ETH zürich

ETH Library

Creating a Flexible Middleware for Low-Power Flooding Protocols

Master Thesis

Author(s):
Bächli, Jonas

Publication date:
2018-06-07

Permanent link:
https://doi.org/10.3929/ethz-b-000270388

Rights / license:
In Copyright - Non-Commercial Use Permitted

This page was generated automatically upon download from the ETH Zurich Research Collection.
For more information, please consult the Terms of use.

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institut für
Technische Informatik und
Kommunikationsnetze

Creating a Flexible Middleware for Low-Power Flooding Protocols

Master Thesis

Jonas Bächli

baechlij@ethz.ch

Computer Engineering and Networks Laboratory
Department of Information Technology and Electrical Engineering
ETH Zürich

Supervisors:
Romain Jacob
Reto Da Forno
Prof. Dr. Lothar Thiele

June 7, 2018

Acknowledgements

I thank my supervisors Romain Jacob and Reto Da Forno for the continuous support by giving invaluable input and feedback throughout the duration of this thesis.

Abstract

Low-power wireless networks allow for easy-deployment and low-maintenance solutions for sensor applications. Glossy, an often used communication primitive to build such low-power wireless networks, is a flooding protocol providing high reliability and resilience. Even though the list of protocols building on top of Glossy is extensive, the current approach to implement Glossy-based protocols could be significantly improved by separating low-level complexity from high-level protocol logic.

The Glossy Middleware presented in this thesis provides system designers with a well-defined, yet versatile interface. Building on top of a round-based framework, the Glossy Middleware offers a vast amount of control over important Glossy parameters on a high-level.

This report presents the design and features of the Glossy Middleware in detail, while also serving as an user manual. The evaluation displays the low performance overhead in terms of memory and energy usage as well as the high usability by reimplementing different existing protocols in addition to others, illustrating the various features of the middleware. The middleware has been designed with hardware portability in mind: currently it is fully supported on the DPP platform and to some extent on the TelosB platform.

This thesis concludes by illustrating possible improvements on the existing work while also outlining exiting possibilities of the developed Glossy Middleware.

Contents

Acknowledgements **i**

Abstract **ii**

1 Introduction **1**

1.1 Low-Power Wireless Networks 1

1.2 Glossy 1

1.3 Middleware 2

1.4 Related Work 3

2 The Glossy Middleware **4**

2.1 Requirements 4

2.2 Design 4

2.2.1 Round-Based Framework 4

2.2.2 State Machine 5

2.2.3 Callbacks 6

3 Implementation **9**

3.1 The Control Packet 9

3.1.1 The *Schedule* Section 10

3.1.2 The *Config* Section 12

3.1.3 The *Slot Config* Section 14

3.1.4 The *User Bytes* Section 15

3.2 Callbacks: The Nitty-gritty Part 15

3.2.1 The Post Control Slot Callback 15

3.2.2 The Pre Slot Callback 17

3.2.3 The Post Slot Callback 18

3.2.4 The Post Round Callback 19

3.2.5 The Bootstrap Timeout Callback 20
3.3 Main Interface 20
3.3.1 The GMW Start Function 20
3.3.2 The Set New Control Function 21
3.3.3 The Control Init Function 22
3.4 Additional Features 22
3.4.1 Drift Compensation 22
3.4.2 Statistics 23
3.4.3 Channel Hopping 24
3.5 Porting 24

4 Minimal Working Example 26
4.1 Include Files 26
4.2 Local Static Function Declarations 27
4.3 Local Variables 28
4.4 The Main Application Process 28
4.5 The Host Node Implementation 30
4.6 The Source Node Implementation 32
4.7 The Control Management Functions 34

5 Evaluation 36
5.1 Usability and Setup 37
5.1.1 GMW-Test 37
5.1.2 LWB Reimplementation 37
5.1.3 eLWB Reimplementation 40
5.2 Evaluation of the Performance Overhead 42
5.2.1 Binary Size 42
5.2.2 Maximum Stack Size 47
5.2.3 Duty-Cycle 48
5.2.4 Reliability 52

6 Conclusion and Future Work 56

Bibliography **58**

Chapter 1

Introduction

This chapter introduces low-power wireless networks in general and Glossy in particular as well as explaining the current state of Glossy-based protocols.

1.1 Low-Power Wireless Networks

The uprising of the Internet of Things (IoT) led to a significant amount of devices in need of connectivity, both in-between the devices itself as well as to the internet. Especially for sensor applications it is crucial to have devices that are both easy-to-deploy as well as low-maintenance. While Wireless Sensor Networks (WSN) have been around for a while now (starting out with military applications in the second half of the 20th century[3]), they were limited in terms of energy consumption and high cost. Major developments, such as the release of the IEEE 802.15.4[10] low-power wireless standard, led to significant improvements in wireless technology. Today, wireless devices can be powered for several years with a small battery, making low-power wireless a reality.

1.2 Glossy

Glossy[9] is a flooding communication primitive for wireless sensor networks. It relies on tightly synchronised concurrent transmissions to generate constructive interference. In the example of Glossy, this means that the same packet it transmitted with sub-microsecond accuracy from multiple nodes, leading to receivers receiving the superposition of those signals. This accuracy is achieved by using radio events to trigger the flooding procedure. Additionally, this tight synchronisation results in highly-accurate timestamps of a shared reference time between multiple nodes.

Figure 1.1 shows a simplified representation of a *Glossy flood*. The node starting or initiating a Glossy flood is called the *initiator*. In this example, the first transmission is received by neighbouring nodes in the 1-hop range. Other

nodes are 2-hops away and unable to receive the transmission from the initiator. After each phase, the roles are switched and nodes receiving a packet retransmit it while nodes that have previously transmitted a packet receive it again. In this example, the retransmission of the packet by 1-hop nodes is now able to be received by nodes 2 hops away. To terminate the flooding process, Glossy uses the *number of retransmissions*, in case of this example 3, visible by the 3 TX phases of each hop.

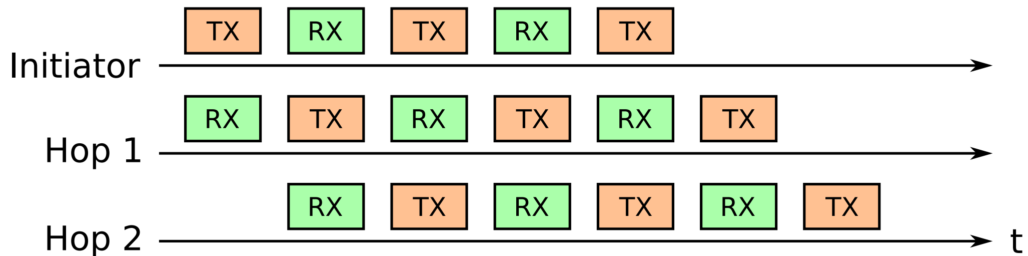

Figure 1.1: Glossy floods allow for reliable packet transmissions in complex network topologies.

The design of Glossy has a high reliability due to the multiple transmissions of the same packet. Increasing the number of retransmissions is a tradeoff between reliability and energy efficiency, allowing to parameterise Glossy according to the requirements. The flooding process also makes it highly-resilient to topology changes, since all nodes in the network will eventually receive the packet, as long as there exists a path from the initiator. Additionally, since every node in a Glossy network is able to reach every other node during a Glossy flood, a higher layer is able to abstract the network topology as a virtual single-hop network, which significantly simplifies the scheduling of communication time slots and therefore the development of higher-level protocols.

1.3 Middleware

The advantages offered by Glossy led to a multitude of protocols using it, such as LWB[8], Chaos[14], Splash[5], Crystal[12], DRP[13], eLWB[22], A^2[1] and many more. However, all these protocols integrate Glossy in their own way while often sharing a round-based approach to schedule Glossy floods. This means that each protocol has its own low-level implementation, interacting with Glossy, using hardware timers to schedule Glossy floods, etc., while at the same time also implementing higher-level functionality, i.e. the protocol logic. Therefore current protocol implementations are quite cumbersome and complex. The goal of this thesis is to simplify the implementation of protocols, like e.g. LWB, by encapsulating the low-level complexity into a separated and fixed middleware

layer, as illustrated in Figure 1.2. This would allow protocol implementations to be designed on a higher abstraction level. In other words, instead of having to micromanage each individual Glossy flood, protocols can now be designed with packets in mind.

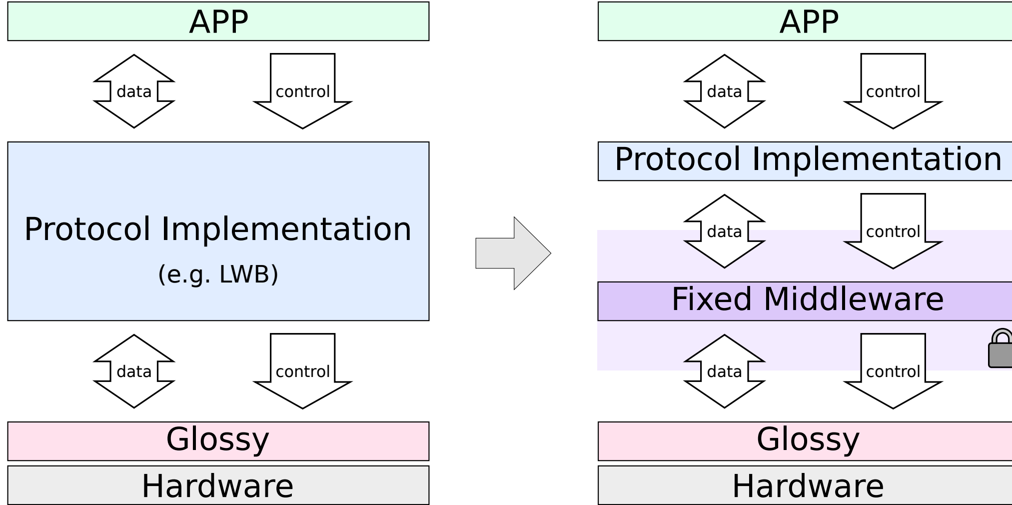

Figure 1.2: We propose to encapsulate low-level complexity common in current Glossy-based protocols, e.g. LWB, into a separated and fixed middleware layer with well-defined interfaces in order to simplify the implementation of such protocols.

1.4 Related Work

There have been extensive studies of middleware in the context of WSN in the past decade, demonstrating the need for WSN middleware to detach higher-level protocol design from the low-level reality of WSN applications ([18], [19], [24], [2]).

The recent work presenting the generic wireless consensus protocol A^2[1] also introduced Synchrotron, a middleware specifically created to accommodate the requirements of A^2. The key difference of Synchrotron to the work presented in this thesis is the use of Chaos[14] instead of Glossy. Chaos actually is a communication primitive derived from Glossy, however it relies only on the capture effect[15], therefore not providing the same performance guarantees as Glossy, especially in networks with high node density.

Chapter 2

The Glossy Middleware

This chapter introduces the *Glossy Middleware* (*GMW* for short), its main concepts and terms used throughout this thesis.

2.1 Requirements

The middleware should be able to serve as a platform to implement existing as well as new protocols. It should therefore reproduce existing principles (such as round-based communication rounds) and abstract them in a flexible manner. Being flexible in this case means to be able to easily adjust important parameters (e.g. regarding timing and Glossy itself) to meet the requirements of an application. Additionally the Glossy Middleware should be able to be integrated in existing code. Since most Glossy implementations are built on top of the Contiki[6] operating system, Contiki is a obvious choice. This project builds on the newly released Contiki-NG[4].

2.2 Design

2.2.1 Round-Based Framework

The Glossy Middleware is built upon a *round-based framework*, see Figure 2.1 for a visual representation. A *round* is a collection of *slots* being transmitted at a specific time. Each slot consists of one Glossy flood, its length (or slot time) is therefore adjusted to the duration of this particular Glossy flood. The first slot is always the *control slot* (marked red in Figure 2.1), in which the host sends the current control packet to all nodes. This control packet is managed by the host and contains the schedule as well as additional data which allow for simple parameter modification. The other slots are freely assignable by the user to individual nodes. Those slots will be referred to as *data slots* from now on. GMW also supports a special slot type, the *contention slot*. This is a slot

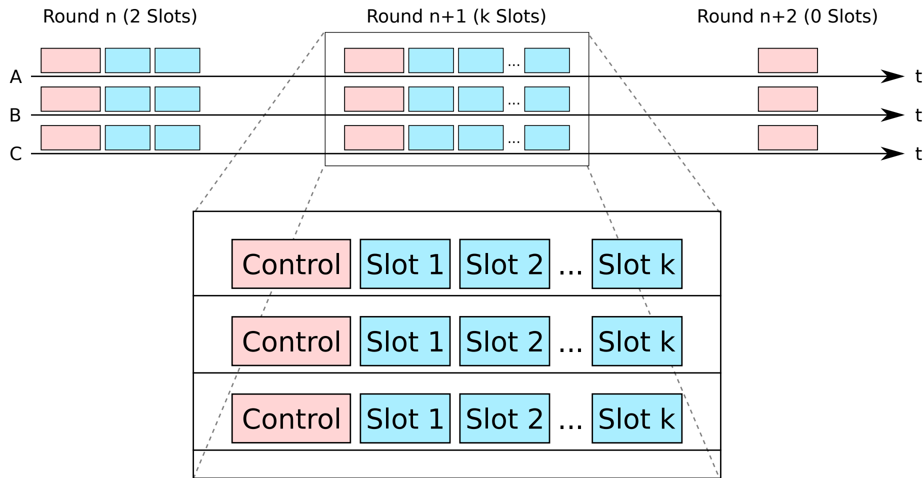

Figure 2.1: The Glossy Middleware is built on top of a round-based framework, providing high flexibility while being very energy efficient.

where multiple nodes might initiate a flood; this is useful for example to discover unknown nodes and adding them to a network of nodes.

Being able to quickly and easily adjust parameters and behaviour makes rapid prototyping of new protocol designs possible, since the user only has to care about when a round is happening and how the slots are used within it without having to carefully synchronise each Glossy flood separately. It is also possible to customise each slot as necessary, tailoring it to a specific use case. Due to the shared middleware, it is also easier to observe the difference between high level design choices and allows for a fairer comparison of different protocols and implementations.

A GMW-based network consists of a host node which is compiling and sending control packets; and source nodes, connected over a multi-hop wireless network to the host, listening for packets of a host node and synchronising to the schedule provided.

2.2.2 State Machine

GMW features a minimalistic state machine, depicted in Figure 2.2. This state machine is used by the source nodes and its states are known as `BOOTSTRAP`, `RUNNING` and `SUSPENDED`. The events triggering transitions are:

- *received control*: the node successfully transceived a control packet
- *received control with config*: a special case of the event above in which

case the control packet also includes the config section, see Section 3.1 for details

- *missed control*: the node fails to transceive a control packet, this might be due to various reasons, such as time misalignments between nodes, RF interference, node failure, etc.

The BOOTSTRAP state is where source nodes wait for the reception of a control packet. In this state the node repeatedly puts the radio in receive mode, until a control packet with a config is received. The node then transitions to the RUNNING state. The RUNNING state is the normal state, in which nodes participate in rounds. This means they receive and transmit in each slot, taking part in Glossy floods. The only event to exit this state is if a control packet is missed, in which case the node transitions to the SUSPENDED state. The SUSPENDED state is a state in which nodes still might be synced to the network, however they do not participate. More precisely, this means the nodes still try to receive and transmit the control packet, however they do not transmit or receive any further slots in the same round. If the node successfully transceived a control packet, it goes back to the RUNNING state. If it missed the control packet, it goes back to the BOOTSTRAP state.

The protocol implementation can overwrite the default state transitions in the post control slot callback, giving the user a way to implement custom behaviour (see Section 3.2.1 and Section 5.1.2 for details).

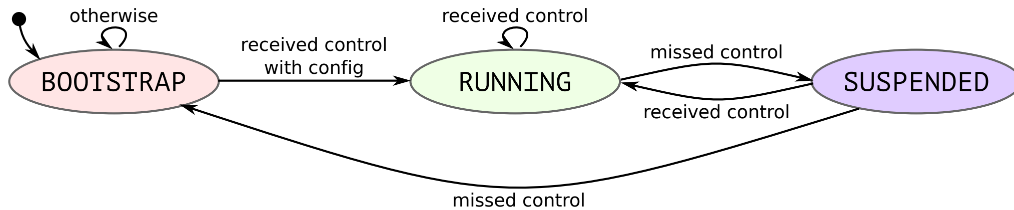

Figure 2.2: The state machine built into the middleware is kept simple on purpose while still offering extensive extension capabilities.

2.2.3 Callbacks

To use the GMW, the user implements a given set of callbacks in which he can implement the desired behaviour, such as preparing or receiving data in a slot. Figure 2.3 depicts when the callbacks are called depending on the context.

- `on_control_slot_post`: called after sending/receiving/missing a control packet, can be used to update state machines

- `on_slot_pre`: called before each slot, to prepare a data transmission in a slot (if any)
- `on_slot_post`: called after each slot, to handle received data or notify about the absence of expected data
- `on_round_finished`: called after the last slot, allowing for more time intensive tasks, for example to calculate a new schedule
- `on_bootstrap_timeout`: here source nodes can decide whether to go to sleep or to continue bootstrapping when failing to receive a control packet

Besides the callbacks, GMW also supports a pre and a post process, for example to poll an application process.

Having a closer look at Figure 2.3, on the left we can see the callbacks for the synced behaviour, shared between host and source nodes, and the bootstrapping behaviour for source nodes syncing to the network. Synced nodes are aware of the timing of the next round and are therefore able to schedule the pre process accordingly, scheduling it just before the start of the round. Synced nodes then wait for the round start and send (host nodes) or receive (source nodes) the control packet, after which `on_control_slot_post` is called. Then the data slots are handled, each slot resulting in an `on_slot_pre` and an `on_slot_post` callback. After the last slot (n), we receive the `on_round_finished` callback, in which we might prepare the received data for the application and update the schedule for the next round. After this, the post process is polled, for example to run the application which processes the received data further. After the post process yields, the Contiki main loop takes over and handles the transition to a low-power mode if applicable.

Source nodes in the `BOOTSTRAP` state repeatedly try to receive a control packet, calling `on_bootstrap_timeout` each time a control packet is missed (depicted on the right side of Figure 2.3). This is where the decision is made whether to go to sleep and try to receive a packet at a later point in time on or to try again right away. Upon receiving the control packet with a config section from the host, the source calls `on_control_slot_post` and is now in `RUNNING` state where the synced behaviour applies again.

Note that those callbacks are called in an interrupt context. Therefore the functions should execute rather quickly, however the timings can be adjusted to be compatible with computationally more expensive tasks or platforms of different speeds (see Section 3.1.2). However, most of the time intensive tasks should be completed during the less time sensitive contexts, such as the `on_round_finished` callback and especially the pre and post processes.

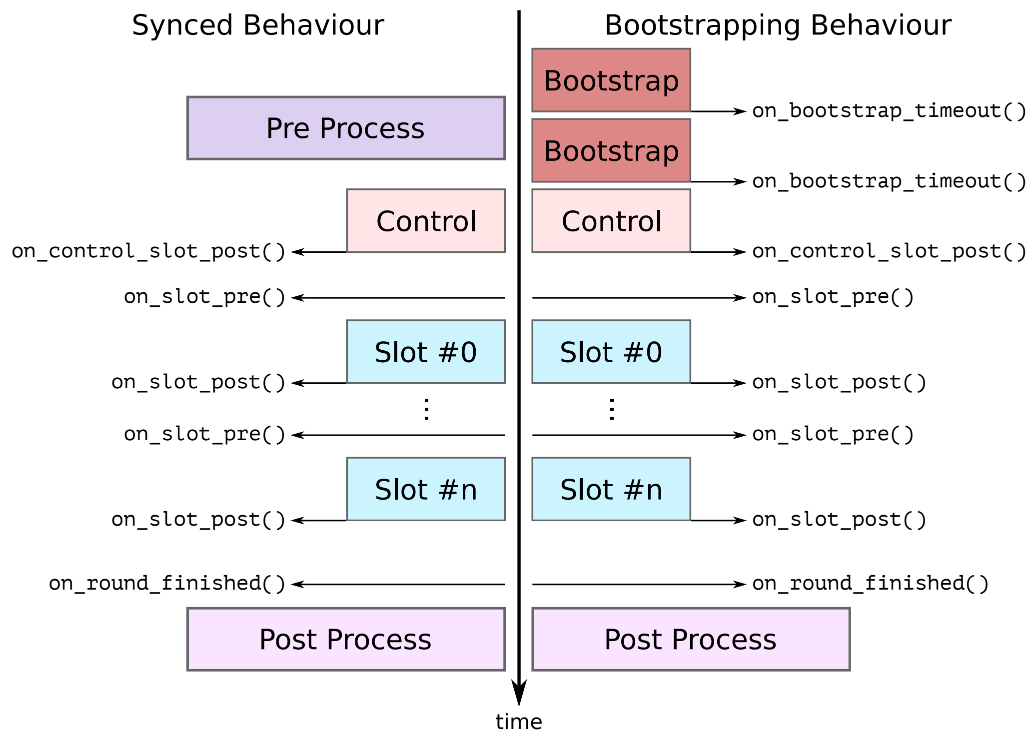

Figure 2.3: The use of processes and callbacks result in a flexible infrastructure to realise a wide range of use-cases.

Chapter 3

Implementation

This chapter aims to serve as an in-depth explanation of the Glossy Middleware as well as an user manual.

3.1 The Control Packet

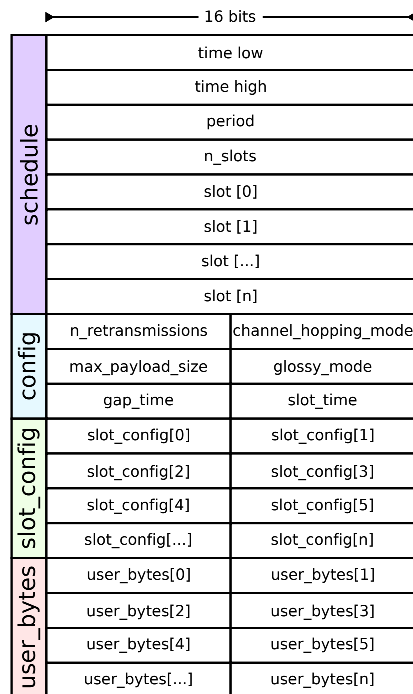

Figure 3.1: A complete control packet consisting of 4 sections.

The control packet (Figure 3.1), sent at the start of each round in the control slot, features up to 4 different sections:

- mandatory schedule section
- optional config section
- optional slot config section
- optional user byte section

Each of those optional sections might be attached to a control packet as required.

3.1.1 The *Schedule* Section

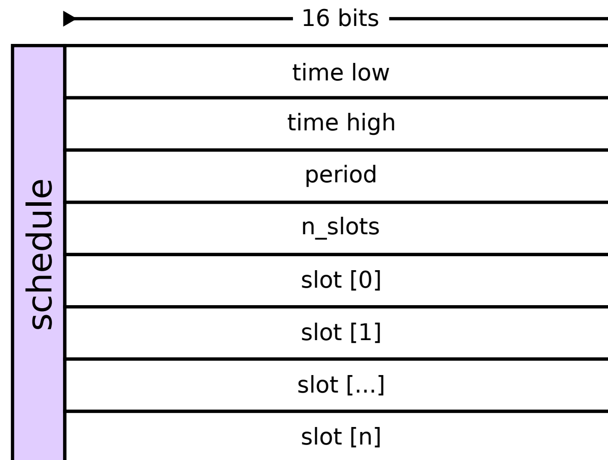

Figure 3.2: The schedule section is a key component of the control packet, defining the slot assignments of the current round and the timing of the next one.

The schedule (Figure 3.2) is where the period and the current global time are set. Additionally it also contains the slot assignment for this round; how many slots there are and which node each slots belongs to.

Period and time both share the same unit of time. It is chosen at compile time, using the `#define GMW_CONF_PERIOD_TIME_BASE`. This `#define` allows the user to choose between four different time resolutions: 1s, 100ms, 10ms and 1ms, depending on the time scale of the project.

The global time field is a 32bit unsigned integer (least significant 16bits first) and represents the current time of the host at the start of each round. It is controlled by the host and informs the source nodes about the host time. It is not directly used by the GMW and more of a informative parameter to be used by higher layers. Usually it should be incremented each round by the period between the last and the current round.

The period is a 16bit unsigned integer depicting the period between two round starts. Hereby the period of the control packet of round n defines the timing of the round n + 1, relative to the start of the current round. It is possible to change the period in each round, allowing for very flexible scheduling. The period is directly used by the GMW to schedule the timer interrupt for the next round at the end of the current round.

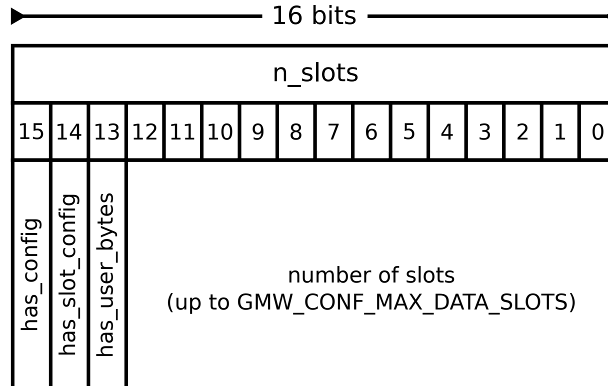

Figure 3.3: The n_slots field not only holds the number of slots in the current round, it also indicates the presence of the optional control packet section.

The n_slots field serves multiple purposes, as shown in Figure 3.3. First and foremost, it informs the network of the number of slots in the current round, using the lower 13 bits, therefore allowing a theoretical maximum of up to 8192 slots in one round. Note that the maximum amount of slots should not exceed `#define GMW_CONF_MAX_DATA_SLOTS`, which corresponds to how much memory is actually reserved for the slot assignment. The amount of slots available is a significant portion of the total maximum size of the control packet. Assuming a maximum size of 256bytes for the control packet with enabled config but no user bytes, the control packet can accommodate up to 121 slots without slot config and up to 80 slots including the slot config. Since the schedule is the only mandatory part of the control packet and full 16bits is exceeding a realistic number of slots used in one round by a huge margin, this field is used to indicate the presence of the 3 optional parts of the control packet, according to Figure 3.3. The corresponding section is attached if the bit is set to 1. There are a set of preprocessor macros in place to simplify the use of those three bits (see the file `gmw-control.h`).

Additionally, the n_slots field allows the middleware to reduce the amount of data being transmitted over radio, since this field defines not only how many slots are present in the current round but also which of the optional extensions are attached to it.

The remainder of the schedule section is used by a 16bit array which denotes

the slot assignment of each slot, or, in other words, which node is to initiate the Glossy flood in the respective slot. Hereby slot[0] is the first slot after the control packet slot, slot[1] the second and so on. The number of slots is denoted by the respective bits in the n_slots field as described above. To assign a node to a slot, one simply sets the `node_id` of the node which is to initiate the Glossy flood in the slot at the corresponding array index. Besides normal `node_ids`, GMW also supports a special id for contention slots: `#define GMW_SLOT_CONTENTION`. There is no limitation on how many contention slots a round can contain and where in the round they are located.

3.1.2 The *Config* Section

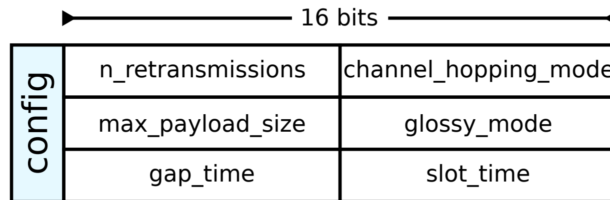

Figure 3.4: The config section holds important middleware parameters shared by the entire network.

The config section (Figure 3.4) contains the middleware parameters that can vary during runtime. This means that it has to be attached to the control packet periodically, since source nodes need to know about its contents before being able to participate. All of its values are 8bit values.

The first field, n_retransmissions defines how many times a packet is transmitted during a Glossy flood in the data slots. Figure 3.5 shows an example where n_retransmissions is set to 2. Note that the number of retransmissions for the control slot is set at compile time by setting `#define GMW_CONF_TX_CNT_CONTROL`.

The field max_payload_size can be used to reconfigure the radio for certain packet sizes, if this is supported by the radio. The reasoning behind this lies in the potentially different sizes of the control packet versus data packets. Since the control packet can get quite big, we need the radio to be able to handle larger packets (up to `#define GMW_CONF_MAX_CONTROL_PKT_LEN` bytes) while still being able to handle smaller data packages. This value basically tells the radio how long it should remain in receive mode before it timeouts. If one, large value is used for control and data packets and the start of a packet is missed, it is possible that an entire Glossy flood happens before the radio timeouts. Using a second, smaller value for the payload size therefore increases the probability of successfully receiving a packet.

The field gap_time defines the amount of time between two consecutive data

slots, see Figure 3.5. This is the amount of time available for processing the received packet and preparing a packet for the next slot. It is set by using the `GMW_US_TO_GAP_TIME(time)` macro, where time is the gap time in micro seconds. The gap time between the control packet and the first slot is fixed at compile time, using `#define GMW_CONF_T_GAP_CONTROL`. This value is directly interpreted as micro seconds.

The last field, slot_time defines the amount of time available for the execution of a Glossy flood, see Figure 3.5. The GMW features a `#define GMW_T_SLOT_MIN`, estimating the minimum duration of a slot according to [25]. Similar to the gap time, the slot time is set by using the `GMW_US_TO_SLOT_TIME(time)` macro, the parameter time again given in micro seconds.

The fields channel_hopping_mode and glossy_mode are not implemented right now. The channel hopping field is currently reserved to extend the middleware with channel hopping capabilities while the Glossy mode field is reserved to support different Glossy modes, such as Chaos-style[14] Glossy floods.

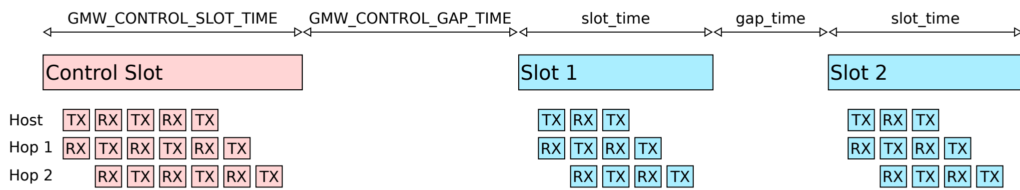

Figure 3.5: Visual representation of the affects of config parameters.

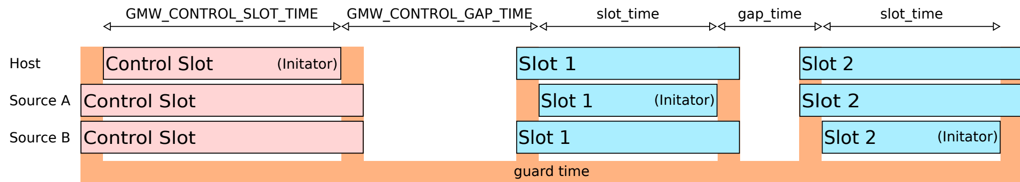

Figure 3.6: The guard time offers some leeway to counter clock-drift between different nodes.

The last parameter influencing slot timing is the guard time `#define GMW_CONF_T_GUARD`. As shown in Figure 3.6, the guard time is the amount of time the non-initiating nodes start a slot before the initiator and also wait longer to finish it, increasing reception rate if devices get out of sync. Therefore the resulting total slot time for non-initiating nodes is $slot_time + 2 \cdot guard_time$.

3.1.3 The *Slot Config* Section

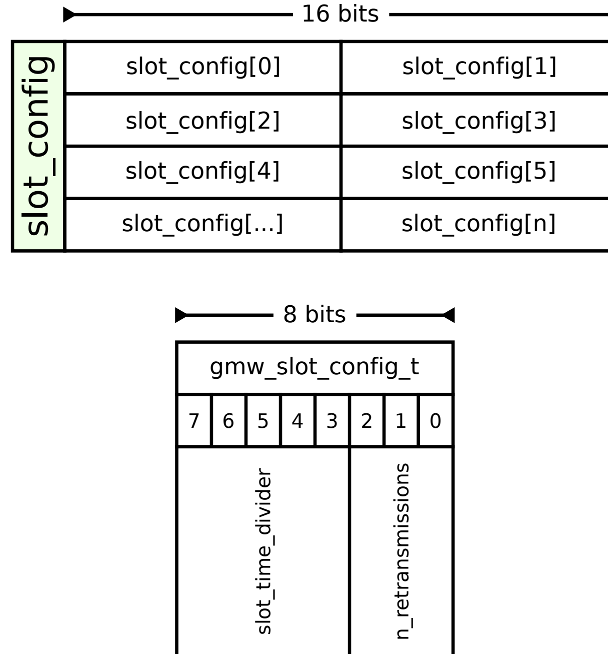

Figure 3.7: The slot_config fields allow individual control over the slot time and the number of retransmissions on a per-slot level.

The slot config section (Figure 3.7) allows further customisation of the slot behaviour on a per slot level. Currently implemented is a solution where the slot time from the config can be divided to allow different slot lengths. Additionally the number of retransmissions can also be set on a per slot basis.

As shown in Figure 3.7, each slot gets assigned an array index, where each index corresponds to the index in the slots array from the schedule section. Therefore one either enables the slot config for all slots or none, there is no in-between.

Each array index currently holds a 8bit wide type `gmw_slot_config_t`, which has two bitfields: bit 7 down to 3 hold the slot time divider and bits 2 down to 0 hold the number of retransmissions. The number of retransmissions is a one-to-one mapping with a offset of +1. Setting it to 0 therefore leads to 1 retransmission in the Glossy flood of the associated slot. The slot time divider however works differently. It uses a divider formula to reduce the slot time set in the config section, allowing a down to 32 times smaller slot length for individual slots. This allows for a wide range of slot times by choosing appropriate values for the config slot time and the corresponding slot time dividers. The formula for slot n is:

$$slot_time = \frac{config.slot_time}{slot_config[n].slot_time_divider + 1} \tag{3.1}$$

Note that while it is currently limited to the number of retransmissions and the slot time, it could be extended or changed to accommodate different parameters if necessary, for example one could extend it to support different gap times as well.

3.1.4 The *User Bytes* Section

To further customise the control packet, it is also possible to attach additional bytes to it. This is especially useful to mark certain control packets or to transmit state information. An example usage can be found in Section 5.1.3.

To use the user bytes, simply set `#define GMW_CONF_CONTROL_USER_BYTES` to the amount of additional bytes required and enable (or disable) it with the `#define GMW_CONTROL_SET_USER_BYTES` set of macros.

3.2 Callbacks: The Nitty-gritty Part

This section explains the callbacks introduced in Section 2.2.3 in more depth, giving the reader everything to know about using them in a project.

3.2.1 The Post Control Slot Callback

Footprint

```
typedef enum {
  GMW_BOOTSTRAP = 0,
  GMW_RUNNING,
  GMW_SUSPENDED,
  GMW_DEFAULT,
} gmw_sync_state_t;

typedef enum {
  GMW_EVT_CONTROL_RCVD = 0,
  GMW_EVT_CONTROL_MISSED,
  GMW_NUM_OF_SYNC_EVENTS
} gmw_sync_event_t;

typedef gmw_sync_state_t
        (*gmw_on_control_slot_post_callback)
                   (gmw_control_t* in_out_control,
                    gmw_sync_event_t event);
```

Description

This callback is called after each control slot, on both host and source nodes.

For the host, it informs the higher layer that a control slot was sent (`event = GMW_EVT_CONTROL_RCVD`). It also includes a pointer to the control structure used in this round. This is mainly to inform the higher layer about the round structure, but it is also possible to alter the control packet here, however those changes do only affect the current node. The host has two valid return values: `GMW_RUNNING` and `GMW_SUSPENDED`. `GMW_RUNNING` is the norm, the slots of the round are processed as defined by the schedule section. `GMW_SUSPENDED` however will ignore the slots of the round and skip it. This can be useful to send informative control packets without having to execute them.

Source implementation learn from the `event` parameter whether or not they received (`GMW_EVT_CONTROL_RCVD`) or missed (`GMW_EVT_CONTROL_MISSED`) a control slot and are able to update their internal, protocol specific state machine accordingly. Similar to the host, they also receive a pointer to the control structure of this round. This might be informative only, but it also allows them to alter it, for example to compensate for a missed schedule. Source nodes can return the following:

- `GMW_RUNNING`: is synced and processes the slots of the round
- `GMW_SUSPENDED`: is synced but ignores the slots of the round
- `GMW_BOOTSTRAP`: lost sync and returns to `BOOTSTRAP` immediately
- `GMW_DEFAULT`: uses the default state transition as defined in Section 2.2.2

Parameters and Return Values

- gmw_control_t* in_out_control pointer to the current control packet, read and write access
- gmw_sync_event_t event informs the node about sending/receiving or missing a control slot
- returns
  - `GMW_RUNNING`: is synced and processes the slots of the round
  - `GMW_SUSPENDED`: is synced but ignores the slots of the round
  - `GMW_BOOTSTRAP` (source only): lost sync and returns to `BOOTSTRAP` immediately
  - `GMW_DEFAULT` (source only): uses the default state transition as defined in Section 2.2.2

3.2.2 The Pre Slot Callback

Footprint

```
typedef void (*gmw_on_slot_pre_callback)
                 (uint8_t slot_index,
                  uint16_t slot_assignee,
                  uint8_t* out_len,
                  uint8_t* out_glossy_payload,
                  uint8_t is_initiator,
                  uint8_t is_contention_slot);
```

Description

This callback is called before each slot in the gap time between two consecutive slots, with identical behaviour on both host and source nodes. The main purpose of this callback is to copy data (if any) from the protocol layer to the Glossy buffer where it is then used to initiate a Glossy flood if `out_len` is not zero after returning.

Parameters and Return Values

- `uint8_t slot_index`: slot index in the current round, starting at 0
- `uint16_t slot_assignee`: the node_id assigned to this slot in the schedule section
- `uint8_t* out_len`: pointer to an `uint8_t` where the number of bytes 'to-be-sent' (if any) should be written to (max value is `#define GMW_CONF_MAX_DATA_PKT_LEN`)
- `uint8_t* out_glossy_payload`: the buffer where the data 'to-be-sent' (if any) should be written to
- `uint8_t is_initiator`: convenience parameter: `node_id==slot_assignee`
- `uint8_t is_contention_slot`: convenience parameter: `GMW_SLOT_CONTENTION==slot_assignee`

3.2.3 The Post Slot Callback

Footprint

```
typedef void (*gmw_on_slot_post_callback)
              (uint8_t slot_index,
               uint16_t slot_assignee,
               uint8_t len,
               uint8_t* glossy_payload,
               uint8_t is_initiator,
               uint8_t is_contention_slot);
```

Description

Similarly to the pre slot callback, the post slot callback is called after each slot in the gap time between to slots (or after the last), again with identical behaviour on both host and source nodes. Complementary to the pre slot callback, this functions main purpose is to retrieve received Glossy data and copy it to the higher layer or to inform about the absence of such data. If the node copied data in the pre slot callback and therefore initiated a Glossy flood, it will simply be called again with the same parameters as in the pre slot callback. If there was no data copied in the pre slot callback, the parameters `len` and `glossy_payload` indicate whether or not data was received: if no data was received, `len` is `0` and `glossy_payload` is `NULL`, otherwise `glossy_payload` points to the buffer where the glossy data is stored and `len` indicates how many bytes were received in the last slot.

Parameters and Return Values

- `uint8_t slot_index`: slot index in the current round, starting at 0
- `uint16_t slot_assignee`: the node_id assigned to this slot in the schedule section
- `uint8_t len`: number of bytes received (0 indicates no bytes received)
- `uint8_t* glossy_payload`: `NULL` if no data received, otherwise start of the buffer used by Glossy
- `uint8_t is_initiator`: convenience parameter: `node_id==slot_assignee`
- `uint8_t is_contention_slot`: convenience parameter: `GMW_SLOT_CONTENTION==slot_assignee`

3.2.4 The Post Round Callback

Footprint

```
typedef struct gmw_pre_post_processes {
    struct process* post_process_current_round;
    struct process* pre_process_next_round;
} gmw_pre_post_processes_t;

typedef void (*gmw_on_round_finish_callback)
               (gmw_pre_post_processes_t*
                    in_out_pre_post_processes);
```

Description

This callback is called shortly after the last slot post callback returned, on both host and source nodes. It informs the higher layer that the round has concluded and allows for tasks that take more time, e.g. updating the control packet for the next round. Additionally it gives the higher layer control about polling of the pre and post processes by writing to the `in_out_pre_post_processes` struct. More concise, writing `NULL` to either of the two pointers inside of the struct will skip its execution. Otherwise, writing of any valid and started process to such a pointer will result in a poll event by the GMW; `post_process_current_round` would affect the post process of the current round, which would be polled shortly after returning the post round callback, `pre_process_next_round` would change the pre process that would be polled just before the next round. On the first callback, the values are the values set with the `gmw_start` call.

Parameters and Return Values

`gmw_pre_post_processes_t* in_out_pre_post_processes` allows altering the polling behaviour of the pre and the post processes. Writing `NULL` to either of the two pointers will skip its execution. `post_process_current_round` accepts valid `process` pointers, which will be polled by GMW after this round concluded. `pre_process_next_round` accepts valid `process` pointers, which will be polled by GMW just before the start of the next round.

3.2.5 The Bootstrap Timeout Callback

Footprint

```
typedef uint32_t (*gmw_on_bootstrap_timeout_callback)
                (void);
```

Description

This callback is only relevant for source nodes, as host nodes never are in `BOOTSTRAP`. During `BOOTSTRAP`, the source nodes tries to receive a control packet. After the duration `#define GMW_CONF_T_MAX_CONTROL` a time out is reached and the GMW calls the bootstrap timeout callback, giving the higher layer control about how long a node should stay in `BOOTSTRAP`. If 0 is returned, GMW tries again immediately, otherwise the returned number is interpreted as the number of milliseconds the node goes to a low-power mode before trying again.

Parameters and Return Values

Returning 0 instructs the node to stay in `BOOTSTRAP`, every other number is the number of milliseconds the node goes to a low-power mode before trying again.

3.3 Main Interface

This section describes the most relevant functions a user needs to call to get the GMW up and running.

3.3.1 The GMW Start Function

Footprint

```
typedef struct gmw_protocol_impl {
  gmw_on_control_slot_post_callback on_control_slot_post;
  gmw_on_slot_pre_callback on_slot_pre;
  gmw_on_slot_post_callback on_slot_post;
  gmw_on_round_finish_callback on_round_finished;
  gmw_on_bootstrap_timeout_callback on_bootstrap_timeout;
} gmw_protocol_impl_t;

void gmw_start(struct process* pre_gmw_proc,
        struct process* post_gmw_proc,
        gmw_protocol_impl_t* host_protocol_impl,
        gmw_protocol_impl_t* src_protocol_impl);
```

Description

This function initialises the GMW, the timers and schedules the GMW main loop. The pre and post processes should be used for any additional processing not closely related to the GMW. The host and source protocol implementations are structs containing the callbacks that define the behaviour as described in Section 3.2.

Parameters and Return Values

- `struct process* pre_gmw_proc`: the initial pre process, can be updated in the post round callback
- `struct process* post_gmw_proc`: the initial post process, can be updated in the post round callback
- `gmw_protocol_impl_t* host_protocol_impl`: a struct filled with callbacks for the host implementation
- `gmw_protocol_impl_t* src_protocol_impl`: a struct filled with callbacks for the source implementation

3.3.2 The Set New Control Function

Footprint

```
void gmw_set_new_control(gmw_control_t* control);
```

Description

This function is where the host implementation sets the next control packet which is sent in the control slot of the next round. This should be completed between the end of the last data slot in a round and the transmission of the control packet in the next round. Good places to call this functions therefore are: the post round callback, the post and the pre process. If the host fails to do so, the GMW will take an naive approach and reuse last rounds control packet, just updating the time field in the schedule. Since the GMW copies the current control packet at the start of the round, it should not be changed between calling `gmw_set_new_control` and the start of the round.

Parameters and Return Values

`gmw_control_t* control`: a pointer to the control packet to be used in the next control packet slot

3.3.3 The Control Init Function

Footprint

```
void gmw_control_init(gmw_control_t* control);
```

Description

This function is used to sanitise the control structure using default values. For the schedule, time is set to 0, period to 5 and number of slots to 0. The config is set to 3 retransmissions, the maximum packet length to `#define GMW_CONF_MAX_DATA_PKT_RF_LEN`, the gap time to `#define GMW_CONF_T_GAP` and the slot time to `#define GMW_CONF_T_DATA`.

Parameters and Return Values

`gmw_control_t* control`: a pointer to the control packet to be initialised

3.4 Additional Features

3.4.1 Drift Compensation

The Glossy Middleware comes with a very simple drift compensation to mitigate clock drift between the host and source nodes. The drift compensation is enabled by setting `#define GMW_CONF_USE_DRIFT_COMPENSATION` to 1. The calculation is as follows:

$$d = \frac{((t_{ref,n} - t_{ref,n-1}) - T_{n-1}) \cdot df}{T_{n-1}} \tag{3.2}$$

$$d_n = \frac{d + d_{n-1}}{2} \tag{3.3}$$

$$T_d = \frac{T_n \cdot d_n}{df} \tag{3.4}$$

where

d	drift of current round
$t_{ref,n}$	Glossy t_{ref} of current round
$t_{ref,n-1}$	Glossy t_{ref} of previous round
T_n	period from current to next round
T_{n-1}	period from last to current round
df	'drift factor', a compensatory term to prevent integer rounding errors
d_n	low-pass filtered accumulated drift term of current round
d_{n-1}	low-pass filtered accumulated drift term of last round
T_d	actual drift compensation term

The resulting term T_d is then added to the period when scheduling the timer for the start of the next round.

3.4.2 Statistics

To help debug and optimise a protocol, GMW features some built in statistics:

```
typedef struct {
  uint8_t   relay_cnt;
  uint8_t   suspended_cnt;
  uint8_t   bootstrap_cnt;
  uint8_t   sleep_cnt;
  int16_t   drift;
  uint16_t pck_cnt;
  uint16_t t_proc_max;
  uint32_t rx_total;
  uint32_t t_round_max;
  uint32_t t_slack_min;
} gmw_statistics_t;
```

- `relay_cnt`: how many times the host received a retransmission of the control packet
- `suspended_cnt`: the number of rounds with state `SUSPENDED`
- `bootstrap_cnt`: the number of bootstrap attempts of a source node
- `sleep_cnt`: how many times a source node went to sleep due to not receiving a packet during bootstrap
- `drift`: low-pass filtered clock drift between host and a source node, see d_n in Section 3.4.1 (source nodes only)
- `pck_cnt`: total number of successful Glossy floods in data slots
- `t_proc_max`: longest time spent in slot post callback in clock ticks

- `rx_total`: total number of received bytes of payload in data slots
- `t_round_max`: longest round time in milliseconds measured by the host only, so the time from the start of the first Glossy flood till between the polling of the post process and the yield of the GMW process
- `t_slack_min`: shortest amount of time in milliseconds for the host between the end of the current round (see `t_round_max`) and the wakeup for the next one, including the preprocess

3.4.3 Channel Hopping

Channel hopping is a feature often used in RF applications to counter interference but currently not implemented by the middleware; however it would be a worthwhile addition, for example on the basis of Robust Flooding[16].

3.5 Porting

The middleware has been designed with portability in mind. There are two main files to offer the necessary flexibility to the lower layers: `gmw-platform.h` and `gmw-platform-conf.h`.

The file `gmw-platform.h` contains the interface of the low-level functions used by the middleware. At this point in time, this interface consists of two functions: `gmw_platform_init(void)` and `gmw_platform_set_maximum_packet_length(uint8_t length)`. The `init` function is called during the execution of `gmw_start()` and is a good place to initialise GMW dependencies that are not initialised during the start-up of Contiki, for example if the Glossy implementation relies on a process that has to be started. The `set_maximum_packet_length` function offers the ability to reconfigure the radio to different packet sizes, which increases the reception probability (see Section 3.1.2 for details). Note that while both functions are required to compile correctly, they can simply be left empty if not needed.

The other file, `gmw-platform-conf.h`, contains preprocessor macros that wrap around low-level calls. Those macros abstract primitives such as Glossy and timer function calls and arguments. Additionally, there are a few required typedefs:

- `gmw_rtimer_t`: the Contiki *rtimer* struct type used by the middleware
- `gmw_rtimer_clock_t`: the data type used by the GMW to store timer values

This abstraction of timer implementations allows the use of extended timers (see the *rtimer_ext* implementation on both the DPP and the TelosB platform),

not directly offered by Contiki. Those extended timers allow scheduling of events that are too far in the future to fit into one timer overflow. Additionally, those extended timer implementations allow the abstraction on multiple hardware timers, a feature not offered by Contiki natively, which simplifies reuse of timer interrupts for different software modules.

Also worth mentioning: Defining `#define GMW_T_HOP(len)` allows the use of `#define GMW_T_SLOT_MIN`, which is an easy way to estimate slot time based on the parameters number of hops, number of retransmission and length of the packet to be retransmitted. `#define GMW_T_HOP(len)` hereby returns the time in microseconds it takes for the radio to transmit a packet with length len.

Last but not least, `#define GMW_CONF_RF_OVERHEAD` should be set to the overhead introduced by the underlying Glossy implementation and the radio hardware. In case of the DPP platform, Glossy adds a 4 byte header, while the radio adds a length byte and a 16bit CRC value, resulting in a total overhead of 7 bytes. This is used to precisely estimate default slot lengths used to initialise the config section values (Section 3.1.2).

Chapter 4

Minimal Working Example

After establishing the middleware and its features, this chapter covers an example usage of the middleware to solve a real-life data-collecting challenge.

A common use-case for low-power wireless networks is periodic gathering of sensor data. This chapter presents a minimal C-code example for a static solution to this problem, simply called GMW-Test. This solution uses the same executable for host and source nodes. Which code is run is decided during runtime by evaluating the value of the variable `node_id` provided by Contiki and comparing it to the `#define HOST_ID` from `project-conf.h`.

The code is available here (`http://bit.ly/glossymiddleware`), this example is found under `<path to repo>/examples/gmw-minimal`.

The first section present the used include files while two following sections show the declaration of the local function and the local variables used in this application. The next section shows the main application process, followed by the definitions of the callbacks for the host and source implementation. Finally, the control management functions used by the host implementations are presented.

4.1 Include Files

We first include the main header files of the modules we want to use in this application. The Contiki header file `contiki.h` gives us access to global configuration parameters and to primitives such a processes. The Glossy Middleware header file `gmw.h` gives us access to the GMW interface while the `debug-print.h` file is included in order to use the asynchronous debug output provided by this module.

```
#include "contiki.h"
#include "gmw.h"
#include "debug-print.h"
```

4.2 Local Static Function Declarations

First off, the callbacks for the middleware as defined in Section 3.2 are declared as forward declarations, a set for both the host and the source implementation. Additionally, there are two functions to initialise the control packet and update it each round.

```
static gmw_sync_state_t host_on_control_slot_post_callback(
                    gmw_control_t* in_out_control,
                    gmw_sync_event_t event);
static void host_on_slot_pre_callback(
                    uint8_t slot_index,
                    uint16_t slot_assignee,
                    uint8_t* out_len,
                    uint8_t* out_glossy_payload,
                    uint8_t is_initiator,
                    uint8_t is_contention_slot);
static void host_on_slot_post_callback(
                    uint8_t slot_index,
                    uint16_t slot_assignee,
                    uint8_t len,
                    uint8_t* glossy_payload,
                    uint8_t is_initiator,
                    uint8_t is_contention_slot);
static void host_on_round_finished(
                    gmw_pre_post_processes_t*
                        in_out_pre_post_processes);

static gmw_sync_state_t src_on_control_slot_post_callback(
                    gmw_control_t* in_out_control,
                    gmw_sync_event_t event);
static void src_on_slot_pre_callback(
                    uint8_t slot_index,
                    uint16_t slot_assignee,
                    uint8_t* out_len,
                    uint8_t* out_glossy_payload,
                    uint8_t is_initiator,
                    uint8_t is_contention_slot);
static void src_on_slot_post_callback(
                    uint8_t slot_index,
                    uint16_t slot_assignee,
                    uint8_t len,
                    uint8_t* glossy_payload,
                    uint8_t is_initiator,
                    uint8_t is_contention_slot);
static void src_on_round_finished(
                    gmw_pre_post_processes_t*
                        in_out_pre_post_processes);
static uint32_t src_on_bootstrap_timeout(void);
```

```
static void control_init(gmw_control_t* control);
static void control_update(gmw_control_t* control);
```

Those function declarations can be left out by rearranging the definitions of the following code, however I personally like the overview it presents right at the start of the file and the ability to arrange the functions in a way that makes sense in an abstract view compared to what makes sense to the compiler.

4.3 Local Variables

Two instances of `gmw_protocol_impl_t` hold the callbacks for the host and the source implementations respectively. The `gmw_control_t` instance holds the memory for the control packet of the host. Finally the `counter` variable is used to generate changing data while `static_nodes` and `number_of_static_nodes` are used to initialise the schedule in this example.

```
static gmw_protocol_impl_t  host_impl;
static gmw_protocol_impl_t  src_impl;
static gmw_control_t        control;
static uint8_t              counter = 0;
static const uint16_t static_nodes[] = { 1, 2, 3, 4, 6,
                                          7, 8,10,11,13,
                                         14,15,16,17,18,
                                         19,20,22,23,24,
                                         25,26,27,28,32,
                                         33};
static const uint8_t number_of_static_nodes = 26;
```

4.4 The Main Application Process

The following extract presents the Contiki process[7] which runs this example. First, the process `app_process` is declared using the `PROCESS` macro. The `AUTOSTART_PROCESSES` macro adds this process to a list of processes automatically run after Contiki has finished initialisation.

In the process definition, defined with the `PROCESS_THREAD` macro, the beginning of the process code is marked with the `PROCESS_BEGIN` macro. First the two implementation instances are filled with function pointers to the callbacks declared above. Then the control structure is initialised with GMW defaults using `gmw_control_init` and customised with `control_init` also declared above and finally handed to the middleware using `gmw_set_new_control(&control)`.

Before entering the main loop of this process, the middleware is started using the `gmw_start` function (see Section 3.3.1). The first argument `NULL` implies the

absence of a pre process while the current process (`app_process`) is used as a post process. The pre and post processes are polled by the middleware before and after a round respectively, giving an application task time to run. By setting the `app_process` in this example, the content of the while loop will be executed every time the process is polled. The last two arguments hold the prepared implementations for host and source nodes respectively.

In the main loop, the application yields until polled with the `PROCESS_YIELD_UNTIL(ev == PROCESS_EVENT_POLL)` macro. If polled, the process polls the `debug-print` process which allows for asynchronous debug output before yielding again. The Contiki operating system automatically takes care of entering and leaving low-power modes during phases of inactivity.

```
PROCESS(app_process, "Application Task");
AUTOSTART_PROCESSES(&app_process);
/*---------------------------------------------------------*/
PROCESS_THREAD(app_process, ev, data)
{
  PROCESS_BEGIN();

  host_impl.on_control_slot_post =
        &host_on_control_slot_post_callback;
  host_impl.on_slot_pre = &host_on_slot_pre_callback;
  host_impl.on_slot_post = &host_on_slot_post_callback;
  host_impl.on_round_finished = &host_on_round_finished;
  src_impl.on_control_slot_post =
        &src_on_control_slot_post_callback;
  src_impl.on_slot_pre = &src_on_slot_pre_callback;
  src_impl.on_slot_post = &src_on_slot_post_callback;
  src_impl.on_round_finished = &src_on_round_finished;
  src_impl.on_bootstrap_timeout = &src_on_bootstrap_timeout;

  gmw_control_init(&control);

  if (HOST_ID == node_id){
    control_init(&control);
  }
  gmw_set_new_control(&control);

  /* start the GMW thread */
  gmw_start(NULL, &app_process, &host_impl, &src_impl);

  /* main loop of this application task */
  while(1) {
    /* the app task should not do anything until it is
     * explicitly granted permission (by receiving
     * a poll event) by the GMW task
     */
    PROCESS_YIELD_UNTIL(ev == PROCESS_EVENT_POLL);
```

```
    /* poll the debug-print task, this will output
     * the prepared debug-print messages
     */
    debug_print_poll();
  }

  PROCESS_END();
}
```

Note that while the post process is very minimalistic in this example, one is free to add additional code there, for example to read sensor data. Also, a pre process would be added in a similar fashion by adding it as an argument to `gmw_start` (after either being started as an autostart process or manually using `process_start` from the `app_process`).

4.5 The Host Node Implementation

For the host node implementation, four callbacks are required:

- `host_on_control_slot_post_callback` directly returns `GMW_RUNNING` to participate in this round
- `host_on_slot_pre_callback` creates the payload if the node is initiator for this slot: the first byte is a magic number (42) and the second byte is an incrementing counter value. Note that writing a non-zero value to the `out_len` pointer will instruct the middleware to initiate a Glossy flood
- `host_on_slot_post_callback` handles received data by printing it or notifying the user about not receiving
- `host_on_round_finished` updates the control structure and hands it over to the middleware

```
static gmw_sync_state_t
host_on_control_slot_post_callback(
                    gmw_control_t* in_out_control,
                    gmw_sync_event_t event)
{
  /* return GMW_RUNNING in order to participate
   * in this round
   */
  return GMW_RUNNING;
}
/*---------------------------------------------------------*/
static void
```

```
host_on_slot_pre_callback(
                    uint8_t slot_index,
                    uint16_t slot_assignee,
                    uint8_t* out_len,
                    uint8_t* out_glossy_payload,
                    uint8_t is_initiator,
                    uint8_t is_contention_slot)
{
  /* if this is our slot, set our payload */
  if(is_initiator) {
    out_glossy_payload[0] = 42;
    out_glossy_payload[1] = counter++;
    *out_len = 2;
  }
}
/*---------------------------------------------------------*/
static void
host_on_slot_post_callback(
                    uint8_t slot_index,
                    uint16_t slot_assignee,
                    uint8_t len,
                    uint8_t* glossy_payload,
                    uint8_t is_initiator,
                    uint8_t is_contention_slot)
{
  /* if this was not our slot, handle the payload */
  if(!is_initiator) {
    if(len == 2) {
      /* print the two bytes */
      DEBUG_PRINT_INFO("Host received %u %u",
                 glossy_payload[0],
                 glossy_payload[1]);
    } else {
      /* we received not what we expected */
      DEBUG_PRINT_INFO("Host did not receive, len=%u", len);
    }
  }
}
/*---------------------------------------------------------*/
static void
host_on_round_finished(
                    gmw_pre_post_processes_t*
                        in_out_pre_post_processes)
{
  /* the round has finished, we can now update the control
   * packet
   */
  control_update(&control);
  /* and set it at the GMW */
```

```
  gmw_set_new_control(&control);
}
```

4.6 The Source Node Implementation

The source node implementation has 5 callbacks:

- `src_on_control_slot_post_callback` similar to the host directly return `GMW_DEFAULT` to use the default state machine included in the middleware (see Section 2.2.2)
- `src_on_slot_pre_callback` similar to the host node, the first byte of the payload is a decrementing counter and the second byte is the magic number (0xaa)
- `src_on_slot_post_callback` similar to the host with slightly different strings
- `src_on_round_finished` in this implementation there is no time intensive task left for the source node
- `src_on_bootstrap_timeout` directly returns 0 in order to stay in `BOOTSTRAP` state (see Section 3.2.5)

```
static gmw_sync_state_t
src_on_control_slot_post_callback(
                    gmw_control_t* in_out_control,
                    gmw_sync_event_t event)
{
  /* here we could implement our own state machine, however
   * we are just using the default state machine implemented
   * in the GMW in this case by returning GMW_DEFAULT.
   */
  return GMW_DEFAULT;
}
/*---------------------------------------------------------*/
static void
src_on_slot_pre_callback(
                    uint8_t slot_index,
                    uint16_t slot_assignee,
                    uint8_t* out_len,
                    uint8_t* out_glossy_payload,
                    uint8_t is_initiator,
                    uint8_t is_contention_slot)
{
  /* same as for host:
   * if this is our slot, set our payload
   */
```

```
  if(is_initiator) {
    out_glossy_payload[0] = counter--;
    out_glossy_payload[1] = 0xaa;
    *out_len = 2;
  }
}
/*---------------------------------------------------------*/
static void
src_on_slot_post_callback(
                    uint8_t slot_index,
                    uint16_t slot_assignee,
                    uint8_t len,
                    uint8_t* glossy_payload,
                    uint8_t is_initiator,
                    uint8_t is_contention_slot)
{
  /* same as for host:
   * if this was not our slot, handle the payload
   */
  if(!is_initiator) {
    if(len == 2) {
      DEBUG_PRINT_INFO("Src received %u %u",
                glossy_payload[0],
                glossy_payload[1]);
    } else {
      DEBUG_PRINT_INFO("Src did not receive, len=%u", len);
    }
  }
}
/*---------------------------------------------------------*/
static void
src_on_round_finished(
                    gmw_pre_post_processes_t*
                        in_out_pre_post_processes)
{
  /* do nothing in this case, however this callback could
   * be used to print debug output or to update statistics.
   */
}
/*---------------------------------------------------------*/
static uint32_t
src_on_bootstrap_timeout(void)
{
  /* returning 0 here instructs the GMW to stay in
   * bootstrap
   */
  return 0;
}
```

4.7 The Control Management Functions

Last but not least the code used to manage the control structure: In `control_init` the content of the `static_nodes` array is copied to the schedule section (see Section 3.1.1) and the `n_slots` field is set accordingly. Then the time is reset to 0 and the period to 2 seconds. Finally the config section (Section 3.1.2) is attached to the control structure and values are set for the gap time, the slot time and the number of retransmissions. The `#defines` used to set the config values are part of the `"project-conf.h"`, a standard header file containing all project specific parameters.

The `control_update` function is called after each round by the host implementation and simply increments the `time` field by the value of the `period` field and prints a debug message. Since the schedule is static, the slot schedule set previously can be reused throughout the duration of the execution.

```
static void
control_init(gmw_control_t* control)
{
  /* initialise the slot assignment with the given array
   * of nodes
   */
  int i;
  for (i=0; i<number_of_static_nodes; i++) {
    control->schedule.slot[i] = static_nodes[i];
  }

  /* we have number_of_static_nodes nodes in our schedule */
  control->schedule.n_slots = number_of_static_nodes;

  /* we start our local time at 0 */
  control->schedule.time = 0;

  /* the period is 2 seconds */
  control->schedule.period = 2;

  /* we want the control packet to include a
   * config section
   */
  GMW_CONTROL_SET_CONFIG(control);

  /* set the parameters from "project-conf.h" */
  control->config.gap_time =
                GMW_US_TO_GAP_TIME(GMW_CONF_T_GAP);
  control->config.slot_time =
                GMW_US_TO_SLOT_TIME(GMW_CONF_T_DATA);
  control->config.n_retransmissions =
                GMW_CONF_TX_CNT_DATA;
}
```

```
/*---------------------------------------------------------*/
static void
control_update(gmw_control_t* control)
{
  /* increment the time by the period */
  control->schedule.time += control->schedule.period;

  /* print debug output */
  DEBUG_PRINT_INFO("t=%lus %lu T=%u",
                   clock_seconds(),
                   control->schedule.time,
                   control->schedule.period);
}
```

This chapter demonstrated a simple example to a common use-case in order to guide the reader through the basic usage of the middleware. More advanced usages are presented in the next chapter where two existing high-level protocols are reimplemented and compared to native implementations.

Chapter 5
Evaluation

There are two major aspects to evaluate for a middleware:

The Usability shows that the design of the middleware allows for flexible use. It is evaluated by reimplementing two existing protocols; additionally there are numerous test applications to cover the entire feature set of the middleware.

The Performance Overhead indicates how much overhead is introduced by the introduction of the middleware and should be comparable to implementations without a middleware. It is measured by the following metrics, comparing an original protocol implementation with a GMW-based reimplementation:

- Binary size of the resulting executables
- Maximum stack size
- Duty cycle of both CPU and radio components
- Reliability of the provided service

While the binary size is measured by simply executing the `size` executable of the toolchain, measuring the duty cycle, the reliability as well as the stack size requires a physical platform. We used the FlockLab[17] testbed for the evaluation. FlockLab allows an easy deployment of executables and provides multiple hardware platforms. Since the currently supported hardware platforms for the middleware include the Dual Processor Platform (DPP[23, 22]) as well as the TelosB[11], 25 FlockLab nodes hosting both of those hardware platforms were chosen.

In Section 5.1 we revisit GMW-Test application and introduce LWB and eLWB, two protocols used to showcase the usability of the middleware; while Section 5.2 presents the performance evaluation, comparing the hardware platforms DPP and TelosB and the overhead with respect to native implementations of LWB and eLWB.

5.1 Usability and Setup

5.1.1 GMW-Test

The application GMW-Test was already described in-depth in Chapter 4. It is built for both the DPP and the TelosB platform with the following parameters. First the parameters for the DPP build:

Parameter	Description
$T = 2s$	Round period T
$T_{ctrl} = 28ms$	Control slot length T_{ctrl}
$T_{data} = 7.5ms$	Data slot length T_{data}
$T_{gap,data} = 2ms$	Gap time $T_{gap,data}$ between consecutive data slots
$T_{gap,ctrl} = 15ms$	Gap time $T_{gap,ctrl}$ between control slot and first data slot
$T_{guard} = 0.5ms$	Guard time T_{guard} for non initiating nodes
$N_{ctrl} = 3$	Number of retransmissions for control slots
$N_{data} = 2$	Number of retransmissions for data slots
$M = 16$	Maximum number of bytes for each data slot

The parameters for the TelosB port deviate slightly. The slot times T_{ctrl} and T_{data} changed due to the different radios used on the platforms while the increased gap and guard times ($T_{gap,data}$ and T_{guard}) are due to issues with the stability of the digitally-controlled oscillator during start up.

Parameter	Description
$T = 2s$	Round period T
$T_{ctrl} = 29ms$	Control slot length T_{ctrl}
$T_{data} = 8.28ms$	Data slot length T_{data}
$T_{gap,data} = 8ms$	Gap time between consecutive data slots $T_{gap,data}$
$T_{gap,ctrl} = 15ms$	Gap time between control slot and first data slot $T_{gap,ctrl}$
$T_{guard} = 2ms$	Guard time T_{guard} for non initiating nodes
$N_{ctrl} = 3$	Number of retransmissions for control slots
$N_{data} = 2$	Number of retransmissions for data slots
$M = 16$	Maximum number of bytes for each data slot

5.1.2 LWB Reimplementation

LWB overview

The *Low-power Wireless Bus (LWB)*[8], presented in 2012, creates a bus like communication network using Glossy floods to allow for different kinds of traffic pattern, such as many-to-one, one-to-many and many-to-many, and is optimised for periodic traffic usage. By now, LWB is a well established and proven protocol

and publicly available, which makes it a perfect candidate to be reimplemented and compared in the scope of this thesis.

LWB features two node types, one host and multiple source nodes. Source nodes are able to request bandwidth on the network by issuing stream requests, where each stream corresponds to a certain traffic demand of this node. Those streams are identified by their stream id and have a *inter-packet interval (IPI)* assigned to them, which allows the host to meet the traffic demands of the network by scheduling slots according to the IPIs of all known streams.

The LWB reimplementation was done while having the original implementation as a point of reference, therefore the behaviour of both implementations is almost identical, which allows for a fair comparison between the native and the GMW-based implementation.

The code is available here (`http://bit.ly/glossymiddleware`), the LWB port is found under `<path to repo>/os/net/mac/glossy/gmw/lwb`, the test application under `<path to repo>/examples/gmw-lwb-test`.

LWB Host Reimplementation

Compared to the original LWB implementation, the GMW-based reimplementation only has a few differences. The LWB scheduler is reused for the most part, however instead of flags in the schedule/control packet being used to indicate the presence of SACK (Stream request ACKnowledgement) and contention slots, those slots are scheduled using the `slots` array-field of the schedule section (see Section 3.1.1). Since only the host is always aware of which slot has which functionality, the common LWB header has be extended to not only accommodate the recipient and stream id, but also the packet type (data, stream request and stream acknowledgement packet), which made use of the previously unused padding byte for 16bit alignment from the original LWB implementation.

LWB Source Reimplementation

The source node reimplementation is also very close to the original implementation. The main differences lie in the use of the packet type field as described above and in the reimplementation of the LWB state machine (Figure 5.1). To reproduce the behaviour of the LWB state machine, each LWB state was mapped to a GMW superstate (see Section 2.2.2). Using the `on_control_slot_post_callback` (Section 3.2.1) to update the LWB state machine allows the LWB reimplementation to return a GMW superstate, therefore giving the LWB reimplementation full control over the state of the middleware.

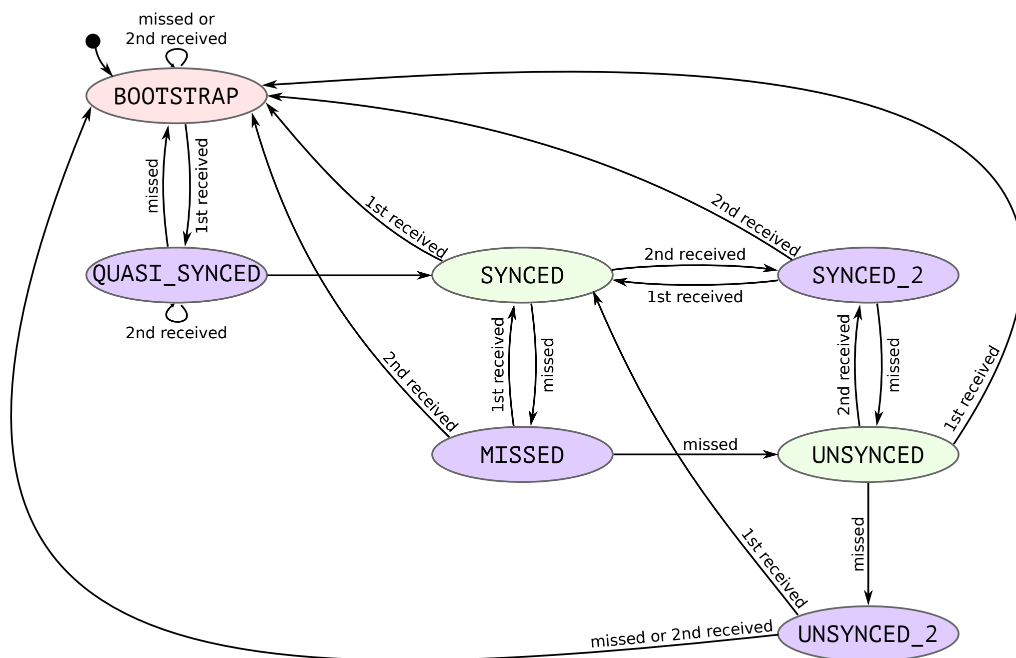

Figure 5.1: The LWB state machine. The colour of the states represent the associated GMW super state (red is `BOOTSTRAP`, purple `SUSPENDED` and green `RUNNING`).

The parameters used by both LWB implementations:

Parameter	Description
$T = 2s$	Round period T
$T_{ctrl,GMW} = 28ms$	Control slot length T_{ctrl}
$T_{sched,LWB} = 15ms$	Schedule slot length T_{sched}
$T_{data,GMW} = 7.3ms$	Data slot length T_{data}
$T_{data,LWB} = 8ms$	Data slot length T_{data}
$T_{gap,data} = 4ms$	Gap time $T_{gap,data}$ between consecutive data slots
$T_{gap,ctrl} = 4ms$	Gap time $T_{gap,ctrl}$ between control slot and first data slot
$T_{guard} = 0.5ms$	Guard time T_{guard} for non initiating nodes
$N_{ctrl} = 3$	Number of retransmissions for control slots
$N_{data} = 3$	Number of retransmissions for data slots
$M = 8$	Maximum number of bytes for each data slot

Note that parameters that are not directly comparable or slightly different are marked with LWB or GMW in the subscript.

5.1.3 eLWB Reimplementation

eLWB overview

The *Event-based Low-power Wireless Bus (eLWB)*[22], presented in 2017, is a modification of the behaviour of LWB to be able to quickly respond to aperiodic event-triggered traffic demands. Instead of scheduling regular data slots for nodes to transmit data, the periodic schedule only contains an *event contention slot*. If an event is detected by one or multiple source nodes, they send a packet in the event contention slot to notify the host about the event. If an event has been detected, the host schedules an *event round*, where each known source node has a data slot assigned in which they can inform the host about the amount of data they have gathered about this event. The event round is then followed by the *data round*, where each node gets assigned a number of data slots according to the amount of data reported in the event round.

Contrary to the LWB reimplementation, the reimplementation of eLWB was done only with the paper as a reference, therefore there are some differences in the behaviour, for example the GMW-based reimplementation does not implement its own state machine and relies on the state machine provided by GMW. Another difference is that the original eLWB detects an event when receiving anything during the event contention slot (i.e. something was transmitted), while GMW is only able to handle packets that were successfully received (i.e. with a correct CRC).

implementation considers receiving anything during the event contention slot an event, while GMW only is able to handle packets that were received completely.

The code is available here (http://bit.ly/glossymiddleware), the eLWB port is found under <path to repo>/os/net/mac/glossy/gmw/elwb, the test application under <path to repo>/examples/gmw-elwb-test.

eLWB Implementation

The host implementation features a scheduler, built on top of a simple state machine (Figure 5.2) to determine the round type. The user byte section of the control packet is used to inform the source nodes about the current round type which makes the source implementation trivial.

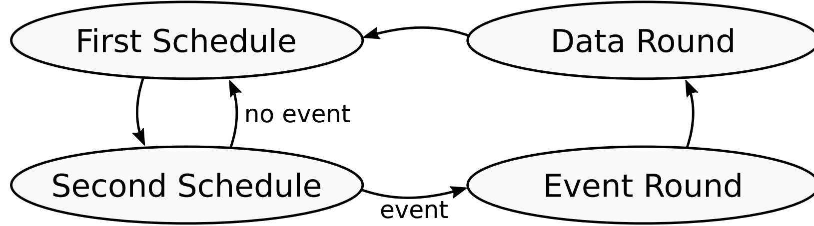

Figure 5.2: The detection of an event in the event contention slot results in the scheduling of two additional rounds, the data and the event round.

The parameters used by both eLWB implementations:

Parameter	Description
$T = 5s$	Round period T
$T_{ctrl,GMW} = 28ms$	Control slot length T_{ctrl}
$T_{sched,eLWB} = 10ms$	Schedule slot length T_{sched}
$T_{data,GMW} = 9.6ms$	Data slot length T_{data} (calculated using GMW_T_SLOT_MIN)
$T_{data,eLWB} = 8ms$	Data slot length T_{data} (hardcoded)
$T_{gap,data} = 2ms$	Gap time $T_{gap,data}$ between consecutive data slots
$T_{gap,ctrl} = 2ms$	Gap time $T_{gap,ctrl}$ between control slot and first data slot
$T_{guard} = 0.5ms$	Guard time T_{guard} for non initiating nodes
$N_{ctrl} = 3$	Number of retransmissions for control slots
$N_{data} = 3$	Number of retransmissions for data slots
$M = 16$	Maximum number of bytes for each data slot

Note that parameters that are not directly comparable or slightly different are marked with eLWB or GMW in the subscript.

Further Protocol Implementations

As mentioned at the beginning of this chapter, besides the reimplementation of both LWB and eLWB, the evaluation of the usability includes other protocols covering the feature set of the middleware. These implementations can be also be found in the repository `http://bit.ly/glossymiddleware` under `examples`.

The main implementation consist of the following builds:

GMW-Config-Test shows the flexibility of the middleware by scheduling round with changing intervals. Additionally it frequently alters the config section by changing the number of retransmissions, the slot length and the gap time.

GMW-Slot-Config-Test features the slot config section and shows rounds consisting of slots with individual parameters for slot length and number of retransmissions.

GMW-Test-Fast illustrates the ability of the Glossy Middleware to handle rounds in quick succession; in this example the round period is 30 milliseconds.

GMW-Test-Slow shows the other end of the round period spectrum with very long round periods of 5 minutes.

Summary

The implementation of different protocols, ranging from existing high-level protocols to protocols specifically designed to showcase and test the features of the middleware, as well as the resulting extensive spectrum of values for the parameters of the middleware successfully demonstrate its usability.

5.2 Evaluation of the Performance Overhead

This section covers the metrics used to evaluate the performance overhead introduced by the middleware, regarding binary size, maximum stack size, duty cycles of CPU and radio, and reliability.

5.2.1 Binary Size

This subsection presents the binary sizes of different builds and compares them. The output of the `size` tool of the msp430-toolchain returns 5 values:

text refers to memory that is either code or holding constant variables that do not change during execution and are stored in flash memory.

data contains variables that have an initialisation value that is not zero. This means those variables use memory in both RAM (value during execution) and flash (values for initialisation).

bss specifies the amount of RAM used by variables, initialised as 0 during the boot process of the micro controller.

dec refers to the sum of text, data and bss as a decimal value.

hex is the same value as dec but in hexadecimal.

The CC430-based DPP platform offers 32KB of flash memory and 4KB of RAM, while the MSP430-based TelosB offers 48KB flash and 10KB RAM. Since the data memory section has to be accounted both in flash and in RAM, the following usage percentages are calculated as shown in Equation (5.1) and Equation (5.2)

$$\%_{Flash} = \frac{M_{text} + M_{data}}{M_{Flash}} \quad (5.1)$$

$$\%_{RAM} = \frac{M_{bss} + M_{data}}{M_{RAM}} \quad (5.2)$$

Hello World

To give a baseline of the size of Contiki-NG, the binary sizes of a Hello World are shown in Table 5.1 for the DPP platform and Table 5.2 for the TelosB platform.

text	data	bss	dec
12870	88	1060	14018

Table 5.1: Hello World built for the DPP platform (39.54% flash and 28.03% RAM usage).

text	data	bss	dec
13348	122	1044	14514

Table 5.2: Hello World built for the TelosB platform (27.40% flash and 11.39% RAM usage).

Compared to the available memory, Contiki itself already uses 39.54% respectively 27.40% of the available flash and 28.03% respectively 11.39% of the RAM.

GMW-Test

Table 5.3 (DPP) and Table 5.4 (TelosB) depict the binary sizes of the GMW-Test build. Flash and RAM usage almost doubled compared to the Hello World build. On the DPP platform, this leaves around 8.4KB of flash memory available for application data. Although this may seem little, the DPP is a dual-processor platform where most computations and memory usages are expected to run on the application side. Thus, 8.4KB of available memory on the communication side can be perfectly sufficient.

text	data	bss	dec
23618	132	2072	25822

Table 5.3: GMW-Test, built for the DPP platform (72.48% flash and 53.81% RAM usage).

text	data	bss	dec
24848	174	2154	27176

Table 5.4: GMW-Test, built for the TelosB platform (50.91% flash and 22.73% RAM usage).

LWB Reimplementation

Table 5.5 (original) and Table 5.6 (GMW-based) show the memory usage for two implementations of the Low-power Wireless Bus, both built for the DPP platform. Compared to the GMW-Test build above which is a minimal example for GMW, the GMW-based reimplementation of LWB gains roughly 5KB in flash use, leaving $\sim$3.3KB unused. The original LWB implementation is only smaller by 1.9KB in terms of flash usage. In terms of RAM usage, the GMW-based implementation added around 1KB, mostly for the buffers used by the FIFOs for incoming and outgoing data. The overhead in terms of RAM between the original LWB implementation and the GMW-based reimplementation is around 200bytes, originating in the larger buffer size used by GMW for the control packet and additional variables used by the features of the middleware.

text	data	bss	dec
26810	164	2814	29788

Table 5.5: Original LWB built for the DPP platform (82.32% flash and 72.71% RAM usage).

text	data	bss	dec
28710	168	3008	31886

Table 5.6: GMW-based LWB reimplementation built for the DPP platform (88.13% flash and 77.54% RAM usage).

eLWB Reimplementation

Table 5.7 (original) and Table 5.8 (GMW-based) depict the memory usage for the two implementations of the event-based low-power wireless bus with a similar outcome as in the LWB case. Again, the flash usage is higher by around 2KB for the middleware build. The lower usage of RAM in case of the GMW-based reimplementation can be explained by the use of a larger FIFOs in the original eLWB build compared to a single buffer solution in the reimplementation.

text	data	bss	dec
24190	218	3250	27658

Table 5.7: Original eLWB built for the DPP platform (74.49% flash and 84.67% RAM usage).

text	data	bss	dec
26242	148	3098	29488

Table 5.8: GMW-based eLWB reimplementation built for the DPP platform (80.54% flash and 79.25% RAM usage).

Summary

The memory footprint of the middleware is significant, especially on the DPP platform where memory is scarce, compared to the TelosB platform. With the DPP being designed as a dual-processor platform, the memory still available on the communication processor should be sufficient for most applications, especially since the native implementations of both LWB and eLWB do not offer a much smaller memory footprint.

5.2.2 Maximum Stack Size

The other memory related metric used in the performance evaluation is the maximum stack size which is expected to remain roughly constant over a runtime of one hour, however smaller differences can occur due to interrupt nesting. This metric is important since it shows the memory usage of the execution, therefore combining it with the amount of available RAM from the previous subsection shows the total RAM usage of an application.

GMW-Test

The stack size in the GMW-Test application for the DPP platform is very consistent at 228bytes for all nodes. On the TelosB it is varying between 202 and 222bytes. Considering the RAM still available is 1892bytes for the DPP platform and 7912bytes for TelosB platform leaves us with plenty of headroom.

LWB Reimplementation

For the LWB builds, the original LWB implementation has a constant stack size of 208bytes for source nodes and 222bytes for the host node. The GMW-based reimplementation is almost constant at 220bytes for all nodes, except for one source node at 222bytes. Taking into account the RAM still available (920bytes for the GMW-based and 1118bytes for the native implementation) amounts to 0.7KB of unused RAM for the GMW-based and 0.7KB of unused RAM for the native implementation.

eLWB Reimplementation

In case of eLWB, the original implementation displays quite a big range of stack sizes: the host node has a stack size of 258bytes, while the source nodes vary between 214 and 242bytes. The GMW-based reimplementation is rather constant again at 228bytes for the host node and 234bytes for all source nodes except one outlier at 220bytes. In this case the native eLWB implementation has the larger total RAM footprint with 370bytes still available while the GMW-based implementation still has 616bytes available.

Summary

Combined with the previous section we can show that, while memory is scarce, we still have a comfortable amount of unused RAM, leaving some room for future development.

5.2.3 Duty-Cycle

This subsection evaluates the duty cycle of GMW-based builds on the DPP and the TelosB platform as well as comparing original protocol implementations to GMW-based reimplementations in order to measure the performance overhead introduced by the middleware. The values shown are the minimal, maximum and average values, for all 25 nodes participating in the test 1 hour test.

GMW-Test

The CPU overhead of the TelosB platform depicted in Figure 5.3 is due to a much less efficient driver implementation of the extended timers on the TelosB platform and offers considerable room for improvement.

The radio overhead is due to the slightly longer slot time and the significantly larger guard time for the TelosB build which had to be increased to accommodate the inaccuracies of the digitally-controlled oscillator used in the TelosB.

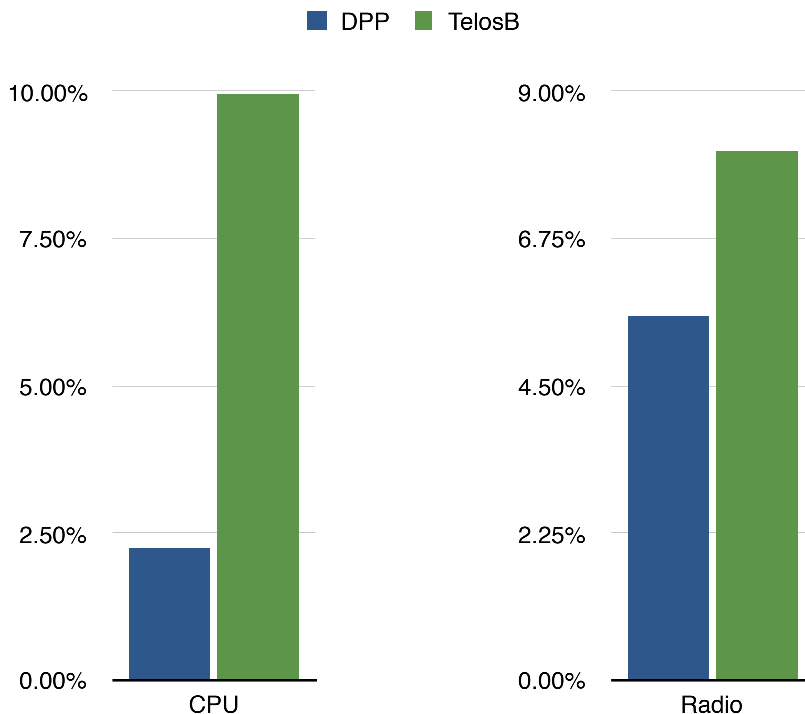

Figure 5.3: The TelosB platform has a substantial overhead in both CPU and radio activity.

	CPU	Radio
Min	2.22%	5.19%
Max	2.27%	6.18%
Avg	2.24%	5.58%

Table 5.9: DPP duty cylce

	CPU	Radio
Min	7.88%	7.75%
Max	10.62%	8.57%
Avg	9.97%	8.10%

Table 5.10: TelosB duty cycle

LWB Reimplementation

Figure 5.4 shows the average duty cycle of the CPU and the radio, comparing the original LWB implementation with the GMW-based reimplementation (with numerical data visible in Table 5.11 and Table 5.12). The slight overhead in CPU is due to the added callback infrastructure and and the additional computations used to support the feature set of the middleware while the radio overhead originates in the slightly bigger control packet used by the middleware compared to the schedule used in the original LWB implementation.

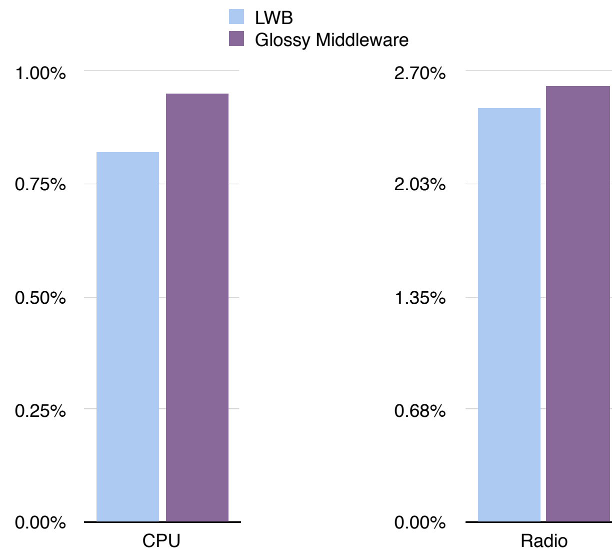

Figure 5.4: A slight overhead in both CPU as well as radio activity for the GMW-based reimplementation.

	CPU	Radio
Min	0.81%	2.19%
Max	0.84%	2.65%
Avg	0.82%	2.48%

Table 5.11: Original LWB

	CPU	Radio
Min	0.94%	2.12%
Max	0.99%	3.12%
Avg	0.95%	2.61%

Table 5.12: GMW-based LWB

eLWB Reimplementation

Figure 5.5 shows the average duty cycle of CPU and radio for the original eLWB implementation as well as the GMW-based reimplementation. The huge discrepancy in radio activity is explained by the original eLWB implementation considering any radio reception during the event contention slot as an event, while the GMW-based reimplementation only considers successfully received packets as an event. Since the source nodes generate a packet with a probability of 10% each round, with 25 nodes there are often multiple nodes transmitting in the event contention. While the original eLWB implementation is able to react to any radio signal and therefore schedule the event and the data round, the GMW-based implementation is not able to detect every event contention successfully, and therefore not scheduling the additional event and data round. Note that, while this is a serious issue, since the time delay between an event taking place and its corresponding data being transferred over the network is an important aspect of the eLWB design, the data is not lost since it will be transmitted whenever the next event contention slot is received successfully by the host. Compared to the LWB reimplementation, the CPU overhead is lower, but that is an artefact due to the missed event contention slots.

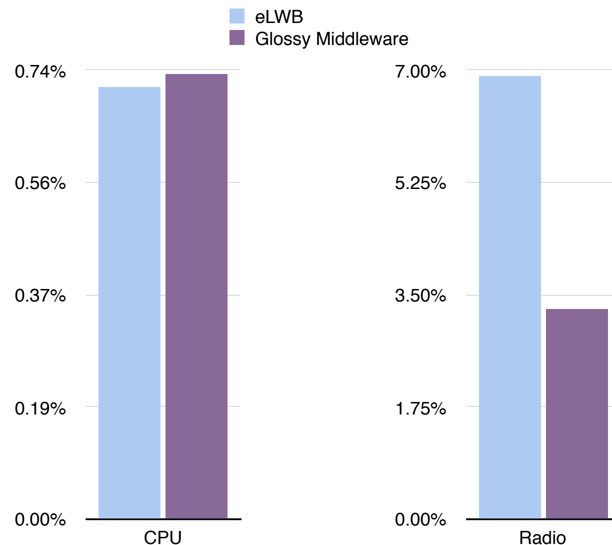

Figure 5.5: The significant difference of radio activity is due to different event detection in the original eLWB and the GMW-based reimplementation.

	CPU	Radio
Min	0.69%	6.86%
Max	0.75%	7.03%
Avg	0.71%	6.93%

Table 5.13: Original eLWB

	CPU	Radio
Min	0.72%	2.81%
Max	0.76%	3.93%
Avg	0.73%	3.29%

Table 5.14: GMW-based eLWB

Summary

The comparison of the duty cycle of the GMW-Test build on the two platforms DPP and TelosB show an overhead for the TelosB in both CPU as well as radio activity. The increased CPU activity is due to less efficient driver implementations of hardware timers while the radio overhead originates in the increased slot and guard times due to inaccuracies of the DCO used in the TelosB.

For the LWB and eLWB implementations we can present an acceptable overhead both in CPU and radio activity, additionally we reveal a shortcoming of the middleware when it comes to the event detection in the eLWB implementation.

5.2.4 Reliability

This subsection depicts the reliability of GMW-based builds on the DPP and the TelosB platform as well as comparing original protocol implementations to GMW-based reimplementations. We use the Glossy *Flood Success Rate (FSR)* as a metric. This value represents the percentage of successful Glossy floods. The total flood count is given by the amount of Glossy floods where the radio detected the beginning of a packet; this prevents to wrongly classify an unused contention slot (i.e. in which no node transmitted) as an unsuccessful flood. If at least one complete packet was received during a flood it is considered successful.

GMW-Test

Figure 5.6 shows the average flood success rate of the GMW-Test build, both platforms perform very well with a success rate of almost 100%.

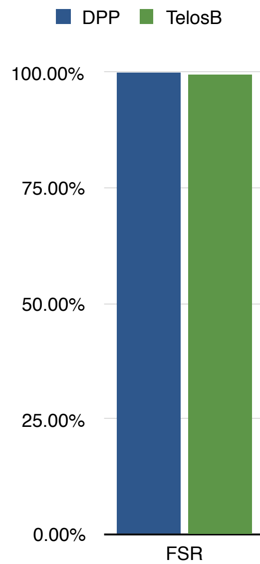

Figure 5.6: Almost 100% FSR for the DPP as well as the TelosB platform.

	DPP	TelosB
Min	99.60%	99.35%
Max	99.96%	99.95%
Avg	99.82%	99.76%

Table 5.15: FSR for the GMW-Test build on DPP and TelosB respectively.

LWB Reimplementation

A similar view is depicted in Figure 5.7, where both, the original LWB as well as the GMW-based implementation score a flood success rate of almost 100%.

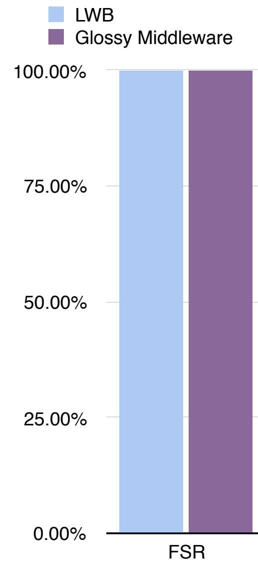

Figure 5.7: Almost 100% FSR for both the original LWB implementation as well as the GMW-based reimplementation.

	Original	GMW-based
Min	99.91%	99.83%
Max	100.00%	100.00%
Avg	99.98%	99.96%

Table 5.16: FSR for the original LWB build and the GMW-based reimplementation.

eLWB Reimplementation

For the eLWB builds we have a different scenario, with both implementation having a comparable flood success rate of ∼97%. The reason for this lower FSR value is that contention slots are more often used by eLWB. While in the standard LWB design the contention slot is only used to request a stream, it remains empty for most of the duration of the test once the network has stabilised. eLWB on the other hand makes frequent use of the event contention slot. This test simulates very frequent events, therefore a lot of nodes use the event contention slot to signal the occurrence of an event. Due to multiple nodes transmitting during the same slot it is correctly counted as an Glossy flood, however interference sometime prevents the successful reception of a packet, leading to a lower flood success rate.

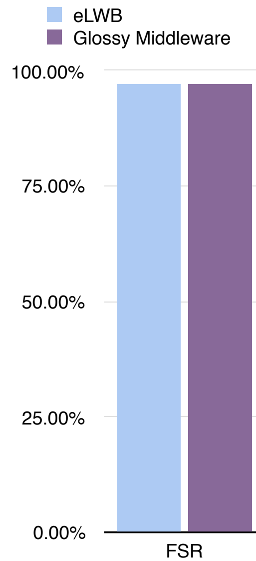

Figure 5.8: The lower FSR value of eLWB implementation is a result of the frequent use of contention slots in the design of eLWB and not necessarily a sign of lower reliability.

	Original	GMW-based
Min	90.68%	92.81%
Max	99.91%	98.27%
Avg	97.21%	96.98%

Table 5.17: FSR for the GMW-Test build on DPP and TelosB respectively.

Summary

In terms of reliability, the middleware demonstrates a good performance on both the DPP as well as the TelosB platform. It also showcases comparable results when being compared to the existing implementations of LWB and eLWB.

This concludes the evaluation of the middleware. The extensive range of implemented protocols successfully shows its usability while the performance evaluation illustrates the acceptable small overhead of the middleware in terms of memory usage and energy consumption while offering a high reliability comparable to existing protocol implementations.

Chapter 6

Conclusion and Future Work

In this thesis, we identified challenges with the current implementation of Glossy-based protocols, especially regarding low-level implementations, where similar frameworks have been reimplemented from the ground up again and again. This thesis proposes a solution to some of those challenges by moving low-level complexity into a separate fixed middleware which allows development of Glossy-based protocols on a higher level.

The design of this proposed *Glossy Middleware* is based on a well-established *round-based framework*, an approach used in many existing Glossy-based protocols. Our middleware features a fixed, yet flexible interface, granting high-level control over important Glossy parameters and the round structure in order to match the runtime behaviour to the application requirements.

Chapter 3 explains the interface of the Glossy Middleware extensively, serving as an user manual for future use.

We evaluated the middleware in terms of its usability and low performance overhead by reimplementing existing protocols and comparing them to native implementations, demonstrating the flexibility of the middleware and its acceptable low overhead in terms of memory usage and energy efficiency.

As shown in the evaluation, while the middleware allows reimplementations of protocols, the flexibility could still be improved, for example by adding the ability to detect and report radio activity during the duration of a slot. Other possible enhancements consist of changes to the behaviour of Glossy, e.g. implementing channel hopping capabilities or adapting the use of additional communication primitives other than Glossy, such as Robust Flooding[16] or Chaos[14].

In terms of its usability, the middleware could greatly benefit from additional platform support, as it currently only fully supports the DPP platform and to some extent the TelosB platform (which is currently lacking a clean driver implementation for the hardware timers). With an existing Glossy port, the OpenMote platform represents a good candidate.

Adding simulation capabilities would also improve the usability of the mid-

dleware, especially during development and debugging of higher-layer protocols. Since the middleware has been implemented using Contiki, the Cooja simulator[20] would be a natural candidate.

Our middleware is a tool made to facilitate the development of Glossy-based protocols. To fully reach this objective, the whole middleware will be published as open-source software, ideally by integrating it into the Contiki-NG repository.

Last but not least, the Glossy Middleware is a useful and versatile tool for the future to develop, compare and port Glossy-based protocols. One application that comes to my mind is the EWSN dependability competition[21] where I believe GMW could be used to participate and maybe even win in the next installation of this competition.

Bibliography

[1] Beshr Al Nahas, Simon Duquennoy, and Olaf Landsiedel. Network-wide consensus utilizing the capture eeect in low-power wireless networks. In *Proceedings of the International Conference on Embedded Networked Sensor Systems (ACM SenSys 2017)*, page 14, 2017.

[2] Ioannis Chatzigiannakis, Georgios Mylonas, and Sotiris Nikoletseas. 50 ways to build your application: A survey of middleware and systems for wireless sensor networks. In *Emerging Technologies and Factory Automation, 2007. ETFA. IEEE Conference on*, pages 466–473. IEEE, 2007.

[3] Chee-Yee Chong and Srikanta P Kumar. Sensor networks: evolution, opportunities, and challenges. *Proceedings of the IEEE*, 91(8):1247–1256, 2003.

[4] Contiki-NG. Contiki-ng. `http://www.contiki-ng.org/`. [Online; accessed 7 June 2018].

[5] Manjunath Doddavenkatappa, Mun Choon Chan, Ben Leong, et al. Splash: Fast data dissemination with constructive interference in wireless sensor networks. In *NSDI*, pages 269–282, 2013.

[6] Adam Dunkels, Bjorn Gronvall, and Thiemo Voigt. Contiki-a lightweight and flexible operating system for tiny networked sensors. In *Local Computer Networks, 2004. 29th Annual IEEE International Conference on*, pages 455–462. IEEE, 2004.

[7] Adam Dunkels, Oliver Schmidt, Thiemo Voigt, and Muneeb Ali. Protothreads: Simplifying event-driven programming of memory-constrained embedded systems. In *Proceedings of the 4th international conference on Embedded networked sensor systems*, pages 29–42. Acm, 2006.

[8] Federico Ferrari, Marco Zimmerling, Luca Mottola, and Lothar Thiele. Low-power wireless bus. In *Proceedings of the 10th ACM Conference on Embedded Network Sensor Systems*, pages 1–14. ACM, 2012.

[9] Federico Ferrari, Marco Zimmerling, Lothar Thiele, and Olga Saukh. Efficient network flooding and time synchronization with glossy. In *Information Processing in Sensor Networks (IPSN), 2011 10th International Conference on*, pages 73–84. IEEE, 2011.

[10] IEEE. 802.15.4-2003. *IEEE Std.*, 2003.

[11] MEMSIC Inc. Telosb datasheet. `http://www.memsic.com/userfiles/files/Datasheets/WSN/telosb_datasheet.pdf`. [Online; accessed 6 June 2018].

[12] Timofei Istomin, Amy L Murphy, Gian Pietro Picco, and Usman Raza. Data prediction+ synchronous transmissions= ultra-low power wireless sensor networks. In *Proceedings of the 14th ACM Conference on Embedded Network Sensor Systems CD-ROM*, pages 83–95. ACM, 2016.

[13] Romain Jacob, Marco Zimmerling, Pengcheng Huang, Jan Beutel, and Lothar Thiele. End-to-end real-time guarantees in wireless cyber-physical systems. In *Real-Time Systems Symposium (RTSS), 2016 IEEE*, pages 167–178. IEEE, 2016.

[14] Olaf Landsiedel, Federico Ferrari, and Marco Zimmerling. Chaos: Versatile and efficient all-to-all data sharing and in-network processing at scale. In *Proceedings of the 11th ACM Conference on Embedded Networked Sensor Systems*, page 1. ACM, 2013.

[15] Krijn Leentvaar and Jan Flint. The capture effect in fm receivers. *IEEE Transactions on Communications*, 24(5):531–539, 1976.

[16] Roman Lim, Reto Da Forno, Felix Sutton, and Lothar Thiele. Competition: Robust flooding using back-to-back synchronous transmissions with channel-hopping. In *Proceedings of the European Conference on Wireless Sensor Networks (EWSN)*, 2017.

[17] Roman Lim, Federico Ferrari, Marco Zimmerling, Christoph Walser, Philipp Sommer, and Jan Beutel. Flocklab: A testbed for distributed, synchronized tracing and profiling of wireless embedded systems. In *Proceedings of the 12th international conference on Information processing in sensor networks*, pages 153–166. ACM, 2013.

[18] Luca Mottola and Gian Pietro Picco. Middleware for wireless sensor networks: an outlook. *Journal of Internet Services and Applications*, 3(1):31–39, 2012.

[19] Martijn Onderwater. An overview of centralised middleware components for sensor networks. *International Journal of Ad Hoc and Ubiquitous Computing*, 21(3):180–193, 2016.

[20] Fredrik Osterlind, Adam Dunkels, Joakim Eriksson, Niclas Finne, and Thiemo Voigt. Cross-level sensor network simulation with cooja. In *Local computer networks, proceedings 2006 31st IEEE conference on*, pages 641–648. IEEE, 2006.

[21] Markus Schuß, Carlo Alberto Boano, Manuel Weber, and Kay Römer. A competition to push the dependability of low-power wireless protocols to the edge. In *Proceedings of the European Conference on Wireless Sensor Networks (EWSN)*, 2017.

[22] Felix Sutton, Reto Da Forno, David Gschwend, Tonio Gsell, Roman Lim, Jan Beutel, and Lothar Thiele. The design of a responsive and energy-efficient event-triggered wireless sensing system. *Proc. of ACM EWSN*, pages 144–155, 2017.

[23] Felix Sutton, Marco Zimmerling, Reto Da Forno, Roman Lim, Tonio Gsell, Georgia Giannopoulou, Federico Ferrari, Jan Beutel, and Lothar Thiele. Bolt: A stateful processor interconnect. In *Proceedings of the 13th ACM Conference on Embedded Networked Sensor Systems*, pages 267–280. ACM, 2015.

[24] Miao-Miao Wang, Jian-Nong Cao, Jing Li, and Sajal K Dasi. Middleware for wireless sensor networks: A survey. *Journal of computer science and technology*, 23(3):305–326, 2008.

[25] Marco Zimmerling, Pratyush Kumar, Federico Ferrari, Luca Mottola, and Lothar Thiele. Energy-efficient real-time communication in multi-hop low-power wireless networks. Technical report, Technical report, TIK Report, 2015.