FLAVIA
FLexible Architecture
for Virtualizable wireless future Internet Access

Grant Agreement: FP7 - 257263

Specific Targeted Research Project

FLAVIA

FLexible Architecture for Virtualizable wireless future Internet Access

Deliverable Report

D 4.2 – 802.11 modules specification

Deliverable title	802.11 modules specification
Version	1.0
Due date of deliverable (month)	M16
Actual submission date of the deliverable (dd/mm/yyyy)	21/11/2011
Start date of project (dd/mm/yyyy)	01/07/2010
Duration of the project	36 months
Work Package	WP4
Task	Task 4.2
Leader for this deliverable	IMDEA
Other contributing partners	CNIT, NEC, TID, MOBIMESH, BGU, IITP RAS, NUIM, AGH
Authors	Pablo Serrano, Vincenzo Mancuso, Pablo Salvador (IMD), Ilenia Tinnirello, Pierluigi Gallo, Pierpaolo Loreti, Claudio Pisa, Francesco Gringoli (CNIT), Antonio Capone, Stefano Paris, Alberto Pollastro, Domenico Schillaci (MOBI), David Malone, Paul Patras (NUIM), Eduard Goma, Yan Grunenberger (TID), Fang-Chun Kuo, Xavier Pérez Costa (NEC), Marek Natkaniec, Szymon Szott, Katarzyna Kosek-Szott, Krzysztof Loziak, Janusz Gozdecki (AGH)

Deliverable reviewer	Paul Patras (NUIM)
Deliverable abstract	The document describes the specification of the 802.11 basic modules according to the FLAVIA's architecture description carried out in D4.1.1.
Keywords	802.11, architecture, interface, module, prototype

Project co-funded by the European Commission within the Seventh Framework Programme		
DISSEMINATION LEVEL		
PU	Public	**X**
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for members of the consortium (including the Commission Services)	

PROPRIETARY RIGHTS STATEMENT

STATEMENT OF ORIGINALITY

This deliverable contains original unpublished work except where clearly indicated otherwise. Acknowledgement of previously published material and of the work of others has been made through appropriate citation, quotation or both.

TABLE OF CONTENT

LIST OF FIGURES

LIST OF TABLES

Executive summary

This report describes the specification of the FLAVIA-based 802.11 modules and provides preliminary prototypes of some of them. Following the work carried out in WP4 and described in D4.1.1 [6] we specify the operational modules planned to be implemented and the corresponding framework that will support them. These modules are chosen as they illustrate FLAVIA's principles: (i) **modularity**, in terms of defining different 802.11 MAC services; (ii) **flexibility**, in terms of dynamic configurability of the 802.11 MAC; (iii) **virtualization**, in terms of managing parallel independent 802.11 MACs accessing the same system resources.

In Section 1 we briefly review the FLAVIA architecture for an 802.11 MAC that is the basis for the modules and Wireless Processor specification. We also identify the set of services and functions to be added and concisely introduce the framework where the modules will be deployed. Then, Section 2 provides the specification of the Wireless Processor, which is the element of the architecture responsible for the direct interaction with the hardware modules. The section describes the MAC Engine that is an executor of Extended Finite State Machines (XFSMs) implemented at the firmware level, and where Wireless Processor is built. We show how we will develop the MAC Engine and introduce its programming interfaces, illustrating how to develop and build MAC programs with a set of examples. Section 3 describes the mac80211framework, the modifications performed to support modularization and provide flexibility in building up the enhanced mac80211 framework, named mac80211++. In addition, this section details the scheduling of a new FLAVIA service, explaining how new services and functions can be added and loaded. Next, we describe the virtualization support, which involves adding an overlay layer between the device drivers and the mac80211 framework. We also provide two specifications of the Information Base, which could be implemented by either programming an ad-hoc module or extending dynamically the structure *ieee80211_local* present in the mac80211 module. Section 4 presents the FLAVIA operation modules to be implemented within the aforementioned framework, describing the mapping and interaction of the modules within the FLAVIA architecture. Section 5 summarizes and concludes the deliverable.

Appendix A contains an update on the FLAVIA architecture for an 802.11 node. We introduce the information on the extended Monitoring service, a new Misbehaviour Detection and Reaction service, and the extended Consistency Manager module. Appendix B contains the pseudo-code for the Service Scheduler and Function Handler specified in Section 3, as well as the pseudo-code corresponding to the mac80211 framework and the Advanced Monitoring module.

1 Introduction

FLAVIA will provide a new MAC architecture where new mechanisms can be easily deployed, improving, e.g., the standardization time for new wireless technologies.

The scope of this deliverable is to describe the specification of the modules to be developed based on the FLAVIA 802.11 architecture. We also report a preliminary implementation stage that confirms FLAVIA's feasibility. Specifically, the flexibility and modularity principles are shown by adding new services that run in parallel as kernel modules, and enable the dynamic configuration of the MAC parameters. In addition, the virtualization support is achieved by implementing a new layer that is inserted between the mac80211 framework and the device driver. Before introducing the implementation, we briefly review the FLAVIA architecture for an 802.11 node and its functional modules.

The FLAVIA architecture builds upon the Wireless MAC Processor (WMP) which is the cornerstone of the FLAVIA architecture. This entity is responsible for the direct communication with the hardware modules. Therefore, the implementation of the WMP becomes one of the key contributions in this deliverable.

A common framework for the implantation of the envisioned services is provided by the mac80211 framework available in the Linux kernels. However, we need to extend and modify this to support FLAVIA's principles. For that reason, this document details the modifications performed to the mac80211 framework deriving an enhanced version, named mac80211++, which will allow the implementation of new services and functions in a flexible and modular way.

More specifically, we focus on a representative set of operation modules specified in the deliverable D4.1.1, namely: SuperSense, Data Transport, Power Saving, Monitoring and Rate Adaptation. We select these implementation examples since they are representative of the 3 fundamental FLAVIA principles, i.e., modularity, flexibility and virtualization. Through these examples we also demonstrate the compatibility between the legacy operation and the new MAC enhancements.

1.1 FLAVIA architecture overview

This section is devoted to briefly describing the architecture proposed by FLAVIA for an 802.11 MAC, following the general WP2 architecture specification [6][5]. Figure 1 depicts the architecture according to the WP2 vision, which comprises several modules, such as: the FLAVIA control, the Information Base, the Service Container, the Function Container and the Wireless Processor.

The FLAVIA architecture is designed according to three main aspects: modularity, flexibility and virtualization. These goals are accomplished by using a set of reusable functions on top of which different services can be built upon. In addition, new

interfaces will be provided such that new services can be easily added and finally, by means of the virtualization modules, wireless processor, services and functions may be instantiated and used in parallel.

We summarise below the main modules that compound FLAVIA architecture for an 802.11 as depicted in Figure 1:

- *Wireless processor*: key component of the FLAVIA architecture intended to handle hardware events and execute medium access programs designed as loadable Finite State Machines (FSMs).
- *Service container*: architecture element in charge of instantiating services, which are composed of functions. A service may implement basic or new MAC-layer functionalities.
- *Function container*: architecture element handling the set of running instances of functions. This container is managed by the FLAVIA control entity.
- *FLAVIA Control*: entity that manages the loading and changes of context of the different services and functions. It is worth noting the importance of the virtualization module as part of the FLAVIA control, which allows creating several virtual wireless processors running on top of a unique physical device.
- *Information Base*: architecture component responsible for managing different data/parameters modified by and shared among different services.

Note that each service is not isolated from the rest, but their interactions are piped through a set of interfaces that enable the exchange of metadata and signalling among different modules.

The modular composition permits to build a more robust and flexible architecture than the already existing MAC 802.11 architecture widely deployed nowadays.

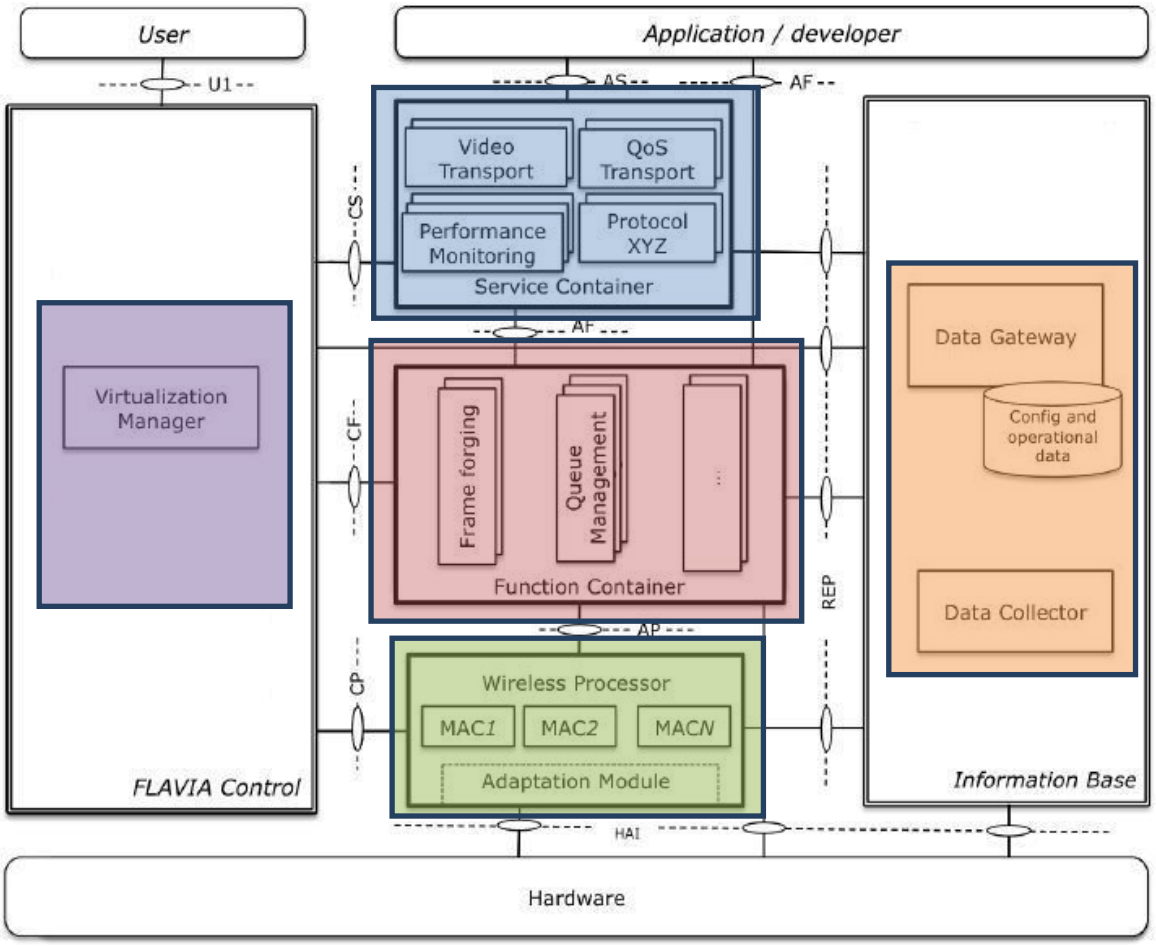

Figure 1: FLAVIA high-level view: 802.11 framework architecture.

1.2 Modules Specification

The FLAVIA deployment is strongly dependent on the framework where the set of modules will be prototyped. For that reason we specify two lines of work: i) Wireless MAC Processor and ii) a specific framework for other hardware platforms as the WMP approach is HW dependent.

The Wireless Processor is defined within the mac80211 framework but for the specific broadcom platform. In order to increase our flexibility we extend our research towards any kind of HW platform. To this end, we develop a new framework named *mac80211++*, starting from the original mac80211 framework.

mac80211 [9], included in the Linux kernel, is the common framework for most of the commonly used wireless drivers. It fulfils the requirements to match the vision in FLAVIA, as it makes possible to modify the operation of the wireless hardware without introducing changes in the actual hardware or drivers. Given the choice of mac80211

framework as the basis of our work, achieving deep knowledge of this module is essential. This task is carried out in Section 3.

We have already started implementing representative envisioned FLAVIA modules previously defined in [6], e.g.: SuperSense, Data transport with QoS capabilities and Power Saving.

2 Wireless MAC Processor

The architecture envisioned by FLAVIA requires hardware support. The Wireless MAC Processor (WMP) is the element of the architecture responsible for the direct interaction with the hardware modules, which provides the architecture with the abstraction to the lowest level of programmable resources. In this deliverable we focus on its specification and a preliminary implementation. This section describes the MAC Engine and the set of its programming interfaces.

We introduce also the Extended Finite State Machines (XFSMs) compiler developed which builds new MAC programs, changing MAC card behaviour on-the-fly without the need of recompiling. Furthermore, we present the specification of a debugger that might help us to monitor the behaviour of the MAC programs.

Finally, we develop a set of MAC programs intended to show the flexibility and modularity derived from the FLAVIA architecture.

2.1 Description

Our design starts from the concern that most modern wireless cards do embed a general-purpose CPU for supporting the hardware control logic. We propose to push this approach further, by transforming the card itself in a specialized processor, named Wireless MAC Processor (WMP). The WMP is devised to specifically handle hardware/PHY events and schedule actions on the hardware/PHY card resources, thus leaving the MAC protocol developer with the much simpler task of describing when and under which events and/or conditions such actions should occur. In other words, similarly to other processors specialized for handling digital signals (DSPs) or graphical images (GPUs), we introduce a processor specialized for handling MAC operations. The wireless MAC processor, whose internal architecture has been presented in D2.2.1 [5], has been conceived as a CPU specialized for handling hardware/PHY events and actions by executing XFSMs. We choose to abstract the definition of the medium access control logic in terms of state machines because they are very effective in modelling the behaviour of sequential control operations. Then, the WMP is built on top of the MAC Engine, which is an executor of Extended Finite State Machines implemented at the firmware level.

XFSMs are a generalization of the finite state machine model and permit to conveniently control the actions performed by the MAC protocol as a consequence of:

- The protocol logic.
- Events such as arrivals and timer expirations.
- Conditions on configuration registers (whose settings can be verified for enabling state transitions and updated when the transition is triggered).

Since the configuration memory is not explicitly represented in the state space, XFSMs

allow modelling complex protocols with relatively simple transitions and limited state space. A user-defined MAC program is thus specified by the set of states S, the triggering conditions F and the transition relations T. The number of states and relations is in principle arbitrary and depends on the device capability. Conversely, the set of events I, the sets of actions O and U, and the set of registers D over which conditions may be enforced is predefined by the Wireless MAC Processor and represent the WMP programming interface, detailed next. These sets represent the wireless device capabilities (e.g., switching to a different frequency band) that cannot be programmed by the user, but must be supported by the device hardware, and can only be invoked and controlled by the user-defined state machine.

Table 1 shows the mapping between the formal definition of an XFSM, in terms of its abstract 7-tuple (S, I, O, D, F, U, T) and the relevant meaning in terms of MAC primitives or parameters.

XFSM formal notation		**MAC Engine meaning**
S	Symbolic states	MAC protocol states
I	Input symbols	Triggering *events*, e.g., hardware signals, timer expiration generated by the interrupt block, etc.
O	Output symbols	MAC *actions*: commands acting on the hardware, performed by atomic functions either native in the device or implemented in the pre-loaded operations module (including arithmetic and logic operations, data creation and deletion, etc.)
D	n-dimensional linear space $D_1 \times \ldots \times D_n$	All possible settings of *n configuration registers*
F	Set of enabling functions $f_i: D \rightarrow \{0,1\}$	Set of *conditions* to be verified on the configuration registers for enabling the transitions
U	Set of update functions $u_i: D \rightarrow D$	*Configuration commands* devised to change the value of the configuration registers
T	Transition relation $T: SxFxI \rightarrow SxUxO$	Indicates the target state, the MAC commands and the configuration commands to be associated to each transition

Table 1: MAC Programs expressed as extensible finite state machines.

2.1.1 Application Programming Interfaces

In order to define an interface covering most of the MAC programmability requirements emerged so far for WLAN systems, we analysed some of the use cases in D2.1.1 [4]. The set of identified events, actions and conditions form a WMP programming interface able to support the examined use cases. Thus, its Application Programming Interfaces (APIs) are summarized in Figure 2, and described in the reminder of this section.

Events
CH UP
CH DOWN
RCV ACK
RCV DATA
RCV PLCP
RCV RTS
RCV CTS
RCV BEACON
HEADER END
COLLISION
MED DATA CONF
MED DATA START
MED DATA END
QUEUE OUT UP
QUEUE IN OVER
END TIMER

Actions
set/get(reg, value)
switch RX()
tx ACK()
tx beacon()
tx data()
tx RTS()
tx CTS()
switch TX()
set timer(value)
set bk()
freeze bk()
update retry()
more frag()
prepare header()
enable_ACK()
register()

Conditions
dstaddr
myaddr
queue length
queue type
cw
cwmin
cwmax
backoff
RTS thr
ACK on
srcaddr
frame type
fragment
channel
tx power

Figure 2: WMP Programming Interface.

Events: These are the set of signals either provided by the hardware interrupt block, or coming from the upper layers. The signals are generated by: i) the energy detection subsystem (CH_UP and CH_DOWN signals, i.e., start and end of channel busy intervals); ii) the receiver subsystem, (RCV_ACK, RCV_DATA, RCV_PLCP, RCV_RTS, RCV_CTS, RCV_BEACON, HEADER_END signals, i.e., the end of reception of different frame types or frame portions; MED_DATA_START and MED_DATA_END signals, which delimit the reception of a generic frame); iii) the frame control subsystem (COLLISION signal, i.e., checksum failure); iv) the transmitter sub-system (MED_DATA_CONFIRM signal, i.e., end of a frame transmission; v) the transmission and reception queues (QUEUE_OUT_UP and QUEUE_IN_OVER signals, respectively enqueuing of a new frame and overflow at the reception queue); vi) the clock (END_TIMER signal when a pre-set timer expires).

Actions: In addition to arithmetic, logic and control flow primitives, the operation block supports MAC-specialized operations, categorized into configuration commands and hardware commands. The configuration commands stores the information about the configuration of PHY and MAC parameters, which refer to: i) the energy detection mechanism: set/get(sensitivity), set/get(detection mode); ii) the transceiver: set/get(channel), set/get(power); iii) the head-of-line frame: update_retry(), more_frag(), prepare_header(); iv) the contention parameters: set/get(cwmin), set/get(cwmax), set/get(cw), set/get(RTS_thr). The hardware commands drive different card sub-systems: i) the transceiver subsystem: switch Rx(), tx_ACK(), tx_beacon(), tx_data(), tx_RTS(), tx_CTS(), switch_Tx(), enable_ACK(); ii) the timers: set_bk(), freeze_bk(), set_timer(value); iii) the upper layer interface: report().

Conditions: The WMP contains registers explicitly updated by WMP actions and/or

implicitly updated by WMP events, which store information on the card configuration and network state. These registers include: the station MAC address and the queue registers (queue_length and queue_type), the transceiver registers (channel and power), the contention registers (contention windows and backoff counter), the handshake registers, the frame registers (frame type, destination and source address, fragment), the medium state register. An example of register updated by a WMP action is the backoff counter register (whose value is set by invoking the set_bk() command), while an example of a register automatically updated by hardware events is the medium state register (which becomes busy when a CW_UP event occurs).

2.2 Implementation Platform

To prove the viability of Wireless MAC processors, we choose to implement its API over an ultra-cheap commodity WLAN network interface card. We select the Air-Force54G chipset from Broadcom, since one researcher of our team has contributed to developing the relevant open source firmware, OpenFWWF [7], and documentation on the internal card structure and its general purpose processor, registers, timers and transmission/reception primitives is available.

The implementation will be carried out by discarding the original card firmware and replacing it by an assembly code whose initial version is already available

The implementation will be carried out by discarding the original card firmware and replacing it by an assembly code executing a MAC Engine, whose initial version was presented in the first year review. In this version we implement the WMP and its state machine execution Engine. In this implementation the previously described WMP programming interface is mapped to actual signals, operations and registers of the card. For supporting the upper-MAC operations and interacting with the other protocol layers, we use the SoftMAC driver b43, which works as a wrapper between the Linux internal mac80211 software and the network card.

The AirForce54G chipset is built upon an 8 MHz processor with 64 registers supporting arithmetic, binary, logic and flow control operations. The other main blocks include:

- Tx and Rx Engine. These blocks correspond to the transmission and reception blocks of the WMP architecture. They (i) encode and decode packets from internal representation to the 802.11b/g CCK and OFDM encodings; (ii) compute and verify the Frame Check Sequence; and (iii) transmit and receive frames. Packet reception is performed by the Rx Engine in parallel to other processor tasks.
- Tx and Rx FIFO queues. These queues are interfaced to the host kernel. On the transmission path, packets forwarded from the driver are enqueued in the Tx queue, from which the chipset moves the frames to the Tx Engine. On the reception path, the processor waits for a packet received by the Rx Engine, and pushes (or drops) the received data towards the host kernel.

- Shared memory. This memory space of 4 KB can be accessed also by the host and can be used for implementing the micro-instruction memory, i.e., the MAC program.
- Internal code memory. This 32-KB memory is used for implementing the MAC commands and the MAC Engine.
- Template RAM. The RAM memory can be used for composing arbitrary frames (including customized frame replies) that can be pushed to the Tx Engine as if they came from the Tx queue.
- Internal registers and external conditions (EC). The internal registers keep hardware configuration settings. They may be set by the processor in response to changes in the EC to program the radio interface and set up timers.

Since the new firmware has to implement a MAC Engine, i.e., an executor of generic XFMSs, we also specify the new firmware work flow as depicted in Figure 3.

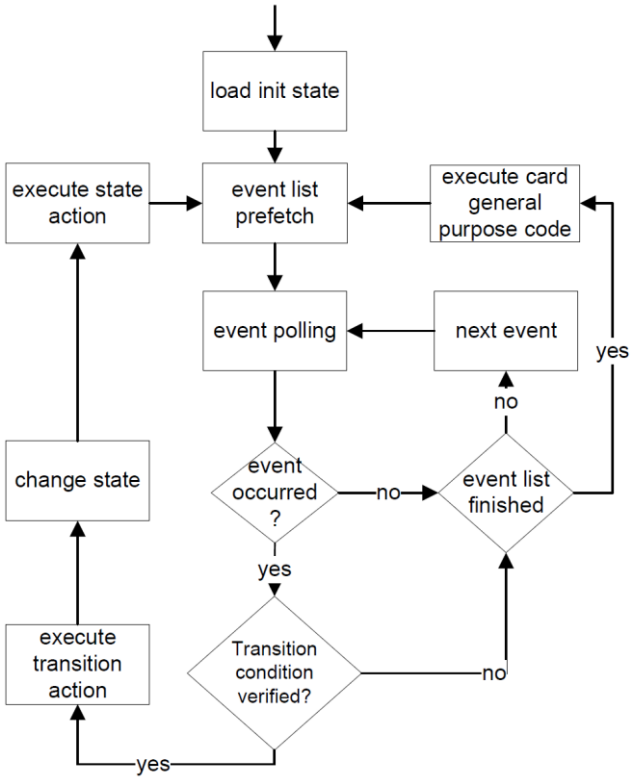

Figure 3: MAC Engine work-flow.

2.3 Compiler and Debugger

To permit the MAC Engine to execute an XFMS, the latter must be coded in a suitable

machine language, analogous to a bytecode, a sort of binary code. Let n_s be the number of symbolic states, and n_e the number of input events in I, the simplest approach is to code the XFSM as an ($n_s \cdot n_e$) table. At each location (I, j), the table stores the state transition when event j is received at state i. Each transition is defined by means of the 6-byte triplet (a, c, s), where:

- a $\in$ (O + U) is a 2-byte MAC transition action, where the first byte identifies the action label, and the second byte the action parameter (needed in case of configuration actions);
- c $\in$ F is a 2-byte condition enabling the transition (first byte = register name, second byte = register state);
- s is the target state, coded with 2 bytes.

Note that we do not limit to 1 byte per state as the number of actual states may become larger than the nominal ones: when multiple actions/conditions are associated to a same transition, as a consequence of the above coding, the state must be split into a sequence of intermediate states, each triggering at most one action and verifying at most one condition. In practice, to cope with the severe memory limitations of the chipset (only 4 KB are available for storing the MAC program table), we have optimized the memory occupancy by replacing each table's row with a list containing only the non-null state transitions. As each state generally reacts to a number of input events much lower than the total inputs number (i.e., the table is sparse), skipping null-transitions significantly reduces the required memory space. Moreover, as a second optimization, we have enabled the possibility to use the second byte of the state labels for encoding an additional state action (with no parameter) to be executed after the state transition.

To avoid writing MAC programs in the above described machine language, we have developed an XFSM builder. It includes a graphic XFSM editor on the eclipse platform [8] for composing MAC program. In addition it includes a bytecode compiler that translates an XFSM graphical representation into the machine language, understandable by the firmware's MAC Engine. The bytecode can be loaded on the memory chipset by using the chipset debug tools, or can be injected from the host to the card by forwarding special packets whose payload carries the MAC bytecode. Loading a new bytecode on the chipset allows changing on-the-fly the card behaviour without any recompiling operation.

For monitoring the behaviour of the MAC program executions, apart from measuring the throughput performance, we are also developing a debugger instrument based on the analysis of channel activity traces. Specifically, by sampling the channel activity by means of an USRP board and by processing this trace with software (whose preliminary implementation has been realised in MATLAB) devised to identify idle and busy intervals, we will double check the medium access operations programmed on the card.

2.4 MAC Programs

We are planning to develop a set of MAC programs conceived to respond to the 'lack of functionalities' emerged in D2.1.1 [2]. To this purpose, we will start with the re-implementation of the legacy DCF as an XFSM executed by the WMP. We will compare its performance with the benchmark provided by the native Broadcom's firmware, as well as with the performance provided by the OpenFWWF firmware (i.e., with the DCF as well reprogrammed on the card, but via straight firmware recoding).

The relevant XFSM is represented by black-lined states and transitions in Figure 4. The same figure shows, with different colours, the extra transitions and states modelling the extensions presented next. In addition to the self-explaining state labels, input events and transition arrows, the figure reports guard conditions – in square brackets - and actions (when associated to a transition) - in italic style. For graphical convenience, the figure separates the Tx state machine (left) from the Rx one (right).

At an initial state we will support at least three different MAC programs that tackle distinct MAC operation aspects, which indeed recur in several literature proposals: i) programmable management of frame replies (in red, the ACK in piggybacking program), ii) precise scheduling of the medium access times (in blue, the pseudo-TDMA program) and iii) fine-grained control of the radio channels (in green, the multi-channel program). In the following section we will focus on the ACK piggybacking program.

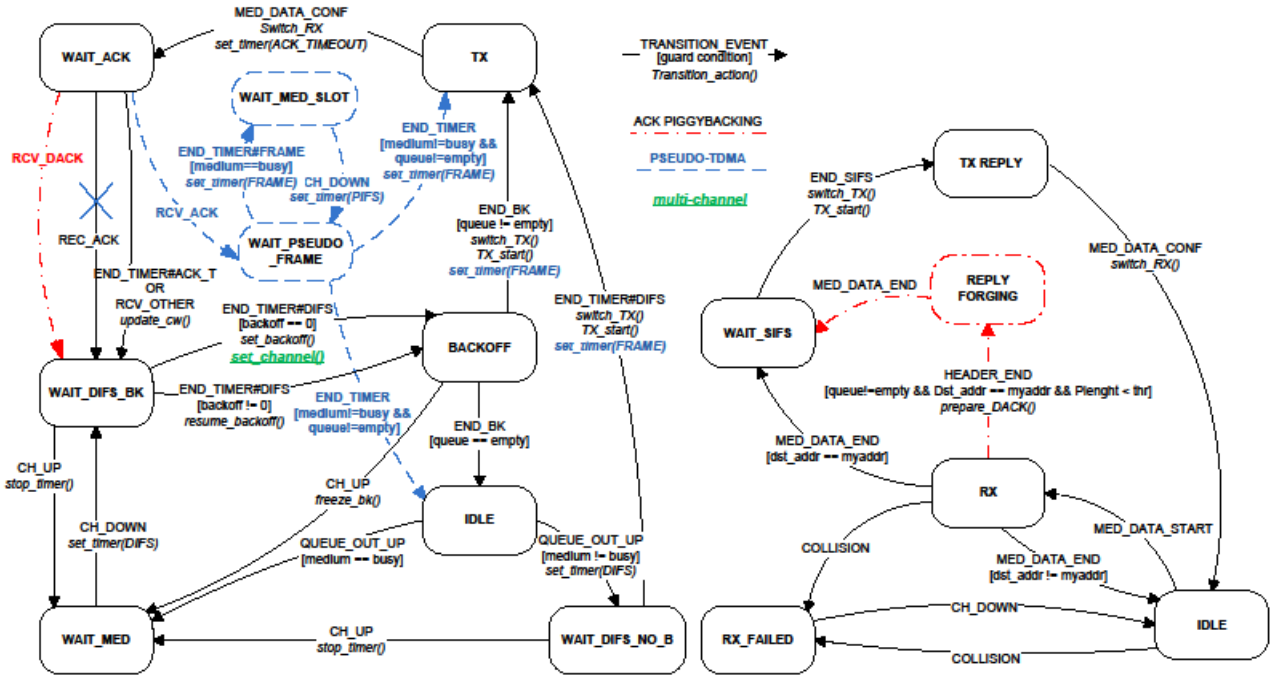

Figure 4: XFSM: Tx-state (left) and Rx-state (right) machines.

2.4.1 ACK Piggybacking

In this section we will describe a MAC program implemented as a DCF extension that optimizes the protocol efficiency, ACK Piggybacking. The key idea is to transmit data encapsulated in acknowledgement frames. Specifically, when a given station gets the access to the medium after a contention period and transmits its data frame, the destination station replies with a data+ACK frame if its transmission queue is non-empty. Note that the transmission of an ACK is always compulsory after the reception of a unicast data frame. Therefore, transmitting data piggybacked on the ACK allows saving air time and thus allocating a higher number of users in the network. Moreover, the number of collisions is reduced as the ACK transmission is protected by the SIFS interval.

Figure 4 shows how this mechanism can be easily defined in terms of an XFSM update. Modifications need to be performed both at the Tx and Rx state machines. At the reception side, starting from an ongoing reception, i.e. from the Rx state, when the header reception is completed (HEADER_END event), the station transits to the REPLY_FORGING state given that: [dst_addr = myaddr] and [queue! = empty]. In this state, the station continues the reception process and simultaneously prepares the data+ACK frame reply. If the transmission queue is empty but the frame is addressed to the target station, the receiver sub-system moves to the WAIT_SIFS state at the reception end (MED_DATA_END event). Otherwise, it moves to IDLE state. When the reply is ready (END_SIFS event), either in the case of a normal ACK or in the case of a data+ACK reply, the FSM switches to the transmission mode and it moves to the state TX_REPLY. When the transmission is completed (MED_DATA_CONF event), the system switches back to the receiver mode and to the IDLE state.

This mechanism can be applied to different types of traffic, such as TCP or voice traffic. For the first case in [17] we show the performance improvements of piggybacking as compared to the legacy DCF when TCP traffic is present.

ACK Piggybacking turns out to be especially appropriate in the case of short frame transmissions, e.g. the ones generated by voice codecs. In VoIP applications, the overhead introduced by the standard mechanism is too large, while collisions and the subsequent backoff procedure add random and unpredictable delays.

We develop a MAC program named *VoIPiggy* [18] that transmits the voice frames together with the acknowledgements, but only in the uplink direction. The VoIPiggy exchange depicted in Figure 5 involves a legacy data frame from the Access Point followed by an ACK frame, with voice data piggybacked, sent by the corresponding station after a SIFS.

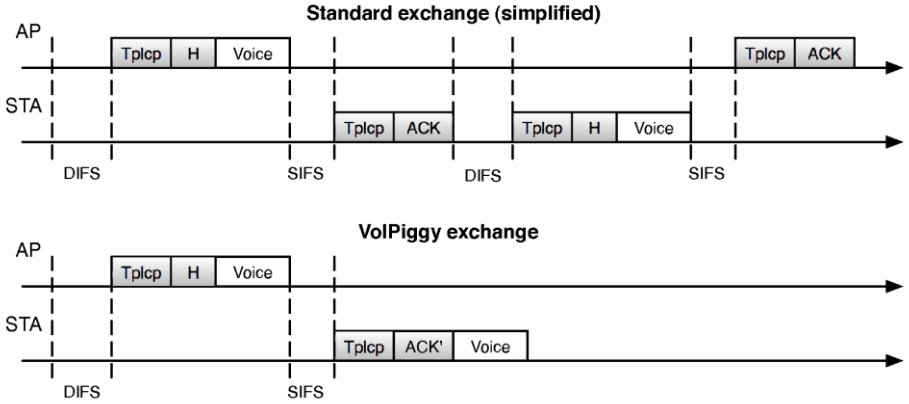

Figure 5: Simplified DCF frame exchange (top) and VoIPiggy proposal (bottom).

The legacy mode is used by the AP for any kind of packets it transmits. Meanwhile, VoIPiggy mode is used by a station whenever it receives a VoIP packet incoming from the AP and the Head-of-Line (HOL) packet in its Tx FIFO queue is VoIP-data. This is identified by checking the data stored in the structure *sk_buff*, which handles the network packets that are received or to be transmitted. In case the queue is empty or the HOL packet is not of VoIP-type, the station will use the legacy mode.

We add a hook in the receiver code to intercept ACK frames longer than the standard ones. The AP will transform them back into full featured voice packets. In addition, the AP is prevented from acknowledging the VoIP packets a station has sent piggybacked on an ACK.

For the transmission state, these changes are subject to implementation at the AP, which needs to wait for the ACK. The access point checks if the length of the received ACK is larger than the legacy one. In that case, the AP needs to send the voice frame up to the host. For the reception state, we modify the behaviour of the station. Upon the reception of a voice frame from the AP, the STA checks if it has pending voice traffic addressed to the AP and piggybacks the voice frames on the acknowledgements.

On the other hand, when a packet is transmitted using the legacy mode because no VoIP packet from the AP was received for more than 25ms, then the MAC Processor will wait for an ACK and the packet will not be removed from the queue, but instead will undergo legacy DCF access. This will be repeated until a maximum number of retransmission attempts (7 in our case) or until a VoIP packet is received by the AP.

Another issue identified when developing this MAC program is the necessity of delaying outgoing packets when the stations have voice traffic to be served. Without this delay a VoIP packet might be transmitted using the legacy access. If this happens once, then the probability of repeating in the near future is not negligible because of synchronization effects (the VoIP traffic is generated using long inter-packet periods, i.e., 20 ms). By delaying VoIP packet transmission for as long as 20 msec., the timeout selected and configured in the *VOIP_PIGGY_TIME_L* and

VOIP_PIGGY_TIME_H registers is very likely to expire before a VoIP packet is received from the AP, hence, the packet will use the VoIPiggy mode. Table 2: Percentage of the piggybacked frames vs. the station delay presents some numbers on the percentage of the piggybacked frames according to the delay introduced at the station.

Maximum Delay [ms]	% Piggybacked Frames	10th Percentile	90th Percentile
0	0,03	0	39
5	1,41	16,9	1202,3
10	65,38	1360,1	2326,5
15	89,36	2138	2935,7
20	99,10	2972	2974,6

Table 2: Percentage of the piggybacked frames vs. the station delay

3 Specification Framework: mac80211++

As already specified in Section 1.2, the mac80211 will be the basis for the development of the FLAVIA prototype. Therefore, we first present an overview of the mac80211 framework, highlighting its main characteristics. Then, we provide a proof of concept of the modularity of mac80211, developing as well a function and a service handler that manage the addition and loading of new services and functions, proving FLAVIA's modularity concept. Besides we describe the virtualization support, by adding an overlay layer between the device drivers and the mac80211 framework. Finally, we explain the development of the Information Base, which is achieved by extending one of the structures contained in the aforementioned framework.

3.1 Overview

The Linux mac80211 [9] layer specifies a framework to enable SoftMAC-capable device drivers used for operating with 802.11 hardware, and implements functionality such as handling several higher-layer components of the MAC, including support for hardware/software crypto, power saving, .11n style aggregation or LED management, while other parts of the MAC functionality are implemented at the hardware level. Figure 6 depicts an overview of the mac80211 framework.

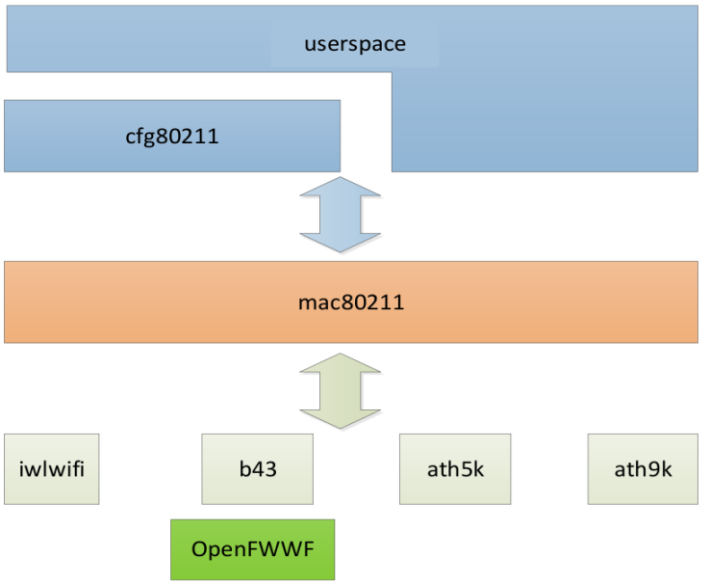

Figure 6: Overview of the mac80211 framework.

The mac80211 module plays two key roles:

- Wrap the packet incoming from the upper layers and translate them into the802.11 frame format.

- Control management operations related to the 802.11 standard.

The difference with respect to the Full-MAC approach is that mac80211 handles the management of MAC Sublayer Management Entity (MLME) within its framework. Still, this happens only for the station mode of operation. In case the node is acting as an AP, it is the *hostapd* module the one in charge of handling the MAC management.

The mac80211 module is composed of different data structures. We point the most relevant ones for the deployment of FLAVIA:

- *ieee80211_ops*: It collects the callbacks required for the communication from the mac80211 to the driver.
- *ieee80211_local:* Each instance of this structure represents a wireless device. It contains all the possible information of the wireless card, such as driver or interface information. It is created when the driver registers to the mac80211 module.

3.2 Modularity

In this section we present the modularity that can be achieved with mac80211 framework. Once identified the limitations of mac80211, we aim to improve the framework modularity by separating some of its components and rebuilding an enhanced version, named mac80211++.

3.2.1 Wireless stack interfaces

A standard wireless driver that uses the Linux wireless facilities includes some kernel modules and may provide some interfaces that can be used by user level tools to configure the device behaviour as depicted in Figure 7. The main modules defined in the framework are the mac8011 and the cfg80211; these modules are loaded and used by the drivers (e.g., ath5k, ath9k, b43) that are implemented in separate Linux kernel modules. Moreover, the framework allows for dynamic composition of the rate control that is linked as a separate component (some driver uses the separate modules provided in the framework such as b43, while other drivers register their operations to implement the rate control mechanism, such as ath9k). The mac80211 registers its callbacks to the *net_device* kernel interface building up the logical network interface.
Bidirectional interfaces are defined among modules as represented in Figure 7 by the arrows. The exported functions provide a direct interface shown with the green continuous arrows. The usage of an exported function introduces a dependency in the direction of the arrow (e.g., the driver depends on mac80211). The interface in the other direction is implemented through the registration of callbacks (i.e., function pointers). In Figure 7 this dependency is represented by the blue dashed arrows and the labels represent the structure containing the function pointers.

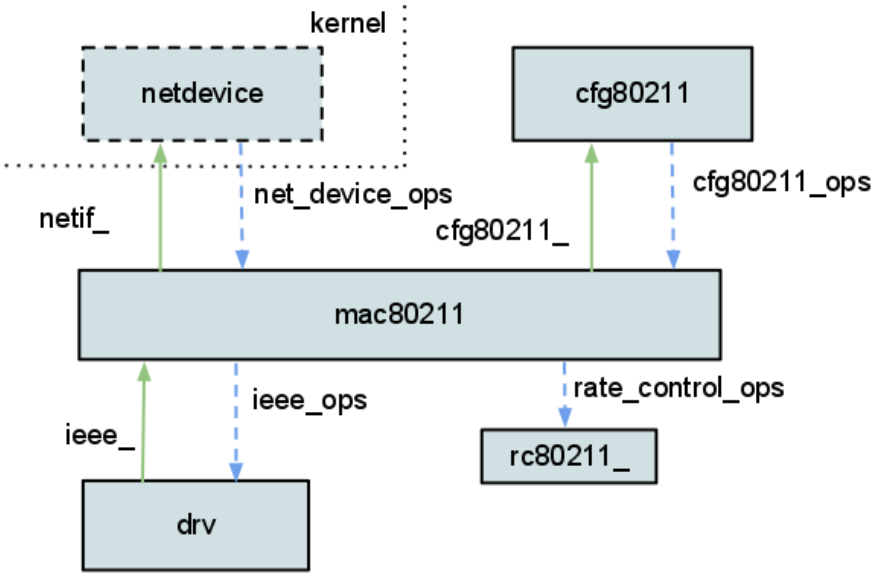

Figure 7: Architecture of a wireless driver.

The mac80211 module is contained in the folder net/mac80211 of the kernel source and exports the *ieee_* interface that is mainly used by the driver. In Table 12 (see Appendix B.1) we report the set of functions implemented in each file.

The mac80211 registers its callbacks to the *net_device,* creating the Linux network interface. The network device operation is defined in the file *include/linux/netdevice.h*, and the registered operations are presented in Table 13 of the appendix. We highlight some of the key operations: i) *ieee80211_subif_start_xmit* is used to transmit packets and ii) *ieee80211_netdev_select_queue* provides a packet classification for the Linux traffic control framework. In addition, Table 14 exposes the set of functions exported by the *net_device*.

From the user level, the wireless card can be configured and controlled by the cfg80211 module. This module acts on top of the mac80211 and consequently onto the driver by means of the *cfg80211_ops* interface, defined in the file *include/net/cfg80211.h*, in order to handle configuration requests on the wireless interfaces. The mac80211 module registers the callbacks of the structure *cfg80211_ops* summarised in Table 15.

The cfg80211 module provides mac80211 with an interface towards the user level, *cfg80211_*. Table 16 summarises the set of functions contained in the files located in the directory *net/wireless/*.

When loaded, the driver activates the mac80211 module and registers its operations through the *ieee80211_ops* interface. In what follows, we report an example of the functions registered by the b43 driver:


```
.tx                         = b43_op_tx,
.conf_tx                    = b43_op_conf_tx,
.add_interface              = b43_op_add_interface,
.remove_interface           = b43_op_remove_interface,
.config                     = b43_op_config,
.bss_info_changed           = b43_op_bss_info_changed,
.configure_filter           = b43_op_configure_filter,
.set_key                    = b43_op_set_key,
.update_tkip_key            = b43_op_update_tkip_key,
.get_stats                  = b43_op_get_stats,
.get_tsf                    = b43_op_get_tsf,
.set_tsf                    = b43_op_set_tsf,
.start                      = b43_op_start,
.stop                       = b43_op_stop,
.set_tim                    = b43_op_beacon_set_tim,
.sta_notify                 = b43_op_sta_notify,
.sw_scan_start              = b43_op_sw_scan_start_notifier,
.sw_scan_complete           = b43_op_sw_scan_complete_notifier,
.get_survey                 = b43_op_get_survey,
.rfkill_poll                = b43_rfkill_poll
```

The last interface depicted in Figure 7, named *rate_control_ops*, is related to the rate control algorithm. A module to provide the rate control functionality registers a set of operations defined in *net/mac80211/* and defined as follows for the case of the popular Minstrel algorithm:

```
.name = "minstrel",
.tx_status              = minstrel_tx_status,
.get_rate               = minstrel_get_rate,
.rate_init              = minstrel_rate_init,
.alloc                  = minstrel_alloc,
.free                   = minstrel_free,
.alloc_sta              = minstrel_alloc_sta,
.free_sta               = minstrel_free_sta,
.add_sta_debugfs        = minstrel_add_sta_debugfs,
.remove_sta_debugfs     = minstrel_remove_sta_debugfs
```

3.2.2 Modularization framework

The mac80211 framework is formed of a large number of sub-components, such as: the MAC layer management entity (*mlme*), the high throughput (*ht*) and the MPDU aggregation (*agg*) as specified in the IEEE 802.11n standard [2]. These parts are defined in dedicated files but not implemented as separated modules. To illustrate the various interfaces that can be defined within the mac80211 modules, we are splitting some parts of the monolithic mac80211 framework and evolving to a new extended

and more modular framework.

The source file of management entity, *mlme.c,* and the code for high throughput functionality, *ht.c*, have been split using the classic export/callback structure to implement the inter-module communication. The *mac8_mlme* and *mac8_ht* modules will be loaded by the mac80211 module.

Table 3 summarizes the new APIs added and declared as exported functions of the new mac80211++ framework to support the new *mac8_mlme* and *mac8_ht* modules:

Method	File
___ieee80211_stop_rx_ba_session	agg-rx.c
__ieee80211_stop_rx_ba_session	agg-rx.c
ieee80211_assign_tid_tx	agg-tx.c
___ieee80211_stop_tx_ba_session	agg-tx.c
ieee80211_tx_ba_session_handle_start	agg-tx.c
__ieee80211_stop_tx_ba_session	agg-tx.c
__ieee80211_request_smps	cfg.c
ieee80211_set_channel_type	chan.c
init_mac80211_ht_ops	ht_ops.c
ieee80211_ht_cap_ie_to_sta_ht_cap	ht_ops.c
ieee80211_sta_tear_down_BA_sessions	ht_ops.c
ieee80211_request_smps_work	ht_ops.c
ieee80211_recalc_idle	iface.c
ieee80211_led_assoc	led.c
ieee80211_hw_config	main.c
ieee80211_bss_info_change_notify	main.c
ieee80211_reset_erp_info	main.c
init_mac80211_ops	mlme_ops.c
ieee80211_rx_bss_put	scan.c
ieee80211_bss_info_update	scan.c
sta_info_get	sta_info.c
sta_info_alloc	sta_info.c
sta_info_insert	sta_info.c
sta_info_destroy_addr	sta_info.c
ieee80211_tx_skb	tx.c
ieee80211_stop_queues_by_reason	util.c
ieee80211_wake_queues_by_reason	util.c
ieee802_11_parse_elems	util.c
ieee802_11_parse_elems_crc	util.c
ieee80211_set_wmm_default	util.c

ieee80211_build_probe_req	util.c
ieee80211_send_probe_req	util.c
ieee80211_recalc_smps	util.c
free_work	work.c
ieee80211_add_work	work.c

Table 3: APIs for* mac8_mlme *and* mac8_ht *support

The callbacks corresponding to the new structure *mac8_mlme_ops* are defined as follows:

```
._ieee80211_dynamic_ps_disable_work    = ieee80211_dynamic_ps_disable_work,
._ieee80211_dynamic_ps_enable_work     = ieee80211_dynamic_ps_enable_work,
._ieee80211_dynamic_ps_timer           = ieee80211_dynamic_ps_timer,
._ieee80211_max_network_latency        = ieee80211_max_network_latency,
._ieee80211_mgd_assoc                  = ieee80211_mgd_assoc,
._ieee80211_mgd_auth                   = ieee80211_mgd_auth,
._ieee80211_mgd_deauth                 = ieee80211_mgd_deauth,
._ieee80211_mgd_disassoc               = ieee80211_mgd_disassoc,
._ieee80211_mlme_notify_scan_completed = ieee80211_mlme_notify_scan_complete,
._ieee80211_recalc_ps                  = ieee80211_recalc_ps,
._ieee80211_send_nullfunc              = ieee80211_send_nullfunc,
._ieee80211_send_pspoll                = ieee80211_send_pspoll,
._ieee80211_sta_quiesce                = ieee80211_sta_quiesce,
._ieee80211_sta_reset_beacon_monitor   = ieee80211_sta_reset_beacon_monitor,
._ieee80211_sta_reset_conn_monitor     = ieee80211_sta_reset_conn_monitor,
._ieee80211_sta_restart                = ieee80211_sta_restart,
._ieee80211_sta_rx_notify              = ieee80211_sta_rx_notify,
._ieee80211_sta_rx_queued_mgmt         = ieee80211_sta_rx_queued_mgmt,
._ieee80211_sta_setup_sdata            = ieee80211_sta_setup_sdata,
._ieee80211_sta_tx_notify              = ieee80211_sta_tx_notify,
._ieee80211_sta_work                   = ieee80211_sta_work,
```

The mac8_ht API is defined below:

```
._ieee80211_send_delba                 = ieee80211_send_delba,
._ieee80211_process_delba              = ieee80211_process_delba,
._ieee80211_ba_session_work            = ieee80211_ba_session_work,
._ieee80211_send_smps_action           = ieee80211_send_smps_action,
._ieee80211_sta_tear_down_BA_session   = ieee80211_sta_tear_down_BA_sessions,
._ieee80211_ht_cap_ie_to_sta_ht_cap    = ieee80211_ht_cap_ie_to_sta_ht_cap,
._ieee80211_request_smps_work          = ieee80211_request_smps_work,
```

Figure 8 illustrates the extensions and changes carried out in the mac80211 framework, turning it into a more modular framework following the FLAVIA specifications.

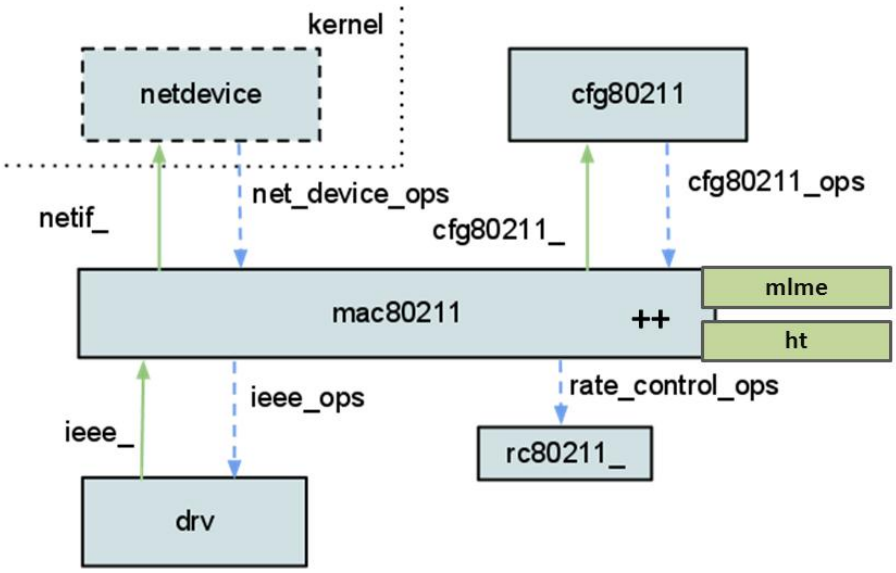

Figure 8: Architecture of a wireless stack under mac80211++.

3.3 Flexibility

In order to keep to a minimum level the number of modifications introduced to the mac80211 code and thus make the FLAVIA architecture more flexible, we implement two auxiliary modules, namely the *Service Scheduler* and the *Function Handler.* These two modules are liable, respectively, for managing the scheduling of a new FLAVIA service and the registration of the FLAVIA functions to be executed at the occurrence of specific events raised/handled by mac80211 (e.g., packet reception, packet transmission or channel switching).

In this section, we first describe the architectures of the Service Scheduler and the Function Handler. Then we illustrate a simple example showing how to use the proposed architecture to register a new service and enhance the behaviour of mac80211 at the occurrence of a specific event (e.g., the association of a STA with an AP).

3.3.1 Service Scheduler

The *Service Scheduler* has been designed and deployed to provide a simple and standardized mechanism to schedule FLAVIA services. Through this system, the

developer of a new service can focus only on the implementation of the main service functions, using the Service Scheduler as a mean to schedule periodically its execution. The Service Scheduler will run the functions registered by the FLAVIA service during its initialization phase.

In addition to simplifying the implementation of a service, the Service Scheduler architecture improves its maintenance, since the implementation of its internal functions can be improved to support enhanced services, as long as its APIs are not modified. Indeed, it can be easily updated with more sophisticated functionalities to meet the requirements of real-time systems.

- **Functions:**

The main function of the FLAVIA Service Scheduler is the *flavia_service_scheduler_init.* It initializes the service scheduler by creating a new single thread workqueue, which is used to manage the services as standalone tasks.

The cleanup function *flavia_service_scheduler_exit* unloads the service scheduler, deleting the auxiliary structures like the workqueue that are used to carry on the scheduling task.

The most important function of the Service Scheduler API is the *flavia_register_service*. This function provides a simple and standardized method for FLAVIA developers to register a function implementing a new FLAVIA service. The process implemented by the registering function comprises the following steps:

1. Initialization of the task implementing the FLAVIA service. This involves initializing a new *flavia_ss_t* object, which contains all the information necessary for executing a new service, and the task that will be registered on the workqueue.
2. Initialization of the new *flavia_ss_t* object on the list used for internal purposes by the service scheduler.
3. Initialization of the timer that will schedule the task, when the corresponding timeout expires.

The behaviour of *flavia_register_service_tsf_sync* function is quite similar to the previous procedure. However, unlike *flavia_register_service*, it registers a function that needs to stay synchronized with the TSF module of the wireless card.

Once a given service is no longer used, it can be removed using the *flavia_remove_service* function that stops the pending timers and, successively, deletes the corresponding work from the workqueue.

The two functions, *flavia_service_set_ieee80211_local* and *flavia_service_get_ieee80211_local,* set and return the internal pointer to the variable struct *ieee80211_local* used by mac80211. These functions provide a simple mechanism to get direct access to the internal information of mac80211.

The auxiliary function flavia_ss_timer_function is executed periodically when the timer

of the FLAVIA service registered by the developer expires. This function simply adds the work implementing *flavia_service_hook_container* on the worqueue used by the Service Scheduler to handle all FLAVIA services.

The *flavia_service_hook_container* invokes the function registered by the module implementing the FLAIVA service and reschedules the timer that, in turn (i.e. when the timer expires), will executes again *flavia_ss_timer_function*. Therefore, *flavia_service_hook_container* represents the main function of the kernel thread put on the workqueue and invoked periodically to execute any FLAVIA service. As illustrated in Figure 9, this auxiliary procedure is the main entry point for the function implementing a FLAVIA service.

The rescheduling timeout is defined by the member *flavia_usec* of the structure *flavia_ss_t*.

Unlike *flavia_service_hook_container*, the *flavia_service_hook_container_tsf_sync* procedure attempts to stay synchronized with the TSF module of the wireless card by rescheduling the execution of the function earlier than the service timeout (the member *flavia_usec* of *flavia_ss_t*).

The preliminary source code of the main data structures and functions of the FLAVIA Service Scheduler described above is detailed in Table 16 of Appendix B.2.

Figure 9 illustrates the data structures used by the Service Scheduler to execute periodically the FLAVIA services. Note that *flavia_ss_wq* is defined as single-threaded. Therefore, only one kernel thread is created to handle and schedule the tasks appended to the corresponding queue. The data structure *flavia_ss_t*, which is assigned to the *data* member of the *work_struct* data structure, contains the items necessary to define a FLAVIA service. These elements are: i) the service specific data (*flavia_data*), ii) the procedure that implements the service behaviour (*flavia_service_hook*), and iii) the time interval that elapses between the execution of two consecutive instances of the service function (*flavia_usec*).

On the contrary, the function *flavia_srv_container* (or the function *flavia_srv_container_tsf_sync* for services that need to stay synchronized with the TSF module) is assigned to the *func member of the *work_struct* data structure, which represents the pointer to the procedure actually executed by the kernel thread. Consequently, when the timer defined by the parameter *flavia_timer* expires, the function invoked by the kernel thread is *flavia_srv_container*. This last function, in turn, executes the service function pointed by *flavia_service_hook* and reschedules the timer to execute again the *flavia_srv_container* later.

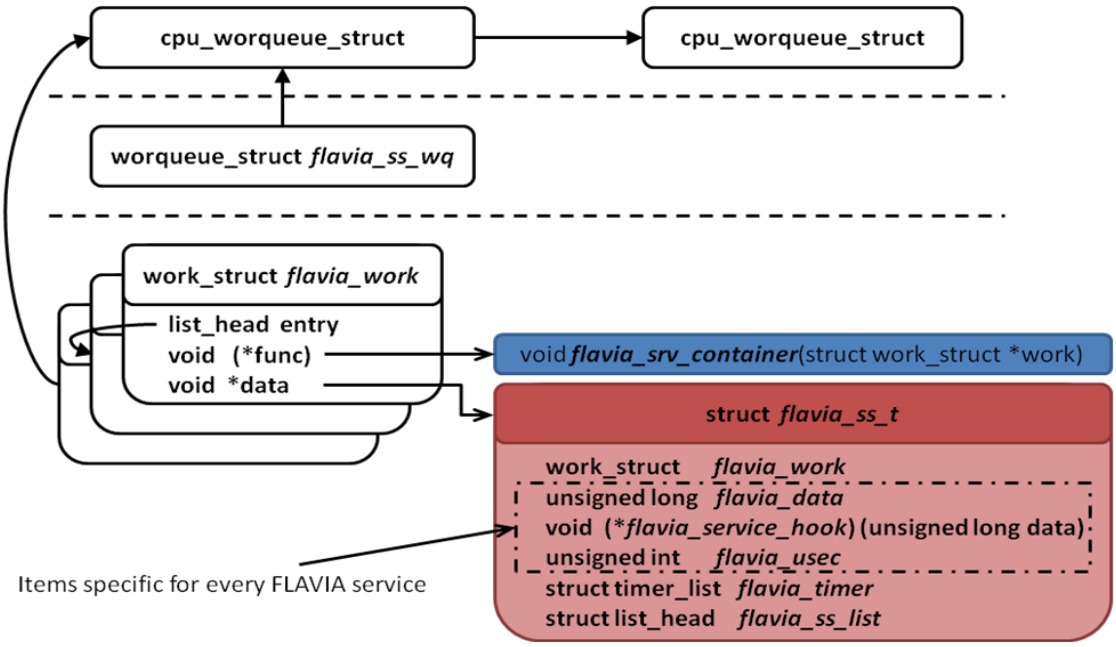

Figure 9: Service Scheduler workqueue structures.

Figure 10 illustrates the main steps executed to register and execute a FLAVIA service. When a new service is registered, a new work_struct object is initialized with the members illustrated in Figure 9 and a new timer is triggered. When the timer expires, the work_struct object is queued on the workqueue defined by the Service Scheduler containing all the tasks that must be executed immediately. Once the task can be scheduled, the kernel thread implementing the workqueue invokes the function *flavia_srv_container* that, in turn, executes the service function pointed by *flavia_service_hook* as described above. Note that the red-coloured phases in Figure 10 are not directly implemented in the Service Scheduler, as they are provided by the Timer and Workqueues of the Linux kernel API.

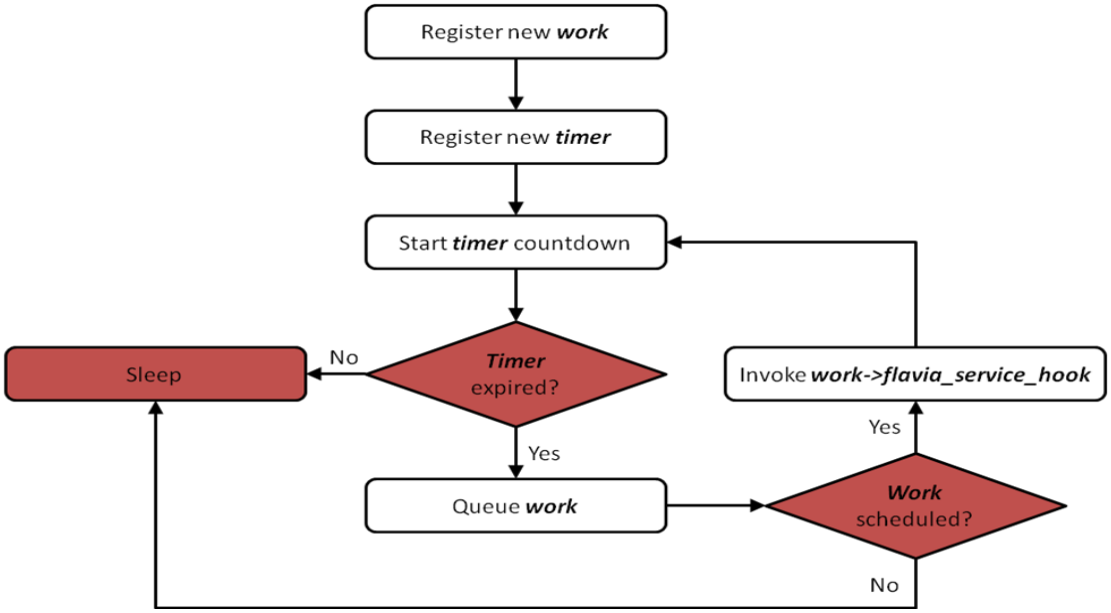

Figure 10: Service Scheduler work-flow.

3.3.2 Function Handler

The *Function Handler* is designed and deployed to provide a standardized mechanism to *hook* the mac80211++ code (i.e., to add piece of code that acts as glue between any FLAVIA function and the mac80211 procedures). More specifically, the Function Handler permits to register a function to any hook added to the mac80211 code; thus improving its functionalities with new FLAVIA functions. At the occurrence of a specific event, the Function Handler will call all the FLAVIA functions previously registered on that hook. For example, when a new frame is received, the control flow of the mac80211 code reaches a FLAVIA hook that transfers the control to the Function Handler, which, in turn, invokes the execution of all functions registered on that hook. Note that the function invoked by the Function Handler can register a FLAVIA service or create a new task executed by an independent kernel thread. Therefore, the Function Handler mechanism provides a high level of flexibility to the developers of new functionalities and services.

The FLAVIA Function Handler defines two main data structures: the *flavia_function_ops*, a structure used to define the list of functions that are called when the corresponding hook is executed in mac80211, and the *flavia_hook_ops*, a structure used to define the set of mac80211 hooks.

- **Functions**

The main function of the FLAVIA Function Handler is the *flavia_function_handler_init*, which performs some consistency checks and initializes the internal data structures used to fulfil the management task. On the contrary, the function *flavia_function_handler_exit* removes the functions registered on all hooks and

deletes the structures to free their memory space.

Every function designed to enhance the functionalities of the MAC protocol through the FLAVIA architecture can be registered on a particular hook using the *flavia_register_function*, which connects the function to a hook invoked from mac80211 when the control flow reaches such hook. The procedure implemented by the registering function consists of two main phases that are detailed next:

1. Initialize the function that will be executed every time the control flow of mac80211 reaches the corresponding hook.
2. Assign the function to the hook passed as argument. More specifically, the registration may proceed as follows:
   - If at least a function has been already registered on the hook, the Function Handler registers the new function on the corresponding hook (i.e., the list of functions), and then it adds the new item to the list of functions executed on the same event/hook;
   - Otherwise, the Function Handler creates a new hook object (i.e., a new list of functions), and registers the function on the new hook.

The *flavia_remove_function* deletes a function registered on a hook. It must be used when a module implementing a FLAVIA service is removed.

In addition to implementing the structures and the procedures necessary to handle a FLAVIA function, the Function Handler provides a standard method to define hooks in the mac80211 code, so that any developer can easily enhance mac80211 with new functionalities. To this end, the Function Handler API provides the *flavia_function_hook_container* function, which can be invoked everywhere inside the mac80211 framework to create new hooks.

The execution of all functions associated with a specific hook is performed by *flavia_exe_hook_functions*.

Finally, the two auxiliary functions *flavia_find_hook_ops* and *flavia_find_function_ops* search the list of FLAVIA functions or a single FLAVIA function registered on a given hook, respectively.

The preliminary source code of the main data structures and functions of the FLAVIA Function Handler described above is detailed in Table 17 Section B.2.

Figure 11 illustrates the data structure used to handle all the functions registered on the corresponding hooks defined in the mac80211 kernel module to enhance its functionality. More specifically, for each item of the dynamic list of hooks, the Function Handler maintains a double linked list of functions registered by a developer through the function *flavia_register_function.*

Note that the binding between a hook and the corresponding list of functions is performed at runtime through the functions *flavia_register_function* and *flavia_function_hook_container* made available by the API of the Function Handler. In

particular, *flavia_register_function* adds a new function *f_nm* to the list assigned to the corresponding hook *hook_n,* on which *f_nm* is being registered. Additionally *flavia_function_hook_container* invokes all functions defined on a specific hook, when these are triggered during the execution of mac80211.

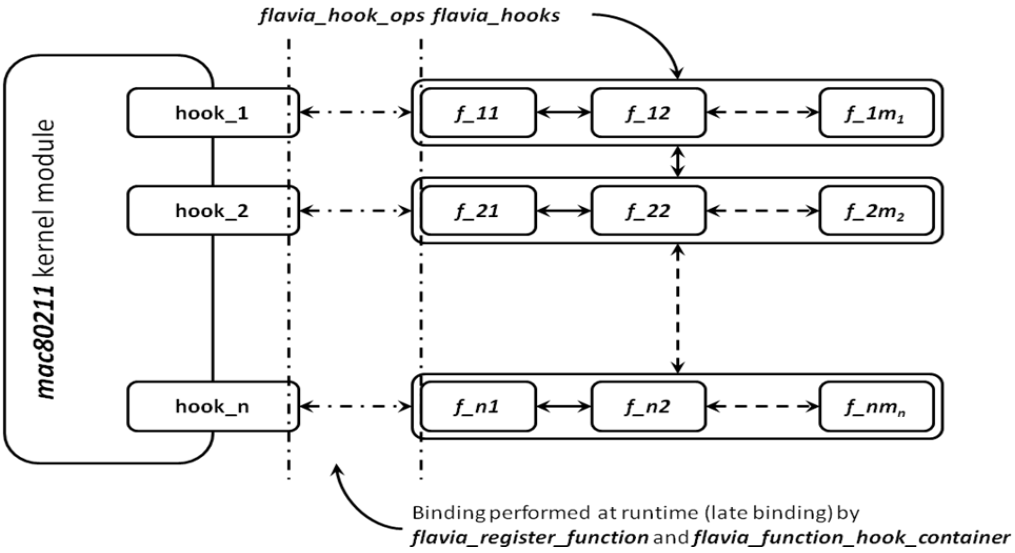

Figure 11: Architecture of the Function Handler.

Figure 12 shows the sequence of steps executed by the *flavia_function_hook_container* function declared in the Function Handler, when the control flow of mac80211 reaches a specific hook (i.e., the hook is triggered).

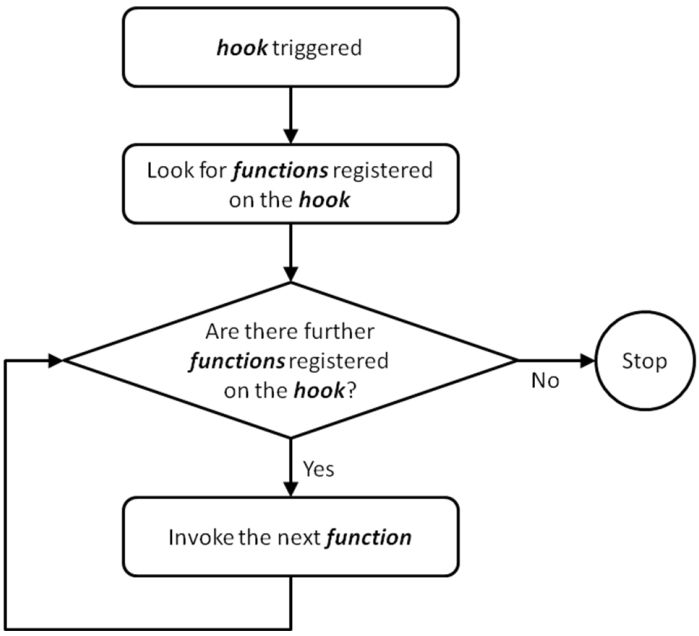

Figure 12: Function Handler work-flow.

3.3.3 A simple example: the FLAVIA hello service

The *flavia_hello_service* is a simple service designed to illustrate the registration mechanism of new services and functions, which occurs through the Service Scheduler and the Function Handler, respectively.

In this simple example, two functions, namely *flavia_hello_service* and *flavia_hello_function*, are defined. Both routines simply print a message in the file used for kernel debug purposes. In particular, the function implementing the Hello Service displays periodically a brief greeting message. Whereas an auxiliary routine assigned to the hook invoked prints the association status and other information of the 802.11 station when an 802.11 station gets associated to an AP.

As any Linux kernel module, the two main functions implemented in this example correspond to the initialization and clean-up macros, defined in *linux/init.h* (*flavia_hello_init* and *flavia_hello_exit*).

The *flavia_hello_init* function, as illustrated in Figure 13, holds two structures named *flavia_ss_item* and *flavia_func_item*, which are used as containers to handle the selected service and function, respectively,. The *flavia_hello_init* function calls the *flavia_register_function*, defined in the *flavia_function_handler.c* file, in order to set the *flavia_hello_function*. Similarly, the *flavia_register_service*, defined and exported by the *flavia_service_scheduler.c* file, is invoked to register the *flavia_hello_service*, which actually implements the service. If any problem occurs during the registration process, an error value is returned to the procedure that invoked the module. Otherwise, the Hello Service is loaded correctly.

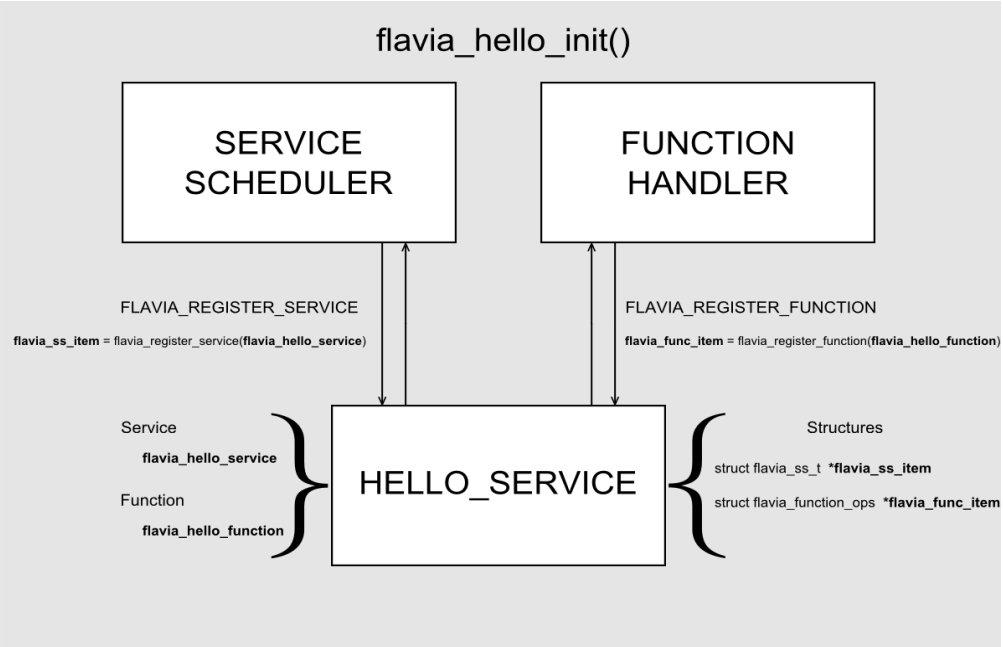

Figure 13: Function flavia_hello_init() in flavia_hello_service.

Figure 14 depicts the *flavia_hello_exit* function that unloads the service, removing both function and service through, the functions *flavia_remove_function* and *flavia_remove_service,* respectively.

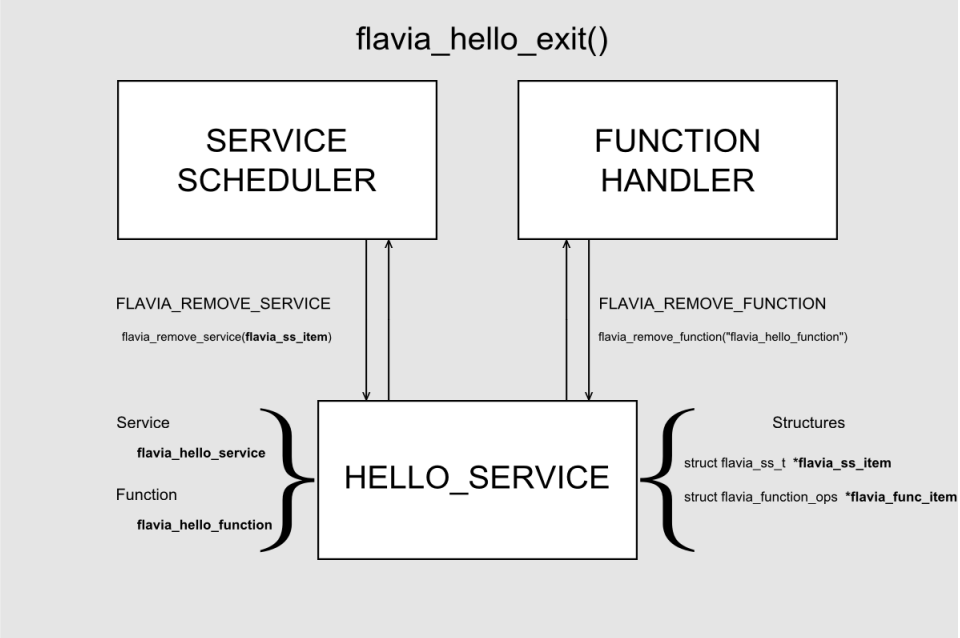

Figure 14: Function flavia_hello_exit() in flavia_hello_service.

3.4 Virtualization

In order to provide true virtualization capabilities to mac80211, FLAVIA is exploiting the current architecture of mac80211 as depicted in Figure 6.

The mac80211 framework is already providing a soft-virtualization mechanism as it can support multiple logical network interfaces on top of the physical one. This mechanism is present in mac80211 as a way to provide multiple BSSID support, as well as simultaneous AP, client and monitoring operations. Therefore, mac80211 and its associated user-land tools such as *iw* are designed to add, delete and change the logical interfaces in a transparent way, by using the *netlink* interface to dynamically change different elements in the kernel.

Unfortunately, these mechanisms are limited by the hardware capabilities and the corresponding mac80211 driver the framework is relying on. Specifically, in the current architecture each logical interface is bound to a physical hardware interface and this interface is bound to a set of physical parameters such as frequency of operation. Consequently, true virtualization in the FLAVIA flavour that aims at exploiting the full capability of a card, e.g., switching on and off a specific channel, is

impossible with the existing mac80211 framework.

To overcome this limitation, FLAVIA is introducing a new element in the mac80211 stack, *FLAVIAn*, which is essentially a kernel module that acts as a mac80211 driver. *FLAVIAn* is used as an overlay layer between the hardware card and the mac80211 framework, allowing to present a single WLAN interface through a set of multiple mac80211 device drivers and thus to enable virtualization. In our case, we will work with the new mac80211++ stack.

The mac80211 family of drivers is using a certain number of hooks to maximize the reuse of element inside the system. The current requirements for interfacing a driver with a Linux kernel and the mac80211 subsystem are the following:

- A kernel driver should provide a sample macro to be recognized by the kernel. Example of these are the MODULE_* family of macros.
- A networking device driver should provide net device hooks as objects of type *net_device_ops*.
- A mac80211 driver should provide mac80211 hooks as mentioned before, exposed as objects of type *ieee80211_ops*.
- A driver could implement a set of supplementary user-space/kernel interfaces by implementing a Netlink hook using the type *genl_ops* to describe the messages to be exchanged.

Figure 15 summarizes the different interactions between drivers and the mac80211 framework.

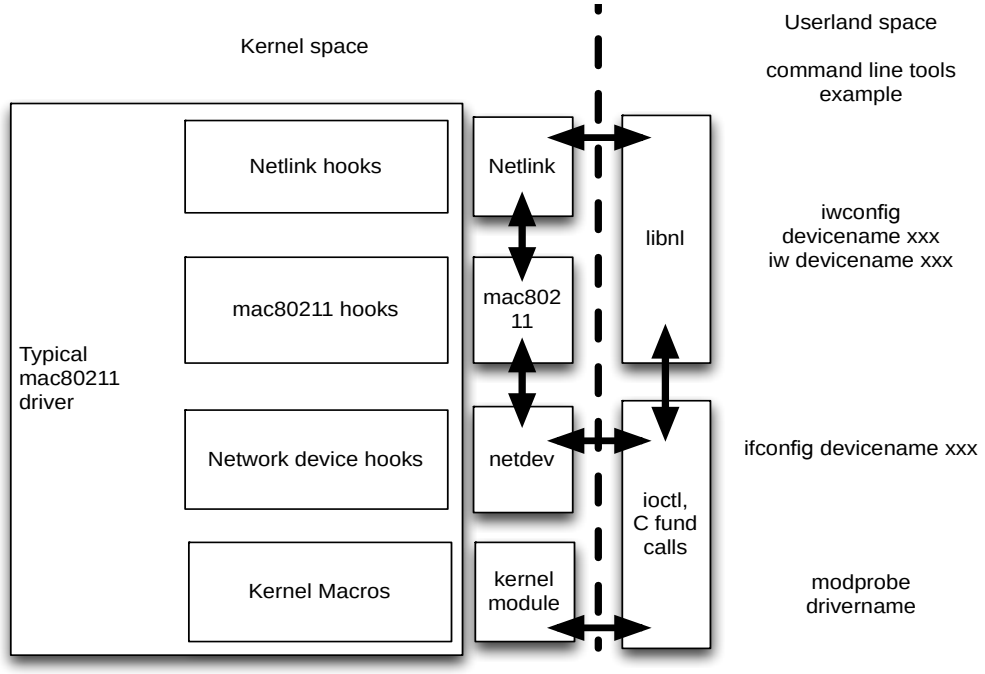

Figure 15: mac80211 typical interface and hooking mechanism.

FLAVIAn as an overlay layer has to provide the same hooks, but additionally is required to provide the following registering functions:

- *flavian_netdev_attach_ops* as a replacement for *netdev_attach_ops* that is needed to managed the netdev hooks;
- *flavian_ieee80211_alloc_hw* as a replacement for *ieee80211_alloc_hw* that is providing the mac80211 hooks
- *flavian_genl_register_family_with_ops* as a replacement for the function *genl_register_family_with_ops* for the Netlink hooks that might be needed.

The overlay mechanism in the new mac80211++ stack is depicted in Figure 16.

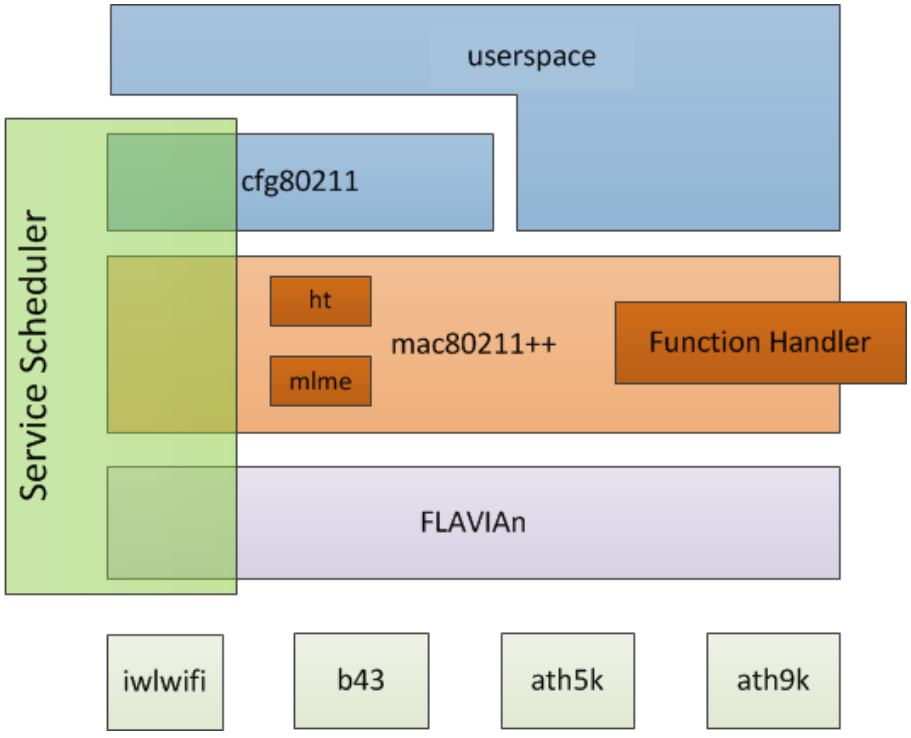

Figure 16: Virtualization overlay driver (FLAVIAn).

In addition to the above, we have determined that FLAVIAn should commit to the following minimum requirements to support mac80211 and the Linux kernel:

- netdev hooks: FLAVIAn should provide a start transmit function (ndo_start_xmit), MTU control (ndo_change_mtu), MAC address control (through ndo_set_mac_address)
- mac80211 hooks: FLAVIAn should provide a packet transmit interface (tx), start and stop capabilities (start/stop), configuration (config) as well as add/ remove interface hooks (add_interface and del_interface). Finally, a

configurable filter is required (configure_filter).

Figure 17 summarizes the above minimum requirements.

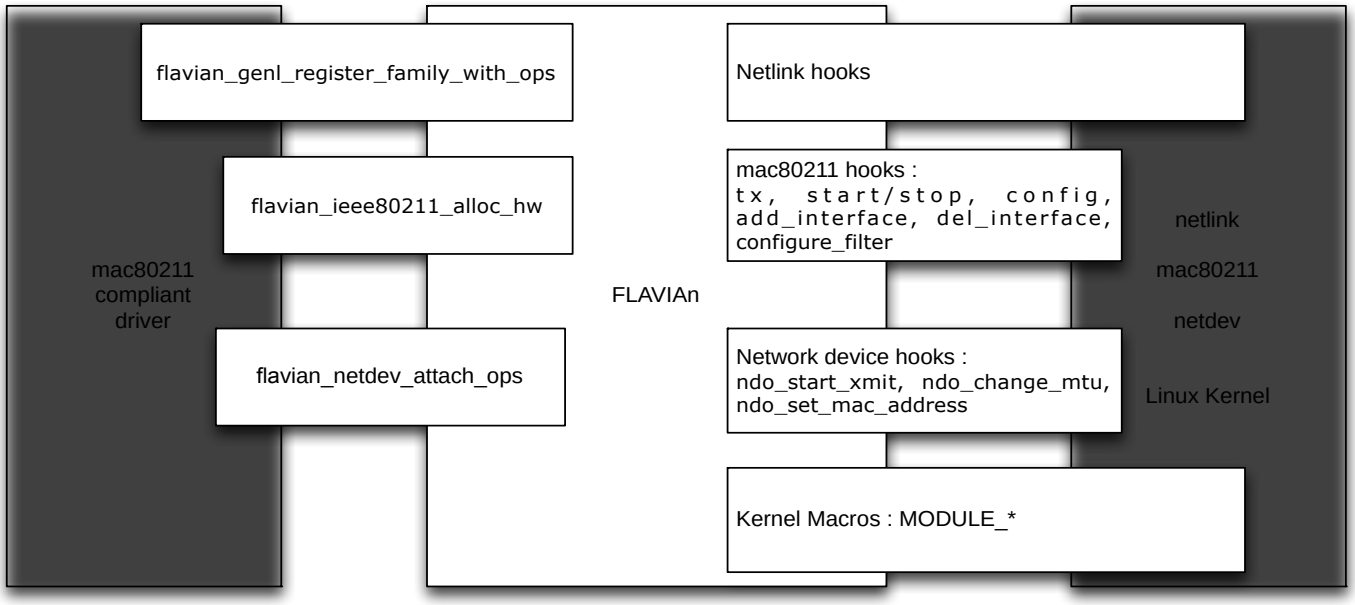

Figure 17: Requirements for FLAVIAn virtualization-enabler driver.

To ensure appropriate interaction of the mac80211 stack and the device drivers through the FLAVIAn overlay, a small modification in the driver code is also required in order to reroute the mac80211 callbacks the driver is capable of handling to point to the equivalent `ieee80211_flavian_ops` structure specified by the FLAVIAn overlay.

3.4.1 Virtualization example

We provide an example for the ath5k driver. The handlers supported by this driver will be defined as follows:

```
const struct ieee80211_flavian_ops ath5k_hw_ops = {
        .tx                     = ath5k_tx,
        .start                  = ath5k_start,
        .stop                   = ath5k_stop,
        .config                 = ath5k_config,
        .prepare_multicast      = ath5k_prepare_multicast,
        .configure_filter       = ath5k_configure_filter,
        .sw_scan_start          = ath5k_sw_scan_start,
        .sw_scan_complete       = ath5k_sw_scan_complete,
        .get_stats              = ath5k_get_stats,
        .conf_tx                = ath5k_conf_tx,
```


```
        .get_tsf                = ath5k_get_tsf,
        .set_tsf                = ath5k_set_tsf,
        .reset_tsf              = ath5k_reset_tsf,
        .get_survey             = ath5k_get_survey,
        .set_coverage_class     = ath5k_set_coverage_class,
        .set_antenna            = ath5k_set_antenna,
        .get_antenna            = ath5k_get_antenna,
        .set_ringparam          = ath5k_set_ringparam,
        .get_ringparam          = ath5k_get_ringparam,
};
```

The key functionality covered by the above handlers cover frame transmission (the low-level driver should send the frame out based on the configuration in the Tx control data and should stop queues appropriately), enabling/disabling the hardware (including turning on frame reception at start-up and clearing queues before shutdown), configuring Rx filtering, notifying about status of the scanning procedure (start/complete), etc.

Note that the add/remove interface capabilities have been removed since virtualization will be handled in the FLAVIAn driver and the device driver will be no longer required to support multiple interface management. BSS association status will also be moved to the FLAVIAn layer, while the capabilities to change the Tx parameters used for channel contention will be implemented but triggered by FLAVIAn for pure virtualization. The same principle holds for antenna and queue configuration. Since TSF is only used in IBSS operation mode, the related handlers may be disabled since we expect the driver to be agnostic of the operation mode with FLAVIAn managing associations of multiple virtual devices. Moreover, if the firmware/hardware takes full care of TSF synchronization, this functionality will not be required but implementing it remains still under discussion.

Similar modifications will be required to support other mac80211 drivers (e.g., ath9k, b43) with the FLAVIAn overlay, but as we explained above, such changes will involve limited programming effort.

3.5 Information Base

The Information Base plays an important role within the FLAVIA architecture since it provides a unified vision of the data for all the FLAVIA modules. Note that at the implementation stage of this entity, existing data management solutions offered by the existing protocol stack must be considered.

In particular, in this section we will introduce the main structures defined in the Linux Wireless stack that includes the mac80211 module. We envisage that the FLAVIA framework may conveniently extend these structures in order to provide the

developer with the functionality required by the FLAVIA architecture.

The structure *ieee80211_local* is a general-purpose collection of information embedded in other structures. For example, it includes the *ieee80211_hw* structure that contains the hardware specific information of the wireless card. Another structure defined here is the ieee80211 operations (*ieee80211_ops*) that allows accessing the driver from the mac80211 modules. In addition, the *ieee80211_local* structure contains the states of the wireless interface (suspended, resuming, started, etc.). Other structures contained herein are related to management and configuration.

The mac80211 framework integrates this structure in the *net device* structure used by the Linux Network Devices to differentiate among the installed wireless cards. For example, each physical card allocates its *ieee80211_local*, thus the different functions can operate separately on the various interfaces.

To interact with the wireless stack, developers have to extend the *ieee80211_local* structure to be able to store custom data. In the FLAVIA framework the different services store the data using the Information Base. Hence, in order to integrate the FLAVIA framework with the mac80211 framework we define a practical way of extending the structure *ieee80211_local* by using the data sharing interface defined in the FLAVIA general architecture.

To this aim the data gateway interface is extended to include a structure of *ieee80211_local* type as a parameter. In case this structure is not provided, the data sharing module will store the data in a general repository. Otherwise, when ieee80211_local is provided, the data sharing module stores the data within that structure, creating separate data storage for each interface.

3.5.1 The Data Sharing module

As defined in the general architecture, the Information Base contains a Data Sharing module that acts as gateway among the FLAVIA Services and Functions. We consider two different approaches for storing data:

a. Multiple Writers Multiple Readers with challenge protection.
b. Single Writer Multiple Readers.

In the first case, the set of functions for the Multiple Writers Multiple Readers solution is:

- *mac8_ds_set*: store the data.
- *mac8_ds_read*: retrieve the data.
- *mac8_ds_remove*: remove the data.
- *mac8_ds_protect*: protect the data against arbitrary data changes.

All the functions receive the structure *ieee80211_local* as first parameter to select the repository and, as second parameter, an integer public key to identify the specific data. In addition, the method *mac8_ds_set* requires as input the data value and size, while the function *mac8_ds_protect* receives also the protection callback. The specific

function signatures are introduced next:

`void *mac8_ds_read( ieee80211_local *local, int publicKey );`	`int mac8_ds_remove( ieee80211_local *local, int publicKey );`
`int mac8_ds_set( ieee80211_local *local, int publicKey, void *data, unsigned int size );`	`int mac8_ds_protect( ieee80211_local *local, int publicKey, int (*consistency_test) (struct mac8_ds_data_protection *) );`

Table 4: Multiple Writers Multiple Readers approach implementation

Note that the description of the protection callback is defined as follows:

```
int (*consistency_test)(struct mac8_ds_data_protection *)
```

where the *mac8_ds_data_protection** parameter represents a structure containing the structure ieee80211_local, the old value, the new value and the public key.

Furthermore, the data sharing module manages an asynchronous procedure that is used for keeping informed about the change of data values. For this purpose, we define two functions, namely *mac8_ds_on_change_listener_add* and *mac8_ds_on_change_listener_rem*. Both functions require as parameter the structure *ieee80211_local* to choose the repository, and the public key to select the observed data. The *mac8_ds_on_change_listener_add* function returns an integer identifier associated to the registered listener. Later this identifier can be later used as parameter in *mac8_ds_on_change_listener_rem* to unregister that specific listener. The function definitions are presented in Table 5.

`int mac8_ds_on_change_listener_add( ieee80211_local *local, int publicKey, void (*notify_function)(void *) );`	`int mac8_ds_on_change_listener_rem( ieee80211_local *local, int publicKey, int idListener );`

Table 5: Data Sharing Listener Management implementation

The notify callback has the following signature:

```
void (*notify_function)(void *)
```

The *notify_function* activates a new Linux task. In the registration process the user is notified about the repository and the public key related to the callback.

For the second approach, Single Writer Multiple Readers, we define the following functions:

- *mac8_ds_create_sw*: for creating and storing the data.
- *mac8_ds_update_sw*: for updating the data.
- *mac8_ds_remove_sw*: for removing the data.

Table 6 presents the declaration of these functions.

`int mac8_ds_create_sw( ieee80211_local *local, int publicKey, void *data, unsigned int size );`	`int mac8_ds_update_sw( ieee80211_local *local, int privateKey, void *data, unsigned int size );`
`int mac8_ds_remove_sw( ieee80211_local *local, int privateKey );`	

Table 6: Single Writer Multiple Readers approach implementation

The *mac8_ds_create_sw* accepts as parameter the structure *ieee80211_local* and the public key. It returns the private key for updating the data with *mac8_ds_update_sw* and for removing the data with *mac8_ds_remove_sw*. The update and create tasks require also the value to be stored and its size. It is worth noting that all the reading functions can be used with the public key.

Table 7 summarises a new group of methods that has been defined for typed reading and writing:

`int mac8_ds_read_int ( ieee80211_local *local, int publicKey );`	`int mac8_ds_set_int ( ieee80211_local *local, int publicKey );`
`long mac8_ds_read_long ( ieee80211_local *local, int publicKey );`	`int mac8_ds_set_long( ieee80211_local *local, int publicKey);`
`double mac8_ds_read_double ( ieee80211_local *local, int publicKey );`	`int mac8_ds_set_double( ieee80211_local *local, int publicKey );`

`char* mac8_ds_read_string(`
`ieee80211_local *local,`
`int publicKey`
`);`
	`int mac8_ds_set_string(`
	`ieee80211_local *local,`
	`int publicKey`
	`);`

Table 7: Set of reading and writing functions

4 Operation Modules

To illustrate the functionality introduced by FLAVIA, new services are added to the wireless architecture. We specify five operation modules, i.e. SuperSense, Data transport with QoS capabilities, Power Saving, Advanced Monitoring and Rate Adaptation.

We specify these services in detail (which service, macro-service or functions are involved) we discuss the mapping and interaction of each module within the FLAVIA architecture and describe the interfaces that are exposed. We highlight the improvements introduced over the standard mechanism, and present some use cases for them as well.

4.1 SuperSense

FLAVIA SuperSense is an innovative sensing service able to select constantly the most advantageous network configuration thanks to a dynamical analysis of the radio spectrum performed using both passive and active techniques.

The SuperSense module comprises three main structures:

1. *flavia_ss_item*: object containing the work to be registered on the scheduler work queue and all the information concerning to the new service;
2. *flavia_func_item*: structure used to define the list of function to be called, when the corresponding hook is executed;
3. *sps_ieee80211_local*: a structure containing all the driver global information.

The module's key functions are the *init* and *exit* ones. In fact, the *flavia_sps_init* function is the most important one since it initializes the module through the following steps:

1. Get the ieee80211_local struct from mac80211.
2. Get the Time Synchronization Function (TSF) value for synchronization issues.
3. Register the function *flavia_sps_init_tsf_polling* that will be executed to start the polling activity.
4. Register the function *flavia_sps_exit_tsf_polling* that will be executed to end the polling activity.
5. Register the function *flavia_sps_set_superframe_ie* that will be executed to set up the new Information Element used to define the SPS Super-frame.
6. Register the function *flavia_sps_start_tsf_polling* that will be executed when the station gets associated.

Once all those functions are correctly registered, the SPS service is considered loaded.

Conversely, the function *flavia_sps_exit* removes the functions previously registered, unloading the service.

The mechanism behind FLAVIA SuperSense allows switching the device between two types of interfaces, which represent two different operation modes that alternate periodically:

1. Tx/Rx mode.
2. Monitoring mode.

Through the introduction of a new data structure, named *super-frame*, the module can set and manage the total duration of a SPS period, and the specific length of the operation modes.

The super-frame always starts with an active monitoring period followed by a transmission period in which all nodes that belong to the same BSS operate using the same medium access mechanisms to transmit their data traffic. During an active monitoring period, only one node is allowed to send probes along the wireless channel in order to estimate the global link quality, based on different metrics.

The *flavia_sps_set_superframe_ie* function sets the new Information Element (IE) contained in the beacon, which will be used to define the FLAVIA SPS Super-frame. The IE contains two variables indicating the global duration of the frame period and the time spent to perform the active monitoring.

To allow all the nodes to switch at the same time from one operational mode to another, synchronization is required and obtained through to the TSF. This ensures that the timers for all stations in the same BSS are kept synchronized. Each node maintains a TSF timer with modulus 2^{64} counting in increments of microseconds. Timing synchronization is achieved by stations periodically exchanging timing information through beacon frames.

Once a given station extracts the TSF value from the beacon, the station is synchronized. Then, the SPS service uses *the flavia_sps_tsf_superframe* function to compute the super-frame length, through the new IE contained in the beacon. In addition, SPS builds the frame mask in order to constantly check when to switch from transmission mode to monitoring, and vice versa. The transition between the two operational modes is handled by:

1. *flavia_tx_to_mon_mode*. It changes the operating mode from Tx/Rx to active monitoring, according to the following steps:
    a. Send a null_func frame with the power save bit on, so that the AP will buffer the frames addressed to the stations while they are not listening.
    b. Disable the *hard_start_xmit* of the AP/STA virtual device.
    c. Change the channel according to the IE sequence and start the active sensing of the channel.
    d. Stop all Tx queues of the non-monitoring network devices (STA/AP) used

by upper layers through *hard_start_xmit*.

e. Start all Tx queues of the monitoring network devices.

f. Disable beaconing.

g. Enable active monitoring (via *flavia_enable_active_probe* function).

2. *flavia_mon_to_tx_mode.* It changes the operating mode from active monitoring to Tx/Rx, according to the following steps:

   a. Enable the *hard_start_xmit* of the AP/STA virtual device.

   b. Tune back to the original channel and send a *nullfunc* frame with the power save bit off to trigger the AP to send the stations all the buffered frames.

   c. Switch back to the operating channel.

   d. Restart all AP/STA interfaces.

   e. Start all Tx queues of the non-monitoring network devices (STA/AP) used by upper layers through *hard_start_xmit*.

   f. Stop all Tx queues of the monitoring network devices.

   g. Enable beaconing.

   h. Disable active monitoring (via *flavia_disable_active_probe* function).

The SuperSense service handles the active probing activity through the *flavia_active_probe_work* function, which enables a single device to transmit during the active monitoring period. In particular, to optimize the active monitoring activity, the above function implements a distributed round robin mechanism that coordinates the medium access for all stations within the same Basic Service Set. The coordination mechanism implemented by the SuperSense service prevents collisions caused by simultaneous transmissions of stations that are hidden to each other (i.e., the intra-interference), thus improving the estimation accuracy of the interference caused by external sources.

Finally, the function *flavia_send_active_probe* transmits specific monitoring packets on a given channel at a specific data rate. Many probes are sent through the wireless channel varying these parameters in order to determine, according to the collected sensing statistics, the best available network configuration.

4.2 Data transport with QoS capabilities

The FLAVIA framework aims at providing the different modules with a direct control of the QoS in the transmission part. For this purpose, we extend the mac80211 framework to provide an interface to the traffic control functionalities that are supported in the Linux network device component.

Typically, the control of these functionalities is handled by the *traffic control* (tc) tool, which is part of the *iproute2* program suite. This tool operates through the netlink framework to implement the user space/kernel space communication. Nevertheless, a kernel module is not able to interact directly with the scheduling component of the traffic control functionalities, and is limited by the provided netlink interface as well.

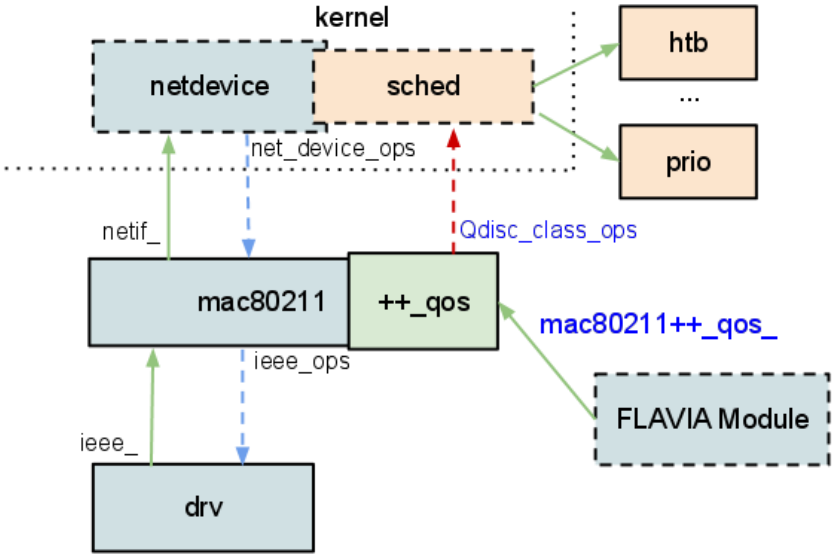

Figure 18: The new mac80211 QoS interface for the data transport.

To provide the FLAVIA modules with a fine grained QoS control of the wireless interfaces, we add to the new maxc80211++ a QoS extension as depicted in Figure 18.

The new interface will use the *mac80211++_qos_* prefix and will wrap the interfaces defined in the Linux scheduler component. In particular, Figure 18 shows the *Qdisc_device_ops* interface that is utilized to control (add/remove/configure) the Linux queuing disciplines (*qdiscs*).

In what follows, we describe the functionalities available through this new interface.

- **Queuing Discipline (qdisc) functions**

Each qdisc type included in the Linux kernel must provide a set of callbacks (collected into a *qdisc_ops* structure):

- *init*: initialize and configure the queuing discipline.
- *enqueue*: enqueue a socket buffer (*skb*) according to the queuing discipline.
- *dequeue*: return the next packet to be sent. If the queue is empty, then NULL is returned.

- *requeue*: allow the re-insertion in the queue of a previously dequeued socket buffer. The socket buffer is re-inserted in the same position from which it was dequeued and not considered for the queue statistics.
- *drop*: remove a socket buffer from the qdisc.
- *change*: change the configuration of the queuing discipline
- *reset*: reinitialize the qdisc by emptying its queue and setting the default configuration. Moreover, the reset function is called recursively for the attached classes and qdiscs.
- *destroy*: destroy a qdisc and remove associated classes and filters. Delete all pending events and free resources.
- *dump*: return status and diagnostic information.

The *qdisc_ops* options are registered by modules through the *register_qdisc* function. To identify a qdisc instance, a 32-bit identifier is employed. This identifier is composed by two fields, the 16 most significant bits identifying the major number, and the other 16 bits corresponding to the minor number.

- **Traffic classification functions**

Defining traffic classes allows classifying the packet stream according to different priorities. Each class is identified by: i) an *InternalID*, assigned by the qdisc to which the class belongs, and ii) a *classID*, which is structured as the qdisc identifier. The major number corresponds to the qdisc to which the classes belong while the minor number must be different for each attached class.

The classes are managed through the callbacks defined inside the *Qdisc_class_ops* structure:

- *graft*: attaches a new qdisc to the class.
- *leaf*: returns the qdisc associated to the selected class.
- *get*: returns the InternalID corresponding to a given ClassID. A class usage counter is incremented.
- *put*: usually called after a get, decreases the usage counter of a class. If the counter reaches 0 the class is destroyed.
- *change*: changes the properties of a class.
- *delete*: deactivates and destroys a class, if it is not used.
- *walk:* iterates through all the classes that belong to the qdisc and invokes a function for each of them.
- *tcf_chain*: returns the list of filters associated to a class.
- *bind_tcf*: links a filter instance to a class.
- *unbind_tcf*: removes a filter instance from a class.
- *dump_class*: returns status and diagnostic information of a class.

When the *enqueue* function of a qdisc is executed, the appropriate class is selected by invoking the *tc_classify* function, which returns a structure (*tcf_result*) that contains the classID. Then, the enqueue function of the qdisc contained inside the class is invoked recursively.

- **Traffic filtering functions**

Filters are employed by the qdiscs to assign an incoming packet to one of their classes. The filters are maintained in ordered lists according to priority values. Filters are identified by handlers, which are 32-bit integers (but not structured into major and minor numbers). The functions associated to filters are grouped into structures of type *tcf_proto_ops*:

- *init*: initializes a filter.
- *destroy*: deletes a filter. If it is linked to a class, the unbind_tcf function of the class is called.
- *classify*: performs the actual classification, returning the classID of a corresponding class .
- *get*: returns a filterID associated to a given handler.
- *put*: should be called when a filter (previously selected through a get call) is no longer in use.
- *change*: configures a new filter (calling the bind_tcf function of the associated class) or modifies the configuration of a pre-existing filter.
- *delete*: destroys the internal element of a filter. To delete the entire filter the destroy function must be called.
- *walk*: iterates over all the elements of a filter and invokes a function for each of them.
- *dump*: returns the status and diagnostic information regarding a filter or an element that belongs to the filter.

- **Driver support for QoS**

To provide an actual control of the QoS in the Linux Traffic Control framework, a direct connection of the wireless card queues has to be created towards the overall net device that includes the mac80211 and the driver.

We will provide an example for the b43 driver. This driver does not use a direct connection between the in-kernel net-device space queues and the underlying DMA rings located in the NIC. In queue saturation condition, this causes packets to be dropped at the driver level, thereby being taken out of the Linux traffic control scope. Thus, the tc qdisc queues are always empty losing the expected shaping functionality. To provide the aforementioned direct connection of the wireless card queues, we extend the b43 driver as a proof-of-concept, namely *b43*.* This solution implements a direct connection between the *net_device* level queues and on-NIC DMA rings. This modification enables the queuing disciplines defined at the *net_device* level to be effective by actually controlling the packet dropping.

Figure 19 illustrates the architecture of the driver *b43**. Having multiple *tx_queue* structures inside the driver allows to separate packet streams. In case of saturation at

the DMA or driver level, the dequeuing process at the corresponding queues inside the *net_device* will be stopped. In this way *net_device* queues will be filled, and the configured Linux traffic control queuing discipline and dropping policy will be implemented, instead of having uncontrolled packet losses, as it happens with the current version of the b43 driver.

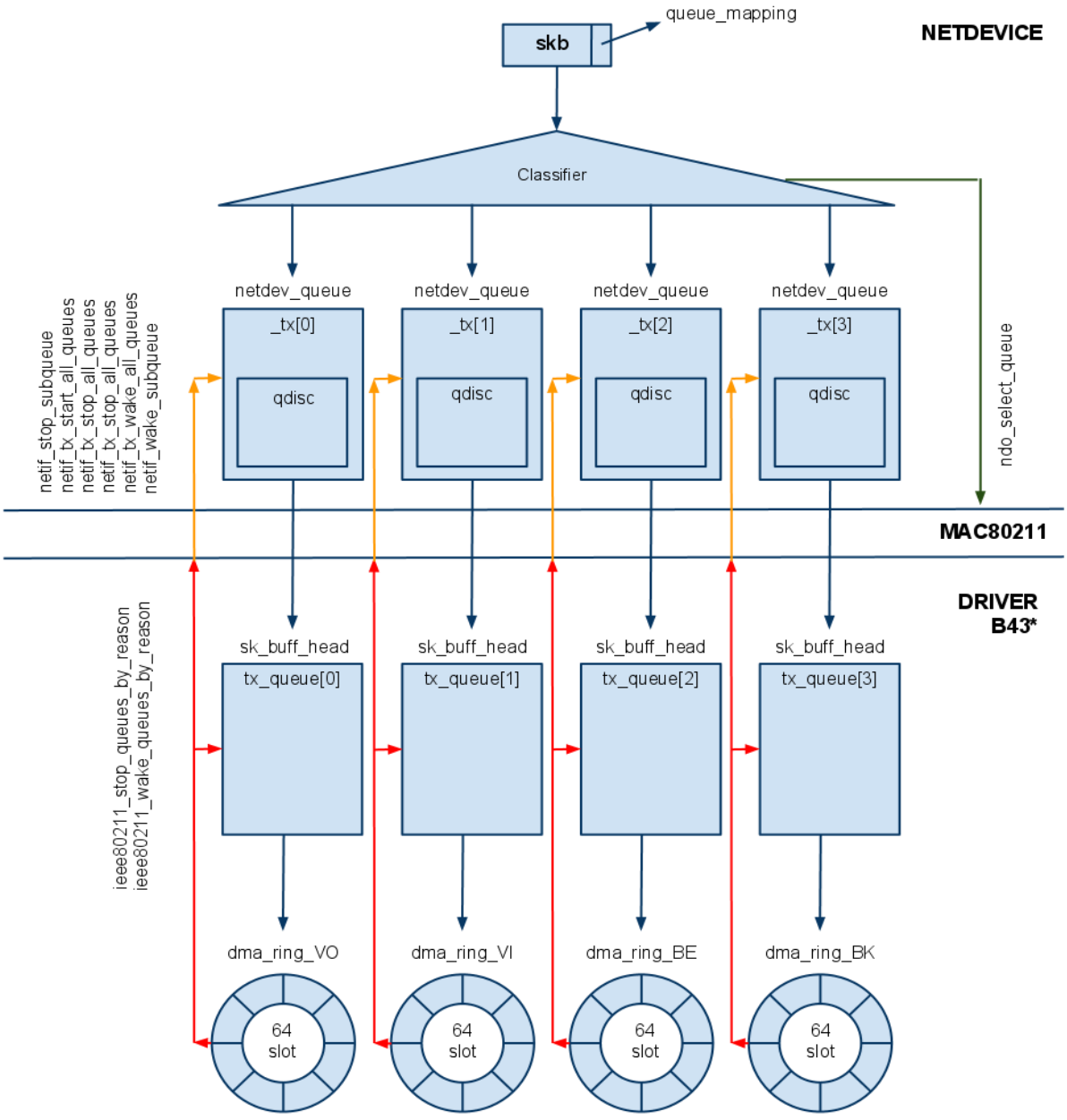

Figure 19: Architecture of the b43* driver.

4.3 Power Saving

The development of power saving mechanisms can also benefit from a modular and flexible architecture. FLAVIA offers a framework whereby tasks such as monitoring, frame forging, or sleep/awake transitions are exposed as online functional blocks that are provided with a common shared data space.

- **Current Framework**

At the testing stage of the custom power saving (PS) algorithms there are two options:

a) Write a complete specific SoftMAC for capable drivers.

b) Hook into current Linux wireless stack the functions needed to support the proposed mechanism.

Needless to say, the second option is the simpler of the two options but still entailing a high degree of complexity. In order to illustrate this, the following introduces an actual PS mechanism, NoA/ASPP [10], and the subsequent implementation within the current Linux wireless stack.

Adaptive Single Presence Period (ASPP) is a power saving algorithm to adaptively control the Notice of Absence (NoA) protocol specified by the WiFi Alliance. NoA has been proposed in order to provide energy savings to all devices in a WiFi-Direct network, a peer-to-peer wireless communication technology specified by the WiFi Alliance. WiFi Direct devices, named peer-to-peer (P2P) devices, must be able to act either as an AP or as a Client. In particular it defines the concept of a P2P Group, where a P2P Group Owner (P2P GO) acts as an AP for a set of connected P2P Clients. Thus, the AP (P2P GO) might be a mobile power-constrained node needing therefore power saving procedures.

Figure 20 demonstrates how this PS scheme can be hooked into the Linux Wireless stack. First, wpa_supplicant[1] should add a static NoA Information element (IE) into the beacon frame template, hand it over to the cfg80211 module and finally to mac80211. Specific functions hooked into the Rx/Tx handlers in mac80211 carry out the measurements required by ASPP and finally, after computing the parameters for the NoA, ASPP needs to edit the static template for every beacon transmission. Upon receiving a beacon frame, the driver (e.g., ath5k) should be provided with the ability of handling the NoA IE and scheduling the absent periods accordingly. The analysis of every piece of the stack, hooking functionalities and reusing code in order to test and to implement new algorithms are clearly difficult tasks using the current architecture.

[1] http://hostap.epitest.fi/wpa_supplicant/

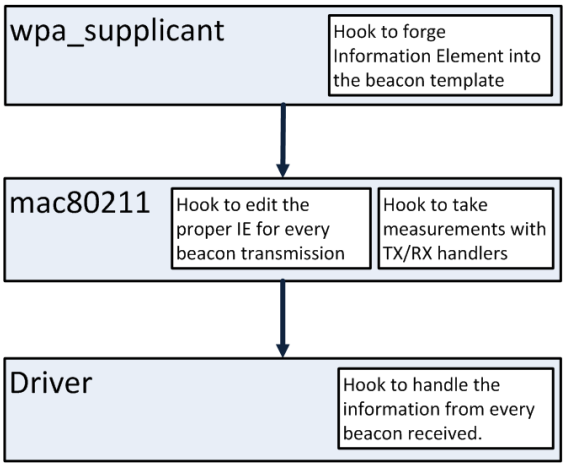

Figure 20: New PS mechanism implementation using the current framework.

- **PS Service module**

The goal of the PS service module is to enable various power saving algorithms to be easily implemented through specifying helpful functional blocks and their interactions with other FLAVIA services, functions or the Information Base.

We take advantage of mac80211 and the FLAVIA service management specified above, and also identify some primitives required to command to the actual hardware to perform certain atomic and hardware-specific tasks, e.g., change its state to a low power state (sleep).

The mac80211 framework exposes the modularity and flexibility required by FLAVIA to easily implement this power saving service as an (un)loadable module. We define the following two functions:

1. flavia_ps_policy(): This function is registered into the FLAVIA Function Handler. It incorporates the logic of the developed algorithms and stores information of the mechanism(s) in operation and its (their) state.
2. *flavia_ps_management():* This function provides management logic to support the PS mechanisms being implemented.

Finally, a generic power saving scheme might require the ability of triggering sleep/awake events. This action is ultimately performed by hardware through setting the proper hardware registers accordingly. We then specify a primitive to communicate with the immediate low layer the notification to execute the chosen event.

- drv_flavia_ps_notify(): This is a notification primitive and requires drivers to provide its proper handling. Thus, we push all the "intelligence" to the upper layer, designing this way a hardware-agnostic power saving framework.

In order to support sleep/awake transitions typically required by power saving algorithms, it is still needed that drivers/firmware support sleep/awake events (issued

by the previously mentioned primitives of the PS service). In order to demonstrate this functionality, we choose the ath9k driver, which does not require any firmware, and we implement the handler functionality required for these notifications.

Following the example introduced in the previous section, the implementation of an algorithm such as NoA/ASPP becomes much simpler using the FLAVIA framework. The developer specifies the monitoring functionality that needs to be supported by the FLAVIA monitoring service. In turn, forging the NoA IE in beacon frames can also be supported by an extended FLAVIA management service. Finally, the computation of the absent periods and the scheduling of sleep/awake events is supported by this new PS service. Figure 21 illustrates the generic functional modules that typically interact to support the power saving mechanism within the FLAVIA architecture.

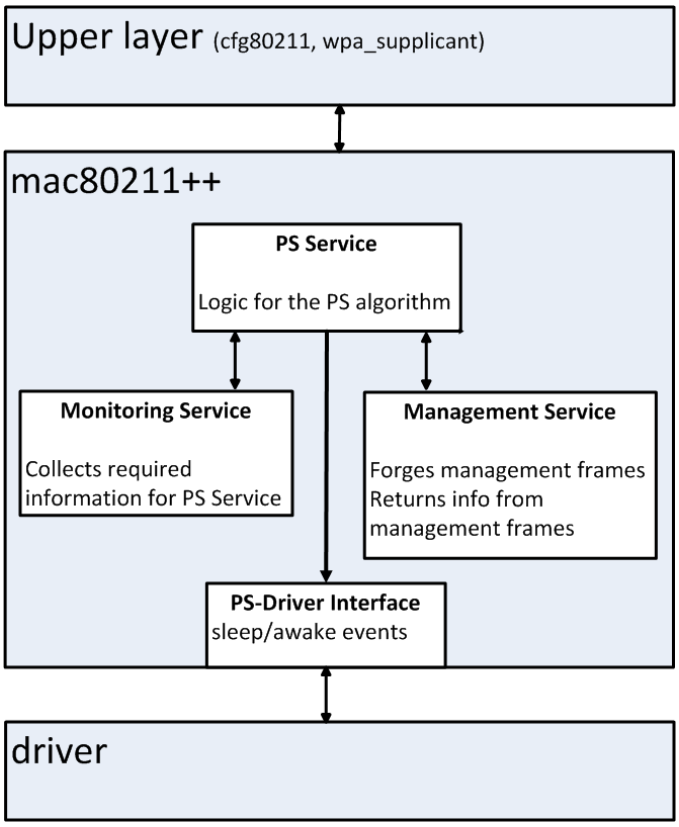

Figure 21: PS mechanism implementation within the FLAVIA architecture.

The implementation of NoA/ASPP is only an illustrative example to demonstrate the benefits of using the PS Service in a modular and flexible architecture. We envision two categories of potential power saving mechanisms that can be implemented:

- Sleep/awake mechanisms: These mechanisms schedule intervals when a wireless interface dozes being in a low-power state, and intervals when it

becomes active by idling, transmitting or receiving. NoA/ASPP is an example within this category.

- PHY/MAC adaptation mechanisms: These mechanisms adapt different PHY parameters (e.g., transmission power, modulation coding scheme), and/or MAC parameters (e.g., EDCA parameters, user association policies).

4.4 Advanced Monitoring

The FLAVIA Advanced Monitoring (FAM) module provides a passive monitoring service able to measure several parameters related to radio channel conditions and capabilities of neighbouring nodes. Each node performs PHY/MAC layer measurements within the time-scale of microseconds, based on all types of 802.11 frames (data, management, and control). The promiscuous mode of operation of wireless network cards is utilized to ensure a comprehensive view of the current wireless channel conditions. This means that all measurements are performed along with the normal activity of the wireless card and reported periodically to the Information Base and MAC parameter calculation service (defined in Appendix A.1). The FAM module supports multiple network interfaces per node. The results of FAM measurements are utilized mainly by the Misbehaviour Detection and Reaction service and the Consistency Manager module. The FAM module works on the frame level – this means that all frames sent and received by each network interface must be examined by the FAM module functions. This imposes high requirements on the FAM module on the effectiveness of the frame analysis (i.e., limited computational power available at the nodes should be taken into account). Therefore, in the first phase of implementation we implemented hooks in the mac802.11 module which are directly accessible for FAM. In the next implementation phase the full integration with FLAVIA Service Management Framework is expected.

The FAM module consists of the following main functionalities:

1. *init* and *exit* functions responsible for initialization of other FAM module functionalities.
2. Functions responsible for communication with user space where the Discovered Capabilities DB (DCDB) component resides and control mechanisms are placed.
3. Measurement functions for uplink and downlink data path.

The FAM module's key functions responsible for initialization are the *init* and *exit.* In particular, the *flavia_fam_int()* is the most important; it initializes the module through the following steps:

1. Initialize netlink communication with the user space.
2. Initialize the function responsible for receiving the configuration and control information from the user space.
3. Initialize the thread responsible for sending measurement data to the user

space applications.

4. Initialize the internal module data structures.
5. Activate the hooks in the mac80211 module.

Conversely, the *flavia_fam_exit()* function deactivates the hooks in the mac80211 module, closes the sending thread and frees resources allocated for data structures.

For communication with user space the *netlink* mechanism is used. The application that requires monitoring data from the FAM module sends the command to the receiving function of the FAM module. This command defines the parameters the application requests and the time interval at which results are to be sent to the application.

The hooks in the mac80211 module have to be placed in the *ieee80211_rx* function for the downlink frame path and in the *ieee80211_tx* function for the uplink frame path. The measurement functions called by the hooks require access to each frame header and frame timing information to discover and calculate the following parameters per each neighbouring station interface: supported rates, preamble type, state of the power saving and WEP security modes, country code, Signal to Noise Ratio (SNR), Frame/Bit Error Rate (F/BER), RTS_Threshold, Fragmentation_Threshold, beacon interval, operation mode, number of retransmissions, channel occupancy. The measurements are also used to determine NAV, backoff, and IFS size independently for each received frame. This calculation is done separately for each station that is within the neighbourhood of the FLAVIA node. Table 19 in Appendix B.3 presents a preliminary source code for the FAM module.

4.5 Rate Adaptation

In order to maximize network performance, current WLAN devices employ rate control algorithms that select the PHY rate used for packet transmission based on the observed channel conditions. By seeking to choose the appropriate transmission rate, these algorithms aim to reduce the packet loss rate while efficiently using the wireless resources in terms of channel time.

As the IEEE 802.11 standard does not provide any guidance for designing robust rate adaptation mechanisms, the solutions encountered nowadays in operational systems are either proprietary to the respective card vendors or have been developed by open source communities. The predominant approach taken by these algorithms is to rely on transmission related statistics to trigger incremental decisions to adjacent rates. This is also the case of the Minstrel[2] and PID[3] (proportional-integral-derivative) rate control algorithms supported by the mac80211 framework which constitutes the foundation of the FLAVIA implementation prototype.

[2] http://wireless.kernel.org/en/developers/Documentation/mac80211/RateControl/minstrel
[3] http://wireless.kernel.org/en/developers/Documentation/mac80211/RateControl/PID

In this section we discuss how rate control algorithms are implemented within the mac80211 framework and how the modularity capabilities can be exploited to support the integration of improved algorithms. As a proof of concept we discuss the extension of the ath5k driver for Atheros based cards with the H-RCA rate adaptation paradigm [11] that minimizes the average time each packet spends on the medium including MAC retries, in a fully decentralized fashion with no message exchange, providing higher and more stable throughput.

- **Current Framework**

Most of the current mac80211 drivers rely on rate control algorithms provided by the framework. These algorithms are encapsulated in independent kernel modules that are linked to the specific driver once a new device is being loaded. The naming convention of these modules is based on an *rc80211_* prefix, followed by the name of the algorithm. As already mentioned, mac80211 implements (i) the Minstrel algorithm, which uses a heuristic that relies on transmission success probability and air-time to select the appropriate rate, and (ii) PID, which employs a proportional-integral-derivative controller that takes as input the frame error rate and controls the transmission rate. Minstrel is currently the default rate adaptation scheme used. On the other hand, the framework allows the drivers to implement their own rate adaptation mechanisms and register them upon device initialization to notify the mac80211 framework that rate selection will be handled by the driver itself. One example of such drivers is ath9k for Atheros cards.

Despite the differences of these two approaches, they both use a common mechanism to interface with the mac80211 framework. Specifically, the *rate_control_ops* callbacks are registered by the rate adaptation module to the framework. These design principles are illustrated in Figure 22.

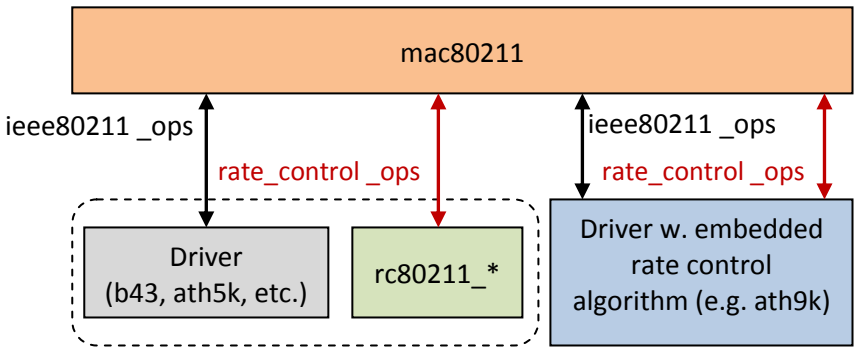

Figure 22: Interfacing Rate Control with mac80211

- **H-RCA**

Existing rate adaptation schemes estimate channel conditions by either directly measuring link SNR or recording packet loss statistics. However, current state-of-the-art algorithms do not distinguish losses due to packet collisions from losses that occur

due to noise. Therefore, they often wrongly select suboptimal transmission rates and incur drastic throughput oscillations.

We investigate how a collision aware rate control algorithm, H-RCA [11], could be implemented in the FLAVIA framework. H-RCA eliminates the abovementioned shortcomings of existing techniques and provides improved network performance. First, given a rate-set $\{r_1, \ldots, r_K\}$ Mb/s with $r_i < r_i{+}1$ for all $i \in \{1, \ldots, K-1\}$ (e.g., for 802.11a {6, 9, . . . , 54}), the rates r_i for which the Packet Loss Ratio (PLR) in a given channel conditions is higher than the one for a higher rate r_j are identified and excluded from the rate-set. To estimate the PLR, H-RCA uses a technique based on IEEE 802.11e TXOP functionality [3] to gain observations of packets solely susceptible to loss through channel noise. As defined in the standard, when a station gains access to the medium and successfully transmits a packet, if the remaining TXOP time is long enough for another packet transmission, the station can transmit the next packet after a SIFS without an additional backoff period. If any packet in the TXOP burst results in an unacknowledged transmission, no further packets are sent. At the time the second or later packets in the TXOP burst are transmitted, all other stations in the network see the medium as continuously busy so there can be no collision. Hence, if transmission of the second or later packets in the burst fails, it can only have been caused by noise.

This technique overcomes a significant limitation of the hardware to distinguish transmission failures that occur due to collisions from those that occur due to noise on the medium. This is important as if the rate of transmission failure increases there are two potential explanations, each of which would dictate distinct corrective action. If the channel is experiencing increased noise, transmission failures will result and the station should change to a lower, more robust rate.

For each rate a critical PLR value above which a lower rate would give higher throughput is determined and Bayesian inference is employed to determine if the PLR of the current rate is above a rate lowering threshold. To explore superior rates, the rate increase frequency is set so that the opportunity-cost of sampling a higher rate is kept below 5%.

We envisioned an enhanced ath5k driver for Atheros cards that incorporates H-RCA, similar to the approach taken by the rate control algorithm specific to ath9k. We note that a more modular design whereby the rate adaption technique is programmed as an independent module that suits different drivers would also be possible. To register the H-RCA algorithm to the mac80211 framework, the driver is required to invoke *ieee80211_rate_control_register* function passing a reference to a *rate_control_ops* structure that contains the handlers implemented by the algorithm. In our example this structure should be defined as follows:


```
static struct rate_control_ops ath_hrca = {
        .name = "h-rca",
        .tx_status = hrca_tx_status,
        .get_rate = hrca_get_rate,
        .rate_init = hrca_rate_init,
        .rate_update = hrca_rate_update,
        .alloc = hrca_alloc,
        .free = hrca_free,
        .alloc_sta = hrca_alloc_sta,
        .free_sta = hrca_free_sta,
};
```

The module will also access the MAC layer parameters through the FLAVIA interface to set appropriate TXOP values to facilitate H-RCA's PLR estimation procedure.

5 Conclusions

This second WP4 deliverable provides the specifications of the Wireless MAC Processor and the modules to be implemented in the FLAVIA prototype. Based on the architecture defined in [6] for an 802.11 node, we have specified the framework to provide the key FLAVIA functionality as well as a set of representative modules to be implemented. Specifically, we have identified as candidate modules to be implemented key representative blocks such as SuperSense, Power Saving and Data transport with QoS capabilities. All these new services reflect the functionalities supported by FLAVIA architecture.

Our specification has started from an almost clean-slate definition of a new architecture, the Wireless Processor, responsible for the direct interaction with the hardware modules. We have described the MAC Engine and the set of programming interfaces. We have also provided a description of the XFSM compiler developed to build new MAC programs. In addition we have introduced a set of such MAC programs to show the flexibility and modularity resulting from the FLAVIA architecture. One of the main achievements accomplished is the possibility to change the MAC behaviour on-the-fly.

One of the main difficulties posed by the envisioned clean-slated design is the need of very specific hardware, and the requirement to implement almost all software from scratch. In order to illustrate the FLAVIA features using other platforms, we have also specified a higher software architecture that can be implemented with almost any existing hardware. This specification has started from an existing framework, namely the mac80211 framework existing in Linux environments, which has been substantially extended it in order to support FLAVIA's main features in terms of modularity, flexibility and virtualization. This new framework, named mac80211++, will become the development platform. More specifically, the extension of the mac80211++ is twofold:

i. We intend to create a modular framework by untangling the existing mac80211, at the present at early stage of development.

ii. We have developed and implemented a Service Handling module that allows loading new services in real-time, based on the mac80211++ framework. This module is one of the essential components of the FLAVIA prototyping.

In order to implement the virtualization as specified in FLAVIA, a new layer, called FLAVIAn, has been specified between the mac80211 framework and the wireless drivers, exceeding the bounded capabilities of the driver that the mac80211 framework is relying on.

Another goal of this deliverable is the implementation specification of the Information Base entity corresponding to the FLAVIA architecture. We have accomplished this by extending the structure *ieee80211_local* contained in the mac80211 module.

Finally, we have also introduced in the Appendix A two new services to be supported by the FLAVIA architecture. In particular, we have described the modifications needed to support a new advanced monitoring service and an additional service (Misbehaviour Detection and Reaction). Note that, to support the new services, we have extended the Consistency Manager module as well.

6 References

[1] IEEE 802.11, *LAN/MAC Specific Requirements - Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) specifications: Medium Access Control (MAC)*. Revision of IEEE Std 802.11-1999, 2007.

[2] IEEE 802.11n, Amendment 5 to Standard for Information Technology. *LAN/MAC Specific Requirements - Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) specifications: Medium Access Control (MAC) Enhancements for Higher Throughput*, IEEE Std. 802.11n, 2010, Supplement to IEEE 802.11 Standard.

[3] IEEE 802.11 WG. Amendment 8 to Standard for Information Technology. *LAN/MAC Specific Requirements - Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) specifications: Medium Access Control (MAC) Enhancements for Quality of Service (QoS)*, IEEE Std. 802.11e, November 2005. Supplement to IEEE 802.11 Standard.

[4] FLAVIA Project Deliverable D.2.1.1 - *Report on Scenarios, Services and Requirements*, January 2011, available at http://www.ict-flavia.eu

[5] FLAVIA Project Deliverable D.2.2.1 - *Architecture Specification*, May 2011, available at http://www.ict-flavia.eu

[6] FLAVIA Project Deliverable D.4.1.1 - *802.11 architecture and interfaces specification*, May 2011, available at http://www.ict-flavia.eu

[7] OpenFWWF: Open FirmWare for WiFi networks, http://www.ing.unibs.it/openfwwf/

[8] Open development platform, Eclipse, http://www.eclipse.org/

[9] Linux kernel mac80211 framework for wireless device drivers, http://linuxwireless.org/en/developers/Documentation/mac80211

[10] D. Camps-Mur, X. Perez-Costa, S. Sallent-Ribes, *Designing energy efficient Access Points with Wi-Fi Direct*, Computer Networks, Volume 55, Issue 13, 15 September 2011.

[11] K. D. Huang, Ken R. Duffy and David Malone, *H-RCA: 802.11 Collision-aware Rate Control*, Hamilton Institute technical report, 2011.

[12] Raya, M., Aad, I., Hubaux, J., and El Fawal, *A. DOMINO: Detecting MAC layer greedy behavior in IEEE 802.11 hotspots*. IEEE Transactions on Mobile Computing, 5:1691–1705, 2006.

[13] Serrano, P., Banchs, A., Targon, V., and Kukielka, J. *Detecting selfish configurations in 802.11 WLANs*. IEEE Communications Letters, 14:142–144, 2010.

[14] Ahn, Y. w., Cheng, A., Baek, J., and Fisher, P., *Detection and punishment of malicious wireless stations in IEEE 802.11e EDCA network*. In Proc. of IEEE Sarnoff Symposium. 2010.

[15] Cagalj, M., Ganeriwal, S., Aad, I., and Hubaux, J.-P., *On Selfish Behavior in CSMA/CA Networks*. In Proc. of IEEE INFOCOM. 2005.

[16] Konorski, J. *A game-theoretic study of CSMA/CA under a backoff attack*. IEEE/ACM Transactions on Networking, 14:1167–1178, 2006.

[17] Gallo, P., Gringoli, F., and Tinnirello, I., *On the Flexibility of the IEEE 802.11 Technology: Challenges and Directions.* Future Network and MobileSummit 2011, to appear.

[18] Salvador, P., Gringoli, F., Mancuso, V., Serrano, P., Mannocci, A., Banchs, A., *VoIPiggy: Implementation and evaluation of a mechanism to boost voice capacity in 802.11 WLANs.* Submitted to INFOCOM 2012.

APPENDIX A: AGH Update on Flexible Architecture for Virtualizable wireless future Internet Access (D4.1.1)

This section contains an update on the FLAVIA architecture described in D4.1.1 [6]. It provides information on the extended Monitoring service, a new Misbehaviour Detection and Reaction service, and the extended Consistency Manager module. Figure 23 shows how the proposed AGH extensions are integrated in the FLAVIA architecture. Furthermore, Figure 24 shows how FLAVIA services interact with the proposed AGH extensions.

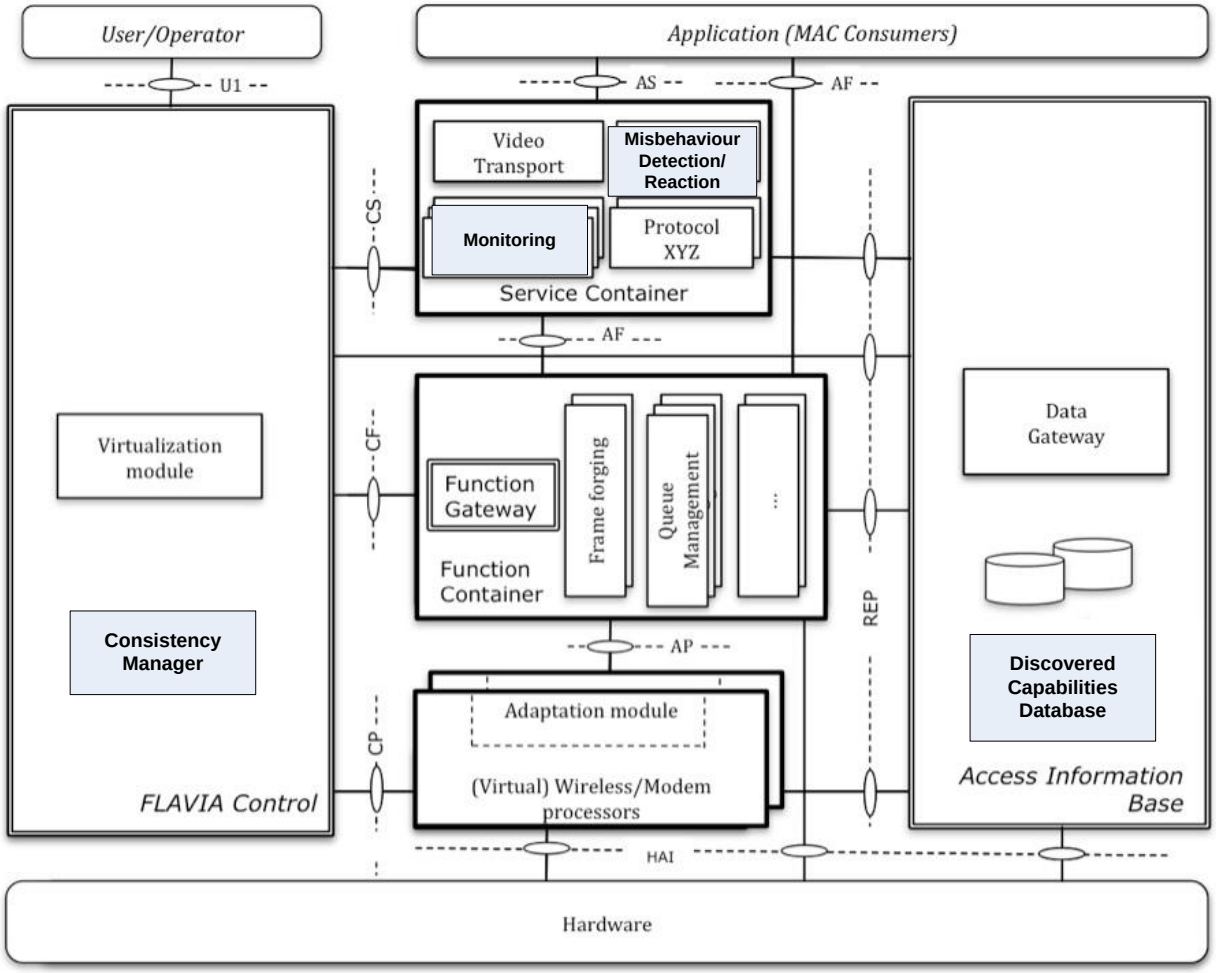

Figure 23: AGH modules integrated in the FLAVIA architecture.

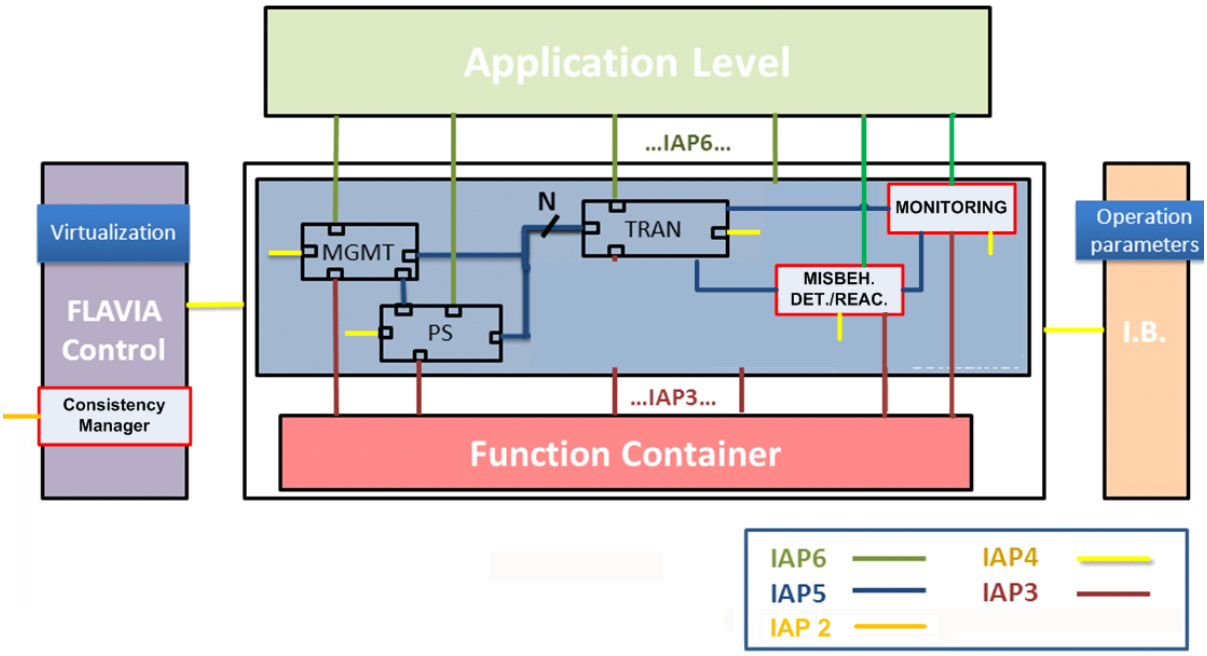

Figure 24: FLAVIA services interaction with AGH modules.

Table 8 summarizes this set of the extended FLAVIA services/modules with their corresponding acronym and a short description. Later on, each of these services/modules is characterized, together with their corresponding list of functions.

Service	**Short name**	**Description**
Monitoring	MONI	Extends the passive monitoring part of the Monitoring Service
Misbehaviour Detection and Reaction	MDR	Based on network measurements it detects misbehaving nodes and applies methods to encourage such nodes to cooperate
Module	**Short name**	**Description**
Consistency Manager	CM	Provides intra- and inter-node consistency

Table 8: Extended FLAVIA 802.11 services scheme

A.1 Monitoring

The FLAVIA monitoring system provides accurate and timely information regarding the status of the network. The Monitoring (MONI) service as defined in [6] analyses the

available radio spectrum and collects important information on the quality of each link. This allows choosing the best available channel for transmission. It also introduces a new scanning mechanism named SuperSense. The FLAVIA architecture supports both types of scanning: active and passive. The extended passive monitoring service described in this Annex supports two new functions: calculation of MAC parameters and capability discovery. Its macro-functions (*capability_discovery* and *mac_parameter_calculation*) are triggered every time a specific request arrives at the monitoring service. They are usually started at the beginning of the wireless card's operation and continued until the card is turned off. The PHY and MAC layer measurements are based on the reception of all types of IEEE 802.11 frames (data, management, and control). Promiscuous network monitoring on the channel used for regular data transmission is utilized to ensure a comprehensive view of the current wireless channel conditions.

Figure 25 provides a vision of the extended passive monitoring and the whole set of functions that are currently supported by the MONI service. The orange colour indicates the services originally proposed in D4.1.1 [6] without any modifications, the blue colour presents the extended passive monitoring service, and finally the green colour shows new services proposed in this Annex. The figure also presents a division according to the macro-functions supported by the MONI Service.

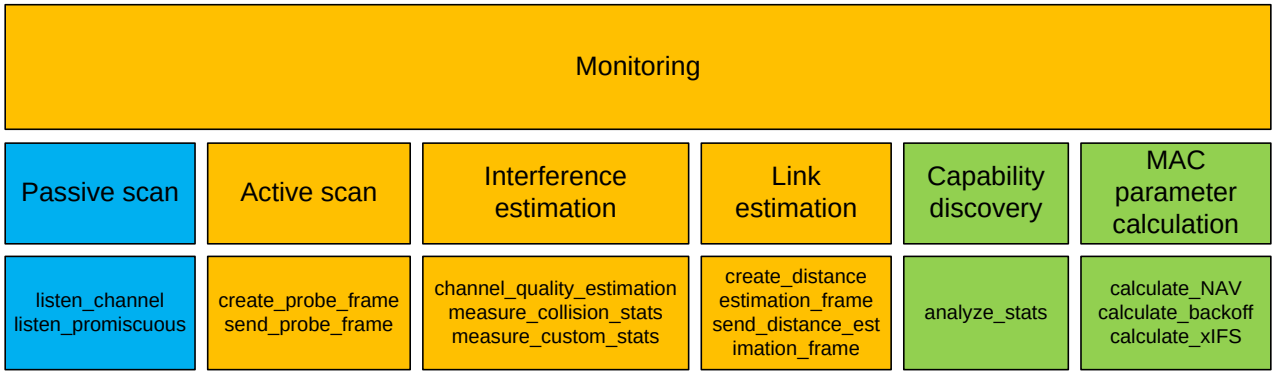

Figure 25: Monitoring service outline.

The *passive_scan* macro-function *passive_scan(frequency range, timeout, dwell time)* requires three parameters:

- A range of frequencies to scan on (it can be a spectrum width or a list of channels). The channel list could also include only one channel, and if the channel value is equal to 0 then the scanning procedure is activated on the currently used channel.
- A time interval within which to perform the scanning. If the timeout is equal to 0, and the *dwell time* is equal to 1 then the passive scanning runs until it is stopped. The stop procedure is called when the *passive_scan* function is run with *timeout*=0, *dwell time*=0.

- The dwell time spent on each channel (only used if *timeout* is different then 0). In all other cases the meaning of this parameter is described above.

Capability Discovery

The information obtained using promiscuous network monitoring is sent to the Information Base where it remains available for other FLAVIA modules in the Discovered Capabilities DB (DCDB) component. DCDB should be considered as an extension to the Data Collector component of the FLAVIA 802.11 framework architecture. The following information about neighbouring nodes and channel state can be obtained using the extended passive monitoring: supported rates, preamble type, state of the power saving and WEP security mode, country code, Signal to Noise Ratio (SNR), Frame/Bit Error Rate (F/BER), RTS_Threshold, Fragmentation_Threshold, beacon interval, operation mode, number of retransmissions, channel occupancy, and number of active nodes in the neighbourhood.

This macro-function is defined as follows:

capability_discovery(parameter list, interval)

MAC Parameter Calculation

The FLAVIA open architecture offers possibilities to misbehave. Therefore, the extended monitoring system should support the MAC parameter calculation service with the required measurements. These measurements are used to determine NAV, backoff, and IFS size independently for each received frame. This calculation is done separately for each station that is within the neighbourhood of the FLAVIA node. This task requires precise recognition of certain events (e.g., reception start, reception stop), which can be provided by the Wireless Processor. The obtained calculations are forwarded to the misbehaviour detection module, which is able, based on the received data, to evaluate the other IEEE 802.11 MAC parameters, in particular CWmin and CWmax. This allows the Misbehaviour Detection service to detect the incorrect configuration of stations.

This macro-function is defined as follows:

mac_parameter_calculation(parameter list, interval)

Summary of the Monitoring Service

Table 9 summarizes the set of macro-functions, functions, and commands corresponding to the extended passive monitoring service.

Macro-functions	passive_scan(frequency range, timeout, dwell time) capability_discovery(parameter list, interval) mac_parameter_calculation(parameter list, interval)
Functions	listen_promiscuous(promisc, on/off) analyze_stats(parameter list, interval) calculate_NAV() calculate_backoff() calculate_IFS() obtain_frame(frame) analyse_frame_parameters(frame) analyze_management_frame() analyze_control_frame() analyze_data_frame() update_MAC_capabilities() update_channel_parameters() update_database() send_MDR_parameters() trigger_MDR_module()

Table 9: Extended passive monitoring service summary

Pseudocode:
The pseudo-code for the macro-functions of the Monitoring Service is the following:

```
capability_discovery() {
  while(capability_discovery == on){
      obtain_frame()
      analyse_frame_parameters()
      case (frame type):
           MANAGEMENT: analyze_management_frame()
           CONTROL: analyze_control_frame()
           DATA: analyze_data_frame()
       update_MAC_capabilities()
      update_channel_parameters()
      if( update_time_out )
           update_database()
}
```


```
mac_parameter_calculation() {
  while(mac_parameter_calculation == on){
      obtain_frame()
      analyse_frame_parameters()
       calculate_NAV()
      calculate_backoff()
      calculate_IFS()
      update_MAC_parameters()
      if( update_time_out )
           send_MDR_parameters()
      if( channel_utilization_threshold )
           trigger_MDR_module()
      }

}
```

A.2 Misbehaviour Detection and Reaction

The goal of the misbehaviour detection and reaction (MDR) service is to first detect modifications of MAC layer parameters (e.g., DIFS, NAV) by analysing wireless measurement data and then to adapt the wireless card behaviour appropriately. By default, the service is activated when the network is saturated (i.e., the network utilization, as reported by the monitoring service, is above a defined threshold).

Misbehaviour Detection

The detection part of the MDR service is dependent on the monitoring service. Based on the measurements of parameters conducted by the extended passive monitoring service, it is able to evaluate IEEE 802.11 MAC parameters, in particular the CWmin and CWmax values set by other nodes in the network. The evaluation of most MAC parameters (e.g., DIFS, NAV, and TXOPLimit) is a straightforward comparison with the standard values. However, the correct setting of the CW values will be done with the use of any of the following three sub-services (methods): chi-square test, mean test, and entropy test. The number of employed CW detection methods can be extended. The methods have configurable parameters, which determine the number of false positives. The methods can also be configured to measure either actual or only consecutive backoff [12] as well as take into account all backoff values or only those for which the frames had their *retry* bit set to 0 [13].

Reaction to Misbehaviour

The reaction part of the MDR service will apply one of the following three methods to encourage correct behaviour: dropping acknowledgement (ACK) frames [14], selective frame jamming [15], and CW manipulation [16]. All these methods send appropriate configurations to the Wireless Processor to change its behaviour. The methods are applied when misbehaviour is detected and are suppressed when the misbehaviour ceases.

Summary of the MDR service

Table 10 summarizes the set of macro-functions, functions and commands corresponding to the MDR service.

Macro-functions	`detect_misbehavior()` `apply_reaction_method ()`

Functions	`obtain_measurements()`
	`estimate_CW(MAC address)`
	`compare_MAC_parameters_with_std(MAC address)`
	`send_configuration()`

Table 10: Misbehaviour Detection and Reaction service summary

Pseudocode:
The pseudo-code for the macro-functions of the MDR service is the following:

```
detect_misbehavior() {
      obtain_ measurements()
      for each MAC_address {
            estimate_CW(MAC)
            is_misbehaving[MAC] = compare_MAC_parameters_with_std(MAC)
      }
}

apply_reaction_method() {
      for each MAC_address {
            if is_misbehaving[MAC] {
                  send_configuration()
            }
      }
}
```

A.3 Consistency Manager

Module Overview

As described in D2.2.1 [5], the FLAVIA Consistency Manager (CM) is designed as a module responsible for intra- and inter-node configuration and parameter detection, as well as analysis and resolving potential or existing configuration inconsistencies. In order to provide such functionality the architecture of the CM module is divided into two basic components: Intra-CM and Inter-CM. In this section, CM functionalities described in D2.2.1 are listed together with several extended functionalities. Macro-functions and functions used by the CM module are also given.

The main responsibility of the Intra-CM component (which operates locally on each node) is to eliminate the possibility of mutually exclusive settings being applied by different services/software modules to the same system parameter. The consistency of information stored in the Information Base (IB) is guaranteed by the Data Gateway (cf. D4.1.1 [6], Figure 23), which defines solutions for both single and multiple write protection (cf. D2.2.1 [5]). The most typical use-case of the Inter-CM application is the Tx Power Control for a wireless interface against the physical characteristic of the end-level power amplifier, as well as local transmission power EIRP regulations, taking into account the antenna gain introduced into a total radiated power. If a requested parameter change surpasses the acceptable value, the CM will report back failure notification with an error code.

The CM module also introduces an additional level of modules/services differentiation within the overall FLAVIA node architecture. It adds the following configurable features:

- Allow or deny a specified module/service to request a specific parameter value change.
- Assign priorities to modules to perform specific actions or decide which request is allowed to take precedence over another (this feature can be applied to both Intra-CM and Inter-CM).

The Inter-CM component takes advantage of neighbouring nodes' capabilities and the parameter discovery function delivered by the extended passive Monitoring service and its Capability Discovery function. Moreover, a dedicated signalling protocol supports the CM in obtaining remote node configuration parameters or a general configuration summary. The Inter-CM operability mode requires a Designated Node (DN) to be nominated among all nodes sharing the same network resources — which typically is a frequency channel a group of nodes is operating on. The natural candidate to be nominated as a DN in an infrastructure operational mode is the Access Point (AP). In an ad-hoc mode a Cell-ID indicates the DN, while Backup Designated Node (BDN), in case of a DN failure, will be the node with the highest MAC. The DN is

responsible for overall group performance analysis, is capable of running group performance optimization algorithms, taking final decisions and generating configuration change indications. The process of the DN election within an ad-hoc mode relies on the Monitoring Service and its Capabilities Discovery function.

Integration of Consistency Manager within the FLAVIA Architecture

As described in D4.1.1 [6], the CM module resides within the FLAVIA Control and Management subsystem (Figure 26). Taking into account the presented overview and the main goals of the CM module it can be derived that it should act as a proxy between services/modules requesting parameters change and the functions implementing them.

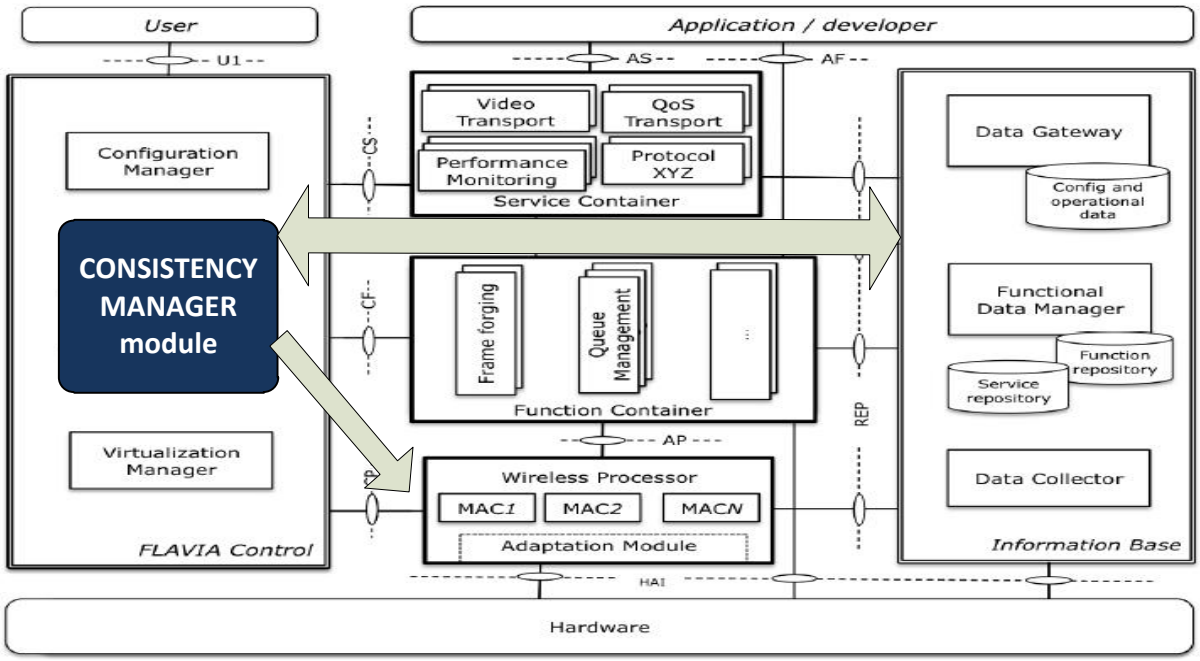

Figure 26: Consistency Manager within the FLAVIA global architecture.

Consistency Manger functionalities

The general functionalities of the CM are the following (this list is a combination of functionalities described in D2.2.1 [5] and extended functionalities proposed in this Appendix):

- Analyse all the requests which are to be applied to wireless interfaces.
- Generate the local set of capabilities with respect to the data stored within

the IB.

- Verify the consistency conditions stored in the IB.
- Confirm the suitability of the requested parameter value within a list of supported parameters and their acceptable value ranges and regulatory limits.
- Confront the requested parameter value with the already configured value.
- Report failed or conflicted requests extended with an error code.
- Organize the parallel access for modules/services to configure system parameters.
- Mutual exchange of consistency conditions.
- Verify the service logic consistency.
- Resolve the conflicts.

As described in D2.2.1 the Intra-CM can take the following actions in case of consistency violation: force the service generation violation to abort its request, return an error code or automatically enforce a correction based on the request.

In the Inter-CM scenario, the DN takes the final decision in order to resolve a group inconsistency. The only use case which requires sending back a response is the situation when a parameter change request is coming from one group of nodes to another one. Then an acknowledgement or error code is generated by the CM. An illustration for such a scenario is depicted in Figure 27, where FLAVIA Node A (which is a DN for Group 1) is requesting a parameter change (e.g., frequency change to a non-overlapping channel) which affects Group 2. FLAVIA Node B, since it belongs to both groups, is able to convey such a request to the FLAVIA Node C (which is a DN for Group2). As a result, Node C runs the Intra-CM consistency check function and decides to apply or reject the requested configuration change.

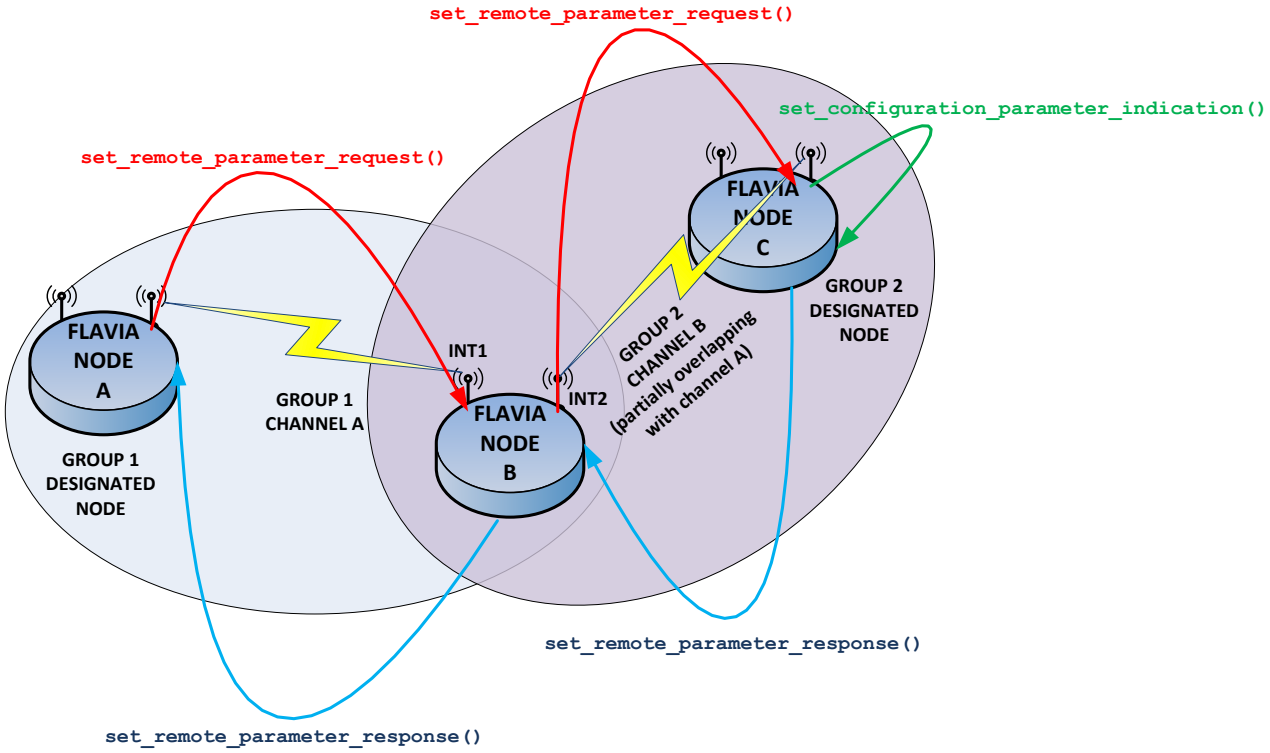

Figure 27: Inter-CM use case.

The following primitives are designed to provide the described CM functionalities:

- *get_configuration_parameter_request()/response()* — allows to receive information from the local IB,
- *get_consistency_conditions_request()/response()* — obtains the summary of consistency conditions stored in the local IB,
- *get_service_logic_request()/response()* — obtains the summary of a requested service logic setup,
- *execute_configuration_change_indication()* — allows to locally trigger a service or function capable of executing a configuration parameter change,
- *get_remote_parameter_request()/response()* — obtains the value of a requested remote parameter, which is used by a remote node, together with its acceptable range,
- *get_remote_summary_request()/response()* — obtains a brief summary of a remote node's configuration: number of interfaces, configured channels, Tx Power levels, modulation schemes used, etc.,
- *set_configuration_parameter_indication()* — allows a DN to indicate a remote node parameter change to a requested value,

- *set_remote_parameter_request()/response()* — allows to invoke the *set_configuration_parameter_indication()* function on a DN, if such an indication requests a parameter change affecting a group of nodes, sharing different resource (i.e., operating on a different channel, other ad-hoc node group, etc.), as depicted in Figure 27. The final decision is taken by a proper DN, which is also responsible to respond with an acknowledgement or error code.

Table 11 summarizes the set of macro-functions and functions corresponding to the CM operation modes: intra- and inter-node. Since the CM relies on commands delivered by the PHY resource management service, the table itself does not contain commands.

Macro-functions	
Intra-CM	`check_intra_consistency()`
Inter-CM	`check_inter_consistency()`
Functions	

Intra-CM	`get_configuration_parameter()`
	`get_consistency_conditions()`
	`get_service_logic()`
	`execute_configuration_change()`

Inter-CM	`get_remote_parameter()`
	`get_remote_summary()`
	`set_configuration_parameter()`
	`set_remote_parameter()`

Table 11: Consistency Manager Summary

Pseudocode:
The pseudo-code for the macro-functions of the CM is the following:

```
check_intra_consistency(){
      switch(consistency_scope){
            case(parameter) : get_configuration_parameter()
            case(conditions) : get_consistency_conditions()
            case(service) : get_service_logic()

      }
      build(consistency_context)
      if(intra_consistency_analysis == positive){
            execute_configuration_change()
```


```
      }else{
            error_code_response()
      }
}

check_inter_consistency(){
      switch(consistency_scope){
            case(parameter) : get_remote_parameter()
            case(summary) : get_remote_summary()
      }
      build(consistency_context)
      if (inter_consistency_analysis == positive){
            if(relay_scenario)
                  set_remote_parameter()
            set_configuration_parameter()
      }else{
            error_code_response()
      }
}
```

Interfaces

Basically CM relies on system state data, registered events or service configuration parameters collected and exposed by the IB via the REP Interface and is able to execute a function from the collection of functions delivered by the PHY resource management service. The inter-node operability and consistency verification requires an additional signalling interface to be introduced. This interface is responsible for ensuring inter-node communication, conveying data related to remote node configuration setup, as well as triggering messages, indicating the specific parameter values to be remotely changed. Such a signalling protocol is to be used in cases when more detailed information is required and capabilities discovered and reported by the Passive Monitoring service to IB is not sufficient for CM to take a required optimizing action in order to resolve inter-nodes configuration inconsistencies. A detailed scheme of the CM architecture and its interfaces is depicted in Figure 28.

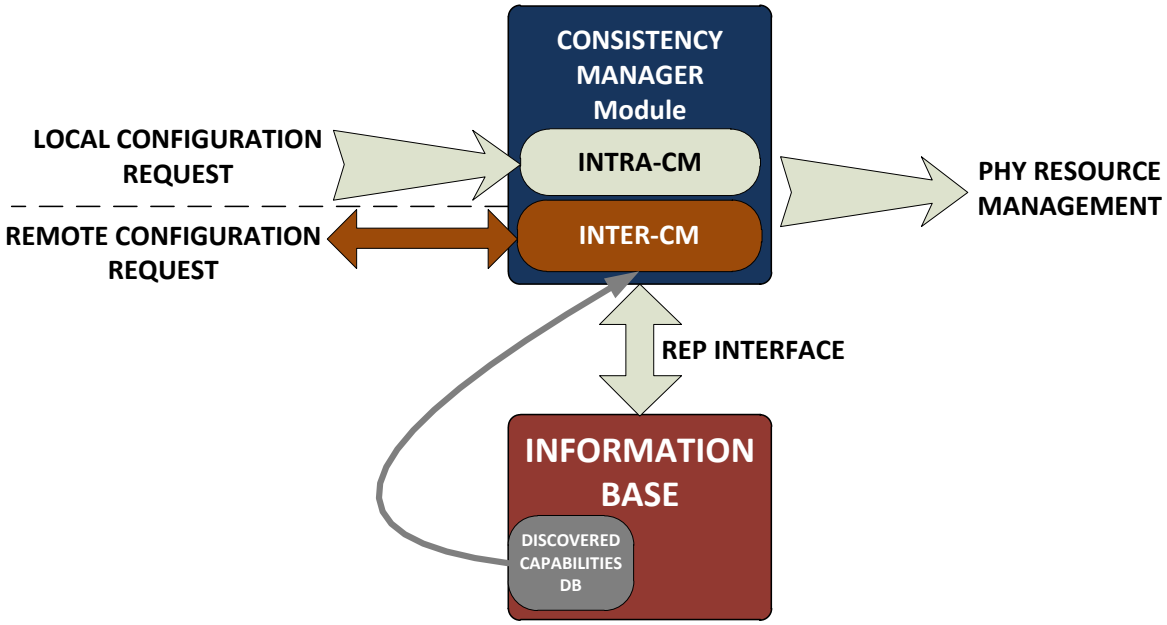

Figure 28: Consistency Manager interfaces.

APPENDIX B: Pseudo-code

This section contains a set of tables describing the source and pseudo-code for: i) the mac80211 framework, ii) the new framework mac80211++ and iii) the advanced monitoring module.

B.1 mac80211

In what follows we summarize the main declarations of the mac80211 framework:

- The set of functions and corresponding declaration files for the mac80211 framework is given below.

Method	File
ieee80211_unregister_hw	main.c
ieee80211_free_hw	main.c
ieee80211_chswitch_done	mlme.c
ieee80211_enable_dyn_ps	mlme.c
ieee80211_disable_dyn_ps	mlme.c
ieee80211_ap_probereq_get	mlme.c
ieee80211_beacon_loss	mlme.c
ieee80211_connection_loss	mlme.c
ieee80211_cqm_rssi_notify	mlme.c
ieee80211_ready_on_channel	offchannel.c
ieee80211_remain_on_channel_expired	offchannel.c
ieee80211_rate_control_register	rate.c
ieee80211_rate_control_unregister	rate.c
rate_control_send_low	rate.c
ieee80211_sta_ps_transition	rx.c
ieee80211_rx	rx.c
ieee80211_rx_irqsafe	rx.c
ieee80211_scan_completed	scan.c
ieee80211_sched_scan_results	scan.c
ieee80211_sched_scan_stopped	scan.c
ieee80211_find_sta_by_ifaddr	sta_info.c
ieee80211_find_sta	sta_info.c
ieee80211_sta_block_awake	sta_info.c
ieee80211_sta_set_tim	sta_info.c
ieee80211_tx_status_irqsafe	status.c
ieee80211_tx_status	status.c
ieee80211_report_low_ack	status.c

ieee80211_get_tkip_key	tkip.c
ieee80211_beacon_get_tim	tx.c
ieee80211_pspoll_get	tx.c
ieee80211_nullfunc_get	tx.c
ieee80211_probereq_get	tx.c
ieee80211_rts_get	tx.c
ieee80211_ctstoself_get	tx.c
ieee80211_get_buffered_bc	tx.c
wiphy_to_ieee80211_hw	util.c
ieee80211_generic_frame_duration	util.c
ieee80211_rts_duration	util.c
ieee80211_ctstoself_duration	util.c
ieee80211_wake_queue	util.c
ieee80211_stop_queue	util.c
ieee80211_stop_queues	util.c
ieee80211_queue_stopped	util.c
ieee80211_wake_queues	util.c
ieee80211_iterate_active_interfaces	util.c
ieee80211_iterate_active_interfaces_atomic	util.c
ieee80211_queue_work	util.c
ieee80211_queue_delayed_work	util.c

Table 12: Set of functions and declaration files of the mac80211 framework

- Set of registered operations in the */linux/netdevice.h* file.

```
.ndo_open               = ieee80211_open,
.ndo_stop               = ieee80211_stop,
.ndo_uninit             = ieee80211_teardown_sdata,
.ndo_start_xmit         = ieee80211_subif_start_xmit,
.ndo_set_multicast_list = ieee80211_set_multicast_list,
.ndo_change_mtu         = ieee80211_change_mtu,
.ndo_set_mac_address    = ieee80211_change_mac,
.ndo_select_queue       = ieee80211_netdev_select_queue.
```

Table 13: Registered operations in /linux/netdevice.h

- The set of functions exported by the *net_device* structure is listed below (The second column specifies the files belonging to the mac80211 framework that call those functions.)

Method	File
netif_addr_lock_bh	iface.c
netif_addr_unlock_bh	iface.c
netif_carrier_off	iface.c, mlme.c
netif_carrier_on	iface.c, mlme.c
netif_napi_add	main.c
netif_receive_skb	rx.c
netif_rx	status.c
netif_rx_ni	cfg.c
netif_stop_subqueue	util.c
netif_tx_start_all_queues	iface.c, mlme.c
netif_tx_stop_all_queues	offchannel.c
netif_tx_stop_all_queues	iface.c, mlme.c
netif_tx_wake_all_queues	offchannel.c, mlme.c
netif_wake_subqueue	tx.c, util.c

***Table 14: Set of functions exported by the* net_device structure**

- The mac80211 module registers the following callbacks within the *cfg80211_ops* structure:

```
.add_virtual_intf          = ieee80211_add_iface,
.del_virtual_intf          = ieee80211_del_iface,
.change_virtual_intf       = ieee80211_change_iface,
.add_key                   = ieee80211_add_key,
.del_key                   = ieee80211_del_key,
.get_key                   = ieee80211_get_key,
.set_default_key           = ieee80211_config_default_key,
.set_default_mgmt_key      = ieee80211_config_default_mgmt_key,
.add_beacon                = ieee80211_add_beacon,
.set_beacon                = ieee80211_set_beacon,
.del_beacon                = ieee80211_del_beacon,
.add_station               = ieee80211_add_station,
.del_station               = ieee80211_del_station,
.change_station            = ieee80211_change_station,
.get_station               = ieee80211_get_station,
```


```
	.dump_station		= ieee80211_dump_station,
	.dump_survey		= ieee80211_dump_survey,
	.add_mpath		= ieee80211_add_mpath,
	.del_mpath		= ieee80211_del_mpath,
	.change_mpath		= ieee80211_change_mpath,
	.get_mpath		= ieee80211_get_mpath,
	.dump_mpath		= ieee80211_dump_mpath,
	.update_mesh_config	= ieee80211_update_mesh_config,
	.get_mesh_config	= ieee80211_get_mesh_config,
	.join_mesh		= ieee80211_join_mesh,
	.leave_mesh		= ieee80211_leave_mesh,
	.change_bss		= ieee80211_change_bss,
	.set_txq_params		= ieee80211_set_txq_params,
	.set_channel		= ieee80211_set_channel,
	.suspend		= ieee80211_suspend,
	.resume			= ieee80211_resume,
	.scan			= ieee80211_scan,
	.sched_scan_start	= ieee80211_sched_scan_start,
	.sched_scan_stop	= ieee80211_sched_scan_stop,
	.auth			= ieee80211_auth,
	.assoc			= ieee80211_assoc,
	.deauth			= ieee80211_deauth,
	.disassoc		= ieee80211_disassoc,
	.join_ibss		= ieee80211_join_ibss,
	.leave_ibss		= ieee80211_leave_ibss,
	.set_wiphy_params	= ieee80211_set_wiphy_params,
	.set_tx_power		= ieee80211_set_tx_power,
	.get_tx_power		= ieee80211_get_tx_power,
	.set_wds_peer		= ieee80211_set_wds_peer,
	.rfkill_poll		= ieee80211_rfkill_poll,
	.set_power_mgmt		= ieee80211_set_power_mgmt,
	.set_bitrate_mask	= ieee80211_set_bitrate_mask,
```


```
    .remain_on_channel          = ieee80211_remain_on_channel,
    .cancel_remain_on_channel   = ieee80211_cancel_remain_on_channel,
    .mgmt_tx                    = ieee80211_mgmt_tx,
    .mgmt_tx_cancel_wait        = ieee80211_mgmt_tx_cancel_wait,
    .set_cqm_rssi_config        = ieee80211_set_cqm_rssi_config,
    .mgmt_frame_register        = ieee80211_mgmt_frame_register,
    .set_antenna                = ieee80211_set_antenna,
    .get_antenna                = ieee80211_get_antenna,
    .set_ringparam              = ieee80211_set_ringparam,
    .get_ringparam              = ieee80211_get_ringparam
```

Table 15: Callbacks within the* cfg80211_ops *structure

- Next we give the set of functions contained in the files located in the directory *net/wireless/*.

Method	File
cfg80211_send_rx_auth	mlme.c
cfg80211_send_rx_assoc	mlme.c
__cfg80211_send_deauth	mlme.c
cfg80211_send_deauth	mlme.c
__cfg80211_send_disassoc	mlme.c
cfg80211_send_disassoc	mlme.c
cfg80211_send_unprot_deauth	mlme.c
cfg80211_send_unprot_disassoc	mlme.c
__cfg80211_auth_canceled	mlme.c
cfg80211_send_auth_timeout	mlme.c
cfg80211_send_assoc_timeout	mlme.c
cfg80211_michael_mic_failure	mlme.c
cfg80211_ready_on_channel	mlme.c
cfg80211_remain_on_channel_expired	mlme.c
cfg80211_new_sta	mlme.c
cfg80211_del_sta	mlme.c
cfg80211_rx_mgmt	mlme.c
cfg80211_mgmt_tx_status	mlme.c
cfg80211_cqm_rssi_notify	mlme.c
cfg80211_cqm_pktloss_notify	mlme.c
cfg80211_classify8021d	util.c

cfg80211_connect_result	sme.c
cfg80211_roamed	sme.c
cfg80211_disconnected	sme.c
cfg80211_scan_done	scan.c
cfg80211_sched_scan_results	scan.c
cfg80211_sched_scan_stopped	scan.c
cfg80211_find_ie	scan.c
cfg80211_get_bss	scan.c
cfg80211_get_mesh	scan.c
cfg80211_inform_bss	scan.c
cfg80211_inform_bss_frame	scan.c
cfg80211_put_bss	scan.c
cfg80211_unlink_bss	scan.c
cfg80211_wext_siwscan	scan.c
cfg80211_wext_giwscan	scan.c
cfg80211_notify_new_peer_candidate	mesh.c
cfg80211_testmode_alloc_reply_skb	nl80211.c
cfg80211_testmode_reply	nl80211.c
cfg80211_testmode_alloc_event_skb	nl80211.c
cfg80211_testmode_event	nl80211.c
cfg80211_ibss_joined	ibss.c

Table 16: Set of functions contained in the directory net/wireless/

B.2 mac80211++

In this section we present the preliminary source code of the prototype of the Service Scheduler and the Function Handler. First, Table 17 summarizes the source code of the main data structures and functions of the Service Scheduler described Section 3.3.1.

Source code for the Service Scheduler

```
struct flavia_ss_t {
  struct work_struct  flavia_work;
  unsigned long       flavia_data;
  void                (*flavia_service_hook)(unsigned long data);
  unsigned int        flavia_usec;
  struct timer_list   flavia_timer;
  struct list_head    flavia_ss_list;
};
```


```
struct flavia_ss_t *
flavia_register_service(void *service_hook,
                        unsigned int usec,
                        unsigned long service_data)
{
  struct flavia_ss_t *srv = alloc_mem(sizeof(struct flavia_ss_t));
  if (srv != NULL) {
    // 1. Initialize the work that will be executed later
    //    and the auxiliary information
    init_work_for_workqueue(&srv->flavia_work, flavia_srv_container)
    srv ->flavia_data = service_data;
    srv ->flavia_service_hook = service_hook;
    srv ->flavia_usec = usec;
    setup_timer(flavia_ss_timer_function, srv);
    // 2. Add the new scheduled service to the list
    add_to_service_list(&srv->flavia_ss_list, &flavia_ss.flavia_ss_list);
    // 3. Start the countdown of the timer that will schedule the work
    start_timer(srv->flavia_usec);
    return srv;
  }
  return NULL;
}
```

```
struct flavia_ss_t *
flavia_register_service_tsf_sync(void *service_hook,
                                 unsigned int usec,
                                 unsigned long service_data)
{
  struct flavia_ss_t *srv = alloc_mem(sizeof(struct flavia_ss_t));

  if (srv != NULL) {
    // 1. Initialize the work that will be executed later
    //    and the auxiliary information
    init_work_for_workqueue(&srv->flavia_work,
flavia_srv_container_tsf_sync)
    srv ->flavia_data = service_data;
    srv ->flavia_service_hook = service_hook;
    srv ->flavia_usec = usec;
    setup_timer(flavia_ss_timer_function, srv);
    // 2. Add the new scheduled service to the list
    add_to_service_list(&srv->flavia_ss_list, &flavia_ss.flavia_ss_list);
```


```
    // 3. Start the countdown of the timer which will schedule the work
    start_timer(srv->flavia_usec);
    return srv;
  }
  return NULL;
}
```

```
void
flavia_ss_timer_function(unsigned long data)
{
  struct flavia_ss_t *srv = (struct flavia_ss_t *) data;
  int ret;
  ret = queue_work_on_workqueue(flavia_ss_wq, srv->flavia_work);
  if (!ret) {
    printk("[FLAVIA] Failed to queue the work after the timer
expired.\n");
  }
}
```

```
void
flavia_srv_container(struct work_struct *work)
{
  struct flavia_ss_t *srv = (struct flavia_ss_t *) work;

  // 1. Call the function implementing the FLAVIA Service
  srv->flavia_service_hook(srv->flavia_data);
  // 2. Reschedule the timer that, in turn, recalls this function through
a work
  start_timer(srv->flavia_usec);

  return;
}
```

```
void
flavia_srv_container_tsf_sync(struct work_struct *work)
{
  struct flavia_ss_t *srv = (struct flavia_ss_t *) work;
  u64 tsf;
  unsigned int next_usec;

  // 1. Call the function implementing the FLAVIA Service
  srv->flavia_service_hook(srv->flavia_data);
```


```
  // 2. Compute the expiration time
  tsf = get_tsf_from_driver();
  r_u32 = module_of(tsf, srv->flavia_usec);
  next_usec = srv->flavia_usec - r_u32;
  // 3. Reschedule the timer that, in turn, recalls this function
  start_timer(srv->flavia_usec);

  return;
}
```

```
void
flavia_remove_service(struct flavia_ss_t *srv)
{
  stop_and_delete_timer(&srv->flavia_timer);
  cancel_work_from_worqueue(&srv->flavia_work);
  delete_from_service_list(&srv->flavia_ss_list,
&flavia_ss.flavia_ss_list);
}
```

Table 17: Source code for the Service Scheduler

Second, Table 18 summarizes the preliminary source code of the main data structures and functions of the Function Handler described in Section 3.3.2.

Source code for the Function Handler

```
struct flavia_function_ops {
  struct module     *module;
  char              *function_name;
  void              (*flavia_function_hook)(unsigned long data);
  unsigned long     function_data;
  struct list_head  flavia_function_list;
};

struct flavia_hook_ops {
  char             *hook_name;
  struct list_head  flavia_function_list;
  struct mutex      flavia_func_list_mtx;
  struct list_head  flavia_hook_list;
};
```

```
struct flavia_function_ops *
```


```
flavia_register_function(void *function_hook,
                         char *func_name, char *hook_name,
                         unsigned long func_data)
{
  struct flavia_hook_ops *hook_item = NULL;
  struct flavia_function_ops *func_item =
                            alloc_mem(sizeof(struct flavia_function_ops));

  if (func_item != NULL) {
    // 1. Initialize the function hook that will be executed each time
          the corresponding mac80211 hook is invoked
    func_item->function_name = func_name;
    func_item->flavia_function_hook = function_hook;
    func_item->function_data = func_data;
    // 2. Look for the hook on which the function will be registered
    hook_item = flavia_find_hook_ops(hook_name);
    if (hook_item != NULL) {
      // 2.1 Register the new function on the corresponding hook
      //     Add the new item to the list of functions executed on the
same hook
      add_to_func_list(func_item, hook_item);
    } else {
      // No function registered for this hook till now...
      // 2.1 Create and register a new hook
      hook_item = alloc_mem(sizeof(struct flavia_hook_ops));
      hook_item->hook_name = kstrdup(hook_name, GFP_KERNEL);
      INIT_LIST_HEAD(hook_item->flavia_function_list);
      add_to_hook_list(hook_item, &flavia_hooks.flavia_hook_list);
      //  2.2  Register  the  function  on  the  new  hook  (new  list  of
functions)
      add_to_func_list(func_item, hook_item);
    }
    return func_item;
  }
  return NULL;
}
```

```
void
flavia_function_hook_container(char     *hook_name,     unsigned     long
mac80211_data)
{
  struct flavia_hook_ops *hook_item;

  list_for_each_entry(flavia_hooks.flavia_hook_list) {
```


```
    hook_item = next_list_entry(flavia_hooks.flavia_hook_list);
    if (hook_item->hook_name == hook_name) {
      flavia_exe_hook_functions(hook_item, mac80211_data);
    }
  }
  return;
}
```

```
void
flavia_exe_hook_functions(struct flavia_hook_ops *hook_item, unsigned
long mac80211_data)
{
  struct flavia_function_ops *func_item;
  // 1. Loop over all functions registered on this hook
  list_for_each_entry(hook_item->flavia_function_list) {
    func_item = next_list_entry(hook_item->flavia_function_list);
    func_item->flavia_function_hook(mac80211_data);
  }
  return;
}
```

```
void
flavia_remove_function(char *func_name, char *hook_name)
{
  struct flavia_function_ops *flavia_func_item =
                          alloc_mem(sizeof(struct flavia_function_ops));

  // 1. Search the function registered on the hook
  flavia_func_item = flavia_find_function_ops(func_name, hook_name);
  if (flavia_func_item != NULL) {
    // 1.1 Remove the function from the corresponding hook
    delete_from_func_list(flavia_func_item->flavia_function_list);
  }
  return;
}
```

Table 18: Source code for the Function Handler

B.3 Advanced Monitoring

This section presents the pseudo-code for the Advanced Monitoring module described in Section 4.4.

Pseudocode for the Advanced Monitoring module

```
//rx.c - definition of the hook for the monitoring function for the
uplink path

void ieee80211_rx(struct ieee80211_hw *hw, struct sk_buff *skb)
{
......
     void  (*p_rx_measure)(  struct  ieee80211_hw  *hw,  struct  sk_buff
*skb);
......
     if(p_rx_measure != NULL )
          (*p_rx_measure)(hw, skb);
......
}
```

```
//tx.c - definition of the hook for the monitoring function for the
downlink //path

static  bool  __ieee80211_tx(struct  ieee80211_local  *local,  struct
sk_buff **skbp, struct sta_info *sta, bool txpending)
{
......
    void (*p_rx_measure)( struct ieee80211_local *local, struct sk_buff
*skb);
......
    if(p_tx_measure != NULL )
         (*p_tx_measure)();
......
}
```

```
//flavia_monitor.c – definition of FAM module functions

flavia_fam_init()
{
    init_netlink_communication();
    create_sending_thread();
    data_structures_init();
    enable_rx_hook();
    enable_tx_hook();
```


```
}

flavia_fam_exit()
{
    disable_rx_hook;
    disable_tx_hook;
    free_resources;
}

flavia_rx_measure(){
     analyze_farme(){
       case(frame type):
       control: analyze_control_frame();
       management: analyze_management_farme();
       data: analyze_data_frame();
     }
     calculate_NAV();
     calculate_backoff()
     calculate_IFS()
     update_mac_parameters();
     update_channel_parameters();
}

flavia_tx_measure(){
     analyze_farme(){
       case(frame type):
       control: analyze_control_frame();
       management: analyze_management_farme();
       data: analyze_data_frame();
     }
     calculate_NAV();
     calculate_backoff()
     calculate_IFS()
     update_mac_parameters();
     update_channel_parameters();
}

receive_command_handler()
{
    analyze_command();
    configure_sending_thread();
}

sending_thread_handler()
```


```
{
      init_wait_queue();
      format_message();
      send_message();
      clear_data_structures();
}
```

Table 19: Pseudo-code for the Advanced Monitoring module