Broadcast



Node `a` sends a broadcast message with a hop limit of 1. All nodes within the circle receive the broadcast.

2-PAN Flooding

Alternatively, a distinction can be made between two neighbouring PANs. By using *two* hop limits—one for the sender's PAN, and one for the receiver's—the number of unnecessary transmissions can be reduced. It is important to note, however, that this implementation goes against the IPv6 standard in regard to the usage of the hop limit field, as it is not decremented at every node, but only at nodes in the source PAN. Figure 3.3 shows an example transmission with 2-PAN Flooding.

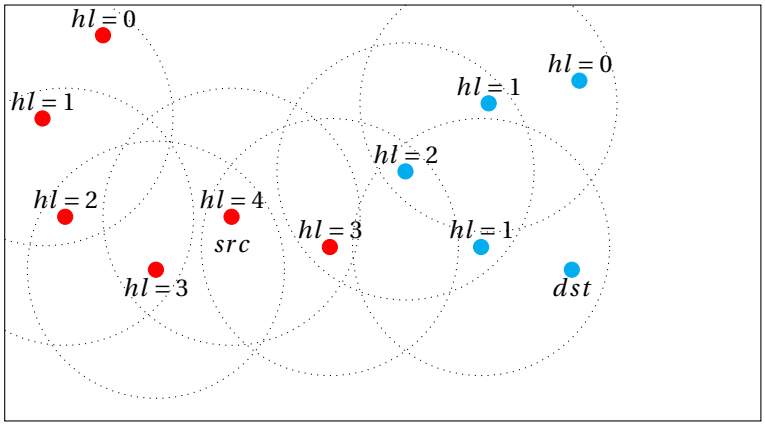


Figure 3.2: Flooding: the source node communicates to the destination node by simply using broadcast messages throughout both PANs.

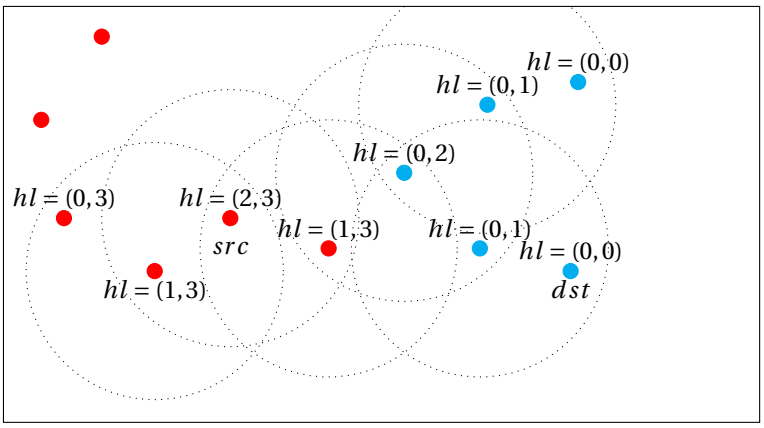


Figure 3.3: 2-PAN Flooding: the source node communicates to the destination nodes by using a broadcast with a separate hop limit for both PANs.

Hybrid

A more sophisticated approach is a hybrid between broadcasting and regular routing within a PAN. This solution uses broadcasting messages in the sending node's PAN to cross the mesh boundary, after which the message is routed directly to the destination using the standard 6LoWPAN routing mechanism. Figure 3.4 shows an example transmission with the Hybrid solution.

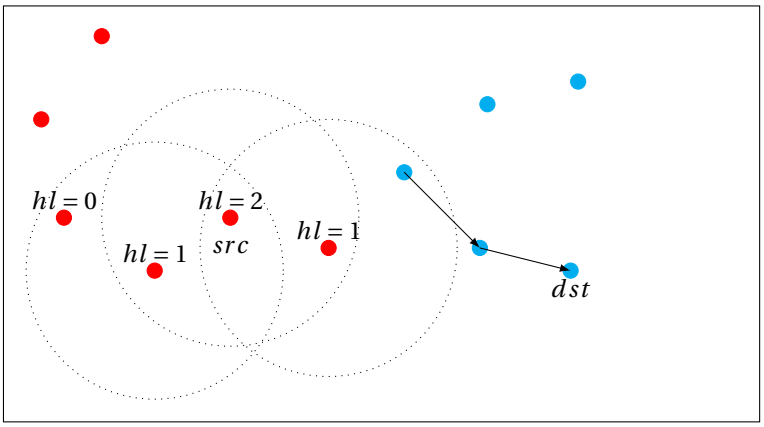


Figure 3.4: Hybrid: the source node communicates with the destination node by sending a broadcast throughout its own PAN, which is routed normally when received by a node in the destination PAN.

Routing-Twice

Finally, to produce even less network traffic, we have designed a fourth solution, which requires only one broadcast message to be sent. In this mechanism, it is necessary to know which node in the sender's PAN is appropriately situated to act as a "bridge", and deliver a message to the receiver's PAN. If this is known, a message can be sent that will be routed directly to the bridge node, which will then broadcast the message. This broadcast is heard by nodes in the destination's PAN, which can again directly route the message to the destination address. Figure 3.5 shows an example transmission with Routing-Twice.

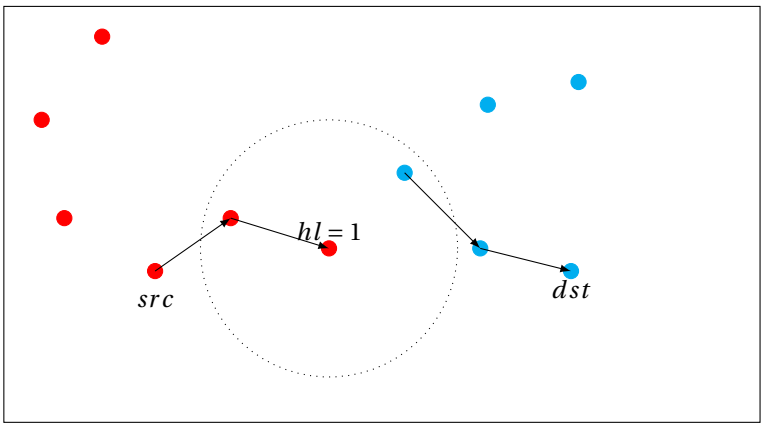


Figure 3.5: Routing-Twice: the source node communicates with the destination node by sending a directed message to a node bordering the other PAN. To bridge the gap to the other PAN, this node then sends a broadcast, which is once again routed normally when received by a node in the destination PAN.

Deduplication

When multiple nodes receive a message that they can route further, all of them will send. This means that nodes may receive multiple copies of the same message. To prevent this from congesting the network and delivering the message to the receiving application multiple times, duplicate messages are filtered out. In the hop-by-hop extension header option, the Sequence Number field is set by the initial sender. Nodes remember the source address and sequence numbers of the 20 most recent packets, although this setting may be overwritten (see Section 4.1). If a message is received with a source address–sequence number matching one of these, the packet is not processed further. This is done in all mechanisms; Figure 3.6 shows this behaviour for Routing-Twice.

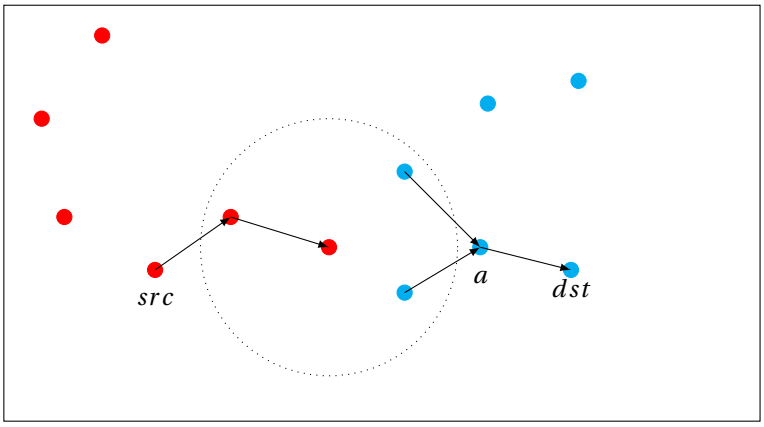


Figure 3.6: Deduplication in cross-mesh communication using Routing-Twice. The *src* node sends a message with the Hop Info field set to a sequence number, which stays the same as the packet is retransmitted. When the message has crossed the PAN border, node *a* receives two messages with the same sequence number, so it only forwards one message to the destination node.

Use Cases

All of the solutions mentioned above allow a node to send a message to a node in another PAN, but they differ in sophistication. The reader may wonder why we bother with Flooding when Hybrid is less network intensive and does not require any extra information. Node-to-node communication may indeed be the main type of cross-mesh communication. However, certain situations like an ambulance driving through a street, may require the sending of low-latency messages that arrive at every node a certain number of hops away (Flooding), or messages that arrive at every node within one or two PANs (2-PAN Flooding).

4

Implementation

The Cross-Mesh Engine

We have extended Contiki with cross-mesh functionality. Cross-mesh engines are used when packets with known hop-by-hop header options are received, as defined in Chapter 3. Contiki's `uip6` module, which processes IPv6 packets, checks incoming packets for hop-by-hop header options. If a known cross-mesh option is found, it calls the cross-mesh engine to process the packet. The cross-mesh engine then decides if the packet should be sent to the upper layers, and retransmits it when this is desirable. We have implemented our solution in the LAD engine (after the initials of the authors' first names). The LAD engine is the default (and currently only) cross-mesh engine. New engines can be created to change or extend functionality, as will be explained later in this chapter. The LAD engine defines four input functions, one for every defined hop-by-hop option type. Upon receiving a packet from `uip6`, the LAD engine calls the appropriate function to process the packet. A diagram describing this process can be found in Figure 4.1.

During retransmission, the packets are processed in Contiki's `tcpip` module, which calls the standard output stack. Special care is taken for the transmission of Routing-Twice packets. If the `tcpip` module notices that the packet to be sent has the Routing-Twice option, a check is performed to see whether the packet can be directly routed to the destination node. If not, the packet must be for a node in another PAN, so it is sent to the node in its own PAN with the short ID specified by the option.

The implementation of cross-mesh engines is similar to that of the multicast engines already existing in Contiki. Similar to `uip-mcast6-engines.h` for the multicast engines, `uip-cross-mesh-engines.h` contains the engine definitions for cross-mesh. To enable cross-mesh functionality, this file needs to be included, and the following macros should be defined in the `project-conf.h` configuration file of the Contiki node:

- `UIP_IPV6_CROSS_MESH`
  Should be defined as `1`

- `UIP_CROSS_MESH_CONF_ENGINE`
  Should be defined as one of the macros listed in `uip-cross-mesh-engines.h`

- `LAD_CONF_STORAGE_SIZE` (optional)
  When using LAD, this sets the number of source address–sequence number combinations that are stored to remove duplicates. If it is undefined, the storage size is 20.

Our implementation assumes that, according to the IPv6 standard, the hop-by-hop extension header is always the first extension header after the IPv6 header, if it is present.

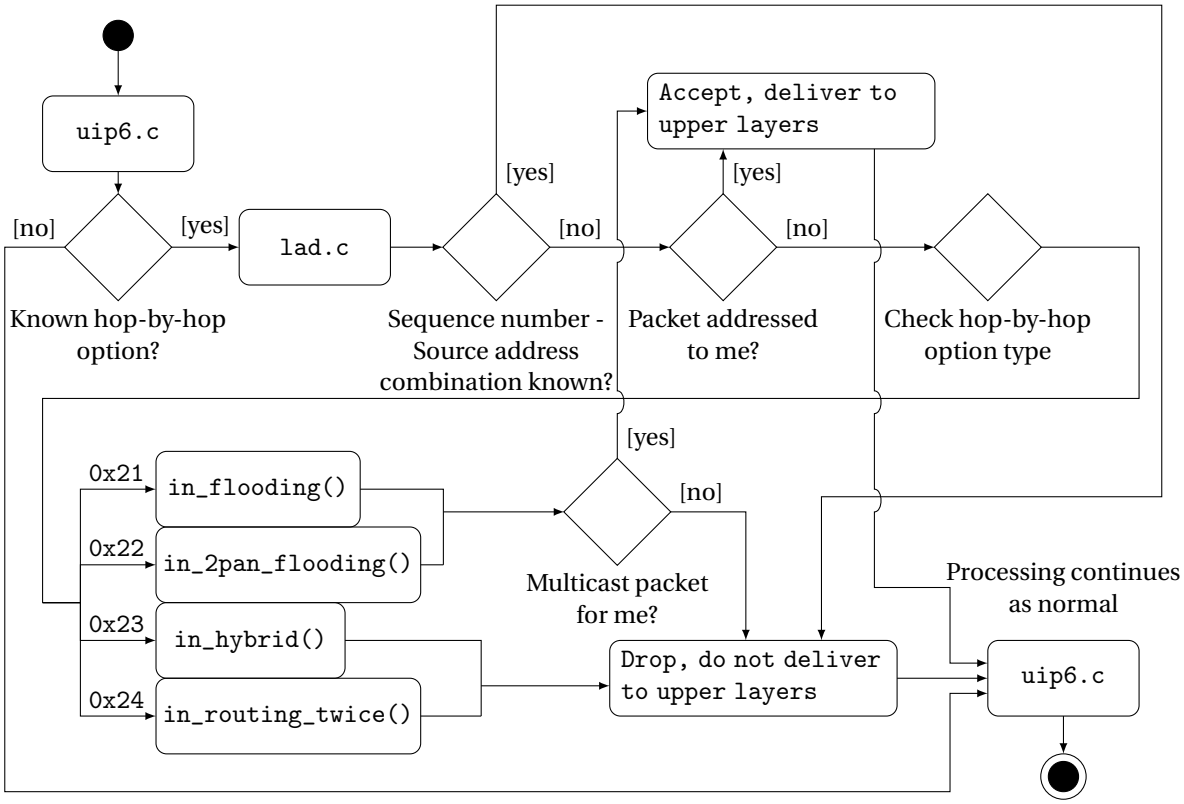


Figure 4.1: Diagram showing what happens when a message with a known hop-by-hop header option is received. When control reaches `uip6.c`, the incoming packet is checked for a known hop-by-hop extension header option. If this is found, control is passed to `lad.c`, otherwise processing continues as normal. `lad.c` then checks if the sequence number is known and if the packet is addressed to its own IPv6 address. If the sequence number is known, the packet is simply dropped. If the processing node is the packet's destination, it accepts without forwarding. If both conditions are false, the hop-by-hop option type is checked and the matching `in_` function is executed, in which the message is retransmitted. Messages to multicast addresses can be sent using a flooding mechanism, so the `_flooding()` functions do an extra check for multicast packets.

Bridging the Gap

To cross a PAN boundary, link-layer messages need to be sent with a destination PAN ID corresponding to the receiving PAN, or the global broadcast PAN ID `0xFFFF`. As it is not known when the second PAN will be reached in situations with multiple hops, `0xFFFF` must be used. To let the NETSTACK framer know whether to use its own PAN ID or the global PAN ID, we have created the `PACKETBUF_ATTR_GLOBAL_BROADCAST` flag in the packet buffer, from which the framer builds the packets.

When a cross-mesh packet needs to be transmitted, the relevant functions in the lad engine set the value of `cross-mesh-flag`, a variable defined in uip, to 1. Sicslowpan then uses this value to check if the `PACKETBUF_ATTR_GLOBAL_BROADCAST` flag should be set to `PACKETBUF_ATTR_GLOBAL_BROADCAST_TRUE` or `PACKETBUF_ATTR_GLOBAL_BROADCAST_FALSE`. When the framer sees that the flag is set to `PACKETBUF_ATTR_GLOBAL_BROADCAST_TRUE`, the global broadcast PAN ID is placed into the packet.

Radio Delay

When two nodes start sending at the same time, this may cause a collision: the receiving node is not able to distinguish the correct message. To decrease the probability of collisions, we have added a random delay before sending to our implementation. This delay is implemented in the LAD engine and works with the non-blocking `ctimer` delay function in Contiki. The engine will randomly allocate a transmission slot between 1 and 8 of its upcoming slots in which a transmission is allowed.

IPv6 Multicast Messages

With our implementation, IPv6 multicast messages can only be sent with the Flooding and 2-PAN Flooding mechanisms. Hybrid and Routing-Twice are not designed for sending multicast messages, since they can

only route a packet to a specific destination. For this reason, attempting to send a multicast message using one of the latter two mechanisms will trigger a warning message in the LAD engine.

Linux Socket

As the FutUrLight project describes a use case where Contiki is run from within Linux, the implementation is designed with interfacing with Linux in mind. For this reason, the standard packet output functions in Contiki are not implemented in the cross-mesh engine. This means that it is not possible to generate cross-mesh packets from within Contiki applications. A packet can be created by utilising the Linux socket API, and can be sent to the network over a tunnel interface. This enables the usage of standard networking applications on Linux, that can interface with the mesh network created by Contiki.

For testing purposes, we have created multiple example socket applications. These applications utilise the Linux socket API to create and send ipv6 packets with dummy data into the desired tunnel interface. All hop-by-hop option values relevant to our solution can be passed on as parameters during execution. In addition to UDP and TCP sockets, to be able to fully control the packet contents, a socket with the `RAW_SOCKET` option is created, in which the whole packet is crafted from scratch. However, it is only possible to send UDP packets with this raw socket.

Extending Current Functionality

Developers using our implementation may want to extend it with additional functions.

One possible extension is the option to route TCP traffic cross-mesh, using the Routing-Twice option. To facilitate this extension, our implementation remembers which node in the second PAN is the first to receive the message in the Destination ID field of the hop-by-hop option. The TCP module can then send its response back through this node to the sending PAN, where, again, the first node across the PAN border is saved.

A similar extension could be to send ICMP messages across PAN boundaries. To do this, the currently existing module responsible for sending ICMP messages, `uip-icmp6.c`, would have to be adapted to include a hop-by-hop extension header with our options. The cross-mesh engine will then forward the packet using the chosen option.

By defining an abstract cross-mesh engine, we give developers the opportunity to create additional cross-mesh engines with their own behaviour. To create a cross-mesh engine, it must first be defined and included by adding macros to `uip-cross-mesh-engines.h`. Next, the new cross-mesh engine must implement the `init()`, `out()` and `in()` functions, which will be called upon for initialisation, sending, and receiving packets, respectively. (N.B. The `in()` function also handles packet retransmissions.) In the current implementation, a packet will only be processed by the cross-mesh engine if it contains a hop-by-hop extension header with a type field set to `0100xxxx` and a length field set to `4`. If another condition for forwarding to the cross-mesh engine is desired, this needs to be changed in `uip6.c`.

Increase in Memory Footprint

Although there is no requirement to keep the compiled code as small as possible, we have looked at the increase in size caused by adding our implementation, because Contiki is designed for limited devices. To do this, we have compiled the `usb-border-router` program created by SOWNet, both with, and without cross-mesh functionality enabled. Afterwards, we have used the `size` program to determine the ROM and RAM used by the compiled program. The results are given in Table 4.1.

	Required ROM	Required RAM
Cross-mesh disabled	230889 bytes	41968 bytes
Cross-mesh enabled	235844 bytes	43832 bytes
Difference	4955 bytes	1864 bytes

Table 4.1: Comparison of the ROM and RAM usage of Contiki with cross-mesh functions enabled and with cross-mesh functions disabled.

Testing

To show the correctness and robustness of the code, the implementation has repeatedly been tested by functional testing and by running existing Contiki regression tests. Additionally, unit tests have been written for

the LAD engine with the use of the Cmocka unit testing framework, with which a line coverage of 98.85% has been achieved. To be able to carefully control the function calls and parameters in the LAD engine, mock files have been created for all files included by LAD. The unit tests can be found in `/contiki/tests/`. Version 1.1.1 of the Cmocka framework is included in the test folder, from where it can be installed before executing the tests. Coverage is calculated with Gcov and the `-fprofile-arcs` and `-ftest-coverage` compilation flags of Gcc. We have also tested our code on hardware, running Contiki on two desktops with the ST-6LP01 module attached for radio communication. We provide more information on hardware testing in Appendix E.

SIG Feedback

On June 1, 2017, after two and a half weeks of programming, we uploaded our code to the Software Improvement Group (SIG) for static analysis. Because much of the code we uploaded was not written by us, we included a list specifying which files should be analysed. After two weeks we received their feedback, which indicated that we had a module with a long function that could be split up into smaller ones. Based on this feedback, we have improved our code by moving code that addresses a subproblem into a separate function. The full SIG feedback is listed in Appendix F.

5 Performance

The four mechanisms (Flooding, 2-PAN Flooding, Hybrid and Routing-Twice) each have their own way of transporting a packet from A to B. Because their routing behaviour is so diverse, their impact on the network is very different. To inspect the behaviour and performance of the described approaches, we ran simulations.

Proof of Concept

As a proof of concept, we verify that cross-mesh communication can indeed provide the desired functionality. Figure 5.1 shows the simulation overview, which is based on Figure 1.2 from Chapter 1. As the source and destination nodes are in each other's radio range, the four cross-mesh algorithms all deliver the message with equal speed and effectiveness. Using those direct cross-mesh communication methods, the average delivery time is 25.4 milliseconds. When using default 6LoWPAN routing, two hops in the blue PAN and three hops in the red PAN are required between the nodes and their corresponding edge routers. This results in an average delivery time of 63.2 milliseconds, without taking the Internet transfer into account — we assume that the Internet connection adds no considerable delay to the transmission time (see Chapter 1). This experiment demonstrates that direct cross-mesh communication works and can be faster than normal routing. However, this example is unsuitable to compare the four different mechanisms, because the sending and receiving nodes are in each others radio range. Therefore, we create another simulation to stress the differences between them.

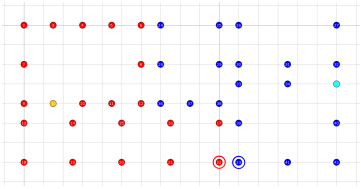


Figure 5.1: Simulation of the street map presented in Chapter 1. The cyan and orange nodes are border routers, the circled blue node is the sender and the circled red node is the receiver.

Simulation Settings

Simulations are done in Cooja, Contiki's built-in simulator. The values used in the IPv6 header and the hop-by-hop extension header are listed in Table 5.1.

The simulations provide insight in network behaviour. However, Cooja emulates different hardware (including the radio module) compared to the street light nodes. Especially the measured radio activity could differ between Cooja simulations and a network running on real hardware.

The nodes are separated by 25 metres to follow the Dutch convention for urban areas, have a transmission range of 50 metres, and an interference range of 100 metres. Figure 5.2 shows a visualisation of the transmission (green) and interference (grey) range of the node with label 10. The radio transmission is simulated using Unit Disk Graph Medium (UDGM) with distance loss. UDGM utilises a probabilistic transmission failure and interference model where a transmit success ratio and a receive success ratio can be set. In our simulation, both ratios are set to 100%.

The full test bed is shown in Figure 5.3. Blue nodes form the first PAN with the green border router; the red nodes form the second PAN with the orange border router. Messages are sent from node 1 to 31.

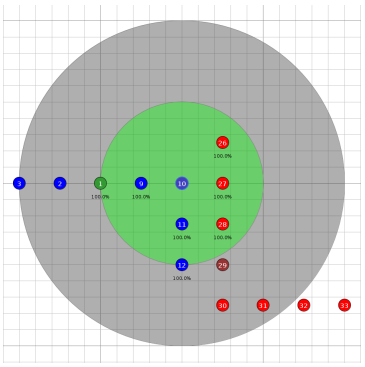


Figure 5.2: The nodes have a transmission range (green) of 50 meters and and an interference range (grey) of 100 metres.

	Flooding	2-PAN Flooding	Hybrid	Routing-Twice
IPv6 header: Hop Limit	8	5	255	255
HBHO: Hop Info	-	4	5	-
HBHO: Destination ID	-	0xbbbb	-	0x000c

Table 5.1: Parameters used in IPv6 packets during the simulations.

	Flooding	2-PAN Flooding	Hybrid	Routing-Twice
ContikiMAC without delay	46%	46%	48%	72%
ContikiMAC with delay	47%	50%	52%	78%
nullrdc without delay	14%	10%	25%	82%
nullrdc with delay	100%	98%	99%	98%

Table 5.2: Packet success rate (percentage of packet received by the destination node) when using different RDC drivers, with or without LAD's random delay.

When comparing different routing algorithms, a dummy packet must be included. As a quick delivery is needed — which is one of the key reasons for direct cross-mesh communication — UDP datagrams are the logical choice, which uses neither acknowledgements nor handshakes.

We have chosen to send 100 packets with an interval of 1500 milliseconds between transmissions. This should approximate a realistic setting where multiple packets are sent consecutively.

Radio Settings

Contiki offers multiple drivers to handle radio duty cycling (RDC). As we started experimenting with Cooja's example simulations and default settings, we used the ContikiMAC RDC driver, which implements radio duty cycles: the antenna is switched off and periodically switched on to see if there is radio activity. To maximize packet delivery, frames are sent multiple times after each other (so nodes waking up can receive them). For many IoT applications, RDC is useful to save energy, however, in the street light application described in Chapter 1, the radio antenna is assumed to play no significant role in the total energy consumption. Also, the nodes must be able to communicate with minimal delay. Therefore, we set the RDC driver to nullrdc, which is a simple pass-through layer: RDC is switched off. This means the radio antenna is kept on all the time.

To illustrate the effect on packet delivery, Table 5.2 shows the ContikiMAC and nullrdc drivers when sending cross-mesh packets, with and without the radio delay described in Chapter 4. ContikiMAC and its radio duty cycling has a dramatic negative effect on the packet success rate. We decided to run all further simulations with nullrdc, as this driver is used in the application that initiated this project.

Measurement Criteria

The following key performance indicators (KPIs) are used to assess the implemented approaches:

- **Success rate**: indicates if the message has arrived at the destination node at all;
- **Delivery time**: the number of milliseconds between the first packet transmission and the first packet arrival at its destination;
- **Transmissions**: the number of packet (re)transmission by all nodes;
- **Receiving nodes**: the number of nodes handling (i.e. receiving) the message;
- **Sending nodes**: the number of nodes (re)transmitting the message;
- **Packet duplication**: the number of times that nodes receives a message that they have handled before;
- **Packet duplication at destination**: the number of times the destination node receives a message that it has handled before.

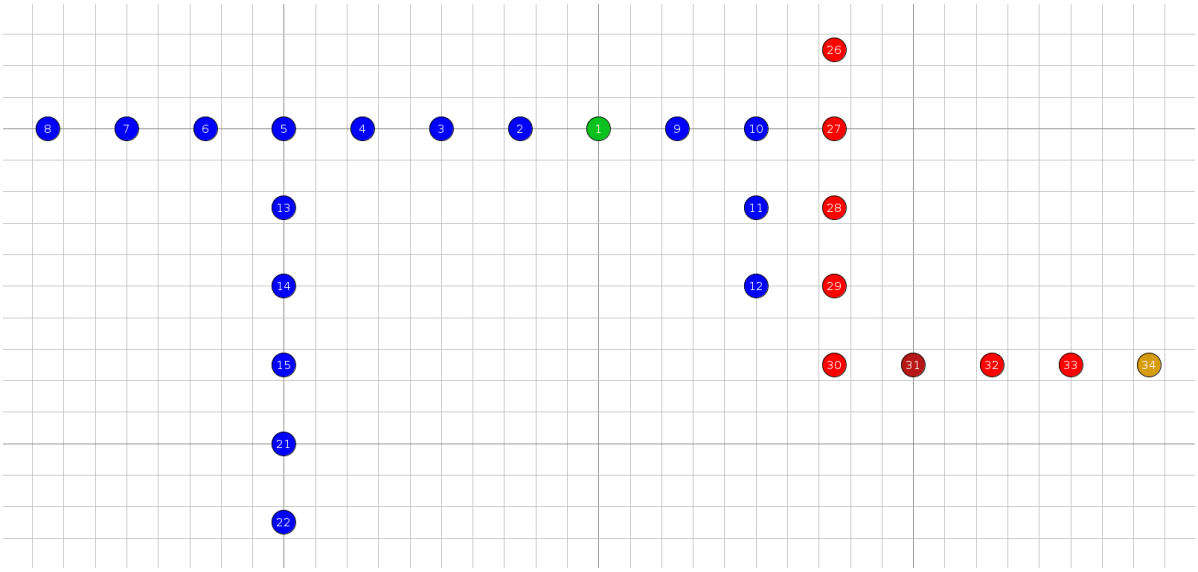


Figure 5.3: Testing scenario resembling a couple of streets. PAN A consists of the green (1, edge router and transmitting to 31) and blue nodes. PAN B consists of the orange (34, edge router), red, and dark red (31, receiving from 1) nodes.

Expected Results

It should take about 5 hops to get a message to the destination node (31), including the original sender (1) (see Figure 5.3). Therefore, in all packets that are received by the destination node, one would expect a *sending nodes* count of at least 5.

Because Flooding is tested using a maximum of 8 hops (IPv6 TTL = 8), this would mean that potentially all nodes within 8 hops of the sending node, except the destination node, will retransmit a packet. These are practically all nodes in the scenario. RF collisions may result in packet loss, leading to less retransmitting nodes. However, as the reception of packets within the interference range (but not the transmission range) have a probability of successful delivery, this may implicate more retransmitting nodes.

Compared to Flooding, the 2-PAN Flooding should reduce the transmission counts, as the spread in both PANs is limited. Hybrid should lower the transmission count even further, as there will be no flooding in PAN B. Because PAN B is the smaller one of the two, this impact may be limited.

Routing-Twice is the most sophisticated of the mechanisms. It should follow the logical path from node 1 to node 31. This includes 6 (re)transmissions on its way. However, as the broadcast send out by the bridge maybe received by more than one node in PAN B, this number may be higher than 6.

Results

Table 5.3 shows the results of the Cooja simulations. All mechanisms perform nicely in this scenario, with all success rates above 98%. From the goal of preventing network congestion, Routing-Twice clearly outperforms the alternatives. As Routing-Twice relies on the default routing strategy in Contiki's RPL implementation, broadcasting only once per message (at the bridge node), the packet retransmission count is lower than the multi-broadcasting alternatives.

Compared to the other mechanisms, Flooding provides the quickest packet delivery. This is a logical consequence of its nature: all nodes, regardless of their PAN and their distance to the destination node, simply retransmit the packet (as long as the IPv6 Hop Limit exceeds one). However, almost all nodes in this scenario retransmit the packets, which could lead to serious network congestion.

The results match the expectations, although the difference between 2-PAN Flooding and Hybrid could be inflated in other scenarios where the destination PAN consists of more nodes.

	Flooding	**2-PAN Flooding**	**Hybrid**	**Routing-Twice**
Success rate	98.33%	99.33%	98.33%	99.33%
Average no. of transmissions	24.12	18.43	18.44	6.89
Arrived packets				
Delivery time	196.74 ms	219.25 ms	217.81 ms	238.89 ms
No. of receiving nodes	25.19	22.50	22.48	8.90
No. of transmitting nodes	24.17	18.44	18.47	6.90
Packet duplication	69.79	39.62	39.93	2.67
Packet duplication at destination node	2.76	0.87	0.90	0.88
Lost packets				
No. of receiving nodes	21.00	20.00	19.22	4.50
No. of transmitting nodes	20.83	16.50	15.89	4.00
Packet duplication	49.50	34.00	38.67	1.00

Table 5.3: The results of the simulations. All four mechanisms have been run three times, with 100 cross-mesh messages sent in each iteration.

6

Considerations

Technological Considerations

We consider here the main technological implications of our implementation: the consequences of enabling flooding, the use of the value range `0x21-0x24` for the hop-by-hop option types, packet space reduction caused by our implementation, and the remaining obstacles for its use in practice.

Firstly, flooding can be a dangerous broadcasting technique if it is not used carefully. Flooding messages with a relatively high hop limit can cause serious network congestion, hampering other traffic. This is why developers using our implementation need to take particular care to set the hop limit field to a reasonable value. Also, if malicious hackers gain access to a node, they can use flooding to shut the network down. In our implementation, there is no sanity check for the hop limits. In practice, this means that a cross-mesh flooding packet with a hop limit set to its maximum value (255) could spread through an entire city.

Next, we consider the consequences of using the values `0x21-0x24` for the hop-by-hop option types. A disadvantage here is the possibility that one or more of these values are used by a conflicting standard in the future. In this case, in order for our code to be compatible with the new standard, these values will have to be changed. Also, as mentioned in Chapter 3, the 2-PAN Flooding solution violates the IPv6 standard because the hop limit header field is not always decremented.

Another consideration is the reduction in remaining packet space. A hop-by-hop header adds 8 bytes to the IEEE 802.15.4 frame(s). This decreases the worst-case maximum UDP payload size to 62 bytes in one IEEE 802.15.4 frame. If fragmentation of packets is to be avoided, this reduced payload size should be taken into account.

Furthermore, we have not created a solution to solve the problems posed by the assumptions made in Chapter 3. This means that cross-mesh functionality in its current state is unlikely to work in situations where different PANs communicate over different radio channels and/or use different encryption keys.

Finally, when our solution is used in practice, a problem can occur when the IPv6 addresses of two nodes in a PAN end in the same 16 bits. Routing-Twice uses short IDs to address the bridge node, however, SOWNet does not use short IDs. We worked around this by treating the last 16 bits of a node's IPv6 address as a short ID. Assuming the last 16 bits are random, the probability of two addresses ending in the same 16 bits is not insignificant (50% in a network of 300 nodes). To use short IDs, the function `is_my_short_id()` in `lad.c` needs to be changed.

Ethical Considerations

Any new technology that impacts society on the scale at which the Internet of Things does needs to be treated cautiously. Taking the IoT to the streets will be able to save a lot of energy, but also creates safety vulnerabilities. In light of the smart city's potential for both beneficial and harmful societal effects, this section sheds light on the ethical side of the story.

On the bright side, turning city lights off when there is no traffic nearby can lead to considerable energy savings. Although traditional, power-consuming street lights are being replaced with more efficient LEDs, there may still be opportunity for gains here.

On the other hand, security vulnerabilities are among the main challenges for IoT devices, and may have far-reaching consequences if they do not come to light. A security breach could leave an entire city in the

dark. Therefore, it is vital that any public service IoT system is thoroughly tested before critical infrastructure comes to rely on it.

7 Conclusion

Results

We have implemented a solution for the problem of cross-mesh communication in Contiki, without affecting any existing functionality. Our solution makes it possible to choose between four types of routing, depending on the situation at hand. We have not solved the problems that occur when the PANs involved use different channels and/or encryption keys, but our implementation is extensible with additional features. Developers can create their own cross-mesh engines with different behaviour by implementing our cross-mesh driver interface.

We have used Cooja simulations and real hardware to test our solution, and it has been found to meet the requirements in both cases: duplicates are filtered and no considerable extra traffic or delays are introduced. Furthermore, we have written unit tests to test our code, attaining 98.85% line coverage, and used feedback from SIG to improve the quality of our code.

The simulations we have done in Cooja show that all routing mechanism achieve a success rate of at least 98% in our setup, in situations where the radio is always on. The average number of transmissions is considerably lower in the Routing-Twice solution, resulting in lower packet duplication, network congestion and processing overhead. Routing-Twice takes a little more time to reach the destination node, because, in contrast to the flooding-based solutions, it does not always find the most efficient route to the destination and depends on the application's choice of bridge node.

Looking at the simulation results, Routing-Twice is the solution with the least network overhead, but it does require that a suitable bridge node is known. Flooding achieves the lowest latency, but will result in considerable network traffic. For general unicast messaging, we advise using the routing twice mechanism, if enough information is known about the network. However, in critical situations, a flooding-based approach might be more suitable.

Learning Experience

Over the course of the project, we have acquired many new skills and experiences.

Primarily, we have greatly increased our knowledge and understanding of standardised network protocols for low-power networks and of IPv6. The second year bachelor course Computer Networks had given us a basic idea of the different standards in existence, but only during this project have we truly driven deep into some them and so gained a solid grasp of their practical aspects.

Additionally, doing the project in a company has allowed us the opportunity to experience what it is like to work full-time in a business, rather than an academic setting. Because we worked at the company's office, we also got to look at the development process from behind the scenes. Within SOWNet, supervision consisted of weekly meetings with our supervisor, during which we reported our progress and considered how to proceed.

Furthermore, reflecting on our Scrum-based process, we find that the model of weekly sprints has been a useful way to organise our work. Discussing what had to be done and dividing tasks once a week made it clear to everyone who was doing what.

8

Recommendations

As mentioned in Chapter 1, we have made simplifying assumptions while designing and implementing our solution(s) for enabling cross-mesh communication in Contiki:

1. The sending node has all the relevant information it needs to send a message.
2. All PANs communicate using the same link layer encryption key or encryption is turned off.
3. All PANs communicate over the same IEEE 802.15.4 channel.

Anyone building on our solution in practice must keep these assumptions in mind and decide how to solve the problems we have left unsolved.

Firstly, developers using our solution need to ensure that nodes sending messages have all the required information to do so. This means that nodes need to know not only such information as destination addresses, short IDs, and PAN IDs, but also need to be able to make a choice between the four available types of cross-mesh communication.

Additionally, if our solution is going to be run on a Linux system with the ability to send TCP responses with our cross-mesh routing-twice option, the Linux socket on the receiving end needs to be enabled to receive hop-by-hop extension headers. In this socket, the hop-by-hop option can then be used to create the response packet. It is, however, important to note that packets containing unknown hop-by-hop options are currently (v4.12-rc6 at the time of this writing) always dropped by the Linux kernel. This means that a custom kernel is required in which the implementation of `/net/ipv6/exthdrs.c` is slightly modified to do one of the following:

- Abide by the IPv6 standard by looking at the first 2 bits of the unknown option type to decide what to do with the packet.
- Recognise the hop-by-hop options defined in this report.
- Never drop any packets with unknown hop-by-hop options.

Next, although using a single 128 bit key for all networks is likely safe from brute force attacks, we recommend implementing a stronger cryptographic scheme. If the key is compromised, all networks using this key become vulnerable. This problem can be solved by using nodes that have multiple encryption keys. In this case, a default key can be used for the initialisation phase, after which different PANs use different keys. In this situation, nodes send broadcast messages with the default key, and messages within their PAN with the PAN-specific key. Other solutions need to be devised in case only one key is available in each node.

Furthermore, in practice not all PANs may use the same channel, or may change channels frequently. To overcome this difficulty, we propose two strategies. A straightforward way to send a message to another PAN that is on a different channel, is to simply send the message on all channels, and hope that potential receivers listen to the right channel at the right time. If the broadcast is successful, and the node receives an acknowledgement, the channel on which it was received can be tried first the next time a broadcast must be sent. A more complex solution would be to distribute channel schedule information via the cloud to

nodes that need to send to other PANs, and coordinate channel switching in this way. Both strategies have the drawback of generating (possibly too much) extra overhead, so more effective solutions may need to be devised.

A

ST-6LP01

The ST-6LP01 radio module has 10 pins with which it can be connected to other devices, and has two communication methods; USB and UART. While communicating over USB creates more code overhead, it is considerably faster - Up to 12Mb over USB vs 1Mb available through UART. The board pin-out can be found in Figure A.1 and their description in Table A.1.

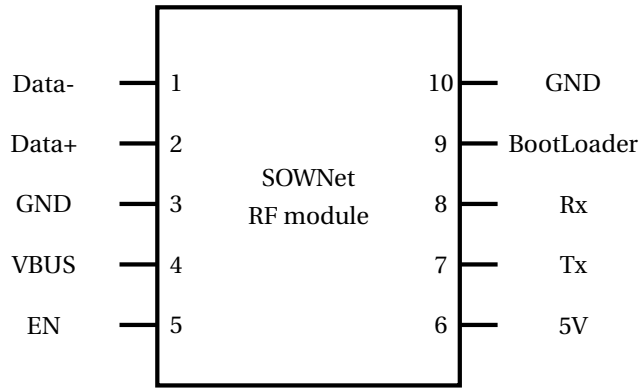


Figure A.1: Pin-out of the SOWNet radio module.

Terminal Name	No.	I/O	Description
5V	6	PWR	5Vdc power supply
GND	3,10	PWR	0V Ground input
VBUS	5	PWR	USB VBUS power supply (only for monitoring purposes)
D+	2	I/O	USB Data signal plus
D-	1	I/O	USB Data signal minus
EN	4	I	Enable Pin. Used to reset the module (active high)
Tx	7	O	Transmit asynchronous data output
Rx	8	I	Receive asynchronous data output
BootLoader	9	I	Bootloader pin. This pin must be held high during reset for entering programming modus

Table A.1: Terminal functions of the ST-6LP01 radio module

B

6LoWPAN

The 6LoWPAN (*IPv6 over Low-power Wireless Personal Area Networks*) protocol connects the IPv6 and IEEE 802.15.4 standards [12]. Concerning the network layer, every device supporting IPv6 should have a Maximum Transmission Unit (MTU) of at least 1280 bytes. In contrast, the 802.15.4 frames have a length of 127 bytes. As the 802.15.4 header takes up to 25 bytes, and the strongest provided encryption at the data link layer adds an overhead of 21 bytes, only 81 bytes are left for higher layers. 6LoWPAN provides a bridge between those layers: outgoing IPv6 packets are split up into 802.15.4 frames, and incoming frames are assembled back to IPv6 packets. To make optimal use of the frames, 6LoWPAN compresses packets from higher layers by eliding information, which is later reinserted on the receiving end.

6LoWPAN adapts the IPv6 standard in a low-power wireless paradigm: unnecessary packet transmission, and therefore power consumption, is avoided as much as possible. The IPv6 Neighbor Discovery Protocol (NDP) is adapted in this mindset: the 6LoWPAN NDP avoids multicast-based address resolution and is based on host-initiated interaction to allow sleeping nodes.

A 6LoWPAN packet header starts with a dispatch byte: an 8-bit sequence that describes the packet type. The 2 most-significant bits are used as quick identifiers, as described in Table B.1a. Table B.1b lists the standardised dispatch bytes. Depending on the dispatch type, additional data is expected, such as fragment information or compression type. Further fields include the following: addressing, hop-by-hop processing, destination processing, and payload.

Bits	Packet type
00	Not a LoWPAN packet
01	Normal dispatch
10	Mesh header
11	Fragmentation header

Dispatch	Header type
00 xxxxxx	Not a LoWPAN packet
01 000001	Uncompressed IPv6 Addresses
01 000011	LOWPAN_DFF
01 010000	LOWPAN_BC0
01 1xxxxx	LOWPAN_IPHC
10 xxxxxx	Mesh header
10 0xxxxx	*... for Critical 6LoWPAN Routing Headers*
10 1xxxxx	*... for Elective 6LoWPAN Routing Headers*
11 000xxx	First fragmentation header
01 100xxx	Subsequent fragmentation headers
11 11xxxx	Page switch

Table B.1: a: (*left*) The first 2 bits at the start of a dispatch, describing the general type. b: (*right*) List of dispatch type bytes currently in use.[1]

The 6LoWPAN header also contains source and destination addresses, and, depending on the compression type, additional IPv6 and UDP header fields. The 16-bit short or 64-bit EUI addresses used by IEEE 802.15.4 need to be converted to IPv6 compatible identifiers, which can be done efficiently in a stateless manner with HC1 header compression[11] by setting requirements on the IPv6 addresses of the nodes. These IPv6 addresses are formed by combining a known IPv6 prefix with the EUI-64 identifier of the device, meaning that only EUI-64 identifiers are required to be able to communicate with other nodes. This is made even

more efficient by introducing 16-bit short identifiers, unique for every node within a PAN. This way, only 32 bits are used to uniquely identify a node: 16 bits for the identifier of the PAN (PAN ID) and 16 bits for the node identifier.

As shown in Table B.1, 6LoWPAN defines the broadcast dispatch type `LOWPAN_BC0`. The dispatch byte is followed by a supporting byte for the multicast mechanism. The simplest mechanism is *flooding*, where nodes simply echo the broadcast packet. With flooding, the supporting byte is used to indicate a packet's *sequence number*. In this way, a node does not need to store an entire broadcast packet, as the sequence number is enough to prevent multiple retransmissions of the same packet.

This broadcasting definition in the standard is rather limited. This may be the result of the low-power philosophy of 6LoWPAN: packet transmission is expensive in terms of power consumption, making broadcasting an inconvenient approach. When a broadcasting technique requires more than one supporting byte, a different routing header can be defined with new dispatch type [12]. We have not used this broadcast dispatch type in our implementation.

[1] A full list can be found at `https://www.iana.org/assignments/6lowpan-parameters/6lowpan-parameters.xhtml`

C

IEEE 802.15.4

With the rise of the IoT, a protocol was needed to allow for robust communication over low-cost, low-power and lossy wireless devices. For this, the IEEE 802.15.4 standard was proposed [1]. It defines a "protocol and compatible interconnection for data communication devices using low-data-rate, low-power, and low-complexity short-range radio frequency (RF) transmissions in a wireless personal area network (WPAN)". The standard includes frame structure models, network topologies, channel access mechanisms and security measures to ensure robust communication.

Addressing within the protocol can be done in two ways. Every device has a 64-bit unique EUI identifier with which a node can be directly addressed. In addition, 16-bit short addresses are assigned to every node within a PAN by the PAN coordinator. These short identifiers are unique within a PAN, but an additional 16-bit PAN ID is needed to distinguish nodes from those in other PANs. The protocol also permits broadcasting by using the broadcast address `0xFFFF` for both the short address and the PAN ID. Broadcast address can also refer to the 64-bit broadcast MAC address which is defined in IEEE Std 802-2014 (`0xFFFF FFFF FFFF FFFF`).

As the number of connected devices within a network increases, setting up the network becomes a difficult task. For this reason, a bootstrap process has been proposed to allow autonomous node and network configuration. During this process, information such as network identifiers and encryption keys is shared between devices according to the diagram in Figure C.1. After this information exchange, a PAN consisting of 6LoWPAN edge (border) routers (6LBRs), reduced-function devices (RFDs) and full-function devices (FFDs) is created [7].

In specific, the bootstrapping protocol is initiated by 6LBRs. These send beaconing messages to the first-hop FFDs in their RF range. The first-hop FFDs then associate with the 6LBR and authenticate to it. The 6LBR checks the authentication with an external trust center (TC) and upon successful validation receives an IP address and a network key. Further bootstrapping of other FFDs and RFDs happens analogously with configured FFDs relaying between the 6LBR and the unconfigured FFDs or RFDs [7].

On the side of security, the protocol enables usage of AES on the link layer for encryption and 32, 64, or 128 bit message authentication codes (MACs) [1]. It provides options for data authenticity (AES-CBC), data encryption (AES-CTR) or both (AES-CCM). The strongest setting, AES-CCM with a 128-bit key, poses an overhead of 21 bytes per packet.

A problem with using RF, especially on generally used frequencies such as 2.4 GHz, is that other wireless networks can interfere with data transmission. To alleviate this problem, a range of frequencies is used instead of a single frequency. This range is then divided into channels, where each channel is a subrange of the full frequency spectrum. The IEEE 802.15.4 standard divides the 2.4 GHz - 2.4835 GHz range into 16 channels, and offers mechanisms to utilize these efficiently, such as channel adaptation and channel hopping [1].

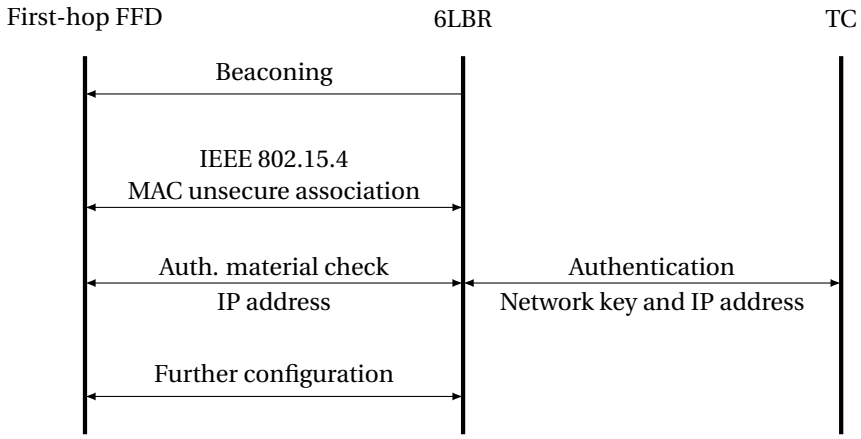


Figure C.1: Schematic depiction of the first phase of the security bootstrapping mechanism. The second and third phase happen analogously.

D

Contiki

Contiki (`http://www.contiki-os.org/`) is an operating system (OS) designed for IoT, aimed to be run on low-power, low-memory devices. Created in 2002 and written in the C programming language, Contiki supports fully standard IPv4 and IPv6 and has implementations of standard low-power wireless standards such as 6LoWPAN, RPL and CoAP.

In order to achieve low power consumption, Contiki makes use of so-called "sleepy routers". Contiki implements a radio duty cycling system, allowing routers to sleep between relayed messages, thus making it possible to have battery powered routers.

Network Call Stack

`NETSTACK` is Contiki's network stack, splitting the network responsibilities into five driver modules: `NETSTACK_NETWORK`, `NETSTACK_LLSEC`, `NETSTACK_MAC`, `NETSTACK_RDC` and `NETWORK_RADIO`. Each driver points to a C program (its implementation) and covers a part of the OSI model. Additionally, the `NETSTACK_FRAMER` is not a driver, but consists of auxiliary functions concerning the frame creation (when transmitting) and parsing (when receiving). Figure D.1 provides an overview of the network stack, of which all drivers use the packet buffer. This section describes the driver responsibilities and connections in Contiki's `NETSTACK` setting for IPv6-oriented communication. [1]

Packet Buffer and Queue

The `NETSTACK` drivers communicate with each other using the *packet buffer* and *packet queue*. To minimise the memory footprint, the buffer is singular, therefore simultaneous packet handling is not possible.

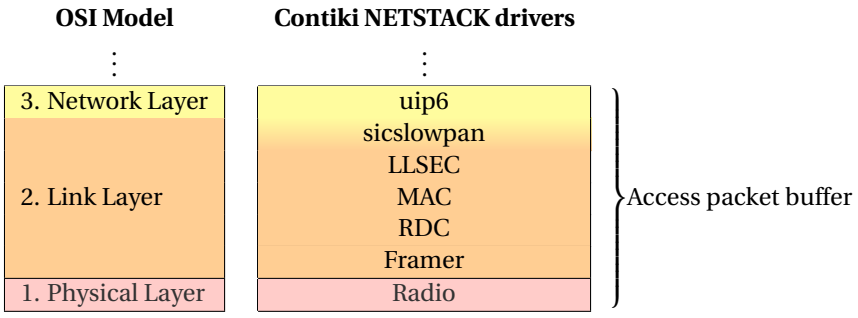


Figure D.1: Overview of the Contiki network stack (NETSTACK)

[1] Further reading on `NETSTACK` and its MAC subdrivers: `http://anrg.usc.edu/contiki/index.php/MAC_protocols_in_ContikiOS`

Network Driver

The network driver is used as an interface by other applications. When using 6LoWPAN communication, the `sicslowpan driver` uses its `input` function pointer as a callback for incoming messages.

LLSEC Driver

The Link Layer Security (LLSEC) driver implements encryption following the IEEE 802.15.4 specification. Currently, Contiki only offers the `noncoresec driver`, which uses a network-wide encryption key. [2] When encryption on the link layer is not required, the `nullsec driver` simply passes through packets.

MAC Driver

The `nullmac driver` simply passes through packets; the `csma driver` implements addressing, sequence numbers and retransmissions.

Radio Duty Cycling Driver

The Radio Duty Cycling (RDC) layer is responsible for the sleeping time of the node, possibly switching off the RF antenna periodically. Contiki offers several RDC drivers, of which some do not use energy savings, such as the default for 6LoWPAN communication (`sicslowmac driver`). Other implementations do switch the antenna on/off for energy saving, such as the `(C)X-MAC driver`, its descendant (`ContikiMAC driver`), and the `LPP driver`. An important setting for these drivers is the *channel check rate* which specifies the number of channel checks per second.

Framer

The appropriate network framer is the `framer-802154` and offers two auxiliary functions: `create` to make a 802.15.4-frame with the provided data, and `parse` to process an incoming frame. [3]

[2] Further reading on `noncoresec`: `https://github.com/contiki-os/contiki/tree/master/core/net/llsec/noncoresec`

[3] The 802.15.4 framer is implemented in `core/net/mac/framer-802154.c` . Notice that the `parse` function is used to filter out messages for other PANs on line 222.

E

Hardware setup

For the testing of the code on hardware, a ST-6LP01 radio module (see Appendix A) is connected to a Linux computer over USB. On the side of the radio module, a USB cable is connected to pins 1 through 4 and pin 4 (VBUS) is bridged with the pin 6 (5V) to be able to power the module from the computer's USB port.

On the computer, Contiki is built natively with the use of a `usb border router` mote provided by SOWNet, based on the already existing `native border router` example in Contiki. This mote enables interconnection between Contiki and an external radio module through SLIP, the Serial Line Internet Protocol. A tunnel interface is opened and SLIP-encapsulated IPv6 packets are sent to Contiki through the Linux sockets API.

Finally, client and server application are created in Linux with the Linux sockets API with options to send standard TCP and UDP packets, and TCP and UDP packets with hop-by-hop extension headers.

F

SIG Feedback

First Feedback

De code van het systeem scoort 3 sterren op ons onderhoudbaarheidsmodel, wat betekent dat de code gemiddeld onderhoudbaar is. De hoogste score is niet behaald door lagere scores voor Duplication en Unit Complexity.

Zoals besproken werken jullie aan een uitbreiding van een bestaand systeem. Dat heeft als gevolg dat jullie voor een deel worden beoordeeld op basis van code die in eerste instantie door iemand anders is geschreven, maar vervolgens later door jullie is aangepast. Jullie zijn in dat opzicht dus in het nadeel ten opzichte van andere groepen die al hun code voor 100 procent zelf geschreven. Dat gezegd hebbend is jullie situatie natuurlijk wel realistischer, in het algemeen zal het vaak voorkomen dat je vanuit al bestaande code moet werken.

Voor Duplication wordt er gekeken naar het percentage van de code welke redundant is, oftewel de code die meerdere keren in het systeem voorkomt en in principe verwijderd zou kunnen worden. Vanuit het oogpunt van onderhoudbaarheid is het wenselijk om een laag percentage redundantie te hebben omdat aanpassingen aan deze stukken code doorgaans op meerdere plaatsen moet gebeuren.

In dit systeem is er bijvoorbeeld duplicatie te vinden in uip.h (waar de IP6 header steeds weer opnieuw voorkomt), en uip6.c (waar de switch op basis van ext_hdr_options_process() meerdere keren voorkomt). Het is aan te raden om dit soort duplicaten op te sporen en te verwijderen.

Voor Unit Complexity wordt er gekeken naar het percentage code dat bovengemiddeld complex is. Het opsplitsen van dit soort methodes in kleinere stukken zorgt ervoor dat elk onderdeel makkelijker te begrijpen, makkelijker te testen is en daardoor eenvoudiger te onderhouden wordt. Binnen de extreem lange methodes in dit systeem, zoals bijvoorbeeld de 'uip_process'-methode in uip6.c, zijn aparte stukken functionaliteit te vinden welke ge-refactored kunnen worden naar aparte methodes. Commentaarregels zoals bijvoorbeeld 'This is where the input processing starts' zijn een goede indicatie dat er een autonoom stuk functionaliteit te ontdekken is. Door elk van deze functionaliteiten onder te brengen in een aparte methode met een descriptieve naam kan elk van de onderdelen apart getest worden en wordt de overall flow van de methode makkelijker te begrijpen.

Bij al deze voorbeelden is het aannemelijk dat deze onderhoudbaarheidsproblemen er al inzaten voordat jullie uberhaupt aan het project begonnen. Het is hier dus de zaak om te kijken of het lukt om de oorspronkelijke code enigszins te refactoren, zodat jullie niet alleen nieuwe functionaliteit hebben toegevoegd, maar ook de technische kwaliteit van de code beter hebben achtergelaten. Hopelijk lukt dit nog tijdens de rest van de ontwikkelfase.

Als we alleen naar de bestanden kijken die jullie zelf hebben geschreven wordt het verhaal inderdaad iets anders. We zien echter nog steeds een aantal voorbeelden waarbij de Unit Complexity nog verbeterd kan worden.

Een van die voorbeelden is in_routing_twice in lad.c. Het commentaar boven de for-loop, "check if one of our ip addresses is the bridge node address", geeft aan dat de inhoud van deze for-loop eigenlijk een apart deelprobleem oplost. Als je dit uitsplitst wordt je code makkelijker te begrijpen, wat met name van belang is als de hoeveelheid functionaliteit in dit bestand op een later moment gaat groeien. Daarnaast wordt het zo makkelijker om onderdelen in de toekomst te hergebruiken, en tot slot kun je de code zo ook makkelijker testbaar maken.

G

Original Project Description

Project description

SOWNet heeft een eigen hardware module ontwikkeld voor een Contiki implementatie van het 6LoWPAN protocol. Ondanks het feit dat er een werkende versie van deze implementatie beschikbaar is, werkt het geheel nog niet naar tevredenheid. Zo zijn er een paar modules en functionaliteiten van het protocol die niet goed werken en/of niet geïmplementeerd zijn. Het doel van deze opdracht is het onderzoeken en verbeteren van huidige Contiki implementatie. Hierbij kan gedacht worden aan "multi homing", "Multi edge routers" en/of communicatie tussen nodes in verschillende 6LoWPAN cellen.

Company description

SOWNet is een innovatief en dynamisch bedrijf gespecialiseerd in de ontwikkeling van hard- en software op het gebied van Mesh Netwerken. SOWNet is 10 jaar geleden ontstaan als een Spin-Off van het onderzoeksinstituut TNO en telt op dit moment 5 engineers in electronica en informatica.

Auxiliary information

SOWNet is op zoek naar studenten die over een bepaalde mate van zelfstandigheid beschikken en bereid zijn zich in te zetten voor het oplossen van actuele problemen vanuit de markt. Daar tegenover biedt SOWNet ondersteuning en begeleiding van professionals op dit gebied alsook een vergoeding van 250,00 per maand.

H

Infosheet

Cross-mesh Communication in Contiki OS

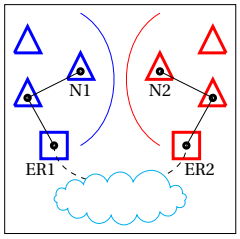
Client: The project was carried out at SOWNet Technologies, an Internet of Things company situated in Pijnacker, The Netherlands.
Final Presentation: July 3, 2017 16:00.
Assignment: The problem we solved was to implement direct communication between devices in two different Personal Area Networks (PANs) in Contiki OS. The problematic and the desired behaviour can be seen in Figure H.1.
Process: We used weekly sprints to steer our development process, as our solution kept evolving throughout the project.
Research: During the research phase we have looked at the standards of the protocols used and at the internal code of Contiki. This gave us a good idea of how communication in Contiki is currently done and how it could be extended.
Product: We have delivered a product that solves the problem, allowing developers to choose between four different types of cross-mesh communication.
Verification: The solution has been tested with unit tests, with Contiki's built-in simulator, Cooja, and it has been tested on hardware. We have documented the still unsolved problems outside of the scope of our project that need to be resolved before our solution can effectively be used in practice.

Team members

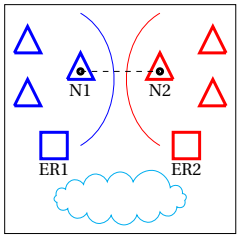
Leendert van Doorn
Interests: Cyber Security, Natural Language Processing
Major contributions: LAD Engine development, Report
Ahmet Güdek
Interests: Embedded Software, Networking, Cyber Security
Major contributions: Socket development, Hardware and unit testing
Daan van der Valk
Interests: Algorithm Design, Networking, Cyber Security
Major contributions: LAD Engine development, Simulation design and scripting

The final report for this project can be found at:
`http://repository.tudelft.nl`

Client Supervisor: Winelis Kavelaars SOWNet Technologies.
TU Delft Supervisor: Koen Langendoen Embedded Software group.
Contacts
Leendert van Doorn leendertvdoorn@gmail.com
Ahmet Güdek agudek95@gmail.com
Daan van der Valk daanvdvalk@gmail.com



(a) Communication via edge routers



(b) Our solution: direct communication

Figure H.1: The current (a) and desired behaviour (b) of cross-mesh communication in Contiki.)

Bibliography

[1] IEEE Standard for Low-Rate Wireless Networks. *IEEE Std 802.15.4-2015 (Revision of IEEE Std 802.15.4-2011)*, pages 1–709, April 2016.

[2] L. Atzori, A. Iera, and G. Morabito. The internet of things: A survey. *Computer Networks*, 54(15):2787–2805, 2010.

[3] D. Daza, R. G. Carvajal, J. Misic, and A. Guerrero. Street lighting network formation mechanism based on ieee 802.15.4. In *2011 IEEE Eighth International Conference on Mobile Ad-Hoc and Sensor Systems*, pages 164–166, Oct 2011. doi: 10.1109/MASS.2011.133.

[4] Stephen Deering and Robert Hinden. Internet Protocol, Version 6 (IPv6) Specification. RFC 2460, IETF, December 1998. URL `https://www.ietf.org/rfc/rfc2460.txt`.

[5] P. Elejoste, I. Angulo, A. Perallos, A. Chertudi, I. J. G. Zuazola, A. Moreno, L. Azpilicueta, J. J. Astrain, F. Falcone, and J. Villadangos. An easy to deploy street light control system based on wireless communication and led technology. *Sensors (Switzerland)*, 13(5):6492–6523, 2013.

[6] Rudolf Giffinger, Christian Fertner, Hans Kramar, Robert Kalasek, Nataša Pichler-Milanovic, and Evert Meijers. *Smart cities – Ranking of European medium-sized cities*, 2007.

[7] Danping He and Behcet Sarikaya. Security bootstrapping of ieee 802.15.4 based internet of things. Internet-Draft draft-he-iot-security-bootstrapping-01, IETF Secretariat, May 2015. URL `http://www.ietf.org/internet-drafts/draft-he-iot-security-bootstrapping-01.txt`. `http://www.ietf.org/internet-drafts/draft-he-iot-security-bootstrapping-01.txt`.

[8] S. Jung, A. Chang, and M. Gerla. Comparisons of zigbee personal area network (pan) interconnection methods. In *2007 4th International Symposium on Wireless Communication Systems*, pages 337–341, Oct 2007. doi: 10.1109/ISWCS.2007.4392357.

[9] Rien Kort. Igov: Onderweg naar een zelfdenkende stad. *instALLICHT*, 2:27–28, 2016.

[10] Silicon Labs. Designing for multiple networks on a single zigbee chip. Application Notes AN724, 2014. URL `https://www.silabs.com/documents/public/application-notes/AN724.pdf`.

[11] Gabriel Montenegro, Nandakishore Kushalnagar, Jonathan W. Hui, and David E. Culler. Transmission of IPv6 Packets over IEEE 802.15.4 Networks. RFC 4944, IETF, September 2007. URL `https://tools.ietf.org/html/rfc4944`.

[12] Zach Shelby and Carsten Bormann. *6LoWPAN: The Wireless Embedded Internet*. John Wiley & Sons, Ltd, 2009. ISBN 9780470686218. doi: 10.1002/9780470686218.ch1. URL `http://dx.doi.org/10.1002/9780470686218`.

[13] A. Sittoni, D. Brunelli, D. Macii, P. Tosato, and D. Petri. Street lighting in smart cities: A simulation tool for the design of systems based on narrowband plc. In *2015 IEEE 1st International Smart Cities Conference, ISC2 2015*, 2015.

[14] Y. M. Yusoff, R. Rosli, M. U. Karnaluddin, and M. Samad. Towards smart street lighting system in malaysia. In *IEEE Symposium on Wireless Technology and Applications, ISWTA*, pages 301–305, 2013.