MAClets: Active MAC Protocols over Hard-Coded Devices

Giuseppe Bianchi
CNIT
Universitá degli Studi
di Roma Tor Vergata
Italy
giuseppe.bianchi@uniroma2.it

Pierluigi Gallo
CNIT / Universitá degli Studi
di Palermo - Italy
pierluigi.gallo@unipa.it

Domenico Garlisi
CNIT / Universitá degli Studi
di Palermo - Italy
domenico.garlisi@unipa.it

Fabrizio Giuliano
CNIT / Universitá degli Studi
di Palermo - Italy
fabrizio.giuliano@unipa.it

Francesco Gringoli
CNIT / Universitá degli Studi
di Brescia - Italy
francesco.gringoli@unibs.it

Ilenia Tinnirello
CNIT / Universitá degli Studi
di Palermo - Italy
ilenia.tinnirello@unipa.it

ABSTRACT

We introduce MAClets, software programs uploaded and executed on-demand over wireless cards, and devised to change the card's real-time medium access control operation. MAClets permit seamless reconfiguration of the MAC stack, so as to adapt it to mutated context and spectrum conditions and perform tailored performance optimizations hardly accountable by an once-for-all protocol stack design. Following traditional active networking principles, MAClets can be directly conveyed within data packets and executed on hard-coded devices acting as virtual MAC machines. Indeed, rather than executing a pre-defined protocol, we envision a new architecture for wireless cards based on a protocol interpreter (enabling code portability) and a powerful API. Experiments involving the distribution of MAClets within data packets, and their execution over commodity WLAN cards, show the flexibility and viability of the proposed concept.

Categories and Subject Descriptors

C.2.1 [**Computer-Communication Networks**]: Network Architecture and Design—*Network communications, Wireless communication*

General Terms

Design, Experimentation, Management

Keywords

programmable MAC; WLAN 802.11, reconfigurability; cognitive radio

CoNEXT'12, December 10–13, 2012, Nice, France.

1. INTRODUCTION

It was the end of the last century since cognitive [1] and active [2] wireless networks were considered near to come. The promise was that wireless networks would have soon become capable of being dynamically reprogrammed so as to best fit unpredictable and dynamically mutating context situations; adapt to user preferences, usage patterns, or service demand variations; smartly exploit temporarily unused radio spectrum, etc.

The aftermath is that such a vision is still fighting with the limited flexibility of *commercial* wireless products and standards. Whereas, in the last decade, the scientific community emerged with open-source, fully programmable DSP, FPGA, or SDR platforms, most wireless manufacturers have consistently pursued a closed products' design strategy, which reduces programmability to the tuning of some pre-defined device parameters. On the other side, although many standards include different configurable operation modes, they are designed to be comprehensive, *one-size-fits-all* and may fail to be effective in specific niche contexts or particular network situations.

With specific reference to the Medium Access Control functionalities, on which this paper is focused, flexibility has been so far accomplished by standardizing a set of configurable parameters, as well as the means to dynamically signal, control, and enforce the relevant settings. Even the 802.22 MAC layer, specifically defined for cognitive networks, is based on *pre-defined* transport modes (combining polling, contention and unsolicited bandwidth grants mechanisms) that can be selected according to the service requirements [3]. Another example is the case of the 802.11 QoS (EDCA) extensions, originally standardized in the 802.11e amendment, which permits dynamic configuration, via beacons, of the key parameters characterizing each traffic category, namely contention windows, transmission opportunities, and arbitration inter frame spaces. Such parameters can then be exploited to optimize performance for dynamically varying context conditions such as number of competing stations [4, 5].

The current solutions prevent to extend configuration facilities to new MAC parameters or functions, unless a rele-

vant standard amendment is first approved and then adopted by manufacturers. With this paper, we aim to show that a radically different approach allowing wireless devices to dynamically download and install *on-the-fly* a full protocol logic (rather than a set of parameter settings), customized for specific contexts and conditions, is technically feasible even in a multi-vendor scenario. As a basic use-case, imagine a terminal associating to an Access Point in a specific context, for instance an home or an airport. The network provider has programmed a number of custom MAC protocols, each optimized for the specific context and for a different operation or traffic/service condition. Upon association (or upon changes in conditions), the terminal downloads a new MAC protocol from the AP, and starts using it. This network management approach (which makes concrete, at least for what concerns the MAC protocol stack, the futuristic vision described in [6]) is not anymore hindered by the reliance on standards; it suffices that all terminals support i) a "default" protocol operation, for instance WiFi, for first communicating with the AP, and ii) an architecture which permits the real time programming of the MAC protocol, and which runs a same software on different hardware platforms, *say something as Java for wireless MAC protocols* (adapting a quote from [6] - indeed this justifies the name "MAClet" we use throughput this paper). Note that in a single vendor scenario (or in an idealistic scenario where all clients are open source devices), this would trivially reduce to load a new firmware in the wireless cards. Rather, the challenging real world hurdle to overcome is how to re-program, or, even better, dynamically reconfigure in real time the MAC protocol operation, under the constraint that such reconfiguration is done in a way that does *not require* vendors to open their platforms.

Our contribution

In order to program MAC protocols on closed devices, suitable Programming Interfaces are mandated. This problem has recently received crucial attention, and the identification of a MAC programming interface appears to emerge (explicitly or implicitly) from a number of recent works [7, 8, 9, 10], where a number of common primitives (core functions in [7], modules in [8], components in [9], actions/event/conditions in [10]) have been proposed for assembling different MAC schemes in an high-level programming language.

This paper leverages, as starting point, our prior findings described in [10], where we proposed to formally describe a MAC protocol through extended finite state machines, executed by a *Wireless MAC Processor* (WMP) architecture running inside the wireless card. The further steps taken in the present work involve both technical and application aspects. From the technical side, several extensions to the framework described in [10] were necessary to support real-time code switching (and MAC multi-threading) with negligible (sub microseconds) delay. On top of the extended WMP architecture, we developed a control framework for supporting MAClet code mobility, i.e. for moving, loading and activating MAC software programs embedded into ordinary data packets (akin to traditional active networks' *capsules*) along with relevant meta-data such as initial state and startup conditions. From the application side, this paper takes a completely different perspective with respect to [10]: from the *node-level* ability to run different MAC protocols, to the *network-level* perspective of managing and dynamically reconfiguring, in real-time, the network, well beyond the parameter-based reconfiguration of today WLAN deployments [11, 12]. The network-level reconfiguration capabilities of the framework have been validated in two interesting use-cases examples (including dynamic spectrum access and support of virtual operators using different MAC protocols over a same, time-shared, infrastructure).

Our contribution focuses on the actual MAC stack reconfiguration via dynamically delivered MAClets. We *do not* claim any contribution on i) the strategies for disseminating mobile code, being many already available in the literature, and on ii) the measurement gathering, learning and decision processes by which a reconfiguration is triggered, being these tasks mostly orthogonal to the "act phase" perspective (using cognitive networking terminology) we tackle in this paper.

2. WIRELESS MAC PROCESSORS

In what follows we briefly review the WMP main concepts [10] and anticipate some discussion on extensions for supporting code switching. Indeed, a pre-requirement of *any* wireless active MAC framework is the ability to support customized MAC operation on general-purpose wireless devices, and the possibility to switch to a desired MAC protocol logic described through suitably formal languages and application programming interfaces.

Concept

The Wireless MAC Processor architecture somewhat 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.

Instruction set: Actions, Events, 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 field, 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 may be considered, so as actions not necessarily restricting to MAC primitives (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 the 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. In [10] we proposed an API able of supporting several MAC behaviors, including a TDMA-like and a multi-channel medium access control. However, this API was not envisioned for supporting code mobility. 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 the WMP *internals* to implement an extended API accounting for new actions, events and registers, tailored to dynamic code management.

Programs: Extended Finite State Machines

MAC protocols are well suited to be described in terms of Finite State Machines. Indeed, they are used in the formal appendices of the 802.11 (and many other) standard. We chose to rely on the more powerful and expressive model of eXtended Finite State Machines (XFSM). XFSMs are a generalization of the finite state machine model and permit to conveniently control the *actions* performed by the MAC protocol as a consequence of the occurrence of *events* and *conditions* on configuration registers.

An XFSM is formally specified through an abstract 7-tuple (S, I, O, D, F, U, T): the meaning of such symbolic states and the correspondence with the MAC terminology above introduced is summarized in Table 1 (configuration commands being a special case of *actions*, devised to update registry status).

A *MAC program* is simply a table listing all possible state transition relations. Note that the number and meaning of the set of *protocol states* is specified by the programmer. By formally describing, per each protocol state, which events and conditions do trigger a state transition, and by associating actions and configuration commands to each state transition, the programmer may access the available hardware primitives, and enforce a desired MAC behavior within the radio hardware. Since the configuration memory is not explicitly represented in the state space, XFSMs allow to model complex protocols with relatively simple transitions and limited state space. For an example, the table *programming* the legacy 802.11 Distributed Coordination Function MAC protocol is coded in less than 600 bytes, and hence *can be transmitted in just a single packet.*

XFSM formal notation		**meaning**
S	symbolic states	MAC protocol states
I	input symbols	Events
O	output symbols	MAC actions
D	n-dimensional linear space $D_1 \times \cdots \times D_n$	all possible settings of n configuration registers
F	set of enabling functions $f_i : D \rightarrow \{0, 1\}$	Conditions to be verified on the configuration registers
U	set of update functions $u_i : D \rightarrow D$	Configuration commands, update registers' content
T	transition relation $T : S \times F \times I \rightarrow S \times U \times O$	Target state, actions and configuration commands associated to each transition

Table 1: MAC programs expressed as Extended Finite State Machines

CPU: 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 a memory space deployed on the hardware. 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 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.

Multi-thread extensions

The MAC engine work-flow allows to easily support *code switching*: in analogy to usual multi-threading, this entails the time-shared execution of *different MAC programs* simultaneously uploaded on the device. Indeed, the MAC engine does not need to know to which MAC program a new fetched state belongs, so that a code switching is basically achieved by moving to a state in a different transition table and by updating the platform configuration registers (e.g. the operating channel, the transmission power, etc.) when needed. The definition of code switching transitions are logically independent of the MAC program definition. Therefore, rather than adding them to the MAC program, we chose to program the switching transitions into a second-level state machine (*meta state machine*), whose states represent the MAC program under execution.

To simplify the management of error conditions due to the dynamic loading of multiple transition tables, we assumed that each table can be loaded starting from a pre-defined memory position, called memory slot. For each memory slot, a dedicated register describes the state of the slot as empty, available for re-writing, ready to be executed, or under execu-

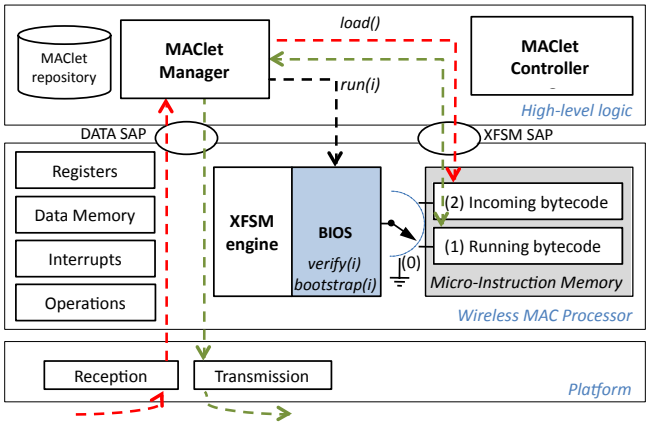

Figure 1: Architecture for MAClet support: extended WMP and external MAClet Control.

tion. Although the number of these slots can be in principle arbitrary, we consider the simplest case of two slots only [1].

Summarizing, the time needed to *change* a MAC program (i.e. the MAC stack reconfiguration delay) consists in the time needed to fetch a new state plus the time needed to update the WMP configuration registers (i.e. a series of memory accesses). All this accounts for a marginal, sub-microseconds, time (a few MAC engine clock cycles, the exact time depending on the platform's clock frequency and on the number of registers to be updated).

3. MACLET CONTROL ARCHITECTURE

In this section we describe how low-level MAC functionalities can be *encapsulated* in a MAClet and transferred from a node to another of the network by exploiting the WMP API and a MAClet distribution protocol.

Figure 1 shows the envisioned system: the control architecture is a *pure* software architecture, running at the application level, that interacts with the enriched WMP by means of an open control API. This approach has several advantages. First, the selection of the MAC protocol can be based not only on low-level performance parameters (such as the link quality, the interference conditions, etc.), but also on high-level context estimates, including the application requirements, the network topology, the user preferences, and so on. Second, the code distribution model (handshaking mechanisms, peer-to-peer code sharing, server-client uploading, etc.) is completely independent on the underlying programmable interface, thus allowing full flexibility and a wide range of applications for the same platform. Moreover, the communication delays between the host and the card have a minimum impact on the MAClet Control, since the dynamics of the networks (which require MAC protocol customizations)

[1]Such a choice has been also confirmed by the analysis of different use cases requiring prompt adaptations of the MAC operations. We never found an actual need to switch between more than two protocols at a frequency so high to require the simultaneous loading of multiple transition tables on the device.

WMP Control Interface	
load i	load a MAC program on memory slot i
run i, e	activate MAC program on slot i (asynch. or at the event e)
verify i	recognize trusted code by means of an hard-coded signature computation
switch $i, j, t, a/r$	add or remove a switching transition t from the slot i to j

Table 2: WMP Commands to be locally or remotely invoked

are reasonably much slower than the processing delay due to an application-level decision module.

More into details, the architecture is based on four main components: the WMP control interface, the MAClet manager, the MAClet Controller and the MAClet repository. The WMP control interface is the interface to the hard-coded device, through which new MAC state machines and switching conditions are loaded on the card, as summarized in table 2. The MAClet manager is responsible of receiving/transmitting MAClets and MAClet protocol messages, enabling the loading on the card, and programming MAC reconfigurations. The MAClet Controller is the intelligent part of the system, dealing with the network-level configuration decisions in a centralized way (e.g. at the Access Point only, as assumed in this paper), or in a distributed way (e.g. by involving multiple cooperating controllers, sharing both the monitored data and the available MAClet tables). Finally, the MAC program repository contains the MAClets available to the network operator, including either standard as well as customized (context-specific) MAC protocols.

3.1 MAClets

A key component of our architecture is the code transport unit, i.e. the MAClet. A MAClet is a *coded state machine* with an *initial state description* to be fed on the wireless device.

Being n_s the number of symbolic protocol states and n_e the number of events revealed by the device, a common approach for coding XFSMs is using a $n_s \times n_e$ table, where at each location (i, j) is stored the state transition when event j is received at state i. A transition is defined by a triplet (a, c, s), specifying the action label a, the enabling condition label c and the target state label s. As each state generally reacts to a number of input events much lower than the total input number, the state machine coding can be optimized by skipping null-transitions. The initial state (from which the state machine has to be run) includes the protocol logic state and the platform configuration registers. For example, according to the API defined in [10], these registers (of equal size) specify the settings of the physical channel, the slot size, the contention window values, the current backoff, the transmission power, the retry limit, generic protocol timers and MAC addresses to be filtered. Optionally, the initial state descriptor can be extended with a signed digest of the MAClet code to be used for verifying trusted code sources[2].

[2]Although most of the security issues can be demanded to the MAClet Control, this function can be used by manufac-

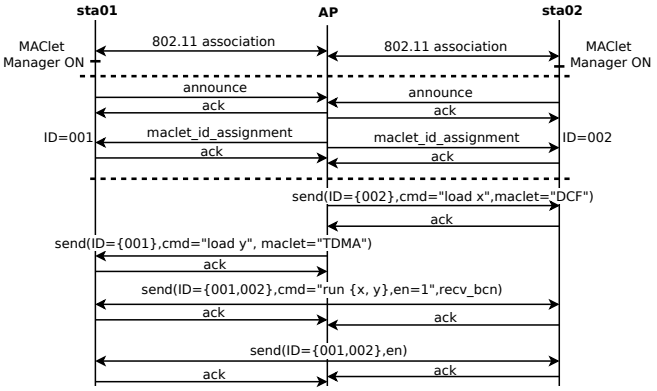

Figure 2: Messages of the MAClet Distribution Protocol: an example.

As detailed in what follows, MAClets are transmitted with a special message of the MAClet Distribution Protocol, called MAClet action message.

3.2 MAClet Distribution Protocol

MAClets can be propagated in the network by means of a *physical transport network*. This means that nodes can negotiate the activation of a new MAC protocol only if they belong to the same network (on a given channel) and employ a compatible MAC protocol. Standard MAC protocols can assume the role of default *common* protocols to be executed (eventually, on a pre-defined *common* channel) for supporting dynamic reconfigurations. We assume that the default protocol and configuration parameters are pre-loaded in each WMP as a *bios* state machine (e.g. in our implementation, the bios machine is a legacy DCF working on channel 1).

The MAClet Control process runs as a normal distributed application, whose messages are defined by a protocol called MAClet Distribution Protocol. This protocol is responsible of: i) collecting information for estimating the network context; ii) negotiating the network reconfiguration decisions; iii) transporting the MAClets and the relative activation signals; iv) verifying the network consistency after a reconfiguration. Although the general definition of the MAClet distribution protocols is out of the scope of the present work, we defined some core messages to be used in case of centralized decision processes. Specifically, in the following we always assume that the reconfiguration decisions are taken in a centralized way (as in [13, 14]) at the AP side, thus significantly simplifying the negotiation phase (which is basically limited to the transport of station measurements to the AP) and the verification of consistency.

The protocol includes two types of messages: MAClet *management messages* for associating each MAClet manager to a MAClet Controller running the distribution logic and confirming control operations, and MAClet *action messages* for transporting MAClets and remotely invoking the desired WMP control functions. When a new station activates, it tries to associate to an AP (acting as a MAClet Controller) by using the bios state machine. In case of success, the MAClet manager is activated for enabling the reception of AP messages. An announcement message is sent to the AP for notifying the activation of the new MAClet Manager and receiving an identifier. According to its decision logic, the AP is then able to send a specific MAClet action messages to each associated station, to a group of stations or to all the network stations (see figure 2).

The MAClet action message comprises the following fields: the list of destination addresses of the relevant MAClet managers, a command to be executed on the addressed WMPs, the MAClet bytecode, the MAClet configuration parameters, and the MAClet activation data. Not all fields are always included in the action messages: for example, it is possible to specify a new set of parameters for a MAClet already loaded on the station without carrying the relevant bytecode.

turers for controlling the MAC program origins and limiting or avoiding third-party reconfigurations.

3.3 MAClet Synchronization

Achieving a network-level reconfiguration is obviously much more complicated than working on a single node, because it is necessary introducing some forms of coordination. In particular, the activation of a new MAClet on different nodes could require a common reference signal for avoiding critical inconsistencies (such a temporary use of different transmitting channels) leading to disassociations or other network errors.

The MAClet Control Architecture provides the primitives for programming the desired synchronization and error recovery operations, but the specific solutions are left to the MAClet Decision Logic (synchronization) and MAC program (management of error conditions) defined by the network operator. For example, in section 5 we describe two different use cases requiring network-level reconfigurations, for which two different synchronization solutions have been envisioned. The synchronization signals can be based on the events and conditions available in the WMP and are specified in the MAClet activation data.

In order to activate a new MAClet on a group of stations, the AP sends a "run" action message to the stations list. If the command does not include an activation data field, each station can starts the program asynchronously, i.e. without a common reference signal. If present, the activation data specifies the triggering event that is usually a control frame sent by the AP or the expiration of a (relative or absolute) timer. While the relative timer is in turns expressed as a function of a network synchronization event (e.g. the next channel busy time), for using an absolute time reference the MAClet Control Process has to rely on a time synchronization function. In our infrastructure scenario, such a synchronization is easily provided by the beacon timestamps, while in general scenarios it has to be explicitly supported by the MAClet Distribution Protocol.

Different activation solutions based on a 3-way handshake mechanism can also be defined in the distribution protocol. After the reception of the run message, each station involved in the network reconfiguration sends a confirmation message. When the AP receives all the confirmation messages, it sends an enabling message. Only after the reception of this message, the stations switch to the new MAC program at the

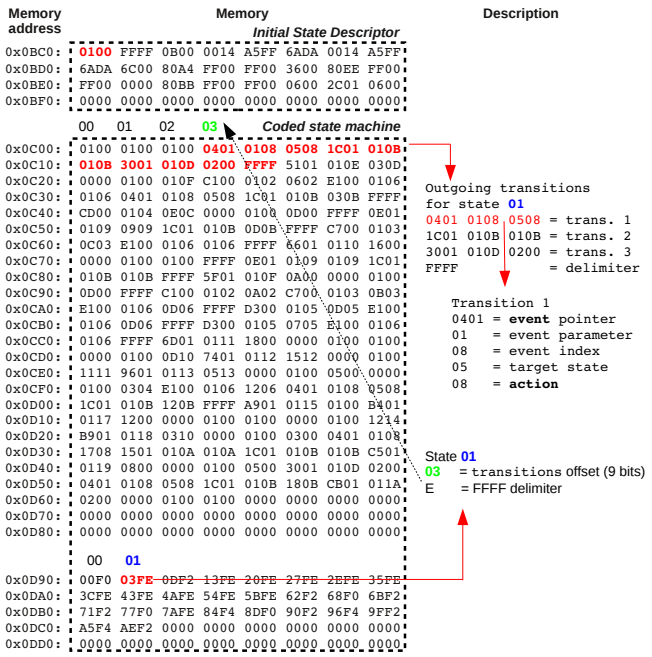

Figure 3: MAClet binary implementation, as stored in the micro-instruction memory

occurrence of the next triggering event. Figure 2 shows an example of messages exchanged between the AP and two stations for loading two different MAClets (a legacy DCF on station 2 and a TDMA protocol on station 1), whose activation is triggered by the first beacon received after the enabling message.

4. NODE-LEVEL VALIDATION

To prove te viability of the MAClet distribution framework, we worked into two different directions: on one side, we modified the MAC Engine and the actions implemented *within* our previous WMP implementation on the Broadcom card (i.e. at the firmware level), by also adding some new condition registers; on the other side, we developed a simple MAClet Control process (including the MAClet manager, repository and controller) at the application level (*outside* the card). For supporting the upper-MAC operations and interacting with the other protocol layers, we used the *b43* soft-MAC driver (without any modifications). Finally, we also extended the WMP machine language (i.e. the labels of actions, events and conditions) in order to code the new API.

4.1 Implementation

The implementation of the MAClet Control Architecture has been based on the development of a new firmware and an application-level software. Regarding the firmware, we developed and pre-installed the micro-code procedures corresponding to the WMP Control API (i.e. the *load*, *bootstrap*, *run*, and *switch* primitives - the *verify* command has not been currently implemented) and added new registers indicating the state of the program slots and the program under execution. We also worked on the MAC Engine work-flow,

MAClet Management Messages	
announce	sent by stations to request a MAClet ID
id_assign	sent by AP to assign unique MAClet ID
poll	sent by AP to check if a station is attached
en	enable command (requires activation message)
ack	message acknowledgment
MAClet Action Message Fields	
ID_set	MAClet_IDs addressed by the message
maclet	MAClet program or BYTECODE
params	set of MAClet parameters
cmd	MAClet command (load, run, en flag, dump, set timer)
activation	MAClet triggering event

Table 3: MAClet management and action messages.

for allowing the execution of the meta machines. Specifically, the new engine pre-fetches the switching transitions (defined in the meta-machine) of the program under execution and adds such a list to the transition list of the program state. In case of events triggering a transition to a new program, the transition action executed by the engine is the bootstrap of the new program. Finally, the WMP machine language has been revised for coding the new API and for trying to guarantee a compact representation of the MAClet bytecode.

Figure 3 shows an *actual* example of MAClet bytecode for the legacy DCF, where we can recognize the initial state descriptor (for configuring the platform registers) and the transition table. The table is coded by: i) a list of transition lists, and ii) a list of states represented by the pointer to the relative transition list. For example, the state in the second position of the list (whose symbolic label 01 corresponds to the position index) points to the transition list coded from the third byte of the table and ends at the occurrence of the first $FFFF$ delimiter. As evident from the figure, the code is very compact (only 544 bytes).

The application-level software has been developed according to a simple client-server paradigm, since we considered a centralized decision logic implemented at the Access Point. The key component of the application is the MAClet manager that implements the MAC Distribution Protocols summarized in table 3 (for loading and executing MAClets on different network nodes). In our implementation, MAClet management messages are unicast, while MAClet action messages are broadcasted to all stations and filtered at the application level according to the "ID_set". The MAClet distribution logic, that is the block responsible of network configuration decisions, has a very simplified structure. Basically, rather than implementing a context estimation module and a programming interface for the operators, we pre-scheduled some decisions at the Access Point and pre-set the other nodes to accept MAClets and switching commands sent by the AP.

4.2 MAClet Switching

We run some simple experiments of MAClet switching in order to test the MAClet *intra-node* functionalities and measure the switching latency of our implementation. A more complete validation also involving multi-node coordination is described in the next section. In our tests, we used a USRP

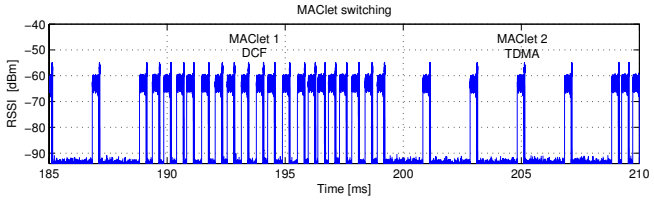

Figure 4: An experimental trace of medium occupancy times under MAClets switching (DCF and TDM) performed at regular time intervals (10 ms).

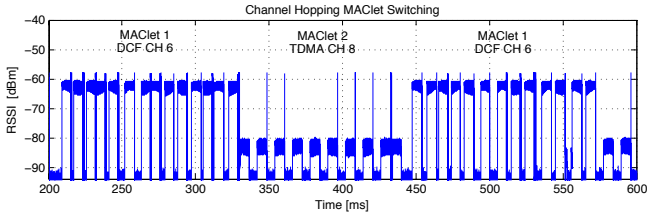

Figure 5: An experimental trace of MAClet switchings involving a channel switching operation.

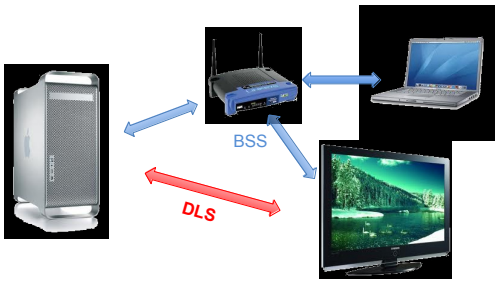

Figure 6: Use case 1: a streaming server (left) delivers HD video to an Internet enabled TV; a laptop (right) is connected to the internet via the AP.

board for acquiring the channel activity trace of the card performing the MAClet switching. The trace is processed by MATLAB for deriving the time-varying RSSI values corresponding to channel idle and busy states. In order to clearly visualize the change of the MAC protocol under execution, we considered the switching between the random-access of standard DCF and the deterministic-access of a TDMA protocol. The two corresponding state machines have been loaded on the two different program slots of the card. A synchronization transition has been programmed by specifying a switching event corresponding to the expiration of a 10ms timer.

Figure 4 shows a channel activity trace of 15 ms, captured by the USRP board when the card is fed with a saturated traffic (generated by the *iperf* tool, with a packet payload of 1470 bytes) transported over UDP. The MAClet initial state descriptor specifies that the card is set to operate on channel 6 and with a modulation rate of 24 Mbps, while the protocol initial state is set to an idle state for both the MAClets. When the switching to the TDMA protocol expires during a frame transmission, the first packet transmission is skipped, in order to avoid deferrals of subsequent packet schedules. The figure allows to easily identify frame transmissions (characterized by an RSSI value of about -60 dBm), acknowledgments (with an RRSI value of about -55dBm), and idle times (with an RSSI value of about -92 dBm). Thanks to the different inter-frame spaces (2ms under TDMA, random under DCF), we can clearly distinguish the protocol under execution at a generic time instant. Moreover, the MAClet switching time is practically negligible and not quantifiable from the figure. Even considering a much longer trace, we practically observed that the channel accesses performed under the TDMA protocol are always scheduled at regular intervals of 2 ms as in the case of a permanent TDMA execution (i.e. cumulative switching delays are not observable in a temporal trace of 5 minutes). In fact, the switching time is practically given by the execution of the bootstrap action, which in turns require to set the configuration registers (12 registers in our implementation), jump to the transition list of the new protocol state and load the new list of events. The execution of these operations requires on average 20 clock cycles, which correspond to about 0.2μs (being the clock frequency of the card at 88MHz).

Figure 5 shows an experiment similar to the previous one, where we also include a different configuration of the PHY layer in the descriptor of the MAClet initial states. Specifically, we set the DCF MAClet to operate on channel 6 and the TDMA MAClet to operate on channel 8, while the physical transmission rate is set to 1 Mbps for both the MAClets. We set the most robust modulation scheme since we verified that, at this rate, the receiver station on channel 6 is able to correctly demodulate the frame transmitted on channel 8, without performing the channel hopping (which would have requested to implement a synchronized MAClet switching also at the receiver side). Similarly, although the USRP receiver is set on channel 6, the board is able to detect part of the power transmitted on channel 8. The switching time is set to 200ms. In the figure, we can recognize the transmissions performed on channel 6 (whose RSSI values are about -60 dBm) from the transmissions performed on the out-of-band channel 8 (whose RSSI values are about -80 dBm). From the figure, it is evident that the radio does not exhibit any remarkable latency for hopping between the two channels.

5. NETWORK-LEVEL VALIDATION

We consider a generic infrastructure network, with an Access Point and a given number of associated stations in radio visibility. Despite of the scenario simplicity, we show two different use cases in which network reconfigurations can be really beneficial. The solutions proposed are on purpose *not general*: they are meant to be just examples (which can be further technically improved) devised to highlight our framework's flexibility and test the MAClet transport, loading, and switching functionalities. In particular, we focus on two important features of the proposed architecture: the ability to coordinate the execution of different MAC schemes at two different stations (*multi-thread*), and the ability to support heterogeneous node configurations performed by two different operators (*virtualization*) and permit their coexistence on a same shared channel.

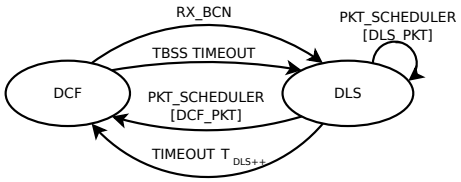

Figure 7: DLS++ protocol definition as a meta machine between two different threads.

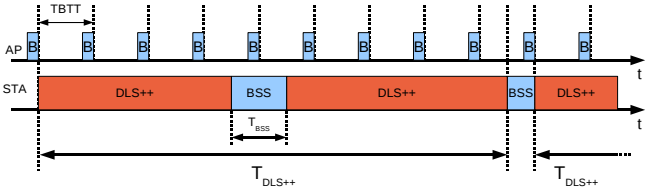

Figure 8: DLS++ timing

5.1 Use case 1: Multi-Thread

Scenario Description

The considered scenario is summarized in Figure 6. A WiFi ADLS domestic router connects three stations to the Internet. This usually works well if traffic to the stations comes from the outside. However, when the kids are at home and start downloading a high definition video from a streaming server (on the left) to the Internet-enabled TVset (in the middle), it is likely that who is trying to work on the laptop will get impaired performance, as the legacy DCF protocol requires traffic to be first routed from the server to the AP and then again to the TVset, thus duplicating the bandwidth used on the wireless channel.

This problem is obviously not nearly new, and indeed was specifically addressed by the 802.11e task group with the introduction of the Direct Link Setup (DLS), further extended in the 802.11z-2010 amendment. However, a direct link setup is not automatic (i.e. the kids should take care of changing the settings of the TVset during the streaming!). Moreover, the direct link uses the same wireless channel, thus, although to a lower extent, the station connected to the Internet still suffers of a bandwidth reduction.

Using the MAClet Control Architecture, the stations in the networks are not expected to implement any specific DLS amendment. By default, their Wireless MAC processor card runs a MAC program implementing just the legacy 802.11 DCF operation. As soon as the AP detects that two associated stations are involved in a greedy data session, it delivers a MAClet to just the two involved stations. Stations are configured to accept and install MAClets coming from the home AP. The AP further signals the (same) time instant at which the two stations will start the installed MAClet. From that time on, the two stations will implement a *custom* MAC protocol.

The custom MAC protocol may be designed to be strictly tailored to the considered context. For instance, the owner of the network *knows* that at most one direct link connection will be deployed, and that this direct link will always involve the two same radio interfaces (the server and the TVset ones). Moreover, the network owner wants to push bandwidth optimization further, by setting the direct link on a *separate frequency channel*, but of course avoiding that the stations will lose the association to the AP.

MAC Customization: Enhanced Direct Link

We have designed a protocol, hereafter referred to as Enhanced Direct Link (DLS++), for coping with the above scenario. The Enhanced Direct Link (DLS++) is meant to be a simple variant of DLS able to simultaneously work on two different channels. The primary channel is that of the AP network; the station has to periodically access such channel for receiving beacons and retaining association. The secondary channel is ad-hoc set up and independently managed by the peer stations. Under DLS++, the channel selection and the associated channel access mode is performed frame by frame. If the head of line frame is directed to the peer station, the frame is sent on the secondary channel as it was sent by the AP (i.e. with the from DS bit set to 1, and with the sender address of the AP)[3]. In absence of collisions, the random backoff on the secondary channel is suspended (by using a backoff counter permanently equal to 0) for optimizing the capacity of the streaming. If the head of line frame is a probe request frame or another frame directed to the AP, the station switches back to the primary channel and to the DCF protocol. It then returns to the secondary channel after a short T_{BSS} time interval. For simplicity of development, the described operation is not yet optimized to prevent packet losses; especially, buffering at the AP side should be performed when a DLS++ station is set on the secondary channel (this extension can be developed by mimicking the standard mechanisms used for power savings).

Multi-threading and Synchronization

The above scheme requires that the peer stations use standard DCF rules on the primary channel and direct-link access rules on the secondary one. This behavior can be programmed by defining a DLS++ meta machine switching from DCF to DLS and vice versa. The DLS machine is derived from the DCF one by changing the addressing operations for both data frames and acknowledgments. Moreover, it can be configured with independent contention window values, thus allowing to support more aggressive access operations. Figures 7 and 8 shows the envisioned meta machine and the relevant timings. The direct link operations are executed after the reception of an AP beacon. At regular time intervals T_{DLS++} (slightly lower than a multiple N of TBTT beacon intervals), the station suspends DLS transmissions in order to receive an AP beacon on the primary channel. This operation is necessary for keeping the synchronization to the AP clock and for receiving other MAClets[4]. A transitions to

[3]Without changing the driver, it is not possible to support simultaneously the ad-hoc and infrastructure addressing modes. Therefore, we chose to provisionally employ a kind of address spoofing for confining the updates in the MAC state machine.

[4]*Clock synchronization* of all the nodes belonging to the same network is a requirement for DCF (clock skews may toughen

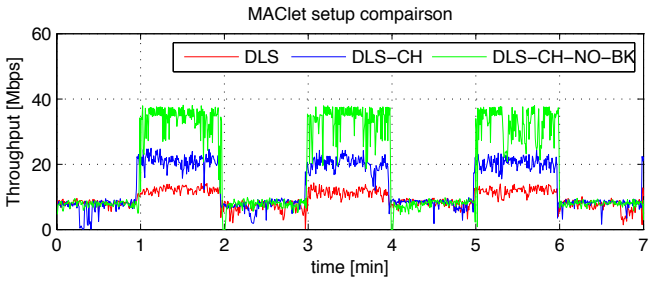

Figure 9: Throughput comparison under legacy DCF, DLS, and two versions of DLS++.

DCF can also occur when the head of line frame is a probe request or another frame directed to the AP. At the expiration of another timer T_{BSS}, the station switches back to DLS.

In order to minimize the frame losses due to the use of the two channels, the peer stations should activate the DLS++ protocol simultaneously. To this purpose, we used a synchronization mechanism based on the specification of an absolute time (after which, the switching are managed by the multi-thread meta-machine). The activation time is computed by adding the desired time offset to the current Access Point time-stamp, to which all the stations are continuously aligned.

Performance evaluation

We setup a testbed in our laboratory with two client stations (the ones with the peer-to-peer traffic) and the AP equipped with our MAClet Control framework. A third client was statically set to the primary channel with a legacy DCF protocol. We repeated the MAClet loading and activation test periodically, by programming the AP to alternatively send (to the two programmable clients) the DLS++ MAClet and the legacy DCF MAClet at regular intervals of 1 minute. The DLS++ MAClet is built by programming the meta machine described above, while the legacy DCF MAClet is activated by sending a run command for the bios program. The DLS++ protocol has been configured by setting T_{DLS++} to 890 ms, T_{DCF} to 6 ms, and N ito 9 TBTT. For the other protocol parameters, we used three different configurations: both the primary channel and secondary channels set to channel 6; the primary channel set to channel 6 and the secondary channel set to channel 11; the primary channel set to channel 6, the secondary channel set to channel 11, and the secondary channel contention window set to 0.

Figure 9 shows the throughput results of the client station sending saturated UDP traffic to the second client under the three settings (labeled, respectively, as DLS, DLS-CH, DLS-CH-NO-BK). The experiments were carried out during the hours of the day (i.e. in presence of background traffic due to students and researchers working in our department). Starting from legacy DCF, the clients switch to the different DLS++ configurations at 1, 3, and 5 minutes, and come back to standard DCF at 2, 4 and 6 minutes. From the figure it is evident that the customized direct-link access may bring dramatic improvements, especially when it is managed on a secondary channel without backoff (from about 12 Mbps of the normal DLS case to about 38 Mbps under the DLS++ without backoff).

frame acknowledgement). Therefore, we always assume that the bios machine provides a clock synchronization function.

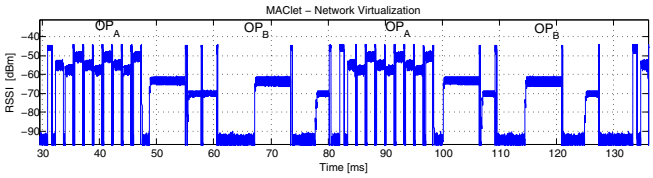

Figure 10: An experimental trace of network virtualization: operator A and operator B use the channel in different time intervals with independent access schemes (TDM and DCF).

5.2 Use case 2: Virtualization

Scenario Description

In this second use case we assume that the same Access Point (belonging to a public network) is shared between two different WiFi operators. The scenario is obviously not new, and indeed it has been specifically addressed by many manufacturers that allow to define Virtual APs, each advertising a distinct SSID and capability set. Virtual APs allow operators to share the same physical infrastructure, while offering access to distinct networks, but they typically suffer of a scarce level of *isolation*, since the resources allocated to each one cannot be really partitioned when stations employ random access schemes and suffer of unpredictable interference.

Suppose that the two operators want to implement a different service model: the first operator (operator A) advertises "FIXED" SSID, offering access to the Internet with a fixed (guaranteed) bandwidth, while the second one (operator B) advertises "BEST" SSID, offering a traditional best effort access. Although the standard includes PCF and HCCA for managing the medium access by means of polling, the lack of support in commercial products prevents an easy solution. Using MAClets, the resource repartition between the two operators can be addressed in a very effective and flexible manner. If all the stations employ a MAClet Control architecture, each operator can send a MAClet to the associated stations for enabling the medium access at regular time intervals (for example, in a fraction of the beacon interval reserved to the specific operator) and preventing it in the rest of the time. Moreover, the time reserved to each operator can be dynamically tuned (by updating the MAClet configuration parameters) according to the traffic conditions and to the agreements between operators.

MAC Virtualization and Synchronization

Different solutions are possible for addressing the beaconing and the MAC pausing in the above scenario. We chose to transmit two SSID Information Elements within each beacon, thus leaving the beacon interval unchanged. The MAC pausing has been programmed in a meta machine between the operator-dependent MAC program and a simple state machine with a waiting state only. At the expiration of the pausing time, each station enters the waiting state until a new activation event is revealed. In the waiting state, the

stations continue to receive beacons from the AP for keeping the synchronization to the time interval of their operator.

According to the SSID specified in the association request, each station receives a different MAClet: a legacy DCF program for the stations associated "BEST" SSID, and a TDMA program for the stations associated to the "FIXED" SSID. The DCF MAClet is a legacy DCF protocol that is suspended at the reception of a new beacon. The reactivation is triggered at the expiration of a parametric timer set before the suspension. The opposite activation and deactivation actions are performed for the TDMA MAClet. This mechanism guarantees a perfect coexistence and isolation between the two networks, since stations accessing the medium during the same time interval employ uniform channel rules, and no station associated to a given operator can interfere with the other operator network. Isolation is not obviously guaranteed with other external interfering networks.

The configuration parameters of the DCF MAClet are the DCF contention parameters that are uniformly set to all the stations (although some forms of user prioritization could be easily supported by differentiating these parameters in the MAClet directed to each station). Conversely, the MAClet transmitted to a new station associated to the "FIXED" SSID specifies a different program parameter indicating the slot numbers allocated to the station (multiple slots can be allocated to the same station). In each TDMA slot, frame transmissions still follow a 2-way handshake mechanism. When the MAClets are loaded on a new arriving station, the reception of the first beacon frame activates the execution of the program. Subsequent beacons are used as synchronization events for pausing the DCF programs and resuming the TDMA ones, as well as for activating the DCF suspension timer and computing the beginning of the TDMA slots. Although beacon frames are scheduled at regular time intervals, they can be delayed because of ongoing frame transmissions. These transmissions can be due to external interference, but also to stations associated to the "BEST" SSID starting a frame transmission right before the expiration of the operator time (no control is indeed implemented on the residual time before starting a transmission). In case of delay, to guarantee the fixed rate of TDMA stations, the time allocated to the "BEST" SSID operator can be reduced in the subsequent beacon interval. The possibility to dynamically tune the DCF activation time can also be exploited for performing a dynamic repartition of the resources allocated to each operator.

Figure 10 shows an example of resource repartitions between operators A and B in two consecutive beacon intervals. The figure plots the channel activity trace captured by the USRP: for better distinguishing the two virtual networks, the TDMA stations transmit at 11 Mbps while the best effort stations transmit at lower data rates (5.5 Mbps and 2 Mbps). Note that in the first TDMA slot the channel is busy (i.e. a transmission has been originated in that slot), but no acknowledgment is received because of channel errors.

Performance Evaluation

We setup a testbed with a fixed number of stations associated to the "FIXED" SSID and a time-varying number of stations

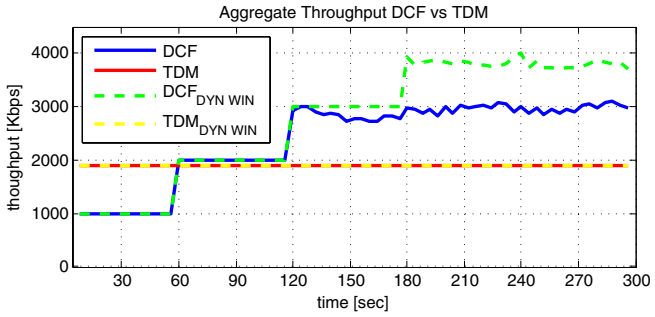

Figure 11: Resource repartition between two different operators using different access rules (TDM and DCF).

associated to the "BEST" SSID. Specifically, three stations access the channel by using TDMA, while five stations join sequentially the best-effort network at regular intervals of one minute. The TDMA frame is organized in nine allocated slots, uniformly assigned to all the stations (three slots each). The beacon interval is set to 50ms, while the slot size is set to 1.7ms (enough to accommodate the transmission of a payload equal to 1470 byte at 11 Mbps). All the stations transmit at 11 Mbps.

We repeated two different virtualization tests: in the first one, each operator receives an equal share of the available bandwidth (i.e. the activation time is one half of the beacon interval), while in the second one, the TDMA operator agrees to release the available bandwidth to the other operator. TDMA stations have a traffic rate of 630 kbps (smaller than the maximum guaranteed bandwidth, namely $3 \cdot 1470 \cdot 8/50ms = 705.6kbps$), in order to have a non-null probability to have some slots empty. DCF stations work with a traffic rate of 1 Mbps.

Figure 11 shows the per-operator throughput results obtained in both the experiments. In case of equal share of the bandwidth, after the third station joins the network the throughput of the best-effort operator (blue curve) saturates to about 3 Mbps (i.e. one half of the total network capacity at 11 Mbps). TDMA network is obviously under utilized because it consumes only 1.89 Mbps (being 3 Mbps the available capacity). By adjusting the time allocated to the best-effort operator, the third station can join the network without causing any throughput degradation. The aggregated network throughput (green line) for the best-effort network is now about 4 Mbps, while TDMA stations performance are not affected by increased DCF traffic.

6. RELATED WORK

Programmable wireless platforms

The advent of WLAN soft-MAC [15] designs, endorsed by several brand-name vendors (including Intel, Ralink, Realtek, Atheros, Broadcom), has transferred non time-critical MAC layer functionalities from the WLAN card to the host, thus permitting their modifications by reprogramming relevant open-source drivers. Moreover, several low level MAC/PHY parameters (contention windows, TX power, TX/RX antenna

settings, etc) can now be accessed through configuration interfaces. However, this level of flexibility is not enough to bring into real world several optimizations solutions which require small changes into the low-level MAC operations. For example, in [16] the experimental validation of receiver-initiated MAC protocols, which have been shown to improve the overall network capacity, could not rely on commercial 802.11 cards, since these cards do not allow to define new frame handshakes spaced of a SIFS time. Other promising solutions, such as the dynamic scheduling proposed in [17], require advanced monitoring operations not supported by current cards and drivers. To overcome this hurdle the research community has developed *custom* programmable wireless platforms, typically revolving around an FPGA or DSP core and software radio. For example, WARP [18] or Airblue [8] are stand-alone software defined radio boards equipped with fast and large FPGAs, hence not constrained anymore (unlike, e.g., GNURadio [19]) by an host back-end PC running part of the needed processing. Similar performance are obtained by SORA [20] by exploiting parallel computing. By (re)implementing all the wireless protocol stack, from the level of signals to that of frame payloads, these solutions support full MAC layer customization and cross-layer designs. More recently, custom MAC programmability was made possible also on commodity card, thanks to the disclosure of a (simplified) open source firmware [21] for a brand name card.

Clearly, the ability to access and modify the source code of a wireless card or wireless custom boards permits *in principle* any modification. In practice, extreme expertise is needed with the device internals and the low level programming languages (e.g. VHDL or assembler, at best C), as well as with the understanding of how software modules do interact with each other. An alternative solution which is currently emerging in other networking domains (such as flow switching technologies [22]) is the shift from open source code modifications to device reprogramming via open and suitably identified *Application Programming Interfaces*. This approach may perhaps restrict generality, but comes along with huge advantages: much simpler and faster programmability, code portability across different vendors' platforms, no need for manufacturers to disclose their internal architecture. A first step in this direction was taken by the *split functionality* approach proposed in [7]. The architecture proposed in this work comprises a radio hardware, which implements (part of) the core MAC functionalities, and a host PC which runs a control software implementing the MAC protocol control logic. The identification of how to most conveniently decompose a MAC protocol into core functions is a further major contribution (a similar analysis is carried out also in [9, 10]).

TRUMP [9] makes the further step of designing an *integrated* platform which permits to compose a MAC protocol operation using elementary modules. The core of this platform is a *Wiring Engine* which connects the core functions according to a programmable control flow, described through a newly introduced language syntax for PHY/MAC protocol description, and an associated compiler. The extended *Wireless MAC Processor* that we propose in this paper promotes a more versatile description and dissemination of MAC protocols in terms of extended finite state machines that can be encapsulated into common data packets. Central, in such approach, is the design of the *MAC Engine*, namely a generic finite state machine executor devised to play the role of MAC program interpreter.

Active Networks and Code Mobility

Despite the hype in the midst of the nineties [23], the application of active networking principles to the wireless domain has lagged behind. In the vision of [2], adaptations, envisioned in terms of selection of PHY functionalities (spectrum access, modulation, and coding) were expected to leverage software radio technologies. But proposed wireless active networking frameworks [24] have mainly addressed issues at layers *higher than low-MAC/PHY* (e.g., QoS, network topology adaptation, mobility, ad hoc network formation, etc).

The interest for code mobility, also embedded in in-band data packets (*capsules*, as per [25]), has more recently emerged in the wireless sensor networks arena. Indeed, in large sensor networks, code mobility may be the only possibility for upgrading the sensors' behavior, given that physical access to the nodes may not be viable. But, again, programmability has been restricted to higher layers, and for tasks such as changes in the monitoring functionalities or in the application operation [26].

Especially in the sensor network field, several issues concerning code distribution protocols [27, 28] and architectures [29] have been considered. Obviously, the programmability requirements for wireless local networks have some differences from the sensor network ones. Sensor nodes deployed in the same network are usually homogeneous, with the same Tiny OS and hardware. A binary code image can be moved from a node to another in active messages (natively supported by TinyOS). Albeit not strictly necessary, bytecode interpreters [30] may significantly improve efficiency of code distribution, i.e. for giving an high-level virtual code representations which significantly reduces the code length and/or facilitate incremental updates. All the above referred solutions limit programmability to network, transport and application protocols, and assume that the lower stack dealing with medium access and single-hop communications is not modifiable [30].

7. WMP DEVELOPMENT PLATFORM

The WMP implementation for the AirForce54G card (by Broadcom) has been released to the research community together with a detailed documentation of the available API (i.e. the list of events, actions and conditions supported by the card) and developing tools [31]. By replacing the original card firmware with the WMP one, the card can work as a generic state machine executor able to run a MAC bytecode. The developing tools include: i) a graphical tool, working as an editor for composing a MAC program in terms of a graphical representation of state transitions and state labels; ii) compiling tool, able to map the graphical representation into a textual transition table and in a bytecode; iii) a MAClet manager, able to load and run the bytecode in the card. The combination of the MAC Engine, graphical editor, compiler, MAClet manager and driver is a complete and cheap toolchain that allows developing and testing a new MAC scheme in a very simple, robust and quick way over an ultra-cheap

platform. The current WMP implementation supports both the infrastructure and the ad-hoc mode, it is compatible (in terms of protocol timings, frame fields, etc.) with legacy DCF stations in b and g mode, and it provides throughput performance comparable with the proprietary card firmware when executing the DCF state machine[5].

8. CONCLUSIONS

This paper proposes a wireless active network framework devised to permit seamless and dynamic MAC stack reconfiguration via MAClets, namely MAC programs conveyed into data packets. This is accomplished by extending our formerly proposed Wireless MAC Protocol architecture with primitives to dynamically handle code inside the radio hardware, by developing an overlay software control framework for moving and launching MAClets, and by experimentally assessing the flexibility and performance of the system operation over commodity WLAN cards.

Besides the specific technical contributions, we believe that a further significance of our proposed approach is that protocol reconfiguration is accomplished via application programming interfaces, rather than via binary images or access to open source devices, thus perhaps permitting its possible future endorsement also in the real commercial world.

Acknowledgement

This work has been carried out in the frame of the EU FP7-FLAVIA project, contract number 257263. We thank the reviewers and our shepherd Ruben Merz for the insightful and constructive feedback that helped us in revising the paper.

9. REFERENCES

[1] J. Mitola III, G. Q. Maguire, "Cognitive radio: Making software radios more personal", IEEE Personal Communications, vol. 6, no. 4, pp. 13–18, August 1999.

[2] V. Bose, D. Wetherall, J. Guttag, "Next century challenges: RadioActive networks", ACM/IEEE MobiCom '99, Seattle, USA, pp. 242–248.

[3] C. R. Stevenson, Z. Lei, W. Hu, S.J. Shellhammer, W. Caldwell, "IEEE 802.22: The First Cognitive Radio Wireless Regional Area Network Standard", IEEE Communication Magazine, January 2009, pp. 130-138

[4] A. Banchs, P. Serrano and H. Oliver, "Proportional fair throughput allocation in multirate IEEE 802.11e wireless LANs," Wireless Networks, 2007, vol. 13, pp. 649-662.

[5] L. Scalia, I. Tinnirello, J.W. Tantra, C.H. Foh, "Dynamic MAC Parameters Configuration for Performance Optimization in 802.11e Networks," IEEE Globecom 2006

[6] C. Partridge, "Realizing the future of wireless data communications," Commun. ACM, Sep. 2011, Vol. 54, issue 9, pp. 62-68

[7] G. Nychis, T. Hottelier, Z. Yang, S. Seshan, P. Steenkiste, "Enabling MAC Protocol Implementations on Software-defined Radios", NSDI'09, 2009.

[8] M. C. Ng, K. E. Fleming, M. Vutukuru, S. Gross, Arvind, H. Balakrishnan, "Airblue: A System for Cross-Layer Wireless Protocol Development", ACM/IEEE ANCS 2010.

[9] X. Zhang, J. Ansari, G. Yang and P. Mahonen "TRUMP: Supporting Efficient Realization of Protocols for Cognitive Radio Networks", IEEE DySPAN 2011

[5]The implementation has been tested on 4311 and 4318 chipset revisions, under the driver b43 and with kernel 3.1.4 (for more information see the specific documentation).

[10] I. Tinnirello, G. Bianchi, P. Gallo, D. Garlisi, F. Giuliano, F. Gringoli, "Wireless MAC Processors: Programming MAC Protocols on Commodity Hardware" IEEE INFOCOM, March 2012.

[11] Wireless Management Suite. Data sheet, Enterasys Networks, Inc., March 2009.

[12] AirWave Management Platform. Data Sheet DS AWMP US 081117, Aruba Networks, Inc., Nov. 2008

[13] Li-Hsing Yen and Tse-Tsung Yeh. SNMP-Based Approach to Load Distribution in IEEE 802.11 Networks. In IEEE 63rd VTC'06-Spring, vol. 3, pp. 1196-1200, May 2006

[14] B-S. Jeon, E-J. Ko, and G-H. Lee. Network Management System for Wireless LAN Service. In 10th International Conference on Telecommunications, 2003. ICT 2003, volume 2, pages 948-953, March 2003.

[15] M. Neufeld, J. Fifield, C. Doerr, A. Sheth, D. Grunwald, "SoftMAC - Flexible Wireless Research Platform" HotNets, Nov. 2005.

[16] T. S. Bonfim and M. M. Carvalho, Reversing the IEEE 802.11 Backoff Algorithm for Receiver-Initiated MAC Protocols, IEEE IWCMC 2012, Cyprus, 2012.

[17] M. Fang, D. Malone, K. R. Duffy and D. J. Leith, Decentralised learning MACs for collision-free access in WLANs, Wireless Networks. To Appear.

[18] Wireless Open Access Research Platform, http://warp.rice.edu/trac.

[19] The GNURadio Software Radio, http://gnuradio.org/trac

[20] K. Tan, J. Zhang, J. Fang, H. Liu, Y. Ye, S. Wang, Y. Zhang, H. Wu, W. Wang, G. M. Voelker, "Sora: High Performance Software Radio Using General Purpose Multi-core Processors", NSDI 2009.

[21] Open firmware for WiFi networks, http://www.ing.unibs.it/openfwwf/

[22] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, J. Turner, "OpenFlow: enabling innovation in campus networks", ACM SIGCOMM Comp. Commun. Review archive, Vol. 38(2), April 2008

[23] D. Tennenhouse, J. Smith, D. Sincoskie, D. Wetherall, G. Minden, "A Survey of Active Network Research", IEEE Communications Magazine, January 1997.

[24] A. Campbell, H. De Meer, M. Kounavis, K. Miki, J. Vicente, D. Villela, "A survey of programmable networks", ACM SIGCOMM 1999, pp. 7 – 23.

[25] D. Wetherall, J. Guttag, D. Tennenhouse, "ANTS: A Toolkit for Building and Dynamically Deploying Network Protocols", IEEE Open Architectures and Network Programming, April 1998, pp. 117-129.

[26] R. K. Panta, I. Khalil, S. Bagchi, "Stream: Low Overhead Wireless Reprogramming for Sensor Networks", IEEE INFOCOM 2007, May 2007, Anchorage, pp. 928-936.

[27] M. Krasniewski, R. Panta, S. Bagchi, C.L. Yang, W. Chappell, "Energy-efficient on-demand reprogramming of large-scale sensor networks", ACM Trans. on Sensor Networks, February 2008, Vol. 2, pp. 1-38.

[28] P. Levis, N. Patel, S. Shenker, and D. Culler, "Trickle: A Self-Regulating Algorithm for Code Propogation and maintenance in Wireless Sensor Network," NSDI 2004.

[29] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong, "TinyDB: an acquisitional query processing system for sensor networks," ACM Trans. on Database Systems., vol. 30, no. 1, pp. 122-173, 2005.

[30] P. Levis and D. Culler, "Mateː a tiny virtual machine for sensor networks," 10th int. conf. on Arch. support for programming languages and operating systems, 2002, pp. 85-95.

[31] WMP Project, http://wmp.tti.unipa.it/.