A Unified Radio Control Architecture for Prototyping Adaptive Wireless Protocols

P. Ruckebusch*, S. Giannoulis*, E. De Poorter*, I. Moerman*, I. Tinnirello+, D. Garlisi+, P. Gallo+, N. Kaminski*, L. DaSilva*, P. Gawlowicz†, M. Chwalisz†, A. Zubow†

*{name.surname}@intec.ugent.be iMinds, Belgium
+{name.surname}@unipa.it CNIT, Italy
*kaminshin|dasilval@tcd.ie TCD, Ireland
†{name.surname}@tu-berlin.de TUB, Germany

***Abstract*—Experimental optimization of wireless protocols and validation of novel solutions is often problematic, due to limited configuration space present in commercial wireless interfaces as well as complexity of monolithic driver implementation on SDR-based experimentation platforms. To overcome these limitations a novel software architecture is proposed, called WiSHFUL, devised to allow: i) maximal exploitation of radio functionalities available in current radio chips, and ii) clean separation between the logic for optimizing the radio protocols (i.e. radio control) and the definition of these protocols.**

I. Introduction

During the last years, research community has achieved an impressive evolution of wireless technologies for short distance communication (like IEEE 802.11, IEEE 802.15.4, Bluetooth Low Energy, etc.) due to the need of coping with the heterogeneous requirements of emerging applications, such as Internet of things, the Industry 4.0, the Tactile Internet, the ambient assistant living, and so on. Indeed, for optimizing the technology performance in these scenarios, it is often required to support some form of *protocol adaptation*, by allowing the dynamic reconfiguration of protocol parameters and the dynamic activation of optional mechanisms, or some targeted *protocol extensions.* In both cases, prototyping, testing and experimentally validating potential solutions is a complex task, which generally requires significant investment in time and resources. On one side, off-the-shelf wireless interfaces are based on radio chips which implement only the obligatory parts of the standard and arbitrarily selected optional parts, with only partially documented interfaces and with drivers being either closed or limited in functionality. On the other side, many powerful Software Defined Radio (SDR) platforms, while offering excellent flexibility at the physical layer, typically have limited performance and lack high-level specifications and programming tools as well as standard APIs for developing protocols.

The consequence is that testing of new solutions often proves problematic, because experimenters can only rely on the limited optimization space enabled by the drivers, while on *open* software architectures many functionalities have to be written from scratch and are tightly dependent on the specific hardware platform. In many cases, different experimentation platforms have to be considered for working on specific optimizations, because each platform supports a different level of complexity and controllability. This heterogeneity further slows down the innovation process, forcing experimenters to familiarize themselves first with platform-specific architectures and programming tools before prototyping their solutions.

To overcome the aforementioned shortcomings and reduce the threshold for experimentation, a novel approach is proposed within the European project WiSHFUL. The project's main goal is the design and development of a software architecture enabling a flexible radio and network control of heterogeneous experimentation platforms, based on standardized wireless technologies and SDRs, through unified programming interfaces. More specifically, the architecture is devised to allow:

- *Maximal exploitation of radio functionalities* available in current radio chips, as opposed to todays radio drivers that restrict radio functionality. For example todays radio drivers for IEEE 802.11 do not support TDMA (Time Division Multiple Access) operation, while the hardware perfectly supports it.
- *Clean separation between radio control and protocol logic*, as opposed to today's monolithic implementations, which prevent the ability to separately work on the logic for enabling specific protocol features and the definition of these features.

In the paper, we present the platforms integrated into the WiSHFUL architecture in §II, the WiSHFUL general architecture in §III, the functions enabled for radio configuration in §IV, and the potentialities of the approach in §V. Finally, some conclusions are drawn in §VI.

II. Programmable Prototyping Platforms

The WiSHFUL project integrates multiple experimentation platforms for which a software architecture designed to simplify MAC or PHY protocol prototyping was already available. The platforms are based on heterogeneous hardware: general purpose devices with a wireless network interface for local area networks (WiFi), microcontroller devices with a radio chip for sensor networks and software defined radios (SDR).

A. WMP

The Wireless MAC Processor (WMP) architecture has been developed for a commercial Broadcom WiFi card and for the

WARP board [1]. The architecture offers the possibility to easy write, load and execute customized MAC protocols, by using a platform-independent, high-level programming language [2]. This capability is achieved by developing a firmware which does not implement a specific protocol, but rather a generic protocol executor called *MAC Engine*.

The MAC programs are specified as extended finite state machines (XFSMs), which are built by composing elementary hardware *actions*, in response of specific hardware *events* and *conditions* of the hardware internal registers. The set of events generated by the hardware, the set of actions coded in pre-defined firmware modules and the set of hardware registers whose settings can be tuned and verified, represent the hardware API that cannot be modified by the user.

The MAC program is coded into a transition table and 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 with that state occurs. It then evaluates the associated conditions on the configuration registers, and triggers the associated action and register status updates (if any). Next it executes the state transition and fetches the new table entry for the destination state. The MAC engine does not need to know to which MAC program a new fetched state belongs to. Therefore, code switching is achieved by simply moving from the current protocol state to a target state in a different transition table, with a latency of a few CPU clocks.

B. TAISC

TAISC (Time-Annotated Instuction Set Computer) [3] aims to simplify the development of new protocols for sensor nodes. It consists of a cross-platform MAC protocol compiler and an execution engine. This design allows to describe MAC protocols in a platform independent language (consisting of a radio platform independent instruction set), followed by a straightforward compilation step, yielding dedicated binary code, optimized for specific radio chips. The cross-compilation approach allows developers to design MAC protocols once, and then compile them for reuse on different radio platforms. To enable time-critical operation, the TAISC compiler adds exact time annotations to every instruction of the optimized binary code. The execution engine running on the radio platform, will execute the instructions with accurate time control thanks to the provided time annotation.

The overall TAISC workflow to develop and execute a MAC protocol is illustrated in Figure 1 and involves the following steps:

- *Step 1: device-agnostic MAC protocol creation. First, the MAC protocol designer creates a high-level, platform independent radio program to describe the MAC logic using predefined commands (instructions) in a C-like language, either using high-level C syntax or using a more intuitive drag- and-drop interface. This human readable code consists of a sequence of commands that describe the generic behavior of the MAC protocol and is largely independent of the final hardware platform where it will be deployed.*
- *Step 2: device specific compilation. Next, this human-readable sequence is compiled by the TAISC compiler into efficient, device-specific binary byte code that can be executed by the TAISC execution engine running on the radio platform.*
- *Step 3: protocol dissemination. Afterwards, the byte code is wirelessly transmitted to the target hardware platform and added to the MAC application repository on the local TAISC execution engine.*
- *Step 4: MAC protocol execution. Finally, the TAISC core executes the byte code.*

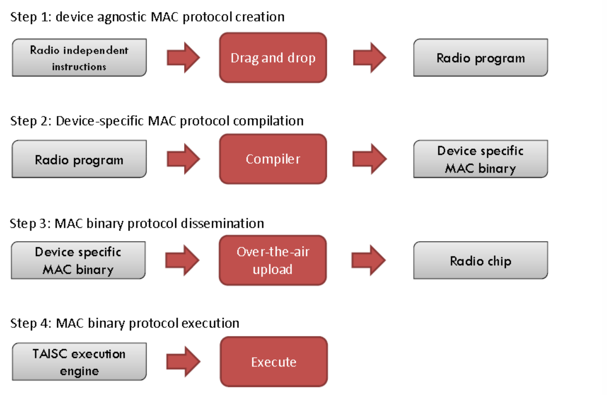

Step 1: device agnostic MAC protocol creation
Radio independent instructions
Drag and drop
Radio program
Step 2: Device-specific MAC protocol compilation
Radio program
Compiler
Device specific MAC binary
Step 3: MAC binary protocol dissemination
Device specific MAC binary
Over-the-air upload
Radio chip
Step 4: MAC binary protocol execution
TAISC execution engine
Execute

Fig. 1. TAISC workflow to develop and execute a MAC protocol.

C. IRIS

The Iris software defined radio framework was developed as a part of the doctoral work of Mackenzie in 2004 [4] and is currently maintained by Software Radio Systems[1]. While the architecture has been extended, the focus of the Iris concept remains on the realization of reconfigurable radio systems. Iris achieves such realization by offering a core framework to manage the execution of signal processing blocks within user defined chains, referred to as flow graphs. Signal processing blocks, such as modulators, scramblers, coders, can be defined by users or can also be selected from a library. In this manner, Iris allows users to define radio transceivers in software, and achieve working implementations of the same functionality with the addition of a Universal Software Radio Peripheral (USRP). As such, Iris is an ideal tool for cutting edge radio research and prototyping.

The core framework provides functionally to support the dynamic reconfiguration of the processing blocks. To this purpose, the scheduling and connection functionality of the system is built in manner to support the unpredictable variation of the rate at which individual blocks produce or consume data. In further support of flexibility, the components of a radio built within Iris are organized into one or more engines. These engines support the various operations of a radio system, such as data passing, on different layers of the network stack to allow users to organize their radio implementation in a convenient manner that support the reconfiguration of any individual component. Such a configuration also allows users to share

[1]https://github.com/softwareradiosystems/iris_core

radio components or incorporate processes and functionality external to Iris into their radio. Further, this approach provides users extreme control over their radio system.

To realize its potential for flexibility, Iris provides several means of interaction.The Iris core framework and the signal processing elements are implemented in C++, which increases performance and portability. Alternatively, radio flow graphs are defined in an XML-based format that allows the description of radios in a human-readable manner. Naturally, the flexibility and convenience offered by Iris has the cost of performance, but the design and approach of Iris supports dynamic research.

D. Atheros-ATH9K

Commodity WiFi devices based on Linux platform, IEEE 802.11n Atheros chipsets (e.g. AR928X) and open source driver ATH9K [5] are widely used in research and academia. Indeed, the open source driver permits to configure a wide set of MAC/PHY protocol parameters (such as the contention window, the antenna diversity scheme or the parameters used in the 802.11e access categories), to monitor some low-level parameters (such as the channel busy or idle intervals) and to modify the upper-MAC protocol mechanisms which do not depend on time-critical operations (softMAC). Therefore, in WiSHFUL we also considered the flexibility enabled by this open source driver for defining a unified radio control architecture.

III. The WiSHFUL Architecture

The WiSHFUL architecture is devised to provide i) unified interfaces to experimenters for easily prototyping novel and adaptable wireless solutions on different radio platforms, ii) a control framework for supporting dynamic on-the-fly re-configuration of the network nodes according to time-varying estimates of the network operating condition. To this purpose, a common programming model is proposed to fit all the heterogeneous experimentation platforms integrated into the project, based on the abstraction of the radio architecture and on the definition of elementary control primitives. In Figure 2 the proposed architecture is presented, showing how heterogeneous devices can be supported through a unified interface.

A. Abstracting Radio Architectures

From the analysis of TAISC, WMP, and IRIS, it is interesting to observe that there is a valuable common set of functionalities (implemented in different ways) and approaches that can be abstracted for the definition of a common programming interface. More into details, each architecture relies on primitive components, called data processing blocks, commands or actions, which depend on the hardware capabilities and cannot be programmed. Each architecture abstracts the hardware systems in a set of *configuration parameters*, which may change the hardware operating conditions (e.g. the transmission channel), *events* which indicate the occurrence of specific hardware operations (e.g. the reception of a new packet), and *measurements* that can be performed by the platform (e.g. the SNR of a received packet). Moreover, the architectures define an execution environment (called PHY engine, Stack engine, MAC engine, or TAISC engine) able

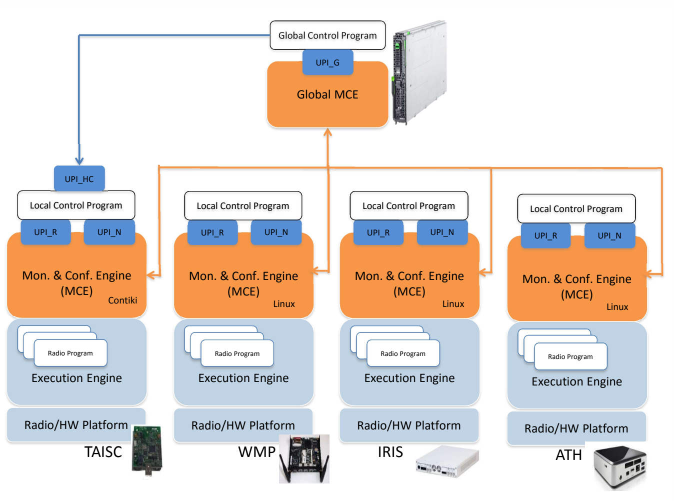


Fig. 2. WiSHFUL general architecture for programming heterogeneous platforms for radio experimentation.

to run radio programs defined in a high-level programming language.

A set of radio programs implementing very similar operations (e.g. CSMA and TDMA access protocols) have been defined for all the architectures by using the architecture-specific programming languages. Both for IRIS and WMP, a local controller can reconfigure the hardware parameters or the radio program itself according to the logic defined in the control program.

Inspired by these considerations, a similar programming architecture has also been proposed for the Atheros-ATH9k platform. A software module (MAC engine) was developed exposing an API for controlling the driver, by enabling the possibility to specify the configuration parameters of the card in a declarative language, such as Python scripts. The API also allows supporting a time-based channel access scheme by specifying the time intervals (slots) in which specific packet flows are allowed to transmit while running the usual DCF scheme. This was made possible because the developed MAC engine executes the MAC programs using the Netlink API to control the ATH9k wireless driver.

Figure 2 furthermore shows the generalization of the WiSHFUL radio programmable architectures (built on top of different hardware platforms) in terms of Radio Programs, Execution Engine, Radio Monitoring and Configuration Engine (MCE) and Control Programs. The Radio Programs specify the logic for driving the hardware platforms and implementing lower-MAC protocols, modulation/demodulation schemes or other processing operations on the hardware platform (e.g. spectrum scanning schemes, interference estimation schemes, localization schemes). The Execution Engine provides the environment for running the Radio Programs. The MCE is responsible of configuring the Radio Programs and the hardware platform during the initialization of the radio or during the radio activity, according to the rules specified in the Control Program. The Control Program configures the platform capabilities (e.g. transmission power, channel, radio program to be activated, etc.) and the program-dependent capabilities (e.g. slot size, contention window, etc.) in a list of parameters and relevant values, which may change as a function of the monitored measurements.

B. Control Framework

As shown in Figure 2, the WiSHFUL control framework is based on a two-tier architecture which enables local, global and hierarchical control programs, thus supporting dynamic adaptations of the wireless nodes according to the aggregation of radio parameters monitored by different nodes and estimates of the network state. Nodes can be monitored and controlled singularly or in clusters, by exploiting some basic control services devised to coordinate the UPI calls. Specific control commands for calling the UPIs can be sent to several nodes in parallel from the global controller, thanks to the ability of the framework to guarantee a timed execution of the function calls network wide.

Services offered by the control framework basically include node bootstrapping and discovery, time synchronization for relying on a common temporal signal, blocking or non-blocking interface calls, management of control programs on the nodes, as well as time-scheduled and remote execution of UPI functions. Depending on the expected timings for interacting with nodes, UPI functions can be called synchronously or asynchronously. In the first case the control program waits until the UPI returns, in the second case the call returns immediately and a callback function must be defined to handle the asynchronous response of the UPI call.

The WiSHFUL control framework supports both proactive and reactive control approaches, leaving to the experimenter the highest flexibility defining his/her control strategy. The control program calls UPIs on the system under test in a proactive scenario. Nodes receive polling from the controller and apply actions defined in the control logic. Conversely, under the reactive approach, local control programs can trigger messages to the global one when specific conditions locally arise on the node.

IV. Unified Interface for Radio Control

In this section the functionalities of the unified interface for radio control, called UPI_R, are presented in detail. This interface is responsible for tuning the radio operating frequency, selecting the transmission format, activating wireless links towards neighbor nodes, collecting statistics and configuring the medium access logic. To this purpose, as a preliminary operation, the interface needs to acquire information about the platform radio capabilities, because different platforms can support different programmability models and configuration parameters. Then, according to the available capabilities, the interface functionalities can work on three aspects: *configuring* the experimentation platform, at both the hardware and radio program levels, *monitoring* the node and network conditions by accessing all the signals and internal state information of the experimentation platforms, *adapting on-the-fly* the node behavior by loading and activating context-specific radio programs.

A. Radio Capabilities

Three different types of platform radio capabilities are defined: configurable *Parameters*, low-level *Measurements* and *Events*. The configurable parameters specify the configuration of the hardware platform and the initialization of the global variables of the loaded radio program. The low-level measurements are provided by the platform in some internal registers which track received signal strength, receiver errors, etc. The events are asynchronous signals generated from hardware or software state changes, which can be directly exposed to the controller or aggregated in a sequence of events whose occurrence can be signaled to the controller.

TABLE I. Abstraction of Radio Capabilties.

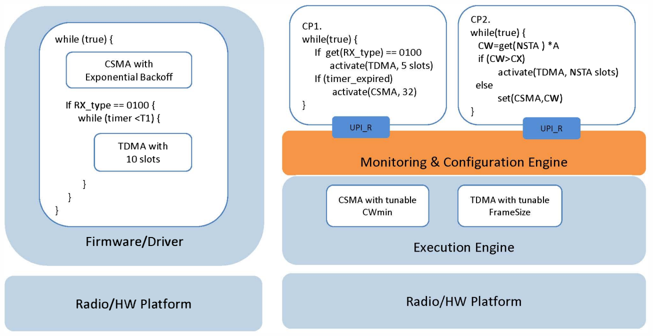
Parameters	Measurements	Events
Channel	RSSI	ChUp
CCA	SNR	ChDown
TxPower	BusyTime	RxPreamble
TxAntenna	TxTime	RxMacHead
RxAntenna	LQI	RxEnd
TxFormat	FER	RxBadCRC
TDMA_SuperFrame	BER	QueueOut
TDMA_NumSlots	goodPreamble	RxQueueOverflow
TDMA_Slot	badPreamble	TxQueueUnderflow
CSMA_CWmin	goodCRC	EndTimer
CSMA_CWmax	badCRC	CSMA_BkExpired

A non exhaustive list of radio capabilities is provided in table I. The capability names are somehow auto-explicative. Note that the list of radio capabilities is intrinsically extensible because they depend on software and hardware releases, which are continuously updated. However a core set of basic capabilities is defined, which are represented by a pre-defined list of identifiers. Each platform can obviously support the whole list of capabilities or a subset of such a list, depending on the hardware flexibility and on the loaded radio programs. Parameters correspond to the configuration registers of the hardware platform and to the variables used in the radio programs. For each parameter, a range of valid values can also be specified.

B. Functionalities

Acquiring Node Information. The information about the number of wireless interfaces available on an experimentation platform and the relevant radio capabilities are retrieved by means of, respectively, the $getRadioNICs()$ and $getRadioNICinfo(NIC\_t)$ functions. The $getRadioNICinfo()$ function is mostly used at bootstrap, but also after the loading of a novel radio program. It returns the radio list of events, monitor measurements and configuration parameters supported by the hardware and by the radio program loaded on the node. The returned data are structured in three lists of $< type, value >$ couples, which enumerate the supported events, measurements, parameters and their current values.

Radio configuration. To configure the experimentation platform, it is possible to work on a parametric configuration model which acts on the hardware and on the active radio program. The $setParameters(NIC\_t, param\_list)$ function allows to specify the settings of the desired parameters, while the $getParameter(NIC\_t)$ function allows to know the current configuration of the platform. It is also possible to define some rules for identifying the packets of a specific traffic flow by using the $FlowDesc()$ function. Each flow can



```
while (true) {
  If RX_type == 0100 {
    while (timer <T1) {
    }
  }
}
```

```
CP1.
while(true) {
    If  get(RX_type) == 0100
         activate(TDMA, 5 slots)
    If (timer_expired)
         activate(CSMA, 32)
}
```

```
CP2.
while(true) {
  CW=get(NSTA ) *A
  if (CW>CX)
        activate(TDMA, NSTA slots)
  else
        set(CSMA,CW)
}
```

Fig. 3. Examples of dynamic protocol mechanisms: monolithic implementation (left case) and flexible control programs defined on top of WiSHFUL UPI_R (right case).

be mapped into a specific radio configuration, by calling the $setPerFlowParameters(flow\_id, param\_list)$ function.

Monitoring. The $getMonitor(NIC\_t, monitor\_id)$ function allows to track single measurements specified by the identifier $monitor\_id$ of the relevant radio capability. Asynchronous events can also be tracked by opportunistically defining the conditions for triggering an interrupt signal, and the program handler for reacting to such an interrupt. The triggering conditions may correspond to the occurrence of an event supported by the platform, to a sequence of multiple elementary events or to the overcoming of a threshold for the counter of elementary events. The function responsible of event definitions is the $defineEvent$ function. Finally, the $getMonitorBounce()$ function allows to configure periodic reports of measurements.

Changing the Program on the fly. The $setActive()$ function is responsible for activating a specific radio program (among the ones loaded on the platform), while the $getActive()$ function is responsible for identifying the radio program currently in use.

V. Radio Control Examples

In order to clarify the potentialities of the WiSHFUL architecture and UPI_R interface, we consider a simple example of *protocol adaptation* in terms of dynamic tuning of protocol parameters, and a simple example of *protocol extension* in terms of dynamic activation of an optional feature. The goal is not designing a novel optimization logic, but rather demonstrating the flexibility of the proposed approach by separating the logic for controlling the experimentation platform from the transmission mechanisms running on the platform.

Suppose that a given wireless technology implements a CSMA access protocol, with the possibility to activate a burst of TDMA slots after an explicit signalling phase between the stations. The example is actually inspired by different real technologies, in which contention-based access mechanisms and reservation-based access mechanisms can coexist.

Current architectures for wireless experimentation platforms impose to work on a monolithic implementation of the protocol. As shown in the left case of Figure 3, the logic

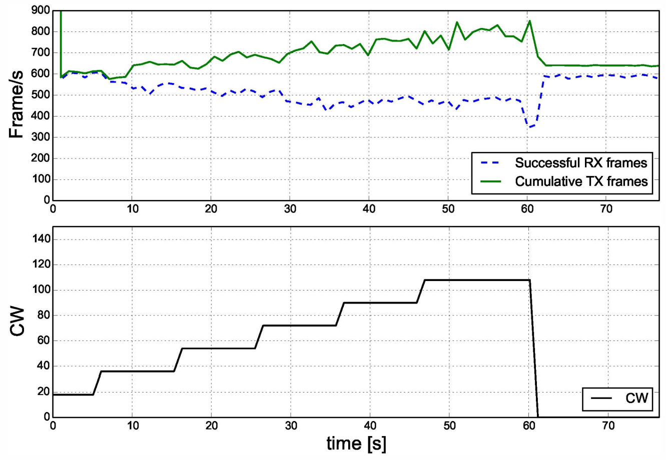


Fig. 4. MAC protocol adaptations performed by the control program *CP2* under a progressive increment of the number of active stations, and for a *CX* threshold equal to 128.

for changing the contention parameters and for activating the TDMA mode is *embedded* into the implementation of the protocol. Conversely, in WISHFUL, the logic for configuring the contention parameters (protocol adaptation) and activating the TDMA mode (protocol extension) is completely decoupled from the implementation of the two elementary radio programs. This implies that such a logic can be easily updated or replaced with a new one. Obviously, such a decoupling is possible when the control logic operates on a time-scale compatible with the latency of the local or global controller, which is indeed the case for protocol optimization depending on an estimate of network conditions.

Figure 3 also shows a high-level description of two control program examples, which work on setting the minimum contention window of the CSMA protocol and on activating the TDMA protocol. For sake of readability, the names of the UPI_R functions used in the control programs have been shortened.

The exemplary control program *CP1* works similarly to the logic of the monolithic implementation: when the node receives a frame of a specific type, the protocol switches to TDMA protocol with 5 slots per superframe. The protocol is switched back to CSMA after the expiration of a timer. In the case of control program *CP2*, the contention window is tuned as a function of the number of active stations and the TDMA protocol is activated when such a number (proportional to the CW value) overcomes a given threshold. Note that the control programs are platform-independent and can work on all WiSHFUL experimentation platforms for which CSMA and TDMA radio programs are available. Details about the implementation of the control program and the synchronization mechanisms available in the WiSHFUL control framework are provided in [6].

An experiment was run, executing the control program *CP2* on the WMP experimentation platform. In this experiment, wireless nodes were progressively activated at regular intervals of 10 seconds. Each station has a greedy traffic source with packets of 1000 bytes size, which are transmitted using a physical layer data rate of 6 Mbps. Figure 4 shows the number of successful received frames, the total number of transmissions and the value of the minimum contention window during the experiment. In the last 10 seconds, when the contention window value overcomes the CX threshold set to 128, the TDMA mode is activated, as evident from the higher stability of the number of transmitted and received frames.

VI. Conclusions

In this paper, starting from the presentation of the programmable radio architectures and prototypes available in WiSHFUL (namely, the IRIS, TAISC, Atheros-ATH9K and WMP architectures), a first specification of a unified interface for radio control has been described. The interface has been conceived for offering a unified programming model to experimenters willing to work on heterogeneous radio platforms and for enabling the definition of platform-independent adaptation logic of the MAC/PHY stack.

References

[1] A. Khattab, J. Camp, C. Hunter, P. Murphy, A. Sabharwal, and E. W. Knightly, “Warp: A flexible platform for clean-slate wireless medium access protocol design,” *SIGMOBILE Mob. Comput. Commun. Rev.*, vol. 12, no. 1, pp. 56–58, Jan. 2008. [Online]. Available: http://doi.acm.org/10.1145/1374512.1374532

[2] I. Tinnirello, G. Bianchi, P. Gallo, D. Garlisi, F. Giuliano, and F. Gringoli, “Wireless MAC processors: programming MAC protocols on commodity hardware,” in *INFOCOM, 2012 Proceedings IEEE*. IEEE, 2012, pp. 1269–1277.

[3] B. Jooris, E. De Poorter, P. Ruckebusch, P. De Valck, J. Bauwens, and I. Moerman, “Taisc: a cross-platform mac protocol compiler and execution engine,” in *Computer Networks Journal,under review*. Elsevier, 2016.

[4] P. Mackenzie, “Reconfigurable Software Radio Systems,” Ph.D. dissertation, Trinity College Dublin, Dublin, Ireland, 2004.

[5] “Atheros Linux wireless driver (ATH9K), https://wireless.wiki.kernel.org/en/users/drivers/ath9k/.”

[6] E. WiSHFUL, “Deliverable 2.3., results of first set of showcases, eu wishful,” Tech. Rep., 2016 (http://www.wishful-project.eu/), Tech. Rep., 2016.