Specific Targeted Research Project

FLAVIA
FLexible Architecture for Virtualizable wireless future Internet Access

Deliverable Report

D 2.2.2 Revision of Architecture Specification

Deliverable title	Architecture Specification
Version	1.0
Due date of deliverable	M20
Actual submission date of the deliverable	13/05/2011
Start date of project	01/07/2010
Duration of the project	36 months
Work Package	WP2
Task	Task 2.2
Leader for this deliverable	CNIT
Other contributing partners	ALL
Authors	Ilenia Tinnirello, Pierluigi Gallo, Pierpaolo Loreti, Domenico Garlisi, Claudio Pisa (CNIT), Jesus Alonso (ALVARION), Stefano Paris, Antonio Capone (MOBI), Artem Krasilov, Evgeny Khorov, Alexander Safonov, Ivan Pustogarov (IITP), Xavier Perez Costa, Andreas Maeder, Peter Rost (NEC), Pablo Serrano, Vincenzo Mancuso (IMDEA), Yan Grunenberger, Eduard Gomà (TID), Lior Uziel, Guillaume Vivier (SEQUANS), Omer Gurewitz, Erez Biton (BGU), David Malone, Paul Patras (NUIM), Marek Natkaniec, Krzysztof Łoziak, Szymon Szott (AGH).
Deliverable reviewer	Giuseppe Bianchi, Pablo Serrano, Xavier Perez Costa
Deliverable abstract	This report describes the architectural solutions envisioned for the FLAVIA system as a result of two design iterations. The first iteration design has been carried out by trying to provide a conceptual model that defines the structure and the behavior of a network framework able to support the FLAVIA goals. Most of the system features and functionalities have been identified during this stage. We summarize these goals into three different

	aspects: i) modularity, in terms of definition of different services on the basis of a hierarchy of reusable functions; ii) flexibility, in terms of dynamic loading, updates and interactions of customized services dealing with MAC layer sub-tasks; iii) virtualization, in terms of launching of parallel independent services and functions accessing the same system resources. The second iteration has been dedicated to the refinement of some specific architecture components on the basis of the feedback provided during the early prototyping activities. The main extensions involved the wireless MAC processor, a key component of the FLAVIA system, that has been revised for supporting code switching, scheduling access schemes, and multi-stack full configuration.
Keywords	Architecture, specification, design, flexibility, modularity, virtualization, wireless processor

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)	

TABLE OF CONTENT

Executive summary

This report describes the architecture of the FLAVIA system as a result of two design iterations. Rather than providing an update on D221 (which reports on the initial architecture design), the report has been organized in order to give a complete and self-contained description of the final design.

The design has been carried out by trying to provide a conceptual model that defines the **structure** and the **behavior** of a network framework able to support: i) *modularity*, in terms of definition of different services on the basis of a hierarchy of reusable functions; ii) *flexibility*, in terms of dynamic loading, updates and interactions of customized services dealing with MAC layer sub-tasks; and iii) virtualization, in terms of launching of parallel independent services and functions accessing the same system resources.

Starting from the basic requirements which are common to both the contention-based and scheduled-based systems, we identified three programmability levels to be included in the FLAVIA architecture: scheduling actions on the hardware and physical layer, programming service/function logic, and specifying frames and system state parameters. The architecture envisioned for satisfying these complex programmability requirements is based on:

- *A wireless processor*, able to handle hardware events and execute medium access programs in terms of loadable Finite State Machines;
- *Function/service containers*, for implementing the system behavior by instantiating and composing different MAC-layer functionalities;
- *A control subsystem*, for defining and controlling the system behavior, by verifying the consistency of the running services and by solving potential conflicts arisen by concurrent services accessing the same system resources.

After an introduction describing the design process and the most significant extensions included in this document, we describe the architecture components (section 2 and section 3), and we introduce the functional architecture envisioned for dealing with: the MAC-layer decomposition and extensible interfaces (section 4, describing the architecture support for modularity), the control sub-system (section 5, describing the architecture support for flexibility), and the system virtualization (section 6, describing the architecture support for virtualization).

1 Introduction to the FLAVIA Architecture

The FLAVIA architecture has been defined by following a spiral model with **two design iterations**. The first iteration design has been carried out by trying to provide a conceptual model that defines the structure and the behavior of a network framework able to support the FLAVIA goals. Most of the system features and functionalities have been identified during this stage. We summarize these goals into three different aspects:

- *modularity*: the architecture provides a hierarchy of reusable functions on top of which different services (with different complexity levels) can be built.
- *flexibility*: the architecture provides interfaces for enabling new services, exporting customized services, composing different MAC layer sub-tasks.
- *virtualization*: the architecture allows the launch of parallel independent services and functions accessing the same system resources.

More specifically, to support modularity, we identified some "confined parts" of the MAC-layer, which would need to be updated according to the specific networking, application and PHY contexts. Each of these confined parts has been mapped into a MAC-layer sub-module, categorized as service, macro-function or function, according to its specific complexity. To support flexibility, we envisioned the need of discovering hardware platform capabilities, and negotiating these capabilities with the peer entities of the network; we also proposed some solutions for revealing and preventing conflicts arisen by the independent configuration of modules working on the same system parameters. Finally, virtualization of the system resources enables multi-thread actions of independent protocols on the same hardware platform (which has to appear as multiple independent platforms) and implies two crucial problems: controlling the partition of the hardware access and providing isolation between multiple virtual entities.

The second iteration has been dedicated to the refinement of some specific architecture components on the basis of the feedback provided during the early prototyping activities. Since most of the implementation activities were focused on the functional sub-system, the main extensions involved the wireless MAC processor (that has been revised for supporting code switching and scheduling access schemes), the virtualization monitor, and the functional containers. Some refinements have also been dedicated to the capability negotiation and consistency management solutions.

2 High-Level View of the FLAVIA Architecture

In this section we present a high level view of the FLAVIA architecture, by enlightening the main enablers and components of the FLAVIA vision. Our approach is based on the identification of a set of important MAC layer sub-tasks as elementary programmable units, which extend the concept of layer 2 services. These services can be, in turns, decomposed into simpler sub-routines and functions, that can be layered according to the abstraction level of the system resources and parameters they work on. On the basis of this functional analysis, we derive a new architecture for programmable wireless interfaces, in which the traditional MAC/PHY layers are replaced by: i) a platform layer, which exposes fixed primitives for managing frame transmissions and hardware events; ii) three layers of functionalities with different complexity (state machines, functions, and services), exposing programmable interfaces.

2.1 MAC Layer Decomposition

Figure 1 shows the envisioned service-oriented architecture of a wireless programmable stack, which includes the following core services:

- *Transport*: It provides the operations for queuing the frames coming from the upper layers and delivering the frames over the wireless medium. It interacts with the QoS policy service, for tuning some configuration parameters of the medium access function and employing different queuing policies according to the traffic type.
- *QoS policy*: It defines the queuing and priority policies according to the application requirements.
- *MAC Management*: It includes the signaling messages and the decisions for link management operations, such as link establishment, authentication and association, beaconing, etc.
- *Monitoring*: It collects statistics on system performance and wireless channel observations to be provided to other services for internal configurations and optimizations.
- *PHY Resource Management*: It allows the dynamic configuration of PHY parameters, thus including rate adaptation (in terms of selection of an available modulation and coding scheme) and power tunings on the basis of an optimization strategy.
- *Power Saving*: It decides on the activation or deactivation of the PHY layer and on the tuning of the transmission power in order to save energy, according to user and application requirements.
- *Application Optimization Support*: It allows exposing internal MAC parameters to the upper layers for enabling cross-layer optimizations.

The figure also shows how these modules can in turn use simpler functions (such as queuing management functions, frame forging, medium access functions, etc.) which can be invocated by multiple services in the same form on in terms of specific implementations. Note that only a sub-set of above services will ultimately interact with upper layers (e.g. data transport and application optimization support), while other services are responsible of MAC/PHY configurations and statistic collections. Additional services can be added for dealing with technology-specific capabilities of different platforms.

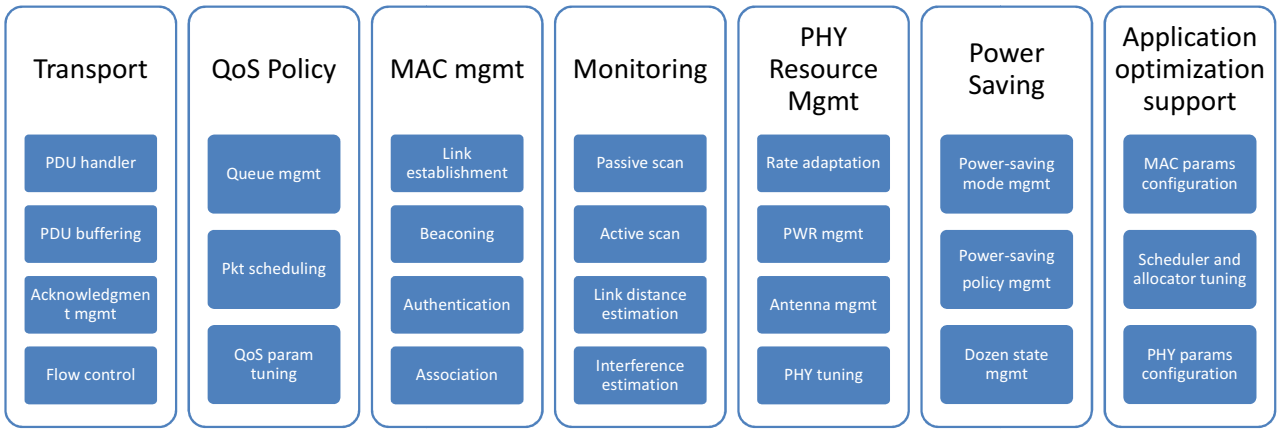

Figure 1-MAC Programmable Units: Services and Functions.

2.2 Functional and Control Sub-System

The system architecture includes a functional sub-system, implementing the system behavior, and a control subsystem, required for configuring the system and guaranteeing its consistency. Figure 2 shows all the architecture components of the functional sub-system (the boxes in the middle) and of the control subsystem (rightmost and leftmost boxes). Details about the architecture specification and relative interfaces are provided in D212.

At the lowest functional layer, there is a special architecture component, called wireless MAC Processor, that is responsible of interacting and scheduling actions on the platform layer, by executing an abstract program specified in terms of a state machine working on the available primitives. On top of this layer, there is a layer called function container that runs data organization and elaboration functions, in terms of frame forging, queue management, definition of signal messages, performance statistics, and so on. This layer interacts with the lower functional layer by passing frames to be transmitted by the platform and/or by invocating the system primitives. At the highest layer, there is the service container in which different layer 2 sub-tasks are executed in terms of independent (replaceable) modules. Each service may expose a customized interface to the higher layers and can invoke functionalities available at the lower layers. Service and functions are defined in terms of generic

bundles in a repository, from which they can be instantiated into the relative containers. The repository is part of a more general system information data base, which aggregates, abstracts and shares different data types, by communicating with all the system functional layers.

A transversal control plane works on the control of the functional layers (i.e. on the service/function logic and data structures), dealing with: i) service configuration, that is adding/removing service and function modules; loading/unloading running instances of services and functions; modifying/updating/reconfiguring service and function parameters (on the fly or off-line); ii) consistency management, that is exporting services definitions to other nodes; verifying service conflicts; guaranteeing inter-nodes and intra-node consistency; iii) resource virtualization, that is managing virtual interfaces; enabling virtualized services and functions; iv) data handling, that is collecting and storing system performance statistics; defining system state data; enabling function and service intercommunication.

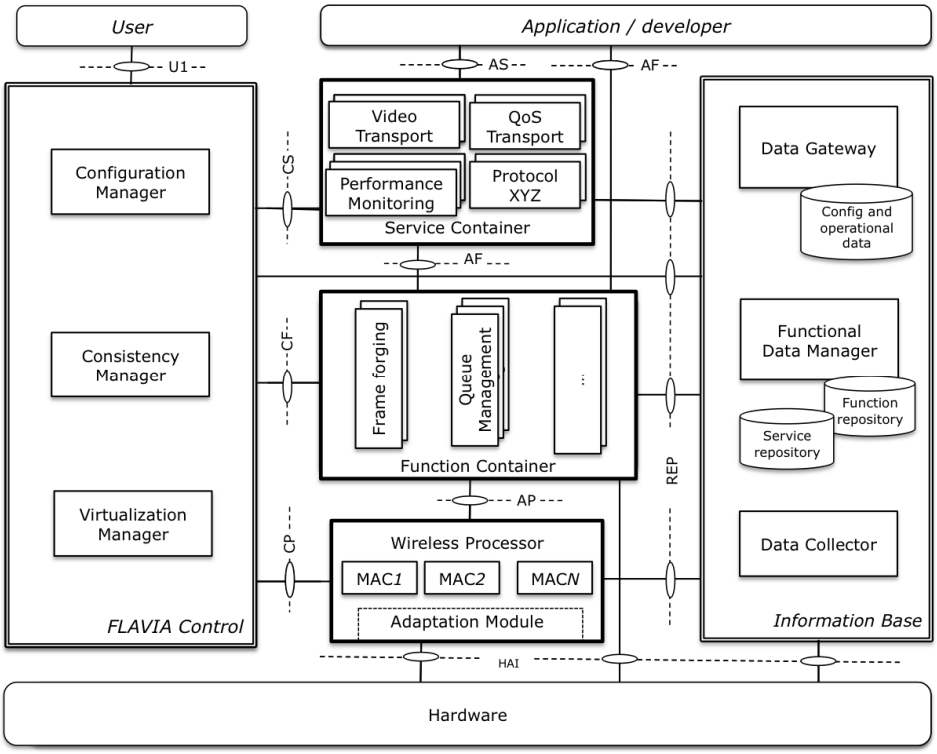

Figure 2: FLAVIA high-level view: framework architecture

2.3 System Primitives

The system primitives represent the basic system features on top of which new functionalities can be defined and added to the system and new wireless stacks can be configured. As emerged in D212, we envision two different levels of system primitives: i) the *functional sub-system primitives*, i.e. the primitives of the Wireless MAC Processor required for programming the operations on the hardware platform, and ii) the *control sub-system primitives*, i.e. the primitives required for loading, uploading and verifying the stack modules and interactions. As far as concerns the first group of primitives, we identified three different types of primitives: the set of events generated and/or revealed by the system, the set of elementary actions natively supported by the system, and the set of core system parameters whose settings can be tuned and verified. As far as concerns the second group, we identified primitive control functions to be exposed to developers and applications. The following table summarizes the primitive list derived from the requirements discussed in D212.

Primitive Description	Sub-System	Type
Encryption	WMP	Action (encrypt)
Contention window configuration	WMP	Parameter configuration
Set events for contention window update	WMP	Parameter configuration
Set events for backoff freezing / resuming	WMP	Parameter configuration
Inter-frame space configuration	WMP	Parameter configuration
Mechanisms for physical medium sensing	WMP	Action (sense)
Dissecting time critical MAC header fields	WMP	Parameter reading
RSSI/SNR/SNRI Measurements	WMP	Parameter reading
Busy/Idle slot detection	WMP	Event (ch_up)
Timer expiration	WMP	Event (timer)
Channel monitoring	WMP	Parameter reading
Controlling ACK generation or artificial dropping	WMP	Action (send_ack)
ACK frame and handshake configuration	WMP	Parameter configuration
Beaconing	WMP	Action (tx_beacon)

Active Scanning	WMP	Action (tx_probe)
Channel synchronization	WMP	Action (set_ch)
Scanning other cells while connected	WMP	Action (set_ch)
Advanced parallel scanning (TDM, two radios..)	WMP	Action (set_ch(vector))
Jamming	WMP	Action (tx)
Dynamic time/frequency/antenna Assignment	WMP	Action (set parameter)
Switching the radio on/off	WMP	Action (switch)
Scheduling activity intervals	WMP	Action (switch) / Event (timer)
Power tuning	WMP	Parameter configuration
Carrier Sense Threshold tuning	WMP	Parameter configuration
Modulation and coding selection	WMP	Parameter configuration
Antenna selection	WMP	Parameter configuration
Mode selection	WMP	Parameter configuration
Parallel streams setting	WMP	Parameter configuration
Opportunistic pre-coding	WMP	Action (pre_code)
Per-carrier channel state information	WMP	Parameter reading
Mapping transport channels to MIMO modes	WMP	Action (bind)
Spectrum scanning	WMP	Action (scan)
Channel bonding	WMP	Action (ch_bond)
Changing the number of subcarriers used for OFDM modulations	WMP	Parameter configuration
Accessing different OFDM subcarrier	WMP	Action (tx_subch)
Scheduling virtual interfaces access to	WMP	Multi-thread support
Detection of system failures	WMP	Event (error)
Handover detection	WMP	Event (rssi_threshold)
Soft-handover support	WMP	Action (set_ch1, set_ch2)
Mapping transport channels into medium access rules	Control	Virtualization Manager

Crete, modify and delete transport Channels	Control	Configuration Manager
Mapping control frames to transport channels	Control	Configuration Manager
Intra- and inter-flow transport differentiation	Control	Configuration Manager
Set up control channels for polling and feedback	Control	Configuration Manager
Defining per-queue data structures	Control	Configuration Manager
Associating queues to transport channels	Control	Virtualization/ Configuration Manager
Associating ACK Policies to transport	Control	Configuration Manager
Configuring flow control parameters	Control	Configuration Manager
Associating flow control policies to transport channels	Control	Configuration Manager
Enabling/disenabling TCP congestion control	Control	Configuration Manager
Associating congestion policy to data transport channels	Control	Configuration Manager
Changing flow priorities	Control	Configuration Manager
Defining virtual interfaces	Control	Virtualization Manager
Aggregating services to virtual interface	Control	Virtualization Manager
Listing/adding/removing available functionalities	Control	Configuration Manager
Instantiating/starting/stopping available functionalities	Control	Configuration Manager/ Consistency Manager
Linking multiple functionalities to a given service	Control	Configuration Manager/ Consistency Manager
Exposing customized SAP to applications	Control	Configuration Manager/ Consistency Manager
Resolving inter-service conflicts	Control	Consistency Manager

3 Wireless MAC Processor

The **wireless MAC processor** is the key component of the FLAVIA architecture, since it provides the framework for handling hardware events and scheduling actions on the hardware, thus supporting the programmability requirements on the lowest level of system resources, i.e. on the hardware and wireless medium resources.

The internal architecture of the wireless processor is expanded in Figure 3. The figure shows five main components: an execution engine for programmable extended finite state machines; a memory block including both the data and the program memory space; a set of registers for saving system state parameters; an interruption block passing the signals coming from the hardware to the execution engine; a set of operations which can be invocated by the execution engine to drive the hardware.
The figure also plots the interface towards the transmission/reception blocks and the PHY, and the interface towards the so called upper-MAC (implemented in the other functional components – the function/service containers - of the FLAVIA architecture).

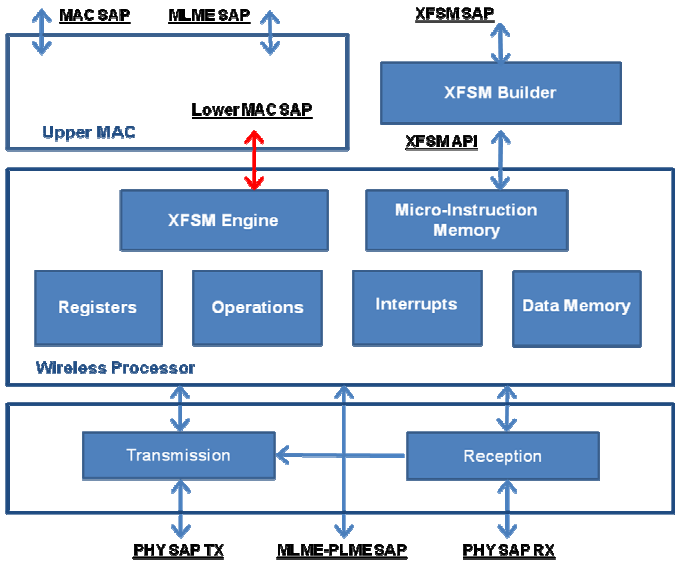

Figure 3 Wireless processor internal architecture

3.1 WMP Basic Architecture

The Wireless MAC Processor architecture [MobSummit2011] [INFOCOM2012]mimics the organization of ordinary computing systems, where programmability is accomplished by specifying: i) an adequate instruction set which permit to perform elementary tasks on a machine; ii) a programming language which conveys multiple instructions (suitably assembled to implement a desired behavior or algorithm) to the

machine, and iii) a Central Processing Unit (CPU), which executes such program inside the machine, by fetching and invoking instructions, updating relevant registers, and so on.

3.1.1 Instruction set: actions, events and conditions

A breakdown analysis of MAC protocols reveals that they are well described in terms of three types of elementary building blocks: actions, events and conditions. Actions are **commands** acting on the radio hardware. In addition to ordinary arithmetic, logic, and memory related operations, dedicated actions implement atomic MAC functions such as transmit a frame, set a timer, build an header, switch to a different frequency channel, etc. Actions are not meant to be programmable. As the instruction set of an ordinary CPU, they are provided by the hardware vendor. The set of actions may be extended at will by the device vendor, and complex actions (not necessarily restricting to MAC primitives) may be considered (e.g. perform a PHY encoding/decoding).

Events include hardware interrupts such as channel up/down signals, indication of reception of specific frame types, expiration of timers, signals conveyed from the higher layers such as a queued packet, and so on. As in the case of actions, also the list of supported events is a-priori provided by hardware design.

Conditions are boolean expressions evaluated on internal configuration registers. These registers are either explicitly updated by actions, or implicitly updated by events. Some registers are dedicated to store general MAC layer information (such as channel used, power level, queue length), frame related information (source or destination address, frame size, etc), or more specific MAC parameters (contention window, backoff, parameters, etc - used to achieve a more compact protocol description in case of specific MAC designs such as CSMA-based ones).

Actions, events, and registers on which conditions may be set, form the application programming interface exposed to third party programmers. This API is implemented (in principle) once-for-all, meaning that programs may use such building blocks to compose a desired operation, but have no mean to modify them.

The specific API described in D221 and summarized in Table 1 was shown capable of supporting several MAC behaviors. However, when faced with the requirements posed by virtualization and dynamic code handling functionalities, it showed structural limitations. For instance, we could not enforce conditions to control the switching between a previously running MAC code and a newly uploaded one. Thus, we needed to extend also the WMP internals, to implement an extended API accounting for new actions, events and registers, tailored to support dynamic code management.

Table 1 Wireless Processor API: lists of events, actions and conditions used for defining contention-based MAC programs

Events	**Actions**	**Conditions**
END_TIMER	*set/get*(reg,value)	dstaddr == value
CH_UP	*switch_RX()*	myaddr == value
CH_DOWN	*tx_ACK*()	queue_length > value
RCV_PLCP	*tx_data()*	queue_type == value
RCV_END	*tx_frame(type)*	cw < value
RCV_DATA	*switch_TX()*	cwmin == value
RCV_ACK	*set_timer(value)*	cwmax == value
MED_DATA_CONF	set_bk()	backoff == 0
MED_DATA_START	freeze_bk()	Frame_length > value
MED_DATA_END	update_retry()	frame_type==value
QUEUE_OUT_UP	more_frag()	channel == value
QUEUE_IN_OVER	prepare_header()	power > value
MED_DATA_CONF		ACK_on == value

3.1.2 Programs: Extensible Finite State Machines

The MAC program is stored in the *instruction memory* in terms of extended and programmable finite state machine. Formally, a state machine is specified by a list of states and transitions. In a Mealy state machine, a transition is given by:

- a *source state*, from which the state machine starts its operations;
- a *trigger event*, such as an hardware signal or a timer expiration;
- an *action*, representing an atomic program code, which can also create, move or destroy data objects;
- an optional *guard condition*, to be evaluated after the trigger event (which can be true or false);
- a *target state*, in which the state machine enters at the end of the action.

On the basis of: i) a set E of pre-defined events revealed and/or generated by the hardware platform, ii) a set A of pre-defined elementary actions, and iii) a set C of available conditions to be verified, a state machine program is given by a table, in which at location (i, j) the transition from state i to state j is specified in terms of the parameters ($event_x$, $action_y$, $cond_z$), with $event_x \in E$, $action_y \in A$, and $cond_z \in C$.
In our architecture, the set of events E is generated by the *interrupt* block, the set of actions A (and logic operators) is implemented in the pre-loaded *operation* block, and the set of state conditions C are memorized in the processor *registers*.

Given the sets E, A and C supported by the processor (which are exposed in terms of Application Program Interface API), different MAC programs can be completely

specified in terms of transition tables. An example of a generic MAC program is provided in Figure 4, where for sake of generality, we also assume that the state transitions can include multiple triggering events, multiple conditions to be verified and a list of actions to be performed. An XFSM builder can be optionally included in the architecture for mapping an high-level XFSM program into a wireless-processor transition table.

	End state					
Start state		*State 1*	…	*State j*	…	*State N*
	State 1		…		…	
	…	…	…	…	…	…
	State i		…	$event_{ij\lambda}, event_{ij\mu}, ..., event_{ij\nu}(*)$ $cond_{ij\alpha}, cond_{ij\beta}, ..., cond_{ij\omega}(**)$ $action_{ijx}, action_{ijy}, ..., action_{ijz}(***)$		
	…	…	…	…	…	…
	State N		…		…	

Figure 4 A MAC program in terms of transition table1

In principle, concurrent MAC programs can also be defined by a single XFSM, whose number of states is the product of the number of states included in each program. However, a multi-thread execution platform allows to us to drastically simplify the complexity of such an artificial definition, reducing the space required for memorizing the transition tables and natively managing potential conflicts arisen by concurrent executions.

3.1.3 MAC Engine

The ability to timely react to events is a crucial property of lower-MAC protocols (e.g. for triggering a transmission right at the end of a timer expiration). In the Wireless MAC Processor architecture, this is accomplished by implementing an XFSM execution engine, called MAC engine, directly on the radio hardware. The MAC program, namely the table containing all the possible state transitions, is loaded in the instruction memory. Starting from an initial (default) state, the MAC engine fetches the table entry corresponding to the state, and loops until a triggering event associated to that state occurs. It then evaluates the associated conditions on the configuration

[1] (*) To trigger the transition from state i to state j, at least one the this events should be caught
(**) conditions can be composed using AND, OR, NOT logical operators
(***) ALL the actions listed will be performed, in the specific order

registers, and if this is the case, it triggers the associated action and register status updates (if any), executes the state transition, and fetches the new table entry for such destination state.

Note that this approach easily supports code switching, in analogy to usual multi-threading. In fact, the MAC engine does not need to know to which MAC program a new fetched state belongs, so that a state transition can be easily associated to a MAC program switching (provided that a dynamic saving and reconfiguration of the registry states is possible).

3.2 API Extensions

3.2.1 Code-switching and multi-thread support

In order to support code-switching, the main modifications required to the WMP involve the possibility to move from one transition table to another (the new program) and to upload the platform configuration registers (the new state). These operations require that the micro-instruction memory has to be organized in multiple program slots with a dynamic pointer to the slot under execution. Moreover, since the events for switching from a program to another have to be revealed during the execution of each machine but they are not included in the transition table of the programs, the MAC Engine has been extended for dealing with default transitions, i.e. transitions to be considered from each state although not explicitly addressed in the transition table. For coping with the above issues, we extended the WMP API in terms of new events, conditions and actions, and we slightly changed the MAC Engine execution flow.

Loading. A new state machine and the initial state descriptor can be written on the WMP data memory by means of the *writing* primitive exposed to the host. However, the writing operation is allowed only when one of the new condition registers *accept_code*$_i$ is in an enabling state. The MAC Engine enables or disenables this register according to the program under execution. All the registers can be disenabled when code uploading is not desired. When a valid MAC code is loaded in the micro-instruction slot *i*, the corresponding (new) register *ready_code*$_i$ is set to 1 by the control process that completed the uploading.

Starting. Since multiple micro-instruction memory slots are available (two at least), the MAC code under execution is indicated by a new pointer register *memory_slot*, whose value is 0 when no code is available. In this condition, the MAC Engine executes a *bios state machine* (i.e. a pre-loaded state machine) that responds to a RUN$_i$ signal event received by the MAC control process (indicating the i-th slot to be run). This event triggers the execution of a *boostrap(slot value)* action devised to copy the settings of the configuration registers indicated in the MAC program initial state on the WMP internal registers. When the platform settings are reconfigured (as

indicated by the RESET event generated at the end of the bootstrap), a transition is performed from the current (bios) state to the initial protocol state and the new list of transition events is loaded.

Switching. The MAC Engine execution flow has to be modified for dealing with code switching. Indeed, since code switching signals are not specifically included in the MAC program state machines (and therefore not addressed in the transition table), they have to be explicitly managed by the MAC Engine. Two default transitions represented by the RUN_i signal, the condition *memory_slot != I && ready_code*$_i$, and the action *bootstrap(slot i)* are hard-coded in the engine (as they were added by default to all the table rows). This means that, in principle, the Engine is able to react to a switching signal coming from the control process regardless of the current state. The switching operations are identical to the starting of a new program, with the only difference that the final transition is now performed from the current state (rather than from the bios state) to the new state indicated in the program descriptor.

Synchronizing. Since the execution of a MAC protocol involves at least two independent nodes, in some cases it could be necessary to start the new protocol execution with strict synchronization requirements. For facilitating such a synchronization, we envision the possibility that the MAC control process writes a new default transition on the WMP, where the RUN_i signal is substituted by the desired synchronization event (such as the expiration of a timer, the reception of a beacon frame, etc.). This operation is enabled by a new *write sync* primitive which assigns the desired event to the symbolic SYNC event.

Verifying. Although most of the security issues related to code mobility can be demanded to the MAC control process, an important functionality that can be added to the WMP is the possibility to recognize trusted code, i.e. state machines coming from trusted sources. For example, a WMP of a given vendor could accept code updates only if produced by the same vendor or by a trusted vendors' alliance. To this purpose, a new *verify()* action (working on the comparison between a local signature calculation and the MAC signature embedded in the MAC code) can be hardcoded in the WMP. In this case, the default transition for starting or switching to a new code has to be modified in order to execute the bootstrap only after a TRUSTED confirmation signal generated by the *verify()* action.

Table 2 Wireless Processor additional API for code switching

Events	**Actions**	**Conditions**
SYNC	*verify*(value)	accept_code$_i$ == true
RUN_i	*bootstrap(value)*	ready_code$_i$ == true
RESET	*write_table*(slot)	current_code == value
TRUST	*write_sync(event)*	

3.2.2 Scheduled access support

We also considered some API extensions devised to support the definition of a scheduler for 802.16/LTE base stations. This is a macro-scheduler, meaning that the actual resource allocation operation is not included in the described operation, i.e., it only considers the selection of the users to be scheduled, based on their traffic demand and their status (e.g., a user might have no new data to transmit, but still might have some ACKs or ARQ/HARQ retransmissions ready in the queue).
In particular, Figure 5 represents the FSM of a simplified macro-scheduler for 802.16 or LTE downlink opportunistic scheduling. Since we only focus on the selection of the users to be scheduled, the amount of resources actually scheduled, and the specific OFDMA resource allocation is out of the scope of the description, and is performed while in the "Frame building" state depicted in Figure 5.

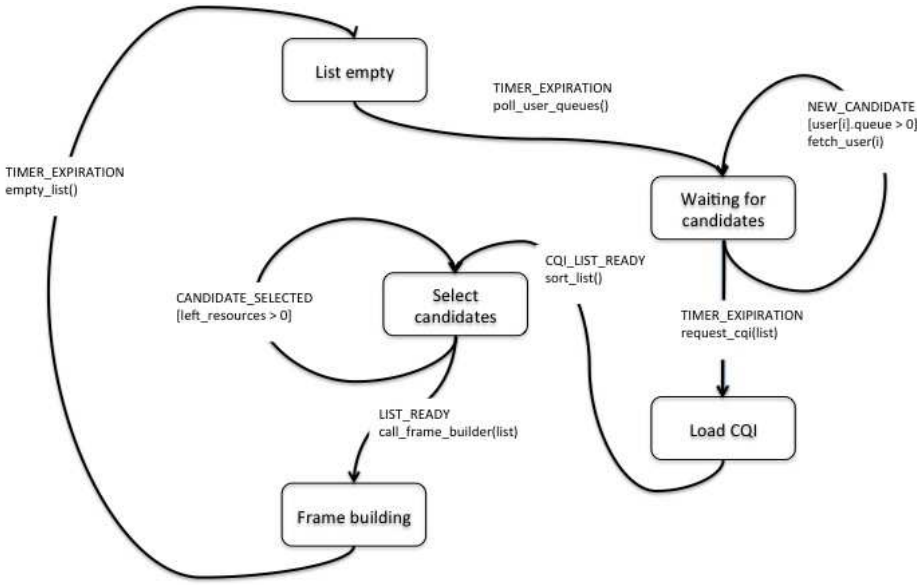

Figure 5 FSM for the macro-scheduler of 802.16/LTE with opportunistic scheduling of the users (e.g., based on CQI reports and priorities). The FSM does not include the frame building process.

The scheduling operation is periodic and then most of its procedures are regulated by means of timers. Specifically, starting with an empty list of users to be scheduled (state "List empty"), the scheduler waits for a deterministic time before checking the content of all downlink transmission queues (action: *poll_user_queues*()), which causes a transition towards the "Waiting for candidates" state. Then, there is a fixed amount of time during which the system goes over all the active connections and counts the amount of queued demand (in terms of new data and control traffic, ARQ, HARQ traffic, and ACKs). If a connection has traffic to transmit, then a user is added to the candidate list (NEW_CANDIDATE event, causing the *fetch_user*(userID) action,

subject to the condition that that user has traffic to transmit). After another timeout expires, the FSM transitions to the "Load CQI" state, in which CQI reports are gathered for each user in the candidate list. As soon as the list is ready (LIST_READY event), a *sort_list*() action is invoked which causes the reordering of the candidate list according to (*i*) HARQ priority and (*ii*) channel quality level. Then the system enters in the state "Select candidates", where candidates are extracted from the candidate list according to the rank assigned in the previous transition. When a candidate is selected (event: CANDIDATE_SELECTED) if the system's has residual resources for extra candidates to be scheduled, then the candidate is added to the final list and the next-in-line candidate's resource request is evaluated. Otherwise, the final list is complete (event: LIST_READY), and the frame building operation is invoked (action: *call_frame_biolder*(list)). The system enters state "Frame building", during which OFDMA resources are allotted to the selected users reported in the scheduling list. The system will wait for another timer to expire (event: TIMER_EXPIRATION) before cleaning the scheduling list (action: *clean_list*()) and starting over the opportunistic scheduling procedure.
A summary of events, actions, and conditions needed to implement the FSM illustrated in Figure 5 is listed in Table 3.

Table 3 Wireless Processor API: lists of events, actions and conditions used for defining the opportunistic user selection in the scheduler of a WiMAX or LTE base station (frame building not included)

Events	**Actions**	**Conditions**
TIMER_EXPIRATION	*poll_user_queues*()	user[i].queue > 0
NEW_CANDIDATE	*sort_list*()	left_resources > 0
CQI_LIST_READY	*request_CQI*()	
CANDIDATE_SELECTED	*fetch_user*(userID)	
LIST_READY	*call_frame_builder*(list)	
	empy_list()	

4 FLAVIA Architecture support for Modularity

4.1 FLAVIA Functional Architecture

In this section we propose a functional MAC architecture to divide MAC functionalities into modules with their corresponding interfaces in order to isolate, as much as possible, each module of interest. While MAC functions are typically considered as closed and "atomic" building blocks which implement specific operations with a limited set of interfaces to use their functionalities and access their internal information, a service offers more complex operations with a rich set of interfaces that can be exploited by other MAC services and functions, as well as MAC modules belonging to a peer entity.
According to the partitioning of MAC functionalities into services and functions modules, we divide MAC interfaces into three different layers: i) intra MAC layer – in and between MAC services and functions; ii) outer MAC layer (within the same entity) – between MAC to lower, upper and management & control subsystems; ii) inter MAC – between two different MAC entities.

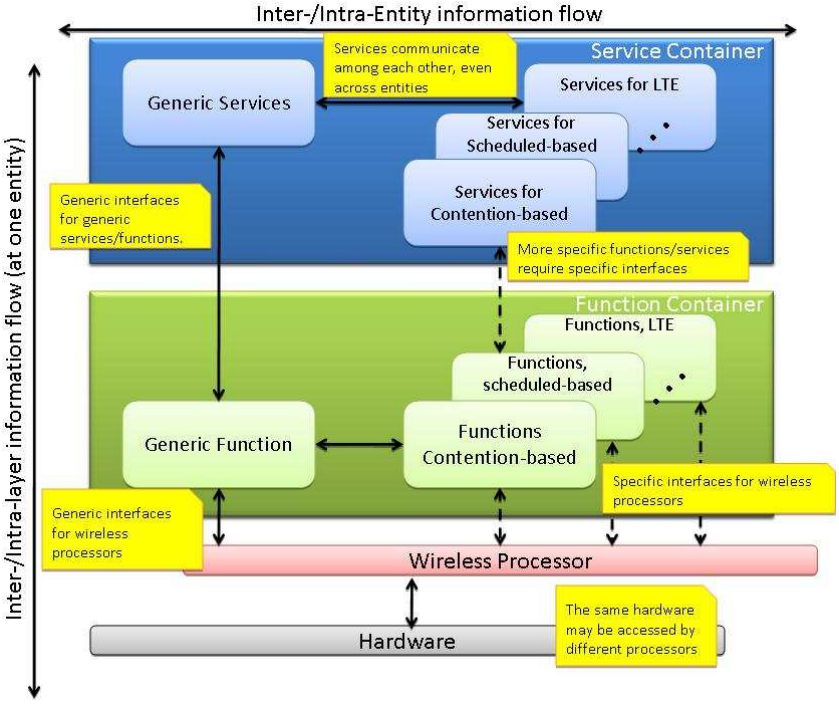

Figure 6 FLAVIA high-level view: Modular architecture

Although the definitions of the interfaces for scheduled and contention based systems are partially overlapped, the complete and detailed definition of the interfaces as well

as their requirements are described in the following for the two different technologies, which have been analyzed within the WP3 and WP4 technical works.

4.2 Functional Components and Containers

As discussed in section 2, the FLAVIA system behavior is described by creating and connecting different *functional components.* While the actions are composed in the MAC state machines and supported by the hardware platform, the functions and services are composed and executed by means of the *Function Container* (FC) and *Service Container* (SC).

The *Service Container* is connected to the control subsystem by means of the CS interface and provides functionalities such as: instantiating, activating, stopping, configuring and interconnecting the desired services. In addition, this container exposes the MAC service functionalities to the upper layers by means of the AS interface, thus fostering the loading of new services. The *Function Container* handles the configuration and execution of the functions that extend the MAC behavior by improving its functionalities. It is connected to the control part through the CF interface that allows function instantiation and configuration. Services register new functions through the AF interface, which enables their easy management through standard and well-defined commands, like registration, discovery, configuration, and invocation.

Figure 7 presents the description of the attributes and relationships of the functional components and containers. The three main attributes that characterize functional components are:

- *Configurability*: the functional component generally exposes a set of parameters used to configure its behavior. Therefore, these parameters can be inquired and modified to change the activities executed by the component.
- *Loadability*: this attribute defines the set of commands supported by the component that can be load into the system at execution time.
- *Accessibility*: functional components and their features can be accessed through different interfaces, which are implemented by Interface Access Points (IAPs).

The services are functional components that are also executable, since they can be started and stopped. Both services and functions access the information data base by means of the data gateway.

Service and function containers are the architectural components handling the instances, respectively, of services and functions. These containers provide functionalities to create, enquire, and maintain the corresponding instances.

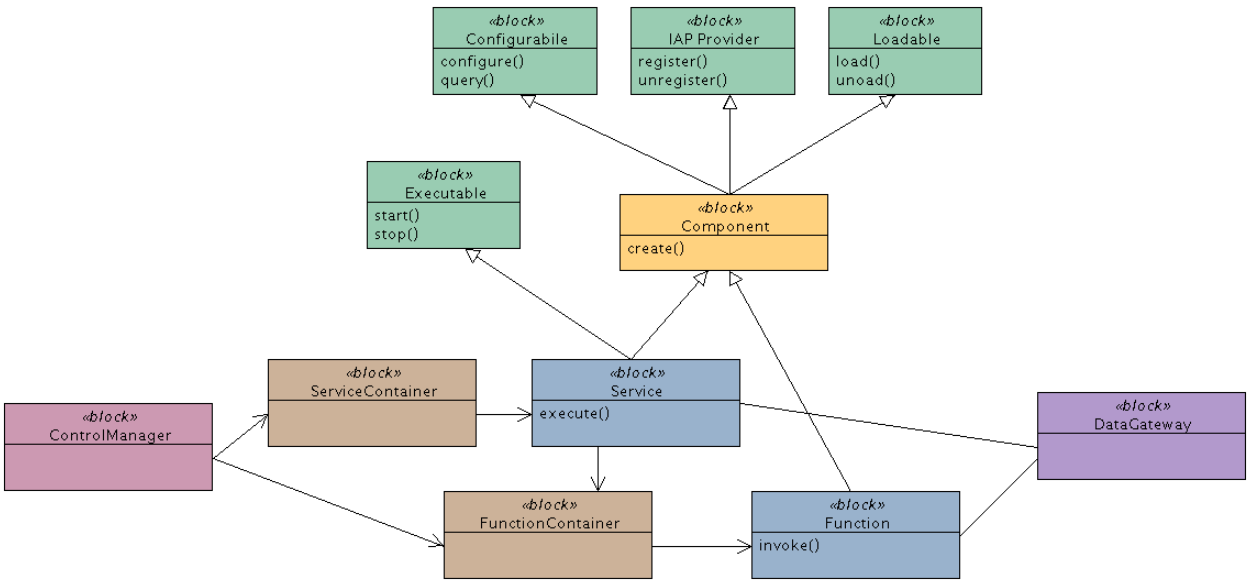

Figure 7 Attributes and relationships of the system behaviour components

4.2.1 Function Container Interactions

The main operations that can be performed by the CF interface are the registration and the configuration of a specific function. The registration operation binds an execution entry point with a name (F1) and a description (parameters, number of simultaneous instances, etc.). The *Function Container* assigns to each registered function a unique Function IDentifier (FID) that can be used for configuration purposes. Once registered, the *Control Manager* can enable the function configuration by using the *Function Container* functionalities and the function identifier.

The interactions with the services take place via the AF interface. A service interacts with the *Function Container* to access a specific function. The function container supports the service in two phases: i) the function *inquiry()* and ii) the function *invoke()*. When a service is initialized, it has to check if the functions required to properly operate are available. This task is performed by the inquiry operation provided by the *Function Container*. A running service can register new functions occurring at the execution of specific events to modify the behavior of the underlying MAC protocol through the invoke function by specifying the function and its corresponding parameters. From the modularity perspective of the FLAVIA architecture, the *Function Container* totally decouples services and functions. Indeed, functions and services can be defined separately focusing on their specific requirements and features, and bound the appropriate methods at execution time through the late binding of their implementation.
In order to simplify functions composition, the Function Container provides also the necessary mechanisms to manage the events that may occur within the underlying

MAC layer (subscribed by means of a subscribe() command) or other functions registered by the FLAVIA Services. On the one hand, events like packet reception or STA association can be captured to modify and extend the behavior of the MAC layer. On the other hand, FLAVIA functions can trigger customized events that are captured by the Function Container, so as to foster the interaction among several functions, thus simplifying their composition to implement enhanced services.
Therefore, the event-based architecture supported by the Function Container provides a high level of flexibility and modularity, since standard or customized events permit to easily implement future optimizations without modifying the code of services and functions already deployed.

4.2.2 Service Container Interactions

The service container allows three main classes of interactions:

- Interactions with the upper layers, which access the services through the AS interface.
- Interactions with the Control Manager, responsible of instantiating, initializing and configuring services using the CS interface.
- Interactions between different services, which allows communicating each other via the service Interface Access Points (IAPs).

The AS interface, which exposes the MAC functionalities to upper layers, is composed of different APIs that include APIs of standardized technologies. Since the MAC layer consists of multiple services, which interact among each other to extend their functionalities, these APIs are not supported by a single module. The *Service Container* is responsible to bind the APIs offered to the higher layers to the service IAPs, i.e., connecting each API functionality and its corresponding service. This specific association is configured by the *Control Manager* and modified at runtime if the MAC is reconfigured. Thus, the same API used by the upper layers can be dynamically changed, by reconfiguring the association between functionalities and services.

The CS interface is used during the system initialization or reconfiguration by the *Control Manager* for instantiating specific services, configuring the service instances, and binding the services IAPs to the MAC exposed API. Service IAPs may, as well, be bound to other service IAPs for inter-service communication. In addition, these binding operations are set-up in the *Service Container* by the *Control Manager*. Basic inter-node operations such as, association or de-association, are performed by connecting IAPs of service instances on different nodes.

4.2.3 Interface Access Point (IAP) definition

Interface Access Points (IAPs) define generic interfaces that provide an abstract framework to access and use the functionalities provided by FLAVIA services. For

example, most FLAVIA services might have "Configuration IAP" for service configuration, "Operational IAP" for interacting with the main service functionalities or "Statistics IAP" which provides statistics to other modules.

In order to maintain a high level of flexibility for the general definition of IAPs, without affecting their implementation strategy, we propose to use primitives to describe the main interfaces. A primitive definition is simply the operation requested to an entity, with the necessary set of parameters required to correctly execute the operation.
The description of IAPs through the utilization of primitives provides the necessary level of abstraction for the general definition of the activities that should be provided by the IAP, while leaving out implementation details such as the mechanism used to execute and coordinate the activity. Indeed, an interface implementation can be synchronous (i.e., function based) or asynchronous (i.e., message based).
For example, when a synchronous operation is implemented between two peer services located within the same MAC and running under the same flow context, primitive definition will simply be implemented as a direct function call with the parameters passed as function arguments. The function return value will include the corresponding primitive response (if it exists) containing the operation result. Asynchronous based interfaces will simply use the primitive parameters as message fields, which will be later sent over an asynchronous transport channel to a peer entity. Similar considerations apply to the transmission of a primitive response between two peers.

A complete notation of a primitive over an interface will begin with the interface identifier, followed by a short operation notation ended with the primitive type:

<Module Name>_<IAP ID>_<Operational Name>_<Primitive Type>

Table 4 shows the identifiers corresponding to each type of interface, according to the possible interactions among the different FLAVIA components.

Num	MAC Service	Description
IAP1	WP – MAC Interface	Control and data transfer between the wireless processor and MAC layer
IAP2	Inter–entity Interface	Interactions between services in different entities (e.g., among mobile stations or between the access point and mobile stations)
IAP3	Services – Functions Interface	Functionality required by services from functions
IAP4	Control & Mgmt Interface	Configuration & Management

IAP5	Inter Services Interface	Services interactions within the same entity
IAP6	Application Interface	Communication between FLAVIA services and upper layers (Control & Data)

Table 4 – Interface Access Points Identifiers

IAP1. WP – MAC Interface
This interface provides the mechanisms used to configure the Wireless Processor and obtain the information used by the FLAVIA services to extend the functionalities of classical MAC techniques.

IAP2. Inter-entity Interface
The Inter-entity Interface enables inter-process communication among entities (e.g., services and processes) running on different devices, like station to station or access point to stations communications. This interface provides the methods to set up a communication channel among the different entities to transfer essential configuration parameters. For example, the active monitoring service requires the coordination among the devices that belong to the same cell in order to prevent intra-interference during the estimation of external interface. The parameters used to synchronize the active monitoring procedures are transmitted using the primitives provided by the IAP2 interface.

IAP3. Services – Functions Interface
This interface provides the necessary mechanisms used by services to invoke the functions handled by the Function Container (FC). The IAP3 interface is implemented by the Function Container, to better coordinate the utilization of functions invoked simultaneously by competing services, thus simplifying the concurrence management.

IAP4. Control & Mgmt Interface
IAP4 provides the methods to control and correctly configure the system. This interface is used also to notify changes to the state of the system so as FLAVIA services can modify their configuration in response to specific events. For example, the activation of power management triggers the modification of the configuration other running services, like the data transport and the monitoring services which should stop their execution during periods of inactivity. Note that the notification of system changes and the following modification request of services configuration needs to be performed through IAP4, since it implements the procedures that guarantee the system consistency.

IAP5. Inter Services Interface

The Inter-Services Interface enables the communication and coordination among different FLAVIA services. IAP5 is implemented by the Service Container that is liable for the scheduling strategy used to execute and coordinate the FLAVIA services. The interface provides also the primitives used by the services to directly communicate information like configuration parameters, performance statistics, and the operating state. For example, through IAP5 the monitoring service provides the information which has collected about the channel quality so as other services can modify their behavior accordingly (the virtualization service can select only the APs tuned on the best wireless channels).

IAP6. Application Interface
IAP6 enables upper layer protocols to use the services provided and implemented by the FLAVIA system. This interface controls the access to the functionalities implemented by the lower layers like reliable data transmission and network monitoring. Note that configuration primitives are provided by other IAP interfaces.

4.3 Service Modules

Services modules and their corresponding functions constitute the essential components of the FLAVIA architecture, as they are the fundamental blocks used for accomplishing specific MAC layer sub-tasks and their interaction define the behavior of the sub-tasks. The decomposition was motivated by the conclusions drawn from the analysis of the operational scenarios previously studied in D211. Specifically, we identified for both scheduled and contention-based paradigms a set of possible use case scenarios with their corresponding operational context, a basic set of services to be mapped into MAC-layer sub-modules that would be required to achieve the desired functionality. The outcome of this analysis is summarized in Table 5.

Contexts	Scenarios	Services
Single cell scheduled systems	*Multicarrier & load balancing strategies*	- Load balancing - Admission Control - QoS Scheduling
Single cell scheduled systems	*Feedback radio resources management*	- Application optimization support
Single cell scheduled systems	*Extending scheduled systems flexibility through user virtualization*	- Support for Self-Organizing Networks - Virtualization
Single cell scheduled systems	*"Swiss Army Knife" Network Nodes*	- Application optimization support - Support for SON

Single cell scheduled systems	*API for upper layer adaptation*	- Data transport - Application optimization support
Multi-cellular and overlapping networks	*Network Nodes Resources Virtualization*	- Virtualization - Power saving - Cell selection & tracking
Hierarchical macro/micro cells	*Cooperative allocations for femto/macro cells*	- Inter-cell coordination & cooperation - Power Control - Mobility Management
Single cell contention-based systems	*Intra-WLAN PHY-based traffic differentiation*	- MAC/PHY resource mgmt - Data transport with differentiated QoS - Data transport configuration
Single cell contention-based systems	*Punishment or incentive mechanisms for programmable nodes*	- Flow/congestion control
Single cell contention-based systems	*Multi-rate selection schemes for emerging PHY layers*	- MAC/PHY resource mgmt - Monitoring
Single cell contention-based systems	*Network analysis and monitoring*	- Monitoring - MAC/PHY resource mgmt
Overlapping networks	*Interference free networks*	- Monitoring - MAC/PHY resource mgmt
Overlapping networks	*Load-based OFDM carrier allocation with concurrent WLANs*	- MAC/PHY resource mgmt
Overlapping networks	*Backhaul bandwidth aggregation*	- MAC/PHY resource mgmt - Mobility mgmt
Overlapping networks	*Broadband hitch-hiking*	- Mobility mgmt - Power saving mgmt
Overlapping networks	*Indoor location*	- Mobility mgmt - Monitoring
Multi-service mesh networks	*Multiple virtual networks*	- MAC/PHY resource mgmt
Multi-service mesh networks	*Infrastructure-less and Infrastructure-independent Mesh for emergency services*	- Data transport with parameterized QoS - Data transport configuration - Layer-2 routing

		- Monitoring - MAC/PHY resource management
Multi-service mesh networks	*Video support in 802.11-based mesh networks*	- Data transport with differentiated QoS - Data transport configuration - Layer-2 routing - Monitoring - MAC/PHY resource management
Multi-service mesh networks	*Distributed allocations in mesh networks*	- Data transport with parameterized QoS - Data transport configuration - Layer-2 routing - Monitoring - MAC/PHY resource management
Multi-service mesh networks	*Advanced multi-antenna techniques*	- MAC/PHY resource mgmt - Monitoring - Layer-2 routing
Special (niche) networking scenario	*Long distance links*	- MAC/PHY resource mgmt - Monitoring
Special (niche) networking scenario	*Highly noisy and attenuated environment*	- Security mgmt - Data transport with differentiated QoS - Monitoring - Layer-2 routing

Table 5 – Services needed for each scenario

In what follows, we discuss general functional aspects of these services, while a more detailed description of the design of these services is given in D311 and D411 for scheduled and contention-based systems, respectively. Notice that the management of these service modules is undertaken by the FLAVIA control subsystem, which manages specific service compositions and creates and configures running instances of them. Details on how services are loaded and how their consistency is maintained are discussed in Section 5.

To provide support for data transport across the network, a data **Transport** service is defined for both types of medium access considered. This handles encapsulation, fragmentation, multiplexing, encryption, as well as queuing routines. The service

interacts also with a **QoS Policy/Transport** module which is responsible for packet classification and service differentiation. A **Flow/Congestion control** service implements various traffic policing functions which depend on the **QoS policy** module. A **Monitoring** service will provide these services with MAC and PHY statistics to ensure they operate efficiently and appropriately react to changes in the network conditions. Given the physical capabilities of the hardware, gathered statistics and constraints imposed by the higher layers, a FLAVIA node may implement certain rate and power control algorithms within dedicated service modules. For hardware supporting power saving operation, a **Power Saving** service is designed to manage transmission power, enable/disable device sleeping, and taking protocol-specific steps such as notifying other stations on entering/exiting power saving modes. Finally, while the QoS and Link/Rate/Power control services provide parameters for transmission, higher layers may also adapt the traffic they generate depending on channel state, available resources, etc. To this end, an **Application Optimization Support** service provides interfaces to facilitate this interaction.

Table 6 summarizes the MAC Service Modules present in the FLAVIA architecture. As already explained, some of them are present in both scheduled-based and contention-based systems, meanwhile others are specific for only one of these two types of systems.

Service name	Scope	Description	Functions
Data transport	Both	Responsible for receiving/sending packets from/to the MAC upper layer, packet manipulation within MAC (including multiplexing, encryption, adding necessary headers, segmentation, aggregation, encryption, etc.).	- Frame forging - Queue mgmt
Data transport configuration	Contention-based	Provides traffic classification functionality and allows selecting a data transport approach between Data transport services with parameterized and differentiated QoS based on the classification	- Traffic classification - Transport service choice
Data transport with differentiated QoS	Contention-based	Provide traffic prioritization and delivery of packets with best-effort quality, based on CSMA/CA.	- Manage access policy - Manage traffic filters

			- Reliable groupcast - Queue mgmt - CCA/NAV/RAV mgmt.
Data transport with parameterized QoS	Both	Accepts details of QoS policy from higher layers and implements the policy by classifying packets passing through the Data Transport service. Provides efficient delivery of packets with specified QoS parameters by means of channel reservation and scheduling.	- Manage one-hop reservation - Manage multi-hop reservation - Scheduling (MCCA mgmt) - CCA/NAV/RAV mgmt - Reliable groupcast
QoS Strategy	Scheduled	Preparing a candidates list to be potentially scheduled in a specific frame, taking into account QoS requirements and time constraints.	
Scheduling Strategy	Scheduled	Implementing a specific scheduling strategy on a set of connections on given resources.	
MAC Scheduler	Scheduled	Managing frame construction process.	
Admission control	Both	Decide whether to accept or refuse a new connection in the cell.	
Load balancing	Scheduled	Once a new connection request arrives, the load balancing service decides on the distribution of users amongst the available cells.	
Power saving	Both	Enables power save modes. Manage power saving intervals synchronized between BS and MS.	- Manage power save mode - Manage power save policy

Inter-cell coordination & cooperation	Scheduled	Inter-cell interference control and inter-cell synchronized scheduling.	
MAC/PHY resource management	Both	Set PHY and MAC radio parameters in order to select the best configuration as a function of the network environment.	- Manage traffic filters - Rate control - MIMO control - Virtualization - OFDM subcarriers management - OBSS mgmt
Mobility Support	Both	Support for mobility-related functionalities, including IP address management hand-over and location updates.	- Manage IP addressing - Association
Support for SON	Both	Automatic intra-cell and inter-cell network monitoring and self-optimization of MAC/PHY parameters.	
Application Optimization Support	Both	Statistics collection and analysis of the run-time performance of MAC/PHY protocols. Application detection and MAC/PHY adaptation to running applications.	
Measurements and Monitoring	Both	Collect measurements and statistics related to low layer and MAC modules; Monitor low layer and MAC key performance indicator to trigger events when necessary. Detect misbehavior nodes and punish them.	Active scanning - Passive scanning - Interference estimation - Link distance estimation - Misbehavior nodes detection and punishment
Cell selection and tracking	Scheduled	MS synchronizing, acquiring and tracking to the DL channel of BSs.	
Layer-2 routing	Contention-based	Responsible for neighbor discovery, path selection and frame forwarding.	- Neighbor discovery and management - Routing

			metrics estimation - Mesh Routing Information Dissemination - Mesh Path Management and Forwarding
Flow and congestion control	Contention-based	Detect, repair and prevent any traffic congestion on the network.	- Manage ACK policy - Manage flow control policy - Manage flow control parameters - Manage congestion control policy - Manage congestion control parameters
Security management	Contention-based	Manage authentication, association and other security-related aspects.	- Authentication - Association

Table 6 – Summary of MAC services

Figure 8 and Figure 9 shows the complete organization of the service architecture for a scheduled-based and a contention-based FLAVIA system. The services summarized in the figures are built upon a set of elementary components which we refer to as '*function modules*' within the FLAVIA architecture. Services and functions mainly differ in the scope of their interfaces and the fact that the functions do not interact with each other.

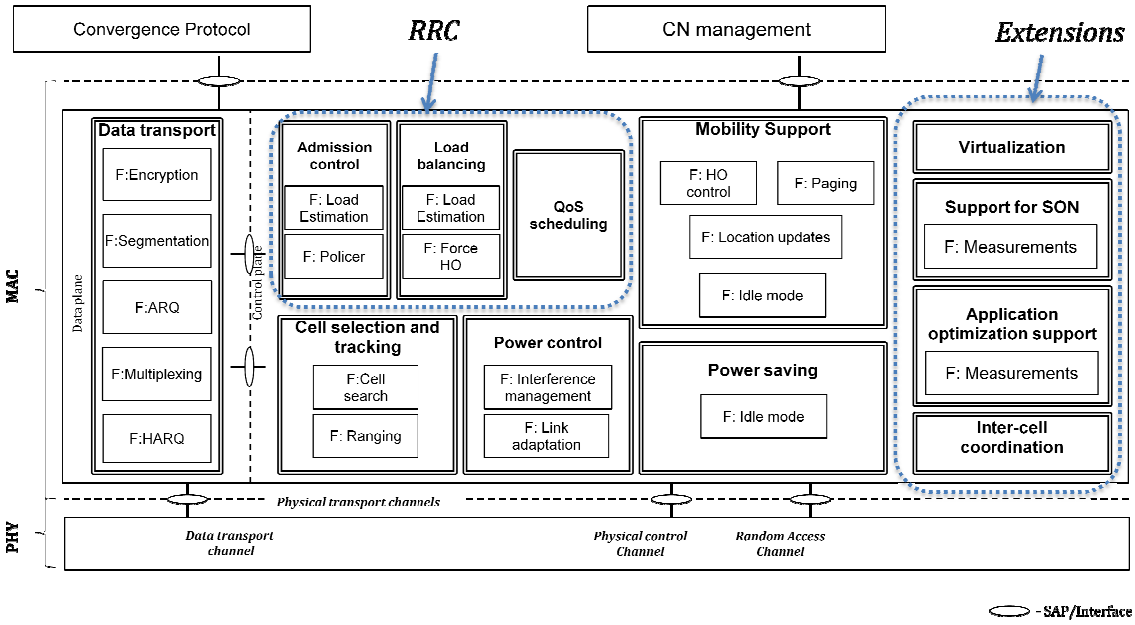

Figure 8 - FLAVIA scheduled MAC functional architecture.

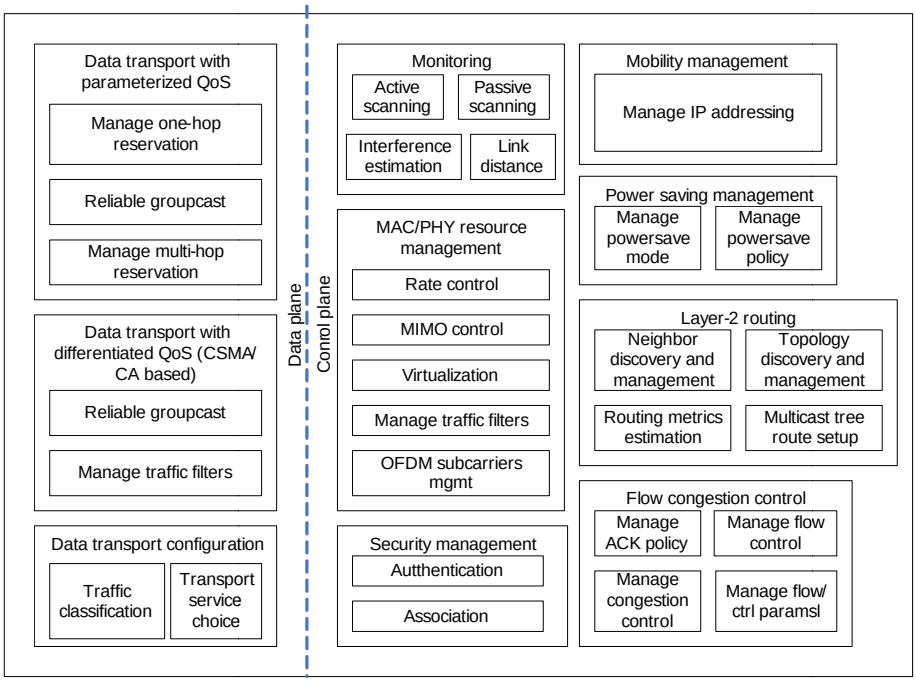

Figure 9 - FLAVIA contention-based MAC functional architecture

Functions only expose interfaces to services but not to other functions or users of the FLAVIA framework. Therefore, functions represent the core enablers and building components for services. In turns, functions rely on the API exposed by the wireless MAC processor in terms of commands, system parameters and events. Generic functions define the behavior of both contention-based and scheduled-based systems. Although their implementation may differ, they provide the same mechanism and potentially use the same interfaces. Specific functions can be added for dealing with the peculiarities of contention-based or scheduled systems.

Among the generic functions, we consider transmission power control routines for reducing energy consumption and interference, power saving techniques that implement different sleep modes, as well as state management functions that operate during power saving. In terms of MAC functionality, FLAVIA provides support for rate adaptation in varying channel environment to optimize the link spectral efficiency, taking into account the channel access mechanism. Advanced channel access methods, such as e.g. Space Division Multiple Access (SDMA), are supported, with the inherent MIMO capabilities. Further, ARQ and hybrid ARQ functions are envisioned to provide more robust data services and increase spectral efficiency. Functions that enable node configuration, synchronization and key management have been also considered, while a set of elementary modules that compose the basic transmission operations have also been defined, namely CRC check, Segmentation, Concatenation, Reassembly, PDU management, data compression and encryption. In both scheduled and contention-based systems, scanning and measurements functions are defined to support network discovery, localization, scheduling, mobility, link adaptation, etc._To obtain cross-layer optimization of applications and to be able to deploy network self-organizing (SON) mechanisms, generic functions handling transport channel, data queue and traffic class management are considered.

In the context of **contention-based systems**, particular attention is paid to functions that handle dissemination of routing information and multi-hop path management and forwarding. The optionally data transport with reservation (similar to MCCA) shall also be supported, while advanced functionality such as access and ACK policies need to be implemented. In this context, it is also important to have a set of functions enabling a fine tuning of the MAC protocol such as CCA/NAV/RAV management, dynamic adjustment of contention parameters and coordination between overlapping BSSs. The above functions are further described in Table 7, while a detailed specification is provided in D411.

In the context of **scheduled systems**, FLAVIA defines a set of functions that permit flow (de)multiplexing, tightly coupled with the PDU management, ARQ, CRC, and encryption functions. Different from scheduled based systems, logical/transport channel mapping functionality is also defined, as well as the typical broadcast and multicast functions found in cellular networks. Additionally, FLAVIA allows for BS-MS Capabilities Exchange, including BS-MS authentication and key exchange, BS-MS

registration, establishing IP connectivity, establishing time of day and transfer operational parameters and establish provisioned connections. Finally, we define a set of functions that aid different handover procedures, initiated both by the MS and the BS. D311 provides more details on the functions applicable to scheduled systems, while Table 7 includes as well a summary of those.

Function Name	Scope	Description
Power control	Both	Adaptively changes the transmit power at the mobile nodes and base station nodes
Power saving support	Both	Exploits the possibility to put MS in a low-power sleep mode. BS/RAP must provide the possibility to buffer data, to wake-up a MS according to a pre-defined mechanism, and to differentiate sleeping and active MSs.
MS state manager	Both	Manage the idle/connected state and the DL/UP activity requirements during power saving operations.
SDMA Support	Both	Creating parallel spatial channels to allow for higher capacity through spatial multiplexing and/or diversity.
Rate adaptation	Both	Different MCS sets but the same mechanism; major part of link adaptation.
RAP and UE Measurements	Both	Estimation of PHY layer parameters, e.g. SNR, RSSI, CFO, Location, etc. Incl. interference profile/estimation.
Profile mgtm	Both	
Scanning	Both	Active, passive; Of particular interest for network entry and cell tracking
Neighbour discovery and link mgmt	Both	Periodic beaconing by stations in ad-hoc and mesh networks, to determine adjacent nodes with whom directly communication is possible. Link management function should only open stable links with required probability of successful data transmission.
Collect statistics & monitoring	Both	HARQ/ARQ stats, radio stats, traffic stats, neighbour stats, load estimation, arrival prediction (?), throughput stats
Configuration	Both	@ UE / BS; antenna, bandwidth, power, coverage, OFDM
Synchronization	Both	Error-handling, time/ freq. sync correction.
Encryption	Both	IEEE 802.16m it is applied after the packing function but before multiplexing. In 802.11, depending on the chosen mode as well

		as the data, which has to be encrypted, different algorithms are applied, e.g. DES, AES, RSA, or ARC4.
Key management	Both	Derivation, update, usage, and exchange of keys in a wireless network. In IEEE 802.16m the PKMv3 is employed to manage three important keys: authorization key, traffic encryption key, and the cipher-based message encryption key. In IEEE 802.11i, the key management is responsible for the authentication, authorization, and accounting key (AAA), the pairwise master key (PMK), the pairwise transient key (PTK), the group master key (GMK), and the group temporal key (GTK).
CRC	Both	CRC checks are performed in order to test the correctness of a transmission. IEEE 802.11 and 802.16, partly use different CRC polynomials, but they apply the same mechanisms.
(De-)Compression	Both	IP header
Segmentation	Both	@ Tx (PDU mgmt)
Concatenation	Both	@ Tx (PDU mgmt)
Reassembly	Both	@ Rx (PDU mgmt)
ARQ	Both	
Transport channel mgmt	Both	Incl. bandwidth mgmt, CSMA based or scheduled based
PDU managment	Both	Reordering, Duplications
Data queue mgtm.	Both	
Traffic class mgmt	Both	Of particular interest for Data transport configuration, Admission control, Application optimization support
Mesh Routing Information Dissemination	Contention	Disseminate routing information (routing metrics) in both proactive and reactive (on-demand) ways
Mesh Path Management and Forwarding	Contention	For the flow(s) with the given transmission requirements choose the best paths to destinations taking into account used channel access method. Forward packets according to chosen paths.
Sheduling (MCCA mgmt)	Contention	Disseminate and process sheduling information (information on reservations) in the network
Access policies	Contention	Maintain and manage a database of configured

		policy. Queried to determine if a course of action is within the policy
ACK policy	Contention	Responsible for the appropriate generation of ACK packets in the correct format.
CCA/NAV/RAV mgmt	Contention	Provide PHY carrier sense feature and notify of virtual busy time intervals
OBSS Mgmt	Contention	Coordinate and reduce interfernce between overlaping BSS working on the same channel
Contention configuration	Contention	Incl. Update persistent RACH probability; Important for Link/Rate adaptation, application optimization support, and flow/congestion control
Network entry	Scheduled	Scan for DL, Ranging, Capabilities exchange, Key exchange, Registration, Connection establishment
Handover support	Scheduled	UE or BS initiated; Intra/Inter-RAT
Admission control enforcement	Scheduled	
HARQ	Scheduled	
DL Broadcast (BCCH)	Scheduled	Incl. cell-info, paging
DL Multicast (MCCH)	Scheduled	
Flow (de)multiplexing	Scheduled	
Logical/Transport channel mapping	Scheduled	

Table 7 List of most important functions in the FLAVIA framework architecture

5 FLAVIA Architecture support for Flexibility

5.1 FLAVIA Flexibility view: Control architecture

Flexibility is one of key concepts of the FLAVIA architecture. It corresponds to the ability of easily modify and re-program the lower layers of the network stack to respond to varying network contexts (i.e. propagation conditions, dynamic topologies, applications, etc.). According to our solution, flexibility is provided through the management of open and interchangeable services and functions which can be promptly reconfigured. To this purpose, the FLAVIA architecture includes a transversal control plane working on the control of the functional sub-system, i.e. on the service/function logic and data structures. Specifically, the control plane deals with the following tasks:

- service configuration:
  - add/remove service and function modules;
  - load/unload running instances of services and functions;
  - modify/update/reconfigure service and function parameters (on the fly or offline).
- consistency management:
  - export services definitions to other nodes;
  - verify service conflicts;
  - guarantee inter-nodes and intra-node consistency;
- resource virtualization:
  - manage virtual hardware interfaces;
  - enable virtualized services and functions;
- data handling:
  - collect and store system performance statistics;
  - define system state data;
  - enable function and service intercommunication.

The architecture component dedicated to the support of these functionalities is the *Control subsystem,* which is composed of two parts: i) the control manager that performs control operations on the *system logic,* and ii) the information base that performs control operations on the *system data*. The control manager exposes a user interface to allow the interaction between the user/operator/developer and the FLAVIA system. The information base acts as a shared memory for storing information about available services and functions, data structures, and low level data.

5.1.1 Control interfaces

The control manager communicates with other architecture components by means of the control interfaces, whose name acronyms are taken from the linked components:

- the *CP interface (Processor Control)*, between the control manager and the processor. This interface allows loading medium access programs, i.e. XFSMs transition tables, on the wireless processor. If present, the XFSM builder is included in the control subsystem and exposes the methods for defining the finite state machine driving the hardware to the user interface.
- the *CF interface (Function Control)*, between the control manager and the function container. It allows loading function modules on the function container in order to be used by the services.
- the *CS interface (Service Control)*, between the control manager and the service container. It provides methods to load/unload or start/stop services acting on the service container.

Moreover, the functional subsystem interacts with the information base by means of the REP (*Repository*) interface.

Finally, there is another interface called Inter-Node Interface used for linking the control subsystems and the service containers of two independent FLAVIA nodes. This interface has two different tasks: i) enable communication between peer services running on two nodes; ii) enable service configuration and activation by negotiating service/function capabilities amongst two nodes. .

Interfaces	Entities connected
Service Control Interface (CS)	Control manager – Service container
Function Control Interface (CF)	Control manager – Function container
Control Processor	Control manager – Wireless processor
Repository	All components – Information base
Inter-Node Interface	Control manager at node 1 – Control manager at node 2 Service X at node 1 – Service X at node 2
User	Control manager - User

Table 8 Summary of control interfaces and related entities

5.2 Control Manager

The *Control Manager* component performs control operation on the system logic and it is mainly dedicated to the configuration, composition and orchestration of the system services. The service configuration and composition is specified by means of a higher level user (U) interface, while the overall orchestration of the running services is managed by means of the CS, CF and CP interfaces.
The control manager architecture can be further decomposed in the following sub-components:

- the command parser, working as a standard code interpreter for translating user commands to system actions;
- the service manager, for loading and activating the service/function modules on the execution environment;

- the consistency manger, for solving potential conflicts arisen by concurrent services;
- the capability negotiator, for identifying the system features which can be used by the running services to be compatible with the peer services on different nodes;
- the virtualization manager, for exposing the system resources to concurrent utilizations.

5.2.1 Command parser

The command parser maps user-level commands into actions to be performed for configuring the system. We do not define the command names and command syntaxes exposed at the user level. Conversely, we envision that the main actions needed for working on the FLAVIA systems should include:

- listing all the available service and function modules loaded in the system;
- listing the running services;
- starting and stopping a given service;
- configuring the tunable knobs of the running services;
- loading/updating a service/function module in the system;
- retrieving information about the system state and performance parameters;
- activating/de-activating virtual interfaces;
- performing consistency testing.

In order to perform these actions as a response to specific commands, the command parser has to invocate the internal interfaces exposed by the other control subsystem components (i.e. the service manager, consistency manager, capability negotiator and virtualization manager).

5.2.2 Service manager

The service manager controls the service and function containers by means of the following functionalities:

- creating a new service;
- loading/unloading modules on the containers;
- monitoring the service/function executions (starting, freezing, stopping executions and handling exceptions).
- registering and finding a specific service/function module.

For creating a new service it is necessary to specify: i) the service access point, ii) the functions required by the service, iii) the service logic (i.e. the service program to be instantiated in the service container). These information elements are stored in the information base, where a new entry is added for each defined service.
Similarly, a new upper-MAC or lower-MAC function can be created by adding to the information base the function interface and the function program.

For loading a new service, the service manager verifies that all the functions required by the service (as specified in the information base) have been instantiated, i.e. the service dependencies are satisfied. If some functions are not available, the service manager first tries to load and instantiate the missing modules. If the extra modules are not available, an error will be arisen and the service will not be loaded properly. After a successful service load, the service container allocates the required resources and the service can be initialized and launched. The unload function may be required if a service has to be removed, updated, or replaced.
Finally, the service manager monitors the active services runtime, by sending commands to the service container for retrieving service state information and/or for freezing or stopping the service execution.

5.2.3 Consistency manager

The Consistency Manager (CM) is a critical element of the control architecture, since it significantly simplifies the service program development, by autonomously verifying and solving potential consistency problems created by the concurrent execution of independent services.
We distinguish two different types of consistency: internal consistency among services or functions on the same node and external consistency, related to different nodes, (e.g. two different versions of the same service running on different nodes).

Internal consistency checks include:

- *Regulatory bounds*: they represent the limits imposed by national regulatory policies to the settings of some system parameters. For example, the transmission power cannot be higher than a maximum value indicated by the national regulation, or the operating channel cannot exceed a given frequency. The most typical use-case of internal consistency verification function is the Tx Power Control for a wireless interface against the physical characteristic of the end-level power amplifier. Setting of the Tx Power should also take into account the antenna gain value 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.
- *Consistency of system parameters*: implies the verification that different system parameters with mutual dependency are set to consistent values. For example, a customized ACK frame definition also affects the ACK timeout duration, whose consistency has to be guaranteed by the consistency manager, even if the timeout is not explicitly changed within the service defining the new ACK frame.
- *Parallel access to the same system parameter*: is required to eliminate the possibility of mutually exclusive settings being applied by different services/software modules to the same system parameter. This is essential to introduce a mechanism to manage simultaneous accesses to the same

system parameters by concurrent services. The CM may be capable of introducing an additional level of modules/services differentiation within the overall FLAVIA node architecture. It adds the possibility of allowing or denying a specified module/service to request a specific parameter value change. It can assign priorities to modules to perform specific actions or decide which request is allowed to take precedence over another.
- *Consistency of service logic*: it represents the verification that the service is correctly built, by aggregating consistent functions.

The verification of all the conflict types cited above is performed by registering specific consistency conditions in the information base storing the data. For example, the creation of a service working on ACK frames might correspond to the registration of a condition on the ACK timeout duration, such as ACK timeout = ACK transmission time + maximum propagation delay. Some of these conditions can be pre-registered (and even hard-wired) in the information base, for imposing regulatory bounds. As clarified in the next section, these conditions are checked at each data or service logic update.

Apart from the problem of revealing inconsistencies by defining consistency conditions, a critical task of the consistency manager is providing a framework for solving the conflicts, thus avoiding that service developers have to explicitly address this type of problems. Whenever a consistency condition is violated, several reactions are possible:

- the consistency manager forces the service which generated the violation to abort. This straightforward solution, however, isn't always acceptable for all services, because sometime a service can't just be stopped;
- the consistency manager can return an error code to the control manager which is performing the call ("*passive behavior*"). Thus, the component which wants to finish the service can decide how to proceed: either to correct the inconsistent state or to explicitly abort the service;
- the consistency manager can automatically enforce a correction to the parameter that cause the violation ("*active behavior*"). Then, the corresponding service is aborted only if this automatic correction fails.

The consistency of information stored in the Information Base (IB) is guaranteed by the Data Gateway (Figure 2), which defines solutions for both single and multiple write protection.

For **external consistency**, that is the consistency between peer services running on different nodes, we envision a solution similar to the previous one, based on the mutual exchange of consistency conditions. In other words, rather than accessing the local information base, the binding between two services running on different nodes requires to access the peer information base and registering remotely a logic consistency condition related to the ongoing service. In case of consistency error, the correction of the violation is managed by the capability negotiator.

The external consistency 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 external consistency managing 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 for contention-based systems and Base Station for scheduled-based systems. 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 external consistency manager is also capable of assigning priorities to modules to perform specific actions or decide which request is allowed to take precedence over another.

In the external consistency verification 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 consistency manager. An illustration for such a scenario is depicted in Figure 10, 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 internal consistency check function and decides to apply or reject the requested configuration change.

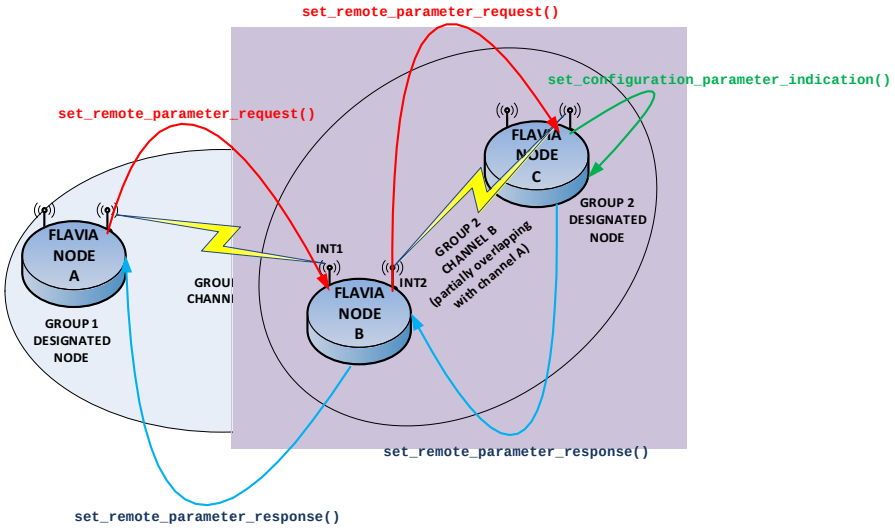

Figure 10 External consistency managing use case

The general functionalities of the consistency managing function are the following:

- 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.

Basically the consistency manager relies on system state data, registered events or service configuration parameters collected and exposed by the IB 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 the consistency manager to take a required optimizing action in order to resolve inter-nodes configuration inconsistencies.

5.2.4 Capability negotiator

The *Capability negotiator* manages service and function definitions and configurations among different nodes. When two services (on different nodes) want to communicate each other, a set of consistency checks must be performed in order to determine their compatibility. In fact, it might happen that a given service does not find a peer on a remote node or finds a peer based on a different logic/version. These problems do not obviously exist in case of traditional technologies, for which the inter-service consistency is natively guaranteed by the adoption of common standardized protocols.

In order to enable the communication between customized services, the FLAVIA architecture relies on the negotiation of service logic and service configurations among the network nodes. These operations can be carried out on the basis of the node

capabilities. The capabilities of a given FLAVIA node are defined as the whole set of available system resources, including hardware resources (e.g. commands for configuring number of antennas, receiver sensitivity, modulation and coding schemes) and service/function modules. While hardware resources depend on the underlying platform and cannot be updated, new service and function modules can be added at each node, provided that the node supports the commands on which the modules are built. Therefore, we assume that in principle different negotiation solutions can be considered: from a simple convergence towards a common pre-loaded service logic (*passive negotiation*), to a distribution of a customized service module from one node to the other ones (*active negotiation*).

We also assume that all the nodes support a minimum set of capabilities that are used for defining a default communication service, such as a standard technology stack (e.g. 802.11/802.16). Such a common service can be used for exchanging information on the additional node capabilities, thus enabling the launch of enhanced services.

Consider two independent nodes A and B, whose hardware capability sets are C_A and C_B. If the two sets coincide, each service running on A or B can be connected to a peer service on the other node (provided that the service is loaded and started). Conversely, if the two sets are different, the negotiation has to only consider the services and functions supported by the hardware resources $C_A \cap C_B$. Different negotiation solutions can be pursued on these modules:

- both nodes agree to rely only on the common services and functions;
- both nodes agree to rely on node B services and functions and node A acquires from node B the missing modules;
- both nodes agree to rely on node A services and functions and node B acquires from node A the missing modules;
- both nodes acquire from the peer node the missing modules, thus enabling all the available services built on the common hardware capabilities.

We are mainly interested in the first solution, since module distributions among the nodes create some security vulnerabilities that are currently out of the scope of the project. Therefore, the main functionalities of the capability negotiator are:

- identifying the hardware capabilities of the network nodes;
- selecting services and functions compatible with the hardware capabilities of all the network nodes.

5.2.5 Virtual MAC monitor

The virtual MAC monitor is responsible of managing multiple virtual hardware interfaces. Its main functionalities are:

- creating/removing virtual interfaces;
- slicing the hardware resources among the active interfaces;

- controlling execution of virtual services and functions running on different virtual interfaces.

The U interface includes the functionalities required for programming the virtualization policies according to the user/developer needs. Different slicing and control schemes on hardware resources and service executions can be specified and managed by this architecture component. Details on this architecture component are discussed in section 6.

Control modules	Description
Command parser	operates control on system consistency problems (inter-nodes, intra-node);
Service manager	provides a list of all the available services; runs new services; modify/deletes existing services; build new services (functions);
Consistency manager	operates control on system consistency problems (inter-nodes, intra-node);
Capability negotiator	gives information about resources usage; manages compatibility issues between nodes; allows code exchange between nodes;
Virtualization monitor	manages virtual interfaces;

Table 9: Summary of Control Manager sub-components

5.3 Information Base

The *Information Base* is the other main component of the FLAVIA Control system. It is responsible of collecting, organizing and exposing different data types for the various (real or virtual) wireless interfaces and for overall MAC. It specifically deals with system states and performance parameters.

In particular, three different data can be grouped:

- low level data, which are obtained by monitoring hardware and physical layer parameters (e.g. channel busy time, received signal strength, number of failure on frame checksums);
- system state data, which include the (time-varying) settings of all the tunable parameters of the system (e.g. modulation and coding scheme, transmission power, header formats);
- system functional data, which represent the list of commands, functions and services available in the system.

Each data type is managed by a dedicated sub-component:

- the low level data collector, which acts as a monitoring module interacting with the wireless processor, on which different data aggregation and filtering operations can be defined;
- the data gateway, which manages multiple accesses on the system state parameters and works in conjunction with the consistency manager;
- the functional data manager, which works on the service/function database, for tracking and saving the modules available in the system.

The information base exposes the *Repository Interface (REP)* to allow the other system components reading and/or writing the stored data.

5.3.1 Data Collector

For the data collector, we envision exploiting the *Hardware Abstraction Interface* for defining a minimum set of hardware parameters and signals which are stored as low-level data. Different polling or event-based data reading schemes can be defined, by taking into account the constraints imposed by the hardware features (such as the minimum polling time, the hardware measurement quantization, etc.).

5.3.2 Data Gateway

The data gateway enables the data sharing among FLAVIA modules and it is responsible for managing the conflicts arising when multiple FLAVIA modules work on the same system state data.
The data can be bounded to a specific wireless interface (wif), that can be real or virtual, or they can have a system wide visibility to allow inter-interface storing. Thus the data gateway acts on each data repository independently.

The data gateway assumes that all the modules know all the data structures representing the system state. Moreover, each data field is bound to a different Unique Identifier (UID), which is notified to all the running modules. The main methods for reading a data value or to be automatically notified about a data value update can be envisioned as follows:

- *get_data*(UID,wif): data.
  This method allows a module to read the data value corresponding to the given identifier on the specific wireless interface or on the system repository if the interface is not provided.
- *on_change_listener*(UID, change_listener,wif): outcome
  This method allows a module to register a change listener related to a given data field on the specific wireless interface or on the system repository if the interface is not provided.

We consider two different solutions for storing data: single writer protection and multiple writers with challenge protection.

Single writer protection is used when there is a single module writing a given data filed and many modules interested in reading such a field. A typical scenario can be the publication of some statistics. The writing module initializes the data value and receives a Modification Identifier (MID) by the Data Gateway to be used for updating the data field or removing the data. The main methods can be represented as follows:

- *create_data*(UID, data value, wif): MID;

This method allows to initialize a data field by using UID bound to the data. It returns the MID to the initializing module. The method acts on the repository associated to specific wireless interface or on the system repository if the interface is not provided as parameter.

- *change_data*(MID, data value,wif): outcome;

This method allows to set a new value for the data field identified by the MID, proving that the module is authorized for such an operation. The method acts on the repository associated to specific wireless interface or on the system repository if the interface is not provided as parameter.

- *clear _data*(MID, wif): outcome;

This method allows the initializing method to remove a data field. The method acts on the repository associated to specific wireless interface or on the system repository if the interface is not provided as parameter.

Multiple writers with challenge protection is used when a data has to be shared among multiple modules. In this case, all the system modules can read and write data. However, optionally, a module can register one or more *consistency_verification* operations, to be verified before any data value updates. The Data Gateway calls sequentially all the *consistency_verification* operations registered at each attempt of changing the data value. If all the consistency tests are passed, the old value is replaced by the new value. The main methods for supporting the above behavior can be the following:

- *set_value*(UID, data value, wif): outcome;

This method enables all the system modules to set a data field not supporting the single writer protection. The method acts on the repository associated to specific wireless interface or on the system repository if the interface is not provided as parameter.

- *delete_data*(UID, wif): outcome;

This method enables all the system modules to remove a data field by using the UID. The method acts on the repository associated to specific wireless interface or on the system repository if the interface is not provided as parameter.

- *protect_data*(UID, consistency_verification, wif): outcome;

This method allows to register a consistency verification operation for a given data field. The method acts on the repository associated to specific wireless

interface or on the system repository if the interface is not provided as parameter.

5.3.3 Functional Data Manager

The Functional Data Manager organizes and interrogates the database containing the information about the system functional resources. At this design stage, we envision storing a minimum set of fields for each functional resource:

- resource name: the name of the service, function or command to be invocated;
- resource interface: the list of basic or advanced data to be passed to the functional resource;
- resource state: the resource availability state that indicates if the resource can be immediately invocated (*running*) or if it has to be loaded (*unloaded*), initialized (*loaded*), or started (*initialized*);
- resource dependencies: the list of other resources called by the current one;
- resource advanced data: the aggregation of basic data in data structures used by the current resource;
- resource consistency conditions: the list of consistency tests related to the system state data affected by the current resource.

Information Base modules	**Functions**
Low level data collector	collects statistics about low-level hardware/PHY parameters
Data gateway	Manages multiple accesses to the same data fields and conditions on data consistency
Functional data manager	provides an interface for accessing the system resource databases

Table 10: Summary table of Information Base modules and their functions

6 FLAVIA Architecture support for Virtualization

FLAVIA's virtualization is based on the execution of **multiple MAC functions on each virtual interface**. Since in our architecture the medium access function is supported by the wireless MAC processor, uch a virtualization scheme implies the possibility to execute parallel medium access threads on the wireless processor. The wireless processor virtualization has two main aspects to be considered: i) sharing the wireless processor itself, and in particular the MAC execution engine; ii) sharing the physical layer and the wireless medium.

As far as concerns the first aspect, we propose some solutions already consolidated for CPU virtualizations, which exploit the WMP extended capabilities to support code-switching. The second aspect is more critical because our architecture has a limited control on the physical layer. Specifically, some physical layer operations, such as frame transmissions and receptions, cannot be stopped and resumed, as required by most virtualization solutions. Therefore, we propose to switch to the next MAC thread regardless of the hardware state. Explicit signals indicating hardware conflicts are included in the API to be treated as normal 'busy' signals in the definition of the MAC programs.

6.1 Virtualization primitives

The definition of the virtualization primitives is based on the following assumptions:
- the physical layer offers the functionality of encoding/decoding frame, i.e. the signal processing elements are hardware based and controlled by parameters through interface to the hardware; these parameters are obviously related to the modulation method (elements such as frequency, rate, modulation method);
- a slicing method of the physical layer is available (time, frequency, code);
- the wireless MAC processor is running in a similar fashion to a real time operating system (RTOS): the embedded operating system guarantees a capability within a specified time constraint, i.e. executing one action in a defined timeframe, with success or not (hard real time). This is a requirement for some MAC protocols operations such as ACK.

With the previous requirements, we redefined the following RTOS concepts to be suited to the wireless MAC processor:

- *Entities*: entities are the MAC protocols running inside the wireless processor;
- *Scheduling*: the wireless processor is responsible for creating a schedule of how contending access to the physical layer will be performed by the entities, i.e. how MAC protocols will access the wireless medium;
- *Dispatching*: the wireless processor is responsible for granting access to the most eligible contending entity;
- *Eligibility:* is the position of an entity in a schedule or other parameters such as priority and deadline;

- *Timeline*: entities or set of entities can have a set of timeline i.e. they have to respect a precise timeline dictated by the MAC protocols;
- *Specifications*: each timeline has two level of specification: the first level is the completion time constraint (deadline) i.e. a packet need to be sent before time X; the second level of specification is how the timeline should be handled in terms of conflicts, i.e the packet can be rescheduled later, retransmitted etc…;
- The *completion time* is a predicate depending on the logic of the MAC layer deployed in the wireless processor; i.e. depending on the selected physical modulation, a certain amount of transmission time has to be accounted, as well as receiving, processing time, etc.;

Code switching functionalities have been added to the WMP architecture, for assisting virtualization operations and pursuing a form of **hardware-assisted virtualization**. Specifically, we assume that a whole MAC program is associated to a virtualization state (READY, RUNNING, BLOCKED) and that the following transitions are possible:

- READY to RUNNING: Wireless processor is released by current MAC protocol, which starts waiting for an event; highest priority MAC protocol starts running.
- RUNNING to READY: Event for another MAC protocol with high priority occurs. It preempts running MAC, which is inserted in list of ready MAC protocol.
- RUNNING to BLOCKED: MAC protocol starts waiting for an event. Processor is assigned to the next ready task;
- BLOCKED TO RUNNING: Event for MAC occurs, which has higher priority than running task : it preempts it.
- BLOCKED to READY: Event for MAC occurs. MAC has lower priority than running current MAC, it is inserted in the list of ready tasks.

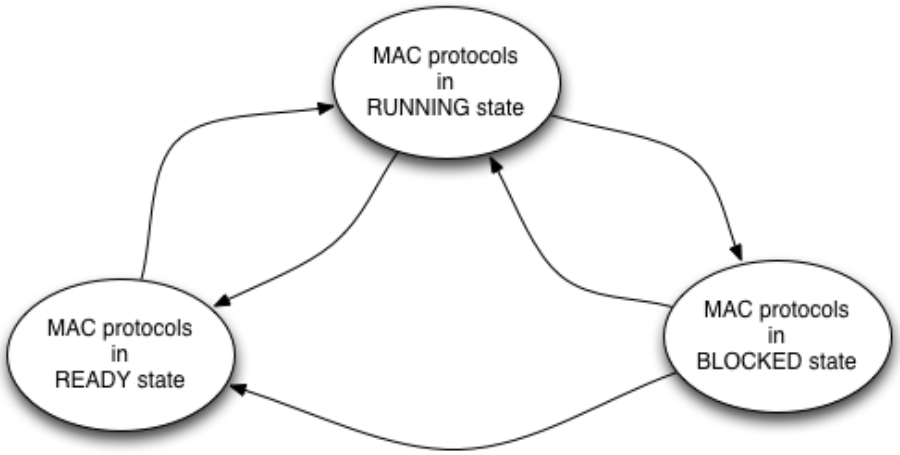

Figure 11 : State of the MAC algorithms within the wireless processor

6.2 Virtualization MAC Monitor

The Virtual MAC Monitor allows exposing multiple virtual MAC Engines, which can independently run their guest MAC machines and relative upper network application. It is a component similar to the Hypervisor or Virtual Machine Monitor in classic computer virtualization. It is the responsible to manage and control the access to the hardware (i.e. the transceiver) and distribute it among the guests. It is also responsible for presenting to the guests the corresponding virtual engines, as shown in Figure 12.

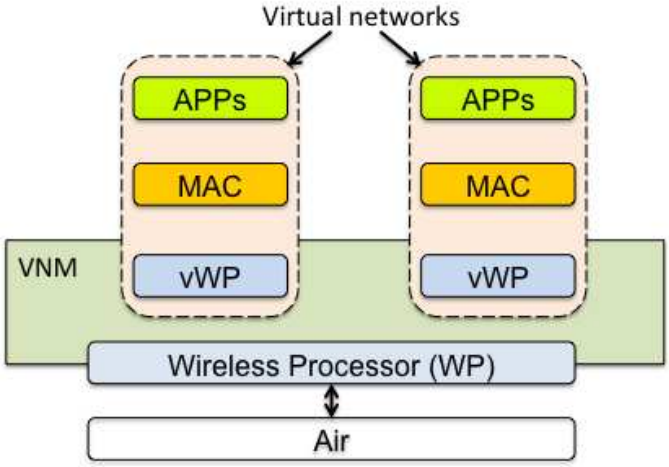

Figure 12 Ideal NIC virtualization

For solving the above issues, we consider two different approaches: i) a classical time-based mechanism that univocally assigns the hardware to a single virtual entity within a given time interval; ii) a virtual collision mechanism according to which multiple accesses to the hardware are resolved (as in case of actual collisions on the medium) following the rules of the guest MAC schemes.

Time slicing. In this case, the virtual MAC Monitor works by enabling the MAC Engine to perform code switching. To this purpose, it creates a virtualization Finite State Machine (also called meta-MAC) in which the virtual state represent the identifier of the guest MAC machine under execution (i.e. the thread), and the transitions define the thread scheduling by means of triggering events for moving to the next thread. The transitions can be programmed on the basis of events and conditions supported by the hardware (packet receptions, time expirations, etc.). For example, as shown in Figure 13, the implementation of a preemptive multi-tasking scheme can be supported by defining cyclic (round robin) transitions between the threads, triggered at the expiration of a fixed interval timer. Note that this mechanism can be generalized and multiple transitions can be specified from a given MAC thread. Each

transition can also include some meta-MAC actions (such as the reset of a timer) to be performed before scheduling the next transition events. Thread state and relative configuration registers are saved for the next execution.

Although this approach apparently avoids conflicts among virtual entities, since at a given time instant a single MAC is executed, it does not explicitly account for the hardware reaction times. In other words, it may happen that a transmission action is still ongoing at the expiration of the thread execution timer. We chose to switch to the next thread regardless of the hardware state (which will be eventually revealed as busy during the incoming thread execution).

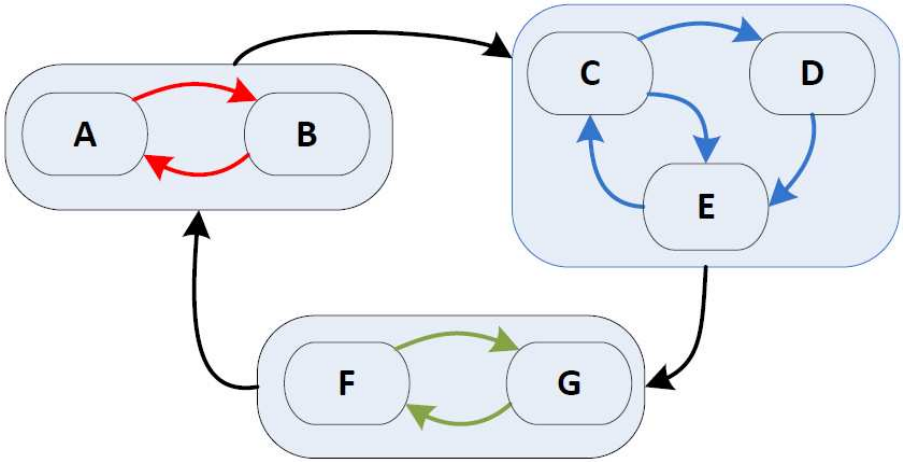

Figure 13- Meta-MAC machines following the time-slice approach

Virtual Collision. In this case, the virtual MAC Monitor works by enabling parallel MAC machine executions. The multi-thread execution is supported by creating a virtualization Finite State Machine obtained by the product of the guest MAC machines (as shown in Figure 14). Being *n* the number of active virtual entities, the resulting n-dimensional state simultaneously tracks the reactions of each guest MAC to the same hardware events. The virtual MAC Monitor is also responsible of addressing n pending frames (corresponding to the head-of-line of each virtual entity) and pushing new frames when the virtual entity removes the pending one (because of successful transmissions or packet drops). It also stores *n* copies of all the card configuration registers, tracking the tunings performed by each virtual entity.

Although this approach makes possible executing multiple MAC machines in parallel, it introduces potential conflicts in accessing and configuring the hardware. In fact, at a given time instant, the hardware resources are not exclusively allocated to a specific virtual entity and sequential actions involving hardware settings or transmission actions could overlap. To deal with this problem, the virtual MAC Monitor generates virtual collision signals when an action on the hardware is triggered while the hardware is busy. If two or more entities need to access the hardware simultaneously,

the virtual collision signal is generated according to a precedence indicator specified for each MAC entity (or randomly if the MAC entities have the same precedence). The hardware is configured according to the registers of the entity winning the virtual contention.

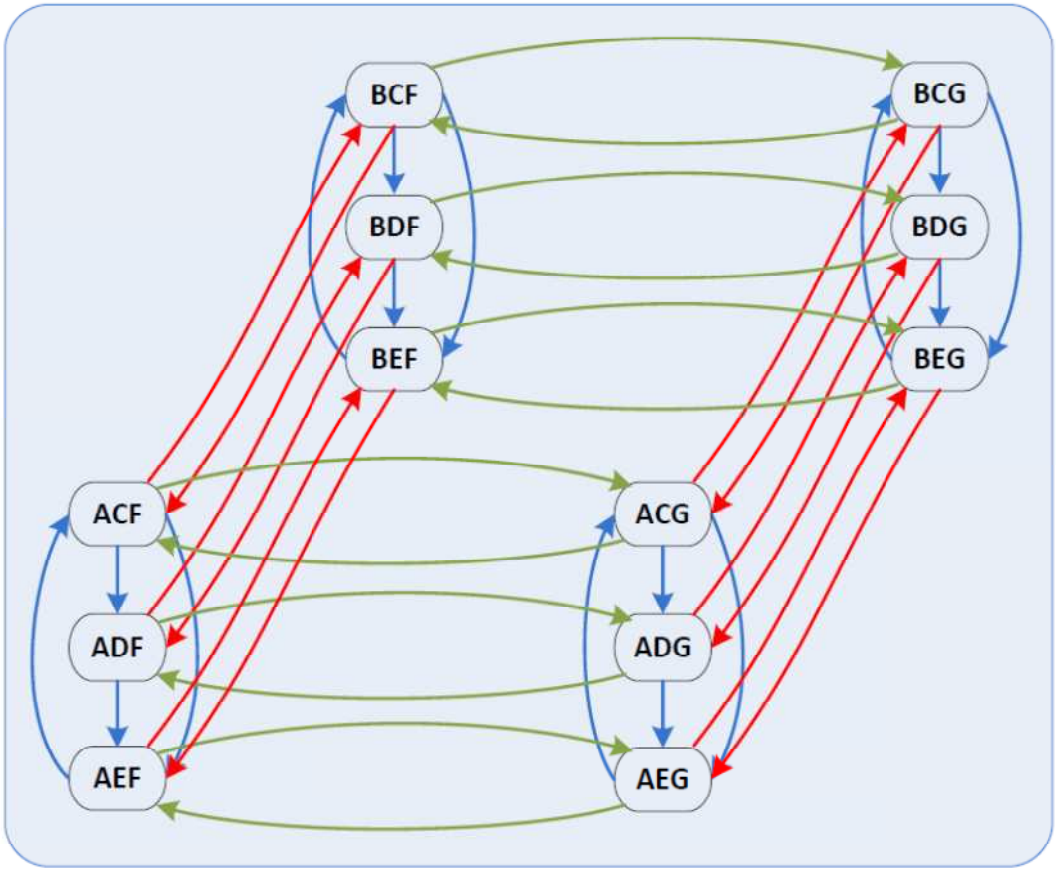

Figure 14 – Meta-MAC machines following the virtual collision approach

6.3 Virtualization examples: Super-frame Building

The virtualization primitives available on the FLAVIA architecture allow to easily implement a super-frame in which the two access schemes are employed in different time intervals (similarly to the coexistence of contention-free and contention access periods in 802.11 networks). The super-frame can be immediately organized by the Virtual MAC Monitor, by defining a virtual MAC-machine moving from an access scheme to another one at regular timer expirations (similarly to Figure 13, where a meta-MAC state may embed the DCF state machine and another one the TDM state machine, and two transitions from one state to another may correspond to timer

expirations set according to the contention-free and contention periods). The thread switching could be also triggered by the reception of a special control frame.

Conversely, in case we want to run the pseudo TDM scheme in parallel with DCF, similarly to the recent standardized MCCA access scheme, we can use the virtual collision approach, by giving priority to the TDM allocations (similarly to Figure 14). Although the machine product can generally result in a complex state machine, in many cases simplifications based on state aggregations are possible.

7 Conclusions

This document provides a description of the FLAVIA system, built on the basis of the requirements emerged during the analysis of the operation scenarios proposed in D211 and on the basis of the feedback provided by the prototyping activities.
The final architecture is based on the following main components:

- the **wireless processor**, which has the critical role of handling hardware events and schedule actions on the hardware;
- the **function container**, which provides the running instances of the functions available for defining customized services;
- the **service container**, which allows to instantiate services dealing with different MAC sub-tasks and to expose application interfaces towards the higher layers;
- the **information base**, which organizes different (shared) data types, including both system-state data and system-performance data;
- the **control module,** which orchestrates the overall configuration and behavior of the system, by enabling function loading and service definition, and by solving conflicts arisen by concurrent services.

Each component has been described under different architecture views, focused on the accomplishment of system modularity, flexibility and virtualization. A complete list of primitives has also been identified for both the functional sub-system (i.e. the architecture components responsible of implementing the system behavior) and the control sub-system (i.e. the architecture components responsible of configuring the protocol stack and verifying consistency). Since the early prototyping activities of the project have been mainly focused on the functional sub-system, the wireless processor, the function container and the service containers have been described with a higher level of details, taking into account the prototyping experience.

8 References

[Orbit] D. Raychaudhuri, M. Ott, and I. Seskar. Orbit radio grid tested for evaluation of next-generation wireless network protocols. In proceedings of IEEE TRIDENTCOM, pages 308–309, 2005.

[PlanetLab] L. Peterson, S. Muir, T. Roscoe, and A. Klingaman. PlanetLab Architecture: An Overview. Technical Report PDN–06–031, May 2006.

[EmuLab] Mike Hibler and Robert Ricci and Leigh Stoller and Jonathon Duerig and Shashi Guruprasad and Tim Stacky and Kirk Webby and Jay Lepreau. Large-scale Virtualization in the Emulab Network Testbed. In USENIX, 2008.

[GENI] GENI. http : //www.geni.net/.

[MobSum2011] P. Gallo, F. Gringoli, I. Tinnirello. On the Flexibility of the IEEE 802.11 Technology: Challenges and Directions. Mobile Summit 2011.

[INFOCOM2012] I. Tinnirello, G. Bianchi, P. Gallo, D. Garlisi, F. Giuliano, F. Gringoli, "Wireless MAC Processors: Programming MAC Protocols on Commodity Hardware", IEEE INFOCOM 2012, Orlando (FL), USA, March 25-30, 2012.

[GENIwirelessvirt] S. Paul and S. Seshan. Virtualization and Slicing of Wireless Networks. GENI Design Document 06-17.

[D211] All FLAVIA partners, ”Report on Scenarios, Services and Requirements”, Deliverable D2.1.1, ICT, FLAVIA, January, 2011; available at http://www.ict-flavia.eu

[D212] All FLAVIA partners, “Revisions of Report on Scenarios, Services and Requirements”, Deliverable D2.1.2, ICT, FLAVIA January, 2012; available at http://www.ict-flavia.eu

[D221] All FLAVIA partners, “Architecture Specification”, Deliverable D2.1.2, ICT, FLAVIA, April 2011; available at http://www.ict-flavia.eu

[D311] ALVARION, IMDEA, BGU, NEC, SEQ, CNIT, “802.16 architecture and interfaces specification”, Deliverable D3.1.1, ICT, FLAVIA, July 2011; available at http://www.ict-flavia.eu

[D411] IMDEA, CNIT, NEC, TID, MOBIMESH, BGU, IITP, NUIM,”802.11 architecture and interfaces specification”, Deliverable D4.1.1, ICT, FLAVIA, July 2011; available at http://www.ict-flavia.eu