IN-DEPTH BREAKDOWN OF A 6LOWPAN STACK FOR SENSOR NETWORKS

Sergio Lembo, Jari Kuusisto, Jukka Manner

Aalto University - School of Science and Technology
Department of Communications and Networking
P.O.Box 13000, 00076 Aalto, Finland

ABSTRACT

There exist several open source 6LoWPAN stacks for researchers to experiment with. However, they often lack sufficient and in-depth description of the internal operation, which makes extending the stacks difficult for many of us. This paper is an extended version of our previous work documenting the internal logic of an implemented and working 6LoWPAN stack, Nanostack (v1.1). We present first the main architecture of the stack and subsequently describe the path followed by a packet transiting the different layers. Then we provide details of each one of the layers in the stack, with exception of the ICMP layer. The main logic in the MAC layer is comprehensively explained, and an undocumented layer used in nodes working as Gateway, the NRP layer, is presented. In addition we provide a conceptual view of the layering of the stack relative to the hardware platform and enumerate the typical tasks running in a sensor node.

KEYWORDS

6LoWPAN, Nanostack, Wireless Sensor Network, WSN

1. INTRODUCTION

Advances in microelectronics, low-power electronics and micro-electro-mechanical systems (MEMS), along with low manufacturing costs, have led to the development of wireless sensor networks (WSNs). These networks consist of individual devices that have been interconnected wirelessly in order to perform diverse tasks. The inter-device communication requires a suitable communication protocol that can be chosen among a diverse range of standard and non-standard protocols. The widely adopted Internet protocol (IP) is used traditionally in computer networks to provide a uniform and standardized way of communicating that is independent from the actual physical communication. Because IP has many advantages (e.g. already supported standards and extensive interoperability) it is also introduced to the WSNs. 6LoWPAN defines the IP version 6 (IPv6) networking in WSNs [1].

Currently there are few known open-source implementations of 6LoWPAN stacks; recent surveys report four known open-source 6LoWPAN implementations [2], Nanostack [3] is one of them. Among the available open-source 6LoWPAN implementations, the authors selected Nanostack due to two strong facts. Firstly, the stack operates on top of a real time kernel (FreeRTOS [4]), making it suitable for time critical tasks like real time control or network synchronization. Secondly, the stack implements mesh-under routing (in contrast of other stacks that adopt route-over routing [2]).

Nanostack (v1.1) is an implemented and working 6LoWPAN stack distributed under GPL license. In this paper we focus on the latest open-source Nanostack release, version 1.1, and hereafter refer to it simply as Nanostack. Nanostack implements most of the RFC4944 [1], namely, processing of Mesh-Header, Dispatch-Header (and its alternatives LOWPAN_HC1, LOWPAN_BC0 and IPv6), mesh-under addressing and compression of IP and UDP headers.

DOI : 10.5121/ijcnc.2010.2614

The implementation currently does not support 6LoWPAN fragmentation and we are unable to report at what extent unicast and multicast address mapping is supported since code related to the processing of native (non-compressed) IPv6 addresses was not considered in our study. The stack was written in C language and designed to work on top of FreeRTOS kernel [4].

Despite that Nanostack is an open-source stack, Nanostack authors and users never created proper documentation of the internal logic of the stack. We provided the first documentation of the stack in [5], documenting and analyzing its internal architecture. This paper extends our previous work documenting Nanostack [5], explaining in detail the main logic implemented in the MAC layer and introducing the NRP layer. In addition we provide a conceptual view of layering of the stack relative to the hardware platform (sensor node), enumerate the typical tasks running in a sensor node and describe the functionality of a sensor node working as Gateway. By exposing the architecture and logic of the stack, the reader can evaluate at what extent Nanostack is suitable for an intended purpose given different requirements and constraints (modularity, concurrency, real-time operation, energy consumption, memory size, etc.).

In the following sections we document the stack architecture and logic. In Section 2 we introduce Nanostack layers, modules and typical tasks. In Section 3 we introduce the architecture of the stack. In the sections that follow (4, 5, 6, and 7) we introduce and explain the operation of each one of the layers in the stack, except the ICMP layer that is not covered in our descriptions. In Section 8 we describe the NRP layer and the case when a sensor node works as Gateway. Finally, we conclude the paper in Section 9.

2. NANOSTACK AND WIRELESS SENSOR NODES

In this section we provide a conceptual view of the layering in the stack relative to the hardware (HW) platform (sensor node), introduce the principal layers (modules) in the stack and enumerate the typical tasks running in a sensor node.

Nanostack operates on top of the FreeRTOS kernel, a multi-platform, mini, Real Time Kernel [4], which in turn runs in a micro-controller located in a hardware platform. In the context of Wireless Sensor Networks (WSN), the hardware platform usually incorporates also sensors and wireless radio circuitry, and receives the name of “wireless sensor node" or “sensor node" among other names used in the literature. Fig.1 provides a relative view of Nanostack and its layers in a hardware platform (wireless sensor node).

In general, in the explanations below we talk simply about nodes and define a node as an entity that implements the stack and is able to transmit and receive 6LoWPAN packets. In the context of this paper it is not a requirement that a node contain sensors.

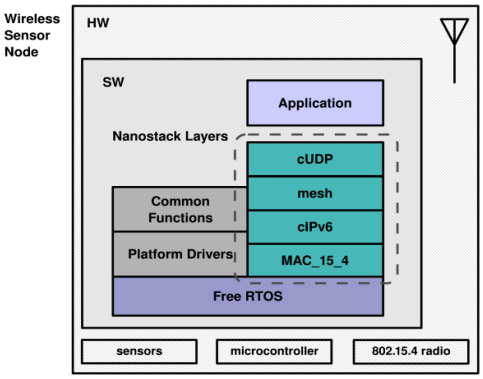

Figure 1: Relative view of Nanostack Layering in HW

2.1. Nanostack Modules and Layers

Nanostack consists of different stack layers, each one in charge of processing the different protocol headers in the packets moving in the stack. The principal layers are shown on Fig. 2. Each one of these layers is introduced in the following sections.

The application layer differentiates from the other layers in the sense that it relates to the stack through a socket interface (Fig. 2), whereas other layers are related to the stack by an elaborated mechanism in charge of moving the packets among the layers (Section 3.2).

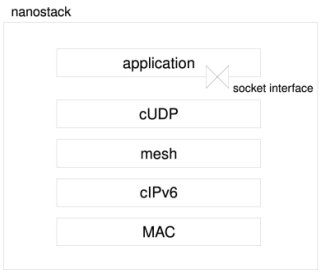

Figure 2: Principal Nanostack Layers

In Nanostack each layer is associated to a module of code. Nanostack modules can be enabled or disabled to modify the functionality of the stack. For example, in Section 8 we describe an optional module called NRP that can be enabled to make a node work as Gateway. In order to simplify the description of the stack the NRP module is not considered until Section 8. Nanostack contains also an ICMP module that introduces ICMP layer functionality. ICMP is a protocol used for communicating exceptional messages between nodes or layers in the stack. ICMP is not covered in our descriptions, mainly due that ICMP logic is distributed in several layers and its inclusion can compromise the simplicity and structure that we intend to adopt in the description of this complex stack. In this paper we assume that a module and a layer are equivalent and talk in general about layers in the rest of the paper.

2.2. Typical Tasks running in a Sensor Node

FreeRTOS is a multitasking operating system. Nanostack was designed to make use of this feature and it executes tasks dedicated to different purposes concurrently.

Typical Nanostack tasks running in a sensor node are shown in Fig. 3. The system task labeled *stack_main()* is the core task of Nanostack; this task is introduced in Section 3.2. The task labeled *mac_task()* process packets at MAC layer level (Section 4). Task *vAppTask()* is a task dedicated to execute user's code at application layer level (Section 7.2). Finally, task *vnrp_task()* is a task that is executed when the NRP module for Gateway functionality is enabled (Section 8).

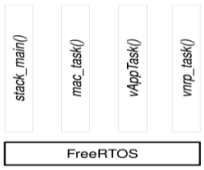

Figure 3: Typical Nanostack tasks running in a sensor node

3. STACK ARCHITECTURE

In this section, we first introduce the main data structure used in the stack for processing packets, followed by the description of a key component in the core architecture, the "dispatcher" module. We also illustrate the overall architecture with the dispatcher module and all the layers in the stack.

3.1. Main Data Structure used in the Stack

In the stack packets are handled by mean of instances of the data structure *buffer_t*. *buffer_t* is defined in file *buffer.h* and reproduced here in Fig. 4.

```
typedef struct
{
  struct socket_t *  socket;
  sockaddr_t         dst_sa;
  sockaddr_t         src_sa;
  module_id_t        from;
  module_id_t        to;
  buffer_direction_t dir;
  uint16_t           buf_ptr;
  uint16_t           buf_end;
  uint16_t           size;
  buffer_options_t   options;
  uint8_t            buf[2];
} buffer_t;
```

Figure 4: Main data structure used in the stack

Throughout this document we denote by *b* an instance of *buffer_t*. The content of a packet is stored in the *buf* field of *buffer_t* : *b.buf*; where the notation expresses a qualified name, in this case *b* is an instance of *buffer_t* and *buf* a field belonging to *b*. (To be more precise in the actual implementation *b* is actually a pointer to an instance of *buffer_t*, and a C language notation *b–>buf* is more appropriate, although not used here to simplify the notation). Linux kernel connoisseurs can imagine *buffer_t* as a structure analogous to the socket buffer structure *sk_buff* in the Linux kernel [6][7].

Packets move through the stack by mean of instances of structure *buffer_t*. (We emphasize here that even when we say that there is a movement of packets inside the stack, Nanostack actually never moves packets but merely pointers to instances *buffer_t*). Incoming packets are directly stored in *b.buf*. Subsequent processing in the different layers of the stack parses the packet and fills accordingly the fields of *buffer_t* structure with the information extracted from the packet. Outgoing packets are constructed layer by layer; each layer adding an appropriate header in *b.buf*.

Three fields in *buffer_t* of particular importance are *b.from*, *b.to* and *b.dir*. Fields *b.from* and *b.to* indicate the previous and posterior layer that processed and will process the packet, respectively. Field *b.dir* indicates the direction of movement of the packet; the value BUFFER_UP indicates an incoming packet moving toward the Application Layer whereas the value BUFFER_DOWN indicates an outgoing packet moving toward the Network Layer.

Hereafter an instance *buffer_t*, *b*, moving in the stack can be thought as a packet under processing (parsing) when the packet is incoming or as a packet under composition when the packet is outgoing. Analogously, when talking about packets moving in the stack we always are talking of the associated instance *buffer_t*, *b*, that carries the packet in the *b.buf* field.

3.2. Dispatcher of Packets and Stack Architecture

Fig. 5 shows the architecture of the stack. In the figure we observe a graphical representation of a queue labeled "Queue *events*". Queue *events* transits incoming and outgoing packets (carried in instances *buffer_t*, *b*) between different layers of the stack. The output of the queue is handled by a dedicated system task labeled *stack_main()* (file *stack.c*) that contains an endless loop checking the arrival of instances *b* from the queue (function *xQueueReceive()*). When a packet is received by *xQueueReceive()* the function *stack_buffer()* is called, passing a pointer of the *buffer_t* instance *b* as parameter. Function *stack_buffer()* behaves as a dispatcher; it contains a suitable logic that delivers the packet to the appropriate destination layer in the stack.

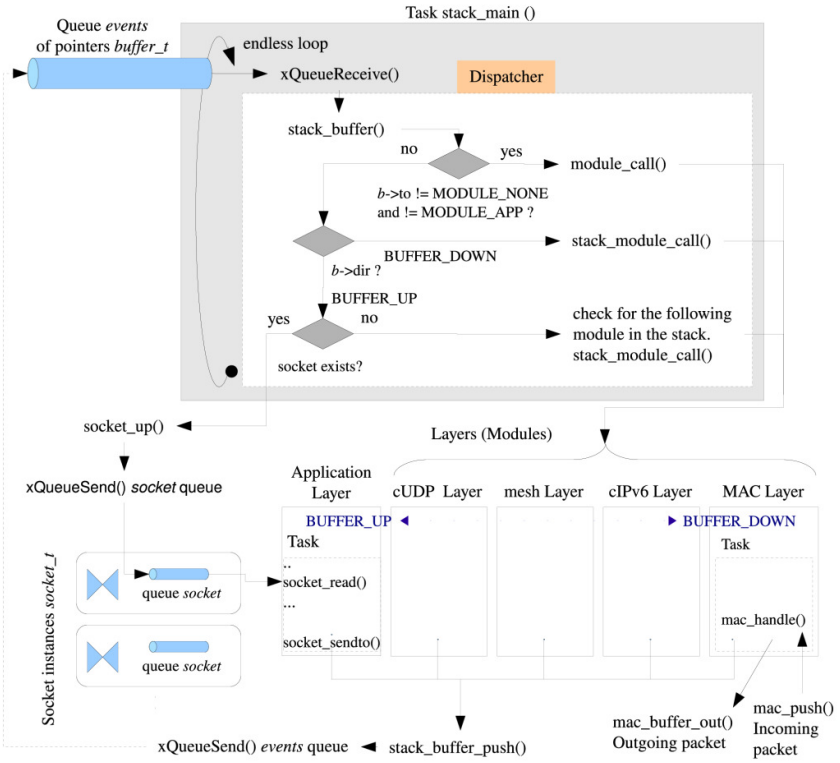

Figure 5: Nanostack architecture

Possible destination layers are depicted below the box representing task *stack_main()*. In the figure we observe the following layers; MAC, cIPV6, mesh and cUDP. Actually the stack contains also an ICMP layer that we do not include in the current description and a NRP layer that we decided to explain separately in Section 8 in order to simplify the description of the stack.

The details of the operation of each one of these layers are described in Sections 4, 5, 6 and 7.1. Note in the figure that the application layer is positioned outside the brace that expands under the dispatcher. The application layer is independent of the stack in the sense that it is not related to the dispatcher and instead receives/sends packets by calling socket functions.

We clarify here that the authors of the stack proposed the name cIPV6 for the layer in charge of processing the headers Mesh-Header, Dispatch-Header, and its alternatives LOWPAN_HC1, LOWPAN_BC0, IPv6, etc. (defined in [1]). And the name cUDP for the layer that processes the

compressed UDP ports (HC_UDP encoding) [1]. In addition each layer is associated to a module of code that can be enabled/disabled in the stack.

In the following subsections we complete the description of the stack by explaining the journey of a packet moving in the stack and introducing additional complementary details.

3.2.1. Journey of a packet in the stack

An incoming packet enters to the stack in the MAC layer through a call to *mac_push()*. In this layer a data packet is stored inside a *buffer_t* structure, in the field *b.buf*. Additional values are stored in *b* fields, for example *b.dir* = BUFFER_UP to indicate that the packet will move toward the application layer in the stack. Then the instance *b* is pushed to queue *events* by calling function *stack_buffer_push()*, which in turn calls *xQueueSend()* to input the instance into the queue (Fig. 5).

The main idea is that an instance *buffer_t b* (actually a pointer) travels among the required layers and in its journey the fields of *b* acquire the information stored in the packet by decomposing it (incoming case) or, conversely, the layers utilize the information in the fields of *b* to compose the packet (outgoing case). The movement between layers is facilitated by every time pushing the instance to queue *events*, and letting the dispatcher deliver the instance to the appropriate layer by doing a function call to a dedicated handler function in the layer, and passing the instance as parameter.

After an incoming packet leaves the MAC layer and enters queue *events*, it is delivered by the dispatcher to the next layer in the BUFFER_UP direction. In the present explanation the next layer is cIPv6. Actually the layers in the stack are modular in the sense that at compilation time some layers can be added or removed from the stack. In our explanation we assume that all the layers shown in Fig. 5 are present.

Layer cIPv6 will decide (as shown in Section 5.2) the next destination layer in the stack. An incoming packet targeted for the current node will move in BUFFER_UP direction towards the application layer, traveling through the cUDP layer. Whereas a packet targeted to another node may move to the mesh layer to obtain routing information, and from there continue BUFFER_DOWN to the MAC layer for forwarding. For outgoing packets the journey involves the displacement of *b* in the BUFFER_DOWN direction through layers cUDP, cIPV6, mesh and MAC.

In essence the processing of the packet in each layer will determine its final fate according to different situations. Subsequent sections in the paper document the different situations and decisions mandated by the logic in the stack. Of particular importance at this point is to observe the logic set in the dispatcher. The dispatcher basically selects the next layer by first checking the existence of a concrete destination in *b.to*, and if this is not present relying on the directions BUFFER_UP or BUFFER_DOWN present in the *b.dir* field (Fig. 5).

3.2.2. Sockets

When an incoming packet leaves the cUDP layer in the stack, it is pushed to queue *events* as usual. At this point the dispatcher recognizes that there is no concrete destination layer (*b.to* = MODULE_NONE), and that the packet is heading BUFFER_UP. At this point the dispatcher looks for an existing socket for the destination address and port number stated in the packet and if the socket exists it moves the instance *b* to a dedicated queue located inside the corresponding socket (queue *socket*) (Fig. 5).

Incoming packets redirected to queue *socket* are retrieved at application layer by calling *socket_read()* function.

Outgoing packets are generated at application layer by invoking function *socket_sendto()*, which will set *b.dir* = BUFFER_DOWN in the instance *b* that holds the packet; finally *b* is pushed to queue *events*.

3.2.3. Stack initialization

The stack is initialized by calling function *stack_init()*. *stack_init()* is in charge of the following:

1 Creating queue *events* by executing FreeRTOS function *xQueueCreate()*.
2 Creating a new system task *"stack_main()"* by executing FreeRTOS function *xTaskCreate()*. This task contains the endless loop shown in Fig. 5, that by means of function *xQueueReceive()* checks for incoming instances *buffer_t* from queue *events*.
3 Allocating memory for a pre-defined number of instances *buffer_t* (set in the constant STACK_BUFFERS_MAX).
4 Initializing indexes *stack_buffer_rd* = *stack_buffer_wr* = 0
5 Creating a dedicated task, *mac_task()*, for processing packets in the MAC layer.

3.2.4. Allocation of pool of instances *buffer_t*

When the stack is initialized a collection (pool) of *buffer_t* instances is created. The pool of instances is arranged in a ring buffer from where the system takes and returns instances during the operation of the stack. The approach to pre-allocate memory in a pool of instances is not only to provide performance to the system. Note that this stack was designed to work in embedded microcontrollers operating with FreeRTOS. FreeRTOS is a multi-platform, mini, Real Time Kernel that provides memory allocation in deterministic time by means of a memory allocation API common for any platform (function *pvPortMalloc()*) [4]. In this sense the simplest RAM allocation scheme does not permit memory to be freed once it has been allocated, and hence the use of a pool of instances.

The pool of pre-allocated instances is managed by means of a ring buffer labeled *stack_buffer_pool[]*. *stack_buffer_pool[]* is indexed by two position pointers, a pointer indicating a reading position (*stack_buffer_rd*) and a pointer indicating a writing position (*stack_buffer_wr*), indicating the next available place where a *buffer_t* instance can be taken or returned respectively (Fig. 6).

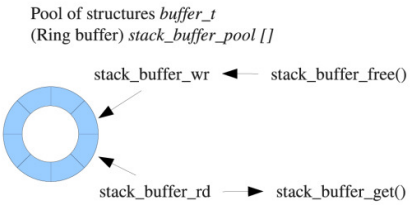

Figure 6: Ring buffer for *buffer_t* instances.

The functions *stack_buffer_get()* and *stack_buffer_free()* are front end functions that will take or return a *buffer_t* instance from the pool and at the same time these functions initialize or reset the fields of the structure. Behind these functions, *stack_buffer_pull()* and *stack_buffer_add()* perform the actual unload/load of *buffer_t* pointers from/to the pool of instances *stack_buffer_pool[]* (Fig. 6).

In the stack a call to *stack_buffer_get()* is always followed by a call to *stack_buffer_free()* at some point in the life of the packet in order to recycle the finite number of instances *buffer_t*.

For example, a packet is returned to the pool after transmission or after discarding it due to some exceptional condition.

4. MAC LAYER

The processing of a packet at the MAC layer level comprises the identification and incorporation of the MAC headers in the MAC sub-layer part of an IEEE 802.15.4 data frame and the logic related to retransmissions when transmission acknowledgements (ACKs) are expected.

Actually the portion of data that we call *packet* is the *MAC protocol data unit* (MPDU) (PSDU at PHY layer level) of the IEEE 802.15.4 Data Frame [8]. This portion of data is the one pointed by the *b.buf* field of the *buffer_t* instance.

The logic in the MAC layer is distributed in two different parts of the code, in the MAC layer module from the stack, and in a dedicated task labeled *mac_task()* (see Section 2.2) that is initialized with the stack.

Fig. 7 depicts the MAC layer module from the stack. As it can be observed in the figure, there is no processing of packets in the BUFFER_UP direction. The processing in this direction is instead carried out in the dedicated task *mac_task()* that among other jobs takes care of processing incoming packets. In the subsections below we discuss the processing of a packet in the BUFFER_UP and BUFFER_DOWN directions considering both, the MAC layer module from the stack and the dedicated task *mac_task()* running concurrently in the stack.

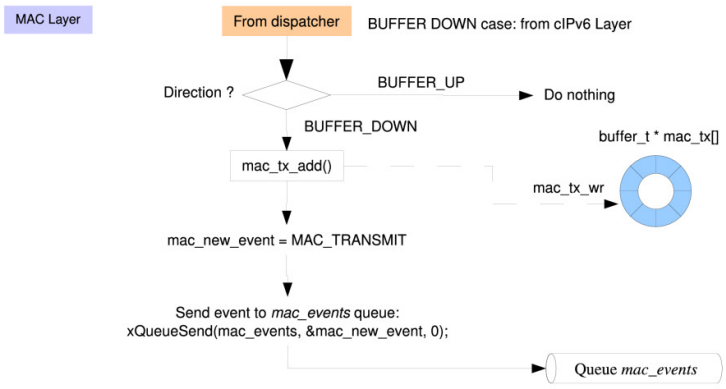

Figure 7: MAC layer module in the stack

4.1. Ring Buffers in the MAC Layer for Reception and Transmission of Packets

Two ring buffers, *mac_rx[]* and *mac_tx[]*, are used for buffering received packets or packets to be transmitted. The ring buffers store instances *buffer_t* and contain two indexes indicating the current writing and reading place in the buffer (Fig. 8).

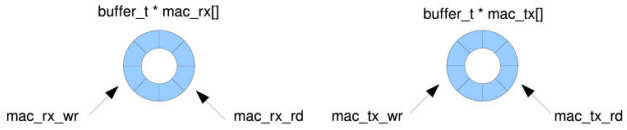

Figure 8: Ring buffers used in the MAC layer for reception and transmission of packets

4.2. Packets in BUFFER_UP Direction

The movement of packets in the BUFFER_UP direction is associated with the reception of packets. Fig. 9 depicts the details of a packet entering to the stack. When a packet arrives to the node, an interrupt service routine calls function *mac_push()* which adds the received packet to the *mac_rx[]* ring buffer and sets the packet direction *b.dir* to value BUFFER_UP; *b.dir* is used later by the dispatcher to move the packet in the BUFFER_UP direction. Then function *mac_rx_push()* is called, which in turn delivers an event type MAC_RECEIVE to a queue named *mac_events*.

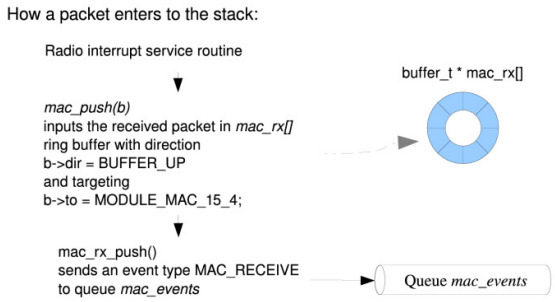

Figure 9: Details of packet entering to the stack

The dedicated task in the MAC layer, *mac_task()* (Fig. 10), contains an endless loop that in each iteration calls function *xQueueReceive()*. This blocking function checks the state of the other end of the queue *mac_events* and unblocks when there is an event available. When the event *mac_event_t* = MAC_RECEIVE is received, the first *switch()* in Fig. 10 branches the flow toward the switch-case MAC_RECEIVE. Then a call to function *mac_rx_pull()* retrieves the received packet from the ring buffer *mac_rx[]*, and the flow of execution continues until the second *switch()* on Fig. 10. In this *switch()* we assume that *mac_mode* evaluates to MAC_ADHOC and function *mac_adhoc()* is executed.

When *mac_adhoc()* is called (Fig. 11), the first *switch()* shown on the figure evaluates to case MAC_RECEIVE. Then the MAC header (IEEE 802.15.4 MPDU) is parsed by calling *mac_buffer_parse()* and the fields *b.dst_sa* and *b.src_sa*, among others, of the *buffer_t* instance *b* are filled with the received destination and source address. A posterior call to *mac_data_up()* pushes the *b* instance containing the packet to queue *events* for further processing by the next layer in the stack (cIPv6 layer).

4.3. Packets in BUFFER_DOWN Direction

The movement of packets in the BUFFER_DOWN direction is associated with the transmission of packets. In this case the dispatcher makes a function call to function *mac_handle()* (the dedicated function to handle packets in this layer) and subsequently the latter calls *mac_tx_add()* (Fig. 7). Function *mac_tx_add()* adds the instance *b* containing the packet in a ring buffer named *mac_tx[]* and next an event MAC_TRANSMIT is pushed to queue *mac_events*. Posterior processing in the system task *mac_task()* (Fig. 10) will process this event and finally deliver the packet to the radio for transmission. In the following paragraphs we explain how the packet is transmitted once the MAC_TRANSMIT event is received in *mac_task()*.

The dedicated task in the MAC layer, *mac_task()* (Fig. 10), is continuously looping and checking for events at the other end of the queue *mac_events* (function *xQueueReceive()*). When the event *mac_event_t* is received with value MAC_TRANSMIT, the first *switch()* in Fig. 10

branches the flow toward the switch-case MAC_TRANSMIT. Assuming that it is the first packet to be transmitted *mac_timer_event* takes value MAC_TIMER_NONE; then the packet contained in the *buffer_t* instance is retrieved from the *mac_tx[]* ring buffer by calling *mac_tx_pull()*. Assuming that this buffer exists, *mac_event_t* is set to MAC_NONE, the *buffer_t* instance is stored in the global pointer *mac_tx_on_air* and a timer is launched. The launch of the timer sets a global event variable *mac_timer_event* to value MAC_TIMER_CCA. Given the case under consideration, and with the given assumptions, the remaining entries in the logic are not executed and the task *mac_task()* returns to the blocking state in function *xQueueReceive()*.

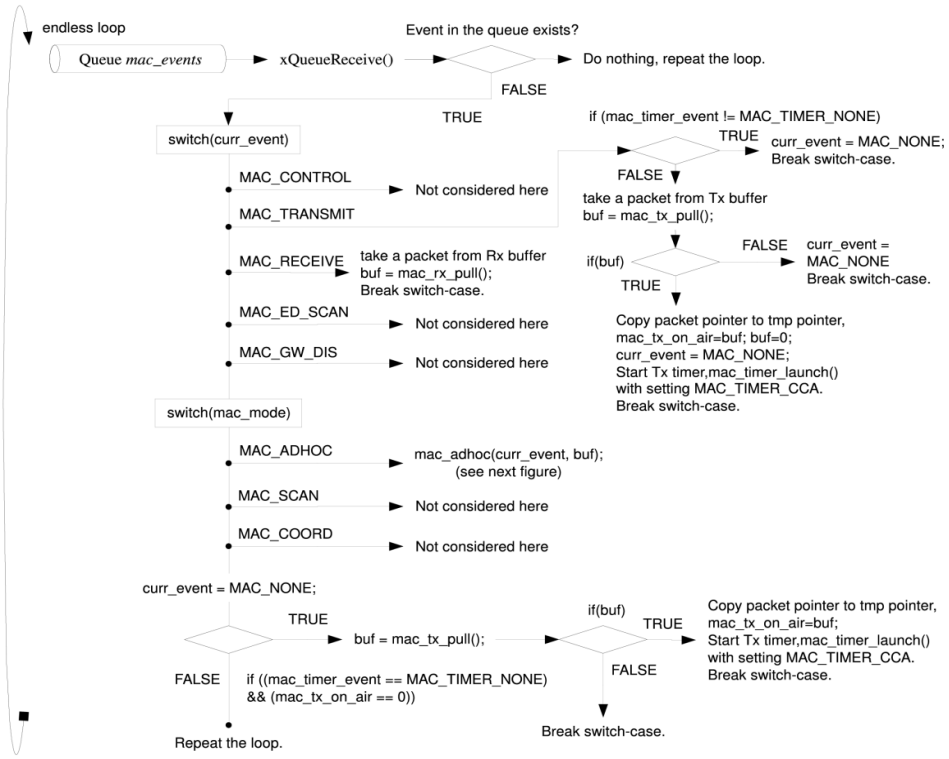

Figure 10: Details of function *mac_task()*

At this point we assume for simplicity that there is no simultaneous transmission or reception of packets. Then when the timer MAC_TIMER_CCA expires (after a random time in a defined interval), it triggers the transmission of the packet as follows. First a timer callback function in the code, *mac_timer_callback()*, is executed. Then function *mac_timer_callback()* (not shown on the figures) pushes to queue *mac_events* the event previously set in the global event variable *mac_timer_event*; i.e. the event value MAC_TIMER_CCA. When at the other end of the queue *mac_events* the event MAC_TIMER_CCA is received, the flow of execution continues until the second *switch()* on Fig. 10. In this *switch()* we assume that *mac_mode* evaluates to MAC_ADHOC and function *mac_adhoc()* is executed. When *mac_adhoc()* is called (Fig. 11), the first *switch()* evaluates to case MAC_TIMER_CCA. In this case the global pointer *mac_tx_on_air* is copied to a local pointer *buf* and the global pointer is set to zero. The flow of the code then executes the same code under the switch-case condition MAC_TRANSMIT. In the switch-case condition MAC_TRANSMIT, function *mac_buffer_out()* is called.

In function *mac_buffer_out()* (Fig. 12) we assume that the packet comes from the cIPv6 layer and *buf.to* is set to value MODULE_MAC_15_4. Then this causes the call to function

mac_header_generate() which creates and adds the MAC header in the packet. Then a local variable *ack* is set to value 1 if the MAC layer is configured with transmission acknowledgements, or otherwise to 0. Next the packet is sent to the radio and transmitted by calling function *rf_write()*, function that does not belong to the stack but to the platform port for radio chip TI CC2420 in file *rf.c*. If the transmission is successful *rf_write()* returns TRUE and function *mac_buffer_out()* returns with a value MAC_TX_OK_ACK if ACK transmission is enabled in the MAC or with value MAC_TX_OK if ACK transmission is disabled in the MAC.

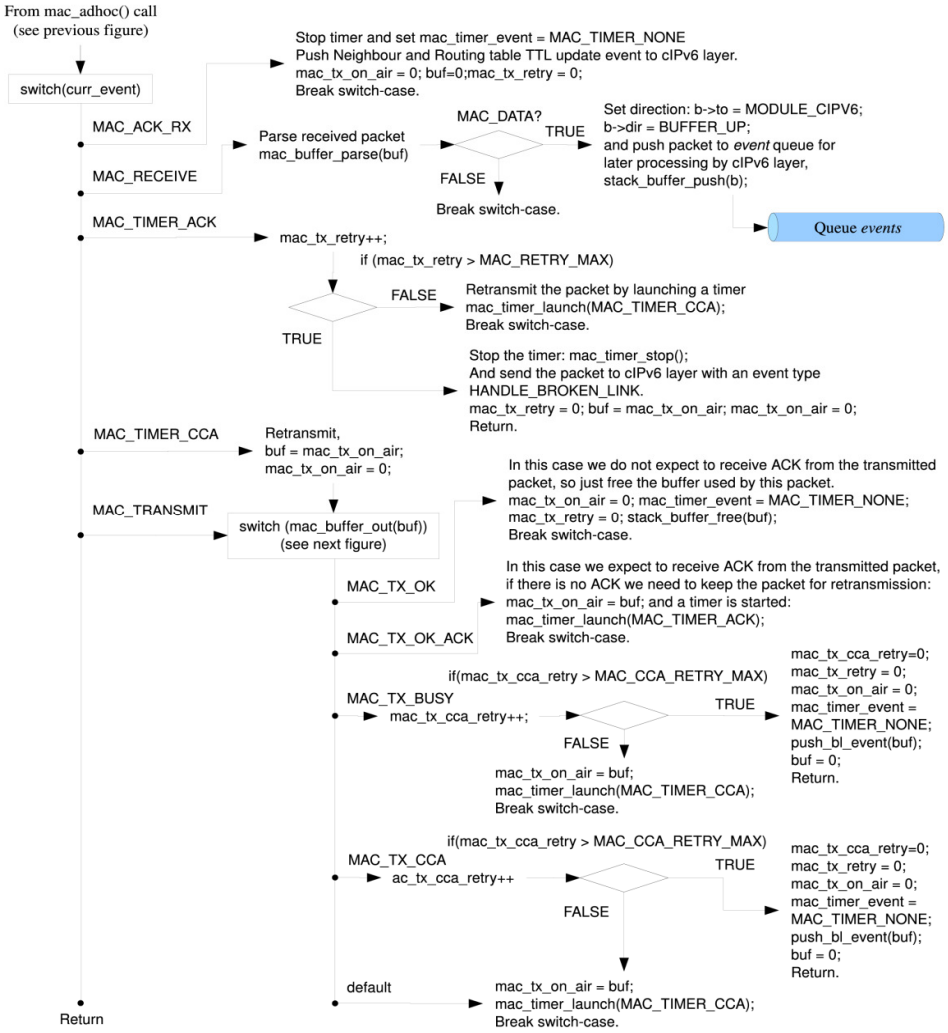

Figure 11: Details of function *mac_adhoc()*

After the packet is sent the flow of the code returns and executes the second *switch()* in Fig. 11, evaluating the return of the function call *mac_buffer_out()*. If ACK transmission is disabled the switch-case evaluates to MAC_TX_OK. In this case we do not expect to receive an ACK from the transmitted packet, so we just free the buffer used by this packet and set *mac_timer_event* to MAC_TIMER_NONE. If ACK transmission is enabled the switch-case evaluates to MAC_TX_OK_ACK. In this case we expect to receive an ACK from the transmitted packet, so we need to keep the packet for retransmission in case that a future retransmission is needed. A

packet for retransmission is kept in the global pointer *mac_tx_on_air* and a timer MAC_TIMER_ACK is launched. In both cases the switch-case breaks and the endless loop in function *mac_task()* blocks again in function *xQueueReceive()* waiting for the arrival of an event in queue *mac_events.*

If ACK transmission is enabled and the receiver properly emits a transmission ACK, it arrives to the node and function *ack_handle()* is called, which in turn delivers to queue *mac_events* an event type MAC_ACK_RX. When function *xQueueReceive()* receives the event, the flow of the code branches in the first *switch()* on Fig. 11 with case MAC_ACK_RX. In this case the ACK timer is stopped, a Neighbor and Routing table TTL update event is sent to the cIPv6 layer and the global pointer *mac_tx_on_air* is set to zero.

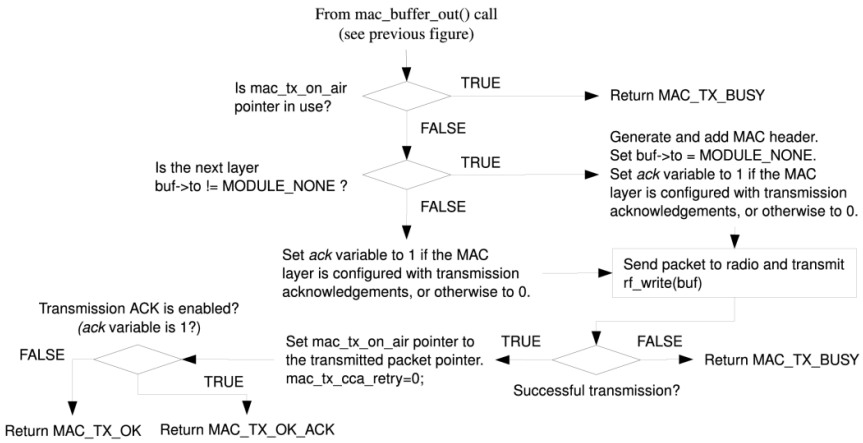

Figure 12: Details of function *mac_buffer_out()*

If ACK transmission is enabled and a transmission ACK never arrives, the timer MAC_TIMER_ACK expires and causes an event type MAC_TIMER_ACK that is pushed to queue *mac_events*. This event is processed in the first *switch()* on Fig. 11. In this case a counter *mac_tx_retry* is increased and a new attempt to transmission is made by starting a timer type MAC_TIMER_CCA, which in turns when expiring triggers a new transmission as explained above. If no ACK is received after the number of re-transmissions attempts is greater than MAC_RETRY_MAX, the timer is stopped and the packet is sent to the cIPv6 layer with an event type HANDLE_BROKEN_LINK.

5. CIPV6 LAYER

The processing of a packet in the cIPV6 layer when it is moving in the BUFFER_UP direction comprises the identification of the 6LoWPAN headers Mesh-Header and Dispatch-Header, and the alternatives of the latter, LOWPAN_HC1, LOWPAN_BC0 and IPv6, defined in RFC 4944 [1]. In the BUFFER_DOWN direction the processing comprises the incorporation of the 6LoWPAN LOWPAN_HC1 and LOWPAN_BC0 Dispatch-Headers.

This version of the stack (Nanostack 1.1) does not include fragmentation and it is not recognizing the Fragmentation-Header. Furthermore in our study we do not cover the processing of IPv6 and LOWPAN_BC0 headers even when these are present on the stack.

The upper part of Fig. 13 and 14 show the logic involved in cIPV6 layer. In Fig. 13 we can observe the entry point of a packet in this layer; packets are delivered by the dispatcher and can

arrive either from the MAC layer (incoming packets) or from the cUDP layer (outgoing packets).

In the subsequent subsections we describe the different conditional branches that form the logic of this layer.

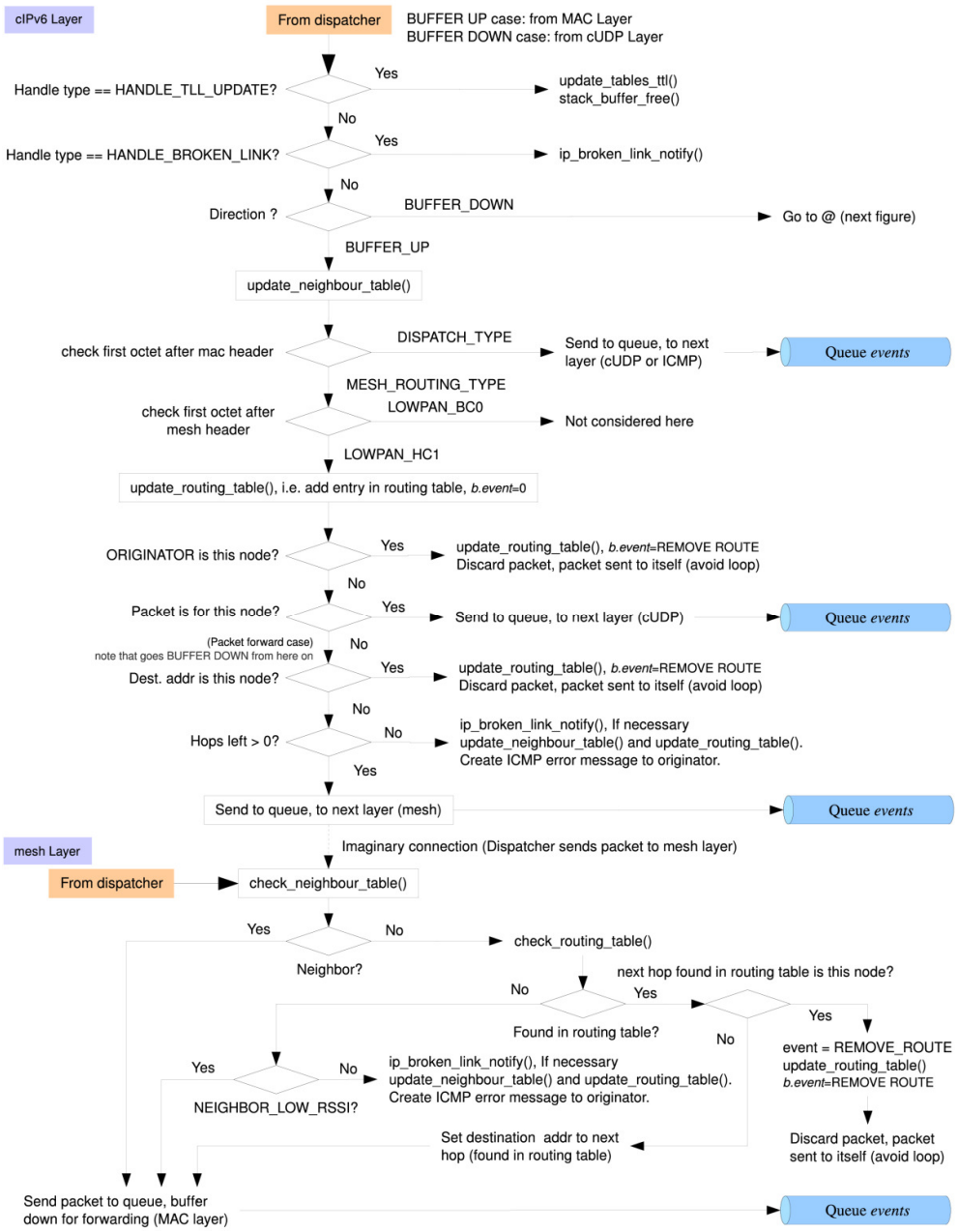

Figure 13: cIPv6 and mesh layers in the stack: BUFFER_UP case

5.1. Exception Messages

On the top of Fig. 13 the first two conditional branches verify particular exception messages set in *b.options*. The first verification is related to an option to refresh a Time-To-Live clock counter in entries of neighbor and routing tables (described in Section 6). The second verification is related to an exception labeled *broken-link* that occurs when the stack does not receive in time an acknowledgement about the reception of a delivered packet from the destination node, or when the transmission of a packet does not succeed after exceeding a maximum number of attempts due to the unavailability of the transmitter to perform the transmission. In the *broken-link* case the MAC layer adds a flag in the packet that is identified in this conditional branch and triggers an exception message. The exception message is delivered to the application layer for packets addressed to the current node through a dedicated queue to inform exceptions (queue *event_queue*), or otherwise delivered by means of an ICMP message to the destination address set in the packet.

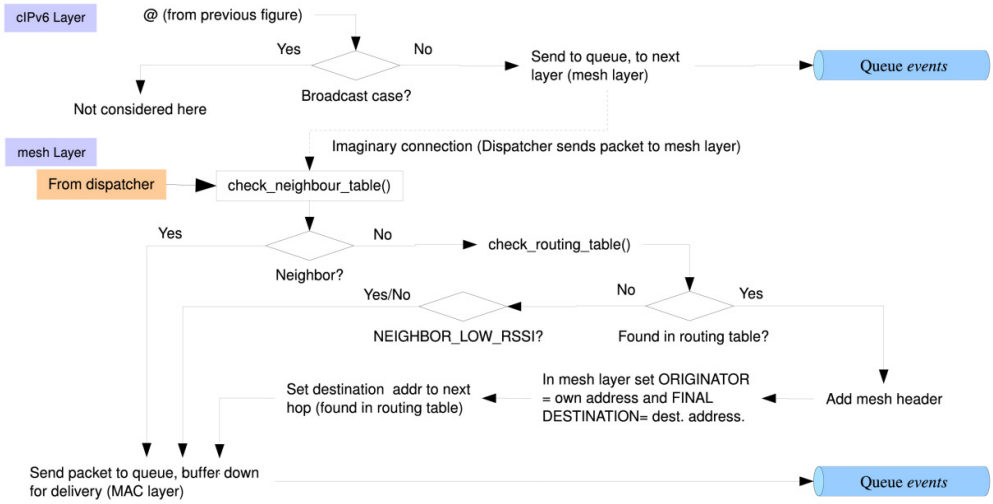

Figure 14: cIPv6 and mesh layers in the stack: BUFFER_DOWN case

Packets not involved with these conditional branches continue their journey toward the next conditional branch that verifies the direction of the packet. The logic for packets in the BUFFER_UP direction is discussed in Section 5.2 and depicted in Fig. 13. The logic for packets in the BUFFER_DOWN direction is discussed in Section 5.3 and depicted in Fig. 14.

5.2. Packets in BUFFER_UP Direction

The upper part of Fig. 13 depicts the logic followed by a packet moving in the BUFFER_UP direction in this layer. The first step is to update a table called *Neighbor Table* by invoking function *update_neighbour_table()*. Studying the logic stated by the code in the stack we concluded that the criteria to define what is a neighbor and what is not is based on the principle of *"any node that can be listened is a neighbor"*. In this sense an incoming packet from other node entitles the other node to become a neighbor of the receiving node. *Neighbor Table* and a related table, *Routing Table*, are described in the next section.

The next step is to check the 6LoWPAN encapsulation header. The conditional branch checks if the header is a Dispatch-Header or Mesh-Header. When a Dispatch-Header is present, the logic verifies the existence of the type-specific header to be LOWPAN_HC1 and then pushes the

packet to queue *events* for further processing in the next layers in the stack (cUDP or ICMP in this case). When a Mesh-Header is present (and assuming that routing is enabled in the stack by defining HAVE_ROUTING directive for the C preprocessor), the parsing process looks for the Dispatch-Header that follows the Mesh-Header. In the Dispatch-Header the type-specific header is checked with a conditional branch that verifies if the header is LOWPAN_HC1 or LOWPAN_BC0. The processing for the LOWPAN_BC0 case is not considered here. In the LOWPAN_HC1 case a table containing the routing of the packets, *Routing Table*, is updated by calling function *update_routing_table()*.

The next phase is to check the 6LoWPAN *originator address* (ORIGINATOR hereafter) present in the Mesh-Header and check if the ORIGINATOR is this node. If the ORIGINATOR is this node it means that the packet looped back to the original sender and therefore should be discarded. The conditional branch that follows checks if the packet is targeted for this node by comparing the *final address* present in the Mesh-Header to the own address of the node. If it is the case the packet is pushed to queue *events* for further processing in the next layer in the stack (cUDP).

A packet with a mesh *final address* other than the address of this node is a packet that reaches this node and requires forwarding. From now on the packet enters in a forwarding stage. In order to forward the packet to the proper destination its traveling direction in the stack will be now changed from BUFFER_UP to BUFFER_DOWN direction.

The first step in forwarding a packet is to check if the destination address (mesh *final address*) is the same that the own address of the node. If this is the case the packet is being sent to itself, so it is discarded. Next the number of hops left is checked. If the number of hops left is zero, a *broken-link* event is generated. Note that this *broken-link* event is a different event to the mentioned at the beginning of this section; in this case an ICMP message is delivered to the destination address set in the packet. If the number of hops is greater than zero the packet is pushed to queue *events* for further processing in the next layer in the stack, in this case the mesh layer.

As additional information we mention that at the beginning of the logic explained in this subsection, the logic checks the presence of an IPv6 header. This is not considered here and not depicted in the figure.

5.3. Packets in BUFFER_DOWN Direction

The upper part of Fig. 14 depicts the logic followed by a packet moving in the BUFFER_DOWN direction in this layer. In the figure we can observe a conditional branch that checks if the packet contains a broadcast address. The broadcast case is not considered in this document. Non-broadcast packets are pushed to the queue *events* for further processing in the next layer in the stack, in this case the mesh layer.

6. MESH LAYER

The processing of a packet in the mesh layer when it is moving in the BUFFER_UP direction comprises the execution of mesh-under routing relying on the information retrieved from the 6LoWPAN Mesh-Header in the cIPv6 layer. In the BUFFER_DOWN direction the processing comprises the incorporation of the 6LoWPAN Mesh-Header defined in RFC 4944 [1].

Routing information is stored in two main tables, *Neighbor Table* and *Routing Table*. *Neighbor Table* contains contact information related to nodes that were heard in the past, either by broadcast or unicast packets. *Routing Table* contains contact information for nodes that are not neighbor nodes. New entries in this table are added with callings to *update_routing_table()* setting a value *b.event* other than REMOVE_ROUTE and ROUTE_ERR in the instance *b*. In

the stack we identified two calls to *update_routing_table()* for adding new entries in the *Routing Table*, both mentioned in Sections 5.2 and 6.2. Here we point out that these function calls are inconsistent in the sense that the logic distributed in different points of the stack does not present a definite criterion to add initial entries in the table: The conditional statement mentioned in Section 5.2 requires the packet to have a 6LoWPAN Mesh-Header before an entry in the table can be processed, and the conditional statement mentioned in Section 6.2 requires an entry in the table before adding a 6LoWPAN Mesh-Header in the packet. Both conditions lock each other.

The lower part of Fig. 13 and 14 shows the logic involved in this layer when the packet moves in the BUFFER_UP and BUFFER_DOWN directions respectively. In the subsequent subsections we describe the different conditional branches that form the logic of this layer for the cases BUFFER_UP and BUFFER_DOWN.

6.1. Packets in BUFFER_UP Direction

The lower part of Fig. 13 depicts the logic followed by a packet moving in the BUFFER_UP direction in this layer. Actually the packet is no longer moving in the BUFFER_UP direction but hereafter in the BUFFER_DOWN direction due to the fact that the packet reaches this part in the stack from the packet forwarding case mentioned in Section 5.2. In the figure we can observe first a conditional branch that checks if the destination address of the packet to be sent matches a previously stored neighbor node (function *check_neighbour_table()*). If this is the case the packet is pushed to the queue *events* for further processing in the next layer in the stack (MAC layer). The remaining conditional branches are self explicative in the figure.

6.2. Packets in BUFFER_DOWN Direction

The lower part of Fig. 14 depicts the logic followed by a packet moving in the BUFFER_DOWN direction in this layer. In the figure we can observe first a conditional branch that checks if the destination address of the packet to be sent matches a previously stored neighbor node (function *check_neighbour_table()*). If this is the case the packet is pushed to the queue *events* for further processing in the next layer in the stack (MAC layer).

If the destination address of the packet being sent does not belong to a previously stored neighbor, the logic checks the routing table by calling *check_routing_table()*. If an entry in the *Routing Table* exists, the 6LoWPAN Mesh-Header is added to the packet with an *originator address* equal to the address of the node and a mesh *final address* equal to the intended address. The destination address in the MAC layer is set to the address found in the routing table.

If an entry in the *Routing Table* does not exist, the logic verifies whether there is a neighbor with low RSSI (Received Signal Strength Indication); state previously retrieved by *check_neighbour_table()*. Despite of the existence of a neighbor with low RSSI or not, the logic pushes the packet to the MAC layer. In other words, this means that if the node does not have the destination address registered in the neighbor or routing tables, the packet is transmitted anyway. In our opinion a more elaborated routing protocol should be developed and implemented, for example initiating a route discovery.

7. HIGHER LAYERS

In this section we describe the remaining layers that form part of the stack; cUDP layer and Application layer.

7.1. cUDP Layer

The processing of a packet in cUDP layer comprises the identification and incorporation of the 6LoWPAN header HC2, in this case for UDP HC_UDP, defined in RFC 4944. In the BUFFER_UP direction this layer checks the HC2 encoding (HC_UDP) and extracts the port numbers. Extracted port numbers are stored in the *b* instance in fields *b.src_sa.port* and *b.dst_sa.port*. In the BUFFER_DOWN direction this layer adds the HC_UDP header and compressed port numbers.

7.2. Application Layer

As mentioned in Section 3.2, in Fig. 5 the application layer is positioned outside the brace that expands under the dispatcher. The application layer is independent of the stack in the sense that it is not related to the dispatcher and instead receives/sends packets by calling socket functions. The application layer consists of an independent FreeRTOS task that implements the socket API to access the stack. Incoming packets are redirected to queue *socket* inside the socket structure, and are then retrieved at application layer by calling *socket_read()* function. Outgoing packets are generated at application layer by invoking function *socket_sendto()*, which sets *b.dir* = BUFFER_DOWN in the instance *b* that holds the packet, and finally are pushed to queue *events* by calling *stack_buffer_push()*.

8. NRP LAYER AND GATEWAY NODES

In this section we introduce an additional layer, the NRP layer. The NRP layer is enabled by using the optional module NRP. This layer was not considered in the previous sections due that it belongs to an optional module that is not essential for the main functionality of the stack. The NRP layer is generally used in the stack when a node acts as Gateway (GW) between the wireless sensor network (WSN) and the rest of the world outside the WSN.

In a typical GW configuration the NRP layer is incorporated to the stack and the application layer does not implement a socket interface. Under this setting we can depict the stack as it is shown in Fig. 15.

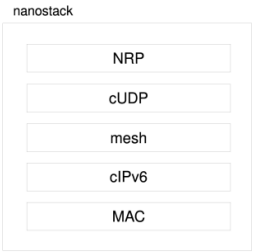

Figure 15: Nanostack Layers including NRP layer

The layer was named NRP due that it uses a Network Routing Protocol (NRP) [9] to communicate the GW node with the external world. In practice a GW node implements the NRP layer in the stack and connects to a workstation via a serial interface. With this setting incoming and outgoing packets to/from the WSN and circulating through the serial interface are encapsulated using the Network Routing Protocol (Fig. 16). The main idea is that a process running in the workstation attends the other end of the serial communication line and helps to transmit and receive packets to the WSN. In Fig. 16 we depict a possible representation of the arrangement of a node acting as gateway, the serial communication line and the workstation. Given this arrangement, the application layer of the stack implementing a socket interface can

be visualized as running in the workstation, thus allowing the implementation of the GW functionality.

The logic followed by the NRP layer is explained with the aid of Fig. 17. The figure depicts the NRP layer (upper part of the figure), and a dedicated task (lower part of the figure), *vnrp_task(),* executing concurrently with the stack. In the figure we can identify the two possible cases of incoming traffic:

1 Packets from the Radio to the Node.
2 Packets from the Serial Interface to the Node.

and the two cases of outgoing traffic:

3 Packets from the Node to the Serial Interface.
4 Packets from the Node to the Radio.

Incoming packets in items 1 and 2 are related to outgoing packets in items 3 and 4 respectively.

In the case of incoming packets from the Radio to the Node, the packet reaches the NRP layer through the dispatcher sending the packet in the BUFFER_UP direction (top of Fig. 17). Then the logic verifies whether there is a NRP subscription (see [9] for details), and then adds the packet into a ring buffer for transmission over the serial communication line. The transmission takes place in function *vnrp_task()* (bottom of Fig. 17).

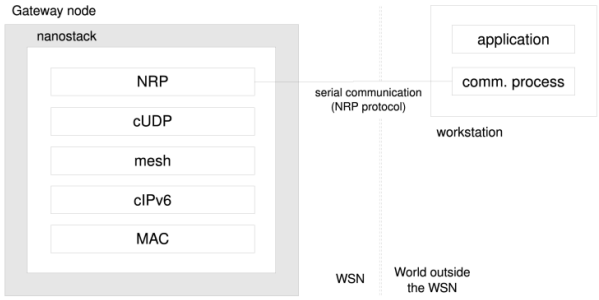

Figure 16: Node implementing Nanostack with NRP layer and serial communication to a workstation using NRP protocol

In the case of incoming packets from the Serial Interface to the Node, the function *vnrp_task()* retrieves one by one bits from the serial line when available, composes a packet and stores its instance, *buffer_t*, into a reception ring buffer. Packets coming from the serial line can contain actual data or control messages. Control messages are used for example to register NRP subscriptions [9]. If the received packet contains data and the next target layer is the IP layer, then the packet is pushed to queue *events* and finally handled by the dispatcher.

9. CONCLUSION

In the previous sections we documented the architecture of the stack and each one of its layers, except the ICMP layer, that was left aside due that the ICMP logic is distributed in several layers and its inclusion can compromise the simplicity and structure that we intend to adopt in the description of the stack. In this section we conclude the paper and provide a list of the outstanding features of the stack. Additional details are provided in [5].

Nanostack v1.1 is a stack designed with a modular architecture that operates moving pointers to instances of the data structure *buffer_t*. This implementation facilitates the movement of data through the stack without misusing memory size or reducing performance. In addition the stack operates in a concurrent fashion using multiple tasks, and over a real-time kernel system.

In our opinion some outstanding points of Nanostack are: 1) The architecture of the stack is modular. Layers can be added or removed. 2) It is suitable for researching mesh-under routing protocols, (in contrast of other stacks that adopt route-over routing [2]). 3) It is based on FreeRTOS, a real time kernel suitable for real time control processes and network synchronization [10]. 4) The programming at application level is extremely simple, facilitated by FreeRTOS and the socket API available in the stack.

To our knowledge Nanostack has been used in diverse academic and research fields, such as in network synchronization [10], real-time wireless control systems [11], structural health monitoring (SHM) [12], and WSN routing.

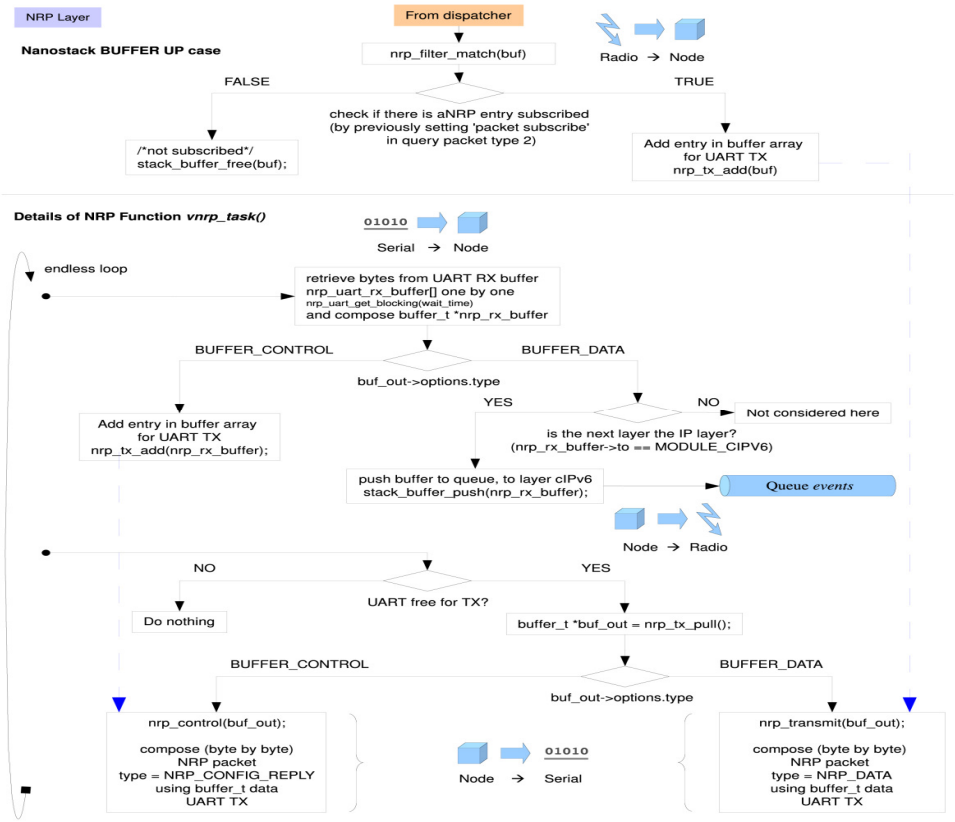

Figure 17: NRP Layer

REFERENCES

[1] Montenegro, G., Kushalnagar, N., Hui, J., Culler, D.: Transmission of IPv6 Packets over IEEE 802.15.4 Networks. RFC 4944 (Proposed Standard) (September 2007)

[2] Mazzer, Y., Tourancheau, B.: Comparisons of 6LoWPAN Implementations on Wireless Sensor Networks. In: Sensor Technologies and Applications, 2009. SENSORCOMM '09. Third International Conference on. (18-23 June 2009) 689 –692

[3] NanoStack 6lowpan: Nanostack v1.1 (2008) `http://sourceforge.net/projects/nanostack/`.

[4] FreeRTOS: The FreeRTOS project (2009) `http://www.freertos.org/`.

[5] Lembo, S., Kuusisto, J., Manner, J.: Internal Map of the Nanostack 6LoWPAN Stack. In Meghanathan, N., Boumerdassi, S., Chaki, N., Nagamalai, D., eds.: Recent Trends in Networks and Communications. Volume 90 of Communications in Computer and Information Science. Springer Berlin Heidelberg (2010) 619–633

[6] Rio, M.: A map of the networking code in linux kernel 2.4.20. Technical Report Data TAG-2004-1 (March 2004)

[7] Wehrle, K.: The Linux networking architecture: design and implementation of network protocols in the Linux kernel. Pearson Prentice Hall, Upper Saddle River, N.J : (cop. 2005.)

[8] IEEE Computer Society: IEEE standard 802.15.4-2006 (2006)

[9] Sensinode Ltd.: nRoute Protocol Specification, v0.7 (2006) `http://sourceforge.net/projects/nanostack/` (distributed with Nanostack v1.1 source code).

[10] Mahmood, A., Jäntti, R.: Time synchronization accuracy for real-time wireless sensor networks. In: Ninth Malaysia International Conference on Communications, 2009. MICC '09. (December 2009)

[11] Kaltiokallio, O., Eriksson, L., Bocca, M.: On the Performance of the PIDPLUS Controller in Wireless Control Systems. In: Proceedings of the 18th IEEE Mediterranean Conference on Control and Automation (MED'10), Marrakech, Morocco. (23-25 June 2010)

[12] Bocca, M., Cosar, E., Salminen, J., Eriksson, L.: A Reconfigurable Wireless Sensor Network for Structural Health Monitoring. In: Proceedings of the 4th International Conference on Structural Health Monitoring of Intelligent Infrastructure (SHMII-4 2009), Zurich, Switzerland. (22-24 July 2009)