SDMAC: A Software-Defined MAC for Wi-Fi to Ease Implementation of Soft Real-time Applications

Gianluca Cena, *Senior Member*, *IEEE*, Stefano Scanzio, *Member*, *IEEE*, and Adriano Valenzano, *Senior Member*, *IEEE*

***Abstract*—In distributed control systems where devices are connected through Wi-Fi, direct access to low-level MAC operations may help applications to meet their timing constraints. In particular, the ability to timely control single transmission attempts on air, by means of software programs running at the user space level, eases the implementation of mechanisms aimed at improving communication timeliness and reliability. Relevant examples are deterministic traffic scheduling, seamless channel redundancy, rate adaptation algorithms, and so on.**

In this paper, a novel architecture is defined, we call SDMAC, which in its current embodiment relies on conventional Linux PCs equipped with commercial Wi-Fi adapters. Preliminary SDMAC implementation on a real testbed and its experimental evaluation showed that integrating this paradigm in existing protocol stacks constitutes a viable option, whose performance suits a wide range of applications characterized by soft real-time requirements.

***Index Terms*—IEEE 802.11, Wi-Fi, Software-Defined MAC, SDMAC, Wi-Fi drivers, real-time communication, real-time wireless, experimental evaluation.**

I. Introduction

Wi-Fi [1] adoption in industrial scenarios has been steadily increasing over the past years. This is mainly due to its high throughput and complete interoperability with Ethernet, which achieve ubiquitous connectivity between devices, reducing at the same time wiring harness complexity. However, when distributed real-time control systems are taken into account where devices are interconnected through Wi-Fi, simple and flexible mechanisms are typically required for configuring and managing network stations. They permit to easily define, develop, and test new effective applications and solutions. This is witnessed, for instance, by the recent introduction of software-defined wireless networks (SDWN) [2].

In order to meet the specific timeliness and reliability requirements of factory automation systems, parameters of the medium access control (MAC) and physical (PHY) layers may have to be suitably tuned, even at runtime. In particular, it should be possible for applications to manage frame transmission on air with finer detail than typical allowed in Wi-Fi (where, e.g., the retransmission process is completely handled in hardware by adapters and hidden to the users).

This work was partially supported by Regione Piemonte and the Ministry of Education, University, and Research of Italy in the POR FESR 2014/2020 framework, Call "Piattaforma tecnologica Fabbrica Intelligente", Project "Human centered Manufacturing Systems" (application number 312-36). The authors are with the National Research Council of Italy, Istituto di Elettronica e di Ingegneria dell'Informazione e delle Telecomunicazioni (CNR-IEIIT), I-10129 Torino, Italy (e-mail: {name.surname}@ieiit.cnr.it).

The ability of the sender to timely start a frame transmission on air and to timely obtain a notification of the delivery outcome enables deterministic overlays to be layered atop Wi-Fi, which help making transmission latencies known in advance and, as much as possible, bounded. Moreover, applications may be interested in receiving management frames, like beacons, which typically are under control of the driver and are not made available to the user, or in accessing information hidden in specific registers of the Wi-Fi adapter.

Currently, many commercial Wi-Fi adapters rely on a SoftMAC architecture, where most functions of the MAC sublayer management entity (MLME) are implemented in software by the device driver and are executed by the CPU of the host computer [3, p. 28]. Only time-critical MAC operations (e.g., managing timeouts, like interframe spaces and backoff, and performing the related actions upon their expiry) are executed in hardware by the adapter. On the contrary, adapters that comply to the FullMAC architecture directly implement the whole IEEE 802.11 protocol stack (both the MAC and the MLME). Thus, they are more complex and expensive.

Unlike FullMAC, advanced customization is possible for SoftMAC devices, which includes redefining protocol parameters and bringing changes to the MLME. This is particularly true for the Linux operating system, where device drivers are often open source and can be easily modified. However, when changes are required to the MAC or PHY layers, which involve operations executed in hardware, other solutions are needed. In these cases, functions of the wireless adapter can be possibly implemented using a software-defined radio (SDR) [4], typically by employing an FPGA [5]. By doing so, practically every aspect of the MAC and PHY layers can be customized to comply with design specifications. Unfortunately, the required effort is considerably high, and the same applies to cost.

In [6] the software-defined MAC (SDMAC) paradigm was first proposed to provide applications executing in user space finer control on operations of Wi-Fi adapters. Its performance, in terms of latency, is not expected to match FPGA-based SDR approaches, and also the ability to manage adapter behavior is more limited. However, SDMAC offers a number of benefits.

To foster its adoption, SDMAC was designed as a flexible framework that relies on commercial Wi-Fi adapters and only requires limited and known modifications to device drivers. In this way, porting to different equipment or dealing with updated driver releases can be accomplished easily and quickly. According to the SDMAC paradigm, operations related to custom protocol functions are implemented in user space, and rely on a suitable application programming interface (API) that exposes the communication primitives provided by

Wi-Fi adapters and the related operating parameters. To ease implementation and testing, diagnostic information are also provided by SDMAC about its inner state.

SDMAC enables direct access to the most basic adapter operations. Probably, the most relevant example are confirmed one-shot transmissions, on which most of the experimental evaluation in this paper focuses. They can be seen as the elementary building block on which every specific protocol overlay (a supplemental mechanism layered above adapters, and meant to improve some aspect, e.g., determinism) can be built. In this case, automatic MAC retransmission upon errors is disabled, and packet transmission simply consists of one DATA frame followed, in case of success, by the related ACK frame in the opposite direction. Having the ACK frame for single attempts managed in the hardware of the adapter unburdens the software of this task and ensures higher reliability and performance, without practically limiting SDMAC flexibility. Multi-shot transmissions, that permit a configurable number of retries, can be also defined.

In addition, SDMAC provides a timely notification when transmission ends, which makes the outcome of delivery available to applications in user space. In this way, the retransmission process can be completely managed in software, which enables additional behaviors besides those foreseen by the standard specification. In general, the exact pattern of frames appearing on air, including the related timings, can be decided in SDMAC according to user-defined rules.

Preliminarily to the stable definition of the SDMAC interface, one must determine if the performance this approach delivers on real devices meets the typical requirements of industrial applications. In this work, a thorough statistical characterization of the latencies introduced by SDMAC was performed, in order to assess whether this approach is adequate for the intended scenarios or more complex and expensive solutions, like SDR, have instead to be pursued.

Compared to the preliminary proposal in [6], a number of enhancements are provided in this paper: 1) a taxonomy of the most important SDMAC service primitives has been sketched; 2) an enhanced measurement system has been developed, where timestamps are acquired in several different places of the protocol stack; 3) a set of specific guidelines and optimizations is provided that sensibly improves SDMAC performance over [6]; and, 4) a more extensive analysis was performed on the dependence between latency and transmission outcome. The paper is organized as follows: in Section II the SDMAC architecture, its basic properties, and possible application scenarios are described. Section III focuses on SDMAC implementation, while Section IV describes the measurement system. A discussion on results is included in Section V, followed by concluding remarks.

II. Software-Defined MAC

SDMAC implementation resides partly in user space and partly in kernel space. In particular, small blocks of code have to be placed in specific positions of the device driver to perform specific operations (e.g., to capture relevant information). Effortless integration in existing device drivers was a key design requirement. Part of SDMAC duties is to transfer information from user space to kernel space, and vice-versa.

A. Taxonomy of SDMAC services

SDMAC services can be subdivided in two broad classes: *transmission-oriented* and *MAC-layer management*.

1) Transmission-oriented services: These services deal with data exchanges among STAs at the *data-link* layer, and are the building blocks that permit applications to precisely coordinate their actions on air, enabling timely data delivery. While they somehow resemble conventional MAC transmission services performed in hardware, the overall protocol execution takes now place in software under user control.

SDMAC services comply to the OSI model: *request* and *confirm* primitives are defined in the originator, and *indication* in the recipient. The *response* primitive is not foreseen, because in Wi-Fi it is mapped on ACK frames, which are automatically sent by the adapter of the recipient STA after a short interframe space (SIFS) has elapsed from DATA frame reception. For unconfirmed traffic, only the *request* and *indication* primitives are needed. SDMAC data exchange services are implemented through the `SDMAC_DATA_req()`, `SDMAC_DATA_con()`, and `SDMAC_DATA_ind()` functions. A conceptual draft of their prototypes, together with the most important parameters, is reported in Table I.

`SDMAC_DATA_req()` is used to send a packet. Parameter `if` identifies the target interface in multi-adapter configurations, whereas `id` is a packet identifier. The value of `id` is initialized by the calling application, and must be unique in the related scope. If more than one application is invoking SDMAC primitives at the same time, each of them has its own scope and is free to select its specific values for `id`. Coherence between the `id` values in user and kernel spaces is managed directly by SDMAC. In the quite common implementation we considered in this paper, a character device is used to link these two execution contexts. Parameter `ac` specifies the queue of the adapter where the packet will be buffered for the forthcoming transmission. Typically, `ac` corresponds to one of the 4 access categories (AC) foreseen by every recent adapter complying with IEEE 802.11e, i.e., *voice*, *video*, *best effort*, and *background*, and permits to differentiate the quality of service (QoS) for specific classes of transmitted packets.

`SDMAC_DATA_con()` is used to obtain a timely notification at the end of packet transmission. It also provides the transmission outcome (either success, when the ACK frame is received, or failure, in case *ACKTimeout* expired and the retry limit was reached). To permit confirmations to be paired by the user with the related requests, the `id` of the packet to which the notification refers is returned. Its default behavior is to block the caller until the outcome is provided.

`SDMAC_DATA_ind()` is called by the recipient STA to wait for the arrival of a packet. The returned packet identifier `idr` is unique on the recipient. Depending on the specific implementation, it could be coupled with the index `id` chosen by the originating STA (this can be useful, e.g., for the implementation of seamless redundancy [7]).

2) MAC-layer management services: These services are meant to complement standard MLME ones, and support the additional features enabled by SDMAC. MAC implementation is partitioned in several functional blocks, either hardware (in the adapter) or software (in the device driver). Real adapters (e.g., Atheros) are typically made up of a number of queue

TABLE I
SAMPLE PROTOTYPES OF THE MAIN FUNCTIONS INCLUDED IN THE SDMAC API (NON-EXHAUSTIVE — FOR INFORMATIONAL PURPOSES ONLY).

Name	Description
`SDMAC_DATA_req(if, id, data, ac, ...)`	Send packet `id`, with payload `data`, on queue `ac` of interface `if`
`SDMAC_DATA_con(if, *id, *outcome, *ac, ...)`	Obtain transmission status `outcome` for packet `id`, on queue `ac` of interface `if`
`SDMAC_DATA_ind(if, *idr, *data, *ac, ...)`	Receive packet `idr`, bearing payload `data`, from interface `if`
`SDMAC_DATA_set(if, name, value, id, ac, ...)`	Set attribute `name` of interface `if` to value `value` (optionally, for packet `id` or queue `ac`)
`SDMAC_DATA_get(if, name, *value, id, ac, ...)`	Get value `value` of attribute `name` of interface `if` (optionally, for packet `id` or queue `ac`)

Star symbol "*" denotes a returned value, while ellipsis "..." indicates additional and optional parameters (still to be defined).

control units (QCU) for managing transmission queues, each of which is linked to exactly one DCF control unit (DCU) for dealing with channel access. Each QCU/DCU pair deals with a specific class of transmissions (e.g., a given AC). A single protocol control unit (PCU) and a single DMA receive unit (DRU) are also there.

`SDMAC_DATA_get()` and `SDMAC_DATA_set()` are designed bearing in mind the above architecture. Parameters to be read/written may refer to: 1) the adapter as a whole (e.g., BSSID, ACK timeout, and general-purpose statistics); 2) a specific queue of the adapter, as specified by `ac` (e.g., TXOP, AIFSN, $\mathrm{CW}^{\mathrm{min}}$ and $\mathrm{CW}^{\mathrm{max}}$); or 3) a single buffered packet, as specified by `id` (e.g., timestamps). In the third case, parameters can be also written/read contextually to the related `SDMAC_DATA` primitives, by augmenting the related functions with suitable arguments. Notable examples are the number of allowed tries and their rates (*request*) and the number of actually performed transmission attempts (*confirm*). To unburden programmers and increase SDMAC performance, default values can be defined for parameters.

A precise definition of the SDMAC API is out of the scope of the current paper, which only focuses on performance aspects, and is left as future work.

B. SDMAC properties

Several properties make SDMAC appealing for a number of application contexts, with and without real-time constraints. First, only features required in a particular application context have to be implemented and integrated in the driver, and the same holds for attributes describing the inner MAC status. For instance, transmission-oriented `SDMAC_DATA` functions are only needed when the default send/receive functions provided by the protocol stack do not offer the capabilities required by applications. In particular, `SDMAC_DATA_con()` is mandatory when notifications on packet delivery are needed.

SDMAC was designed bearing in mind easy porting on a wide range of devices, including updated releases of existing drivers. As a matter of fact, a number of projects exist for hard real-time implementations of protocol stacks, but they are seldom up-to-date and hardly work with the most recent network adapters. Remarkable examples are RTNet and EtherLab. RTNet provides a hard real-time protocol stack and driver for wired and wireless adapters. Unfortunately, Wi-Fi drivers are only available for few legacy adapters and are mostly in a prototype stage. Likewise, a specific hard real-time driver [8] was released for EtherLAB, but it only targets a very specific kind of Ethernet adapter (Wi-Fi is not even envisaged in this case). While adopting hard real-time drivers is definitely the best choice from a performance viewpoint, it is only acceptable for applications based on specific hardware, for which no updates are foreseen over time.

The ability to precisely control IEEE 802.11 MAC behavior by means of code in user space is the most important feature of SDMAC. It also enables the main Wi-Fi communication parameters to be tuned, even at runtime and with per-packet granularity. Besides, the availability of a well-defined SDMAC framework may dramatically reduce the time for prototyping wireless systems with specific communication needs. Developers can benefit from this flexibility, e.g., by defining custom traffic management and enhanced retransmission schemes. For example, Fig. 1 shows how a data transmission is performed, both in conventional Wi-Fi and with a custom SDMAC-based implementation. When using existing transmission services in Linux (based on sockets), management of retries and backoff is entirely performed by the adapter, and no confirmation is provided to the user when transmission ends. Conversely, by using SDMAC, it is up to the user to decide the exact way retransmissions are carried out (e.g., how many attempts can be performed and how much they are spaced).

As shown in the figure, code in user space is executed upon failure of transmission attempts, which can drive protocol operations at runtime, enforcing a specific behavior. The only drawback is that, unlike operations carried out by the adapter with precise timings (tolerances are below 1 μs), delays and jitter introduced by real SDMAC implementations may lower determinism tangibly, which directly affects the benefits it can actually offer. In order to prove SDMAC practical feasibility, this paper focuses on the experimental characterization of time-critical functions, and in particular on the use of `SDMAC_DATA_req()` and `SDMAC_DATA_con()` to perform confirmed one-shot transmissions. Mainly for space

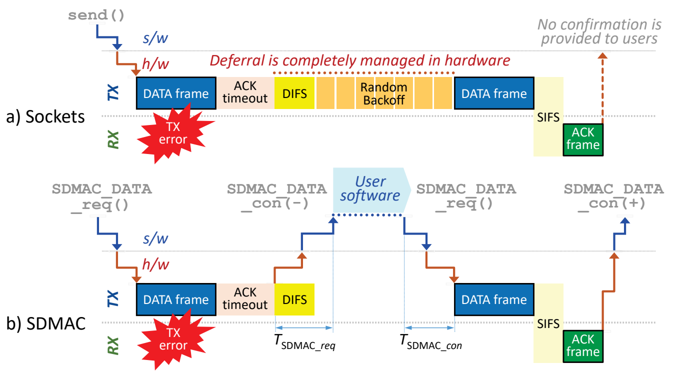


Fig. 1. Management of retransmissions (conventional sockets and SDMAC).

reasons, evaluation of `SDMAC_DATA_ind()` was not performed. In the experiments, we measured how much it takes for SDMAC to send a frame on air and obtain the outcome of delivery (kind of a round-trip delay). This latency is what actually matters for many deterministic MAC overlays, as sharing the wireless spectrum among STAs can be carried out more efficiently and predictably if delays and jitters due to frame originators (not recipients) are known in advance.

It is worth pointing out that using SDMAC to manage retransmissions in software, as shown in the lower part of Fig. 1, negatively affects throughput. However, this is not the most important performance metric for real-time applications. In these cases, the ability to precisely control how and when any single attempt, related to a specific frame transmission, takes place on air, is more important.

We decided not to analyze MAC-layer management services because they are not time-critical.

C. Application contexts that can benefit from SDMAC

1) Time Division Multiple Access: Recent Wi-Fi specifications [1] define the enhanced distributed channel access (EDCA) and hybrid-coordination-function controlled channel access (HCCA). EDCA is completely *distributed*, but relies on a *random* access scheme. Conversely, while offering *deterministic* access, HCCA relies on a *centralized* coordinator. The optimal choice in many industrial scenarios, however, is *deterministic distributed* channel access.

Time division multiple access (TDMA) [9], [10], [11], [12] is a popular distributed approach for sharing the communication support in time-critical applications, and can be applied to wireless networks by constraining each node to access the channel exclusively during its assigned time slots. If nodes are synchronized, so that they share the same time base [13], [14], it is possible to coordinate their access to the underlying network even in the absence of a repeated superframe (e.g., in Wi-Fi). The ability of STAs to send frames on air at precise instants permits to correctly dimension safety margins of time slots. The more accurate timings are, the lower the wasted bandwidth. In turn, this means higher process data rate. Therefore, knowing a priori SDMAC transmission latencies is essential to correctly configure TDMA exchanges.

2) Deadline-driven traffic scheduling: Deciding transmission order of data according to their deadlines is another context where SDMAC can be advantageous. Besides bounded transmission latency, a prompt notification of the transmission outcome (success or failure) permits the scheduler to timely select the next frame to be sent on air according to specific strategies. In addition, statistical information collected in the driver and made available to the scheduler via the SDMAC interface enables it to proactively react to environmental changes. Knowledge of the round-trip delay measured at the user layer for single confirmed transmission attempts can be profitably exploited in scheduling algorithms, e.g., the earliest deadline first (EDF), to account for the actual channel occupation of data exchanges.

For example, in [15] transmissions (and retries) of pending frames take place according to their absolute deadlines. The related EDF scheduler can be implemented as a soft real-time application in user space that relies on SDMAC. Clearly,

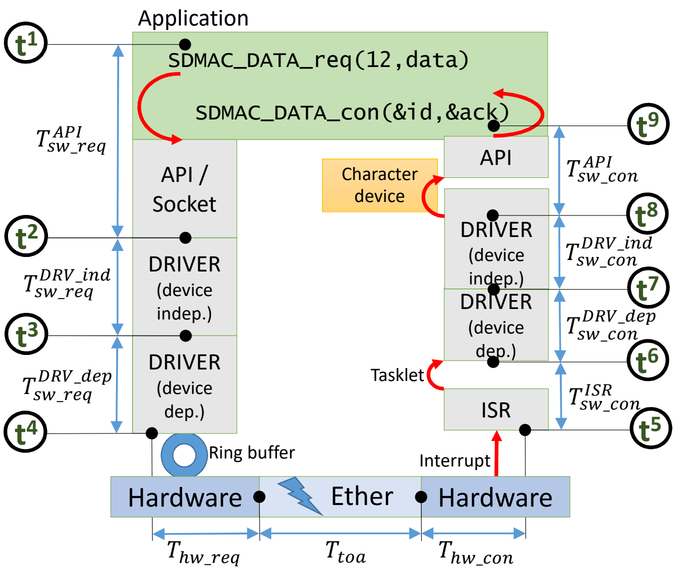


Fig. 2. SDMAC implementation schema.

resulting performance depends on SDMAC overhead and accuracy. Another relevant example is the SchedWiFi proposal [16], which combines distributed channel access and deadline-driven traffic scheduling and also supports aperiodic traffic. It is based on TDMA and makes use of a Time-Aware Shaper module to isolate high-priority traffic in predefined slots.

3) Seamless redundancy: SDMAC can be profitably employed to efficiently implement seamless Wi-Fi redundancy [17]. The ability to perform confirmed one-shot transmissions, remove packets from transmission queues, and, possibly, abort ongoing transmissions on adapters, enables sophisticated strategies, like duplication avoidance in Wi-Red [7], also including proactive heuristics based on network statistics. A preliminary, simplified implementation is described in [18].

4) Additional application contexts: Scenarios that can benefit from SDMAC include, e.g., enhanced rate adaptation techniques and real-time roaming of mobile STAs. Besides, experimental results reported here can be used to characterize in a realistic way in-node delays in network simulators.

III. SDMAC Implementation

Practical implementation of `SDMAC_DATA_req()` and `SDMAC_DATA_con()` in a typical Linux system (and the related device drivers) is schematically shown in Fig. 2.

1) `SDMAC_DATA_req()`: A reasonable option is to map the SDMAC primitive for sending data directly on POSIX *raw sockets*, and in particular on the `sendto()` function, as done in this work, or `sendmsg()`, if ancillary information have to be conveyed with the packet (e.g., parameters `id` and `ac`, as well as attributes managed by `SDMAC_DATA_set()`). The use of standard POSIX implies that the *request* path (before transmission or air) coincides with the conventional Linux protocol stack. Consequently, no changes at all have to be made to the driver. As shown on the left side of Fig. 2, the application in user space issues a call to `SDMAC_DATA_req()`, which in turn invokes `sendto()` on the related socket. After execution of the *device dependent* and *device independent* components of the driver, the packet is inserted into the *ring buffer*, which coincides with the transmission queue of one of the QCU/DCU blocks of the Wi-Fi adapter.

Implementation of confirmed one-shot transmissions requires automatic MAC layer frame retransmissions to be disabled. Depending on the specific driver and network adapter, this can be easily achieved in user space by configuring the retry limit to 0 (e.g., using the `iwconfig` command) or, if this option is not supported, by modifying the driver (as we did here). In the latter case, developers are required to find the correct position in the driver code where retransmissions can be disabled. In the following experiments, Atheros Wi-Fi adapters based on the `ath9k` driver were used. They are very popular in the research community, since the driver is available as open source and is not based on proprietary firmware.

For `ath9k`-compliant adapters, every outgoing packet is associated with a suitable memory structure (*TX descriptor*) that contains packet-specific attributes. Upon packet transmission request, the related TX descriptor is instantiated and inserted into the ring buffer by the driver. It will be fetched autonomously (in DMA) by the network adapter for transmission on air. To provide finer control on single transmission attempts, up to four *transmission series* can be defined for each descriptor. Specific fields exist to configure, e.g., the maximum number of attempts that can be automatically performed for each series (`tx_tries0/1/2/3`) and the transmission rate the adapter has to use for them (`tx_rate0/1/2/3`). Actual values for such fields are selected by the driver on a per-packet basis, typically according to the Minstrel [19], [20] algorithm. Rate adaptation is an effective way to increase communication reliability. For custom SDMAC transmission services, e.g., confirmed one-shot transmissions, it has to be implemented at user space level, either by SDMAC or by the application.

2) `SDMAC_DATA_con()`: The originating STA is notified of the outcome of each packet transmission either directly, when an ACK frame is received from the recipient, or indirectly, when it is not received before *ACKTimeout* expiration. Both events cause the network adapter to raise an interrupt, which is served as soon as possible by the operating system through the related interrupt service routine (ISR) [21]. The ISR is aimed at managing time-critical tasks. Instead, other activities are executed as a *tasklet*, which is a Linux mechanism aimed at deferring code execution so as to decrease interrupt response latencies. Tasklets are managed by the same CPU that served the interrupt (i.e., that ran the ISR), and are executed in interrupt context (i.e., they cannot be preempted by other tasks). Tasklet code can be divided in two components, *device dependent* and *device independent*. The first makes access to registers of the specific network adapter, while in the latter the same code is shared among all the devices managed by the device driver. To make integration of SDMAC into device drivers easier, we placed the code to detect transmission outcomes in the device independent part, to the detriment of latency, which worsens slightly. Specifically, for `ath9k`, it fitted in the `ieee80211_tx_status()` function.

With POSIX sockets, the *confirmation* path does not reach applications directly. Conversely, the outcome of each packet transmission in SDMAC is transferred in user space by means of a *character device*, and made available to applications

TABLE II
DESCRIPTION OF MEASUREMENT PLANES FOR TIMESTAMPS.

	Timestamp	Place in the code where the timestamp is taken
Request path	t^1	Just before `SDMAC_DATA_req()` invocation
	t^2	First instruction executed in the device driver
	t^3	First instruction of the *device dependent* code in the device driver
	t^4	Just after the packet has been queued into the *ring buffer*
Confirm. path	t^5	First instruction of the ISR associated to the device driver
	t^6	First instruction of the *tasklet*
	t^7	First instruction of the *device independent* code in the device driver
	t^8	Just before transmission outcome is written in the character device
	t^9	Just after `SDMAC_DATA_con()` releases control

through `SDMAC_DATA_con()`. To this purpose, we used a semaphore in kernel space and a blocking `read()` system call in the user space `SDMAC_DATA_con()` function.

Determinism of communication between kernel and user spaces could be improved [22], but this requires to set up a hard real-time environment, which sensibly increases configuration effort. Besides, doing so does not practically guarantee hard real-time behavior because, at present, only few prototype implementations exist of hard real-time Wi-Fi device drivers. Experimental results highlighted that the jitter induced by the character device is negligible when compared to other components. A possible alternative to character devices is the `ioctl()` function, but it requires many more changes to the driver code and was replaced in newer drivers by the *netlink* interface. Unfortunately, *netlink* worsens determinism [23].

IV. MEASUREMENT SYSTEM

To profile the most significant software blocks in the *request* and *confirmation* paths, small, specific pieces of code were added to SDMAC. As shown in Fig. 2 and Table II, several meaningful reference planes were identified. For every packet transmission, a timestamp was obtained as close as possible to each reference plane, by reading the Time Stamp Counter (TSC) register of the CPU. The path between the invocation of `SDMAC_DATA_req()` (at time t^1) and the unblocking of `SDMAC_DATA_con()` (at time t^9) can be decomposed as

$$T_{path} = t^9 - t^1 = T_{\mathrm{SDMAC}\_req} + T_{toa} + T_{\mathrm{SDMAC}\_con} \quad (1)$$

where $T_{\mathrm{SDMAC}\_req} = T_{sw\_req} + T_{hw\_req}$ and $T_{\mathrm{SDMAC}\_con} = T_{hw\_con} + T_{sw\_con}$ refer to the delays introduced by SDMAC (both software and hardware) on the request and confirmation paths, respectively, while T_{toa} is the time taken for transmission on air. Software latencies on the two paths can be further subdivided as, respectively,

$$\begin{aligned} T_{sw\_req} &= T^{API}_{sw\_req} + T^{DRV\_ind}_{sw\_req} + T^{DRV\_dep}_{sw\_req} \\ T_{sw\_con} &= T^{ISR}_{sw\_con} + T^{DRV\_dep}_{sw\_con} + T^{DRV\_ind}_{sw\_con} + T^{API}_{sw\_con}. \end{aligned} \quad (2)$$

By defining $T_{hw} = T_{hw\_req} + T_{hw\_con}$ as the overall round-trip delay due to Wi-Fi adapter hardware, the total overhead introduced by SDMAC on confirmed transmission (including both T_{SDMAC_req} and T_{SDMAC_con}) comprises all contributions to T_{path} in Fig. 2 except T_{toa} and can be expressed as

$$T_{\mathrm{SDMAC}} = T_{sw\_req} + T_{hw} + T_{sw\_con} = T_{path} - T_{toa}. \quad (3)$$

When automatic MAC retransmissions are disabled, as for confirmed one-shot transmissions, T_{toa} for packets for which an ACK was successfully received (*acked*) is given by

$$T_{toa} = T_{acc} + (T_{\text{DATA}} + T_{\text{SIFS}} + T_{\text{ACK}}) \tag{4}$$

where T_{DATA} and T_{ACK} are the durations of DATA and ACK frames, respectively, T_{SIFS} is the SIFS duration, and T_{acc} is the time taken by the MAC to access the wireless medium. T_{SIFS} is constant and only depends on the specific PHY layer. When rate adaption algorithms are disabled, also T_{DATA} and T_{ACK} are fixed and can be easily computed. Instead, T_{acc} is, by its nature, not deterministic, as it depends on the interference due to transmissions performed by nearby wireless devices.

Not every transmitted packet receives a confirmation, as either the DATA or the ACK frame may be corrupted. From the originator viewpoint, these situations coincide and denote a transmission failure. However, in the second case the recipient receives data correctly. Statistics on latencies for packets for which no ACK was received (*non-acked*) were computed separately. The related quantities are identified with superscript "*", and not necessarily correspond to acked frames. An exception is the request path, which is unaffected by the transmission outcome. For non-acked frames (3) becomes

$$T^*_{\text{SDMAC}} = T_{sw\_req} + T^*_{hw} + T^*_{sw\_con} = T^*_{path} - T^*_{toa} \tag{5}$$

where, for confirmed one-shot transmissions,

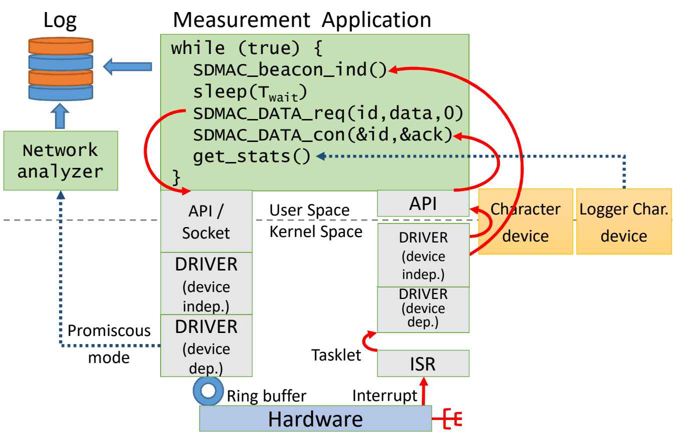
$$T^*_{toa} = T_{acc} + T_{\text{DATA}} + T_{\text{ACKTimeout}}. \tag{6}$$

A. Software architecture and testbed

Experimental evaluation of SDMAC performance was carried out using a purposely developed testbed, whose architecture is depicted in Fig. 3. The measurement system was implemented on a PC equipped with a 3.5 GHz Intel® i3-4150 CPU, Intel® B86 Chipset, and 4 GB 1600 MHz DDR3 Dual Channel RAM, running the Linux kernel v. 3.14.61 and the Ubuntu 14.04.4 LTS distribution. Energy management features and frequency scaling were disabled to improve determinism, according to Intel® guidelines [24], [6]. A dual-band TP-Link TL-WDN4800 was used as Wi-Fi adapter, managed by the `ath9k` device driver v. 4.1.1, and configured to comply with IEEE 802.11a, since this is the simplest way to prevent frame aggregation (the option to explicitly disable aggregation is planned for the final SDMAC version). A generic access point (AP) was configured to set up an infrastructure Wi-Fi network. In the context of this paper, its performance is irrelevant, because it only has to reply with an ACK frame to every DATA frame sent by the PC. Further details on the measurement system can be found in [6].

B. Measurement technique

To correctly characterize SDMAC latencies due to the hardware, we need to make $T_{acc} = 0$ in the experiments. In particular, delays caused by the clear channel assessment, which defers frame transmission when the channel is sensed busy, must be avoided. To this extent, a peculiar technique was employed, which guarantees that the channel is idle (practically) every time a transmission request is performed by the measurement



```
while (true) {
  SDMAC_beacon_ind()
  sleep(T_wait)
  SDMAC_DATA_req(id,data,0)
  SDMAC_DATA_con(&id,&ack)
  get_stats()
}
```

Fig. 3. Implementation schema of the measurement system.

TABLE III
CONFIGURATION OF PARAMETERS FOR EXPERIMENTS

Name	Description	Value
T_{beac}	Time interval between adjacent beacons	51.2 ms
T_{wait}	Offset between beacons and measurement frames	20 ms
R	Fixed transmission rate for all experiments	54 Mb/s
T_{DATA}	Duration of DATA frame transmission	36 μs
T_{SIFS}	Duration of short interframe space (SIFS)	16 μs
T_{ACK}	Duration of ACK frame transmission	44 μs
T_{slot}	Slot time	9 μs

task. After T_{acc} contribution has been removed, T_{hw} can be easily evaluated as $T_{hw} = t^5 - t^4 - (T_{\text{DATA}} + T_{\text{SIFS}} + T_{\text{ACK}})$ for acked frames and $T^*_{hw} = t^5 - t^4 - (T_{\text{DATA}} + T_{\text{ACKTimeout}})$ for non-acked ones.

A number of countermeasures were taken to prevent interference between the frames sent by the measurement task and concurrent traffic on air, as described below.

1) Channel selection: The AP in the testbed (and, as a consequence, the Wi-Fi adapter in the PC) were configured on a channel in the 5 GHz band not currently in use by others STAs. At the time the experiments were carried out, traffic on such band was, on average, by far lower than on the quite crammed 2.4 GHz band. The `iwlist` Linux command was used to discover a Wi-Fi channel on which no APs were visible in the place where the testbed was deployed. We found that channel 165 satisfied this property. Traffic on channel 165 was then monitored using WireShark, before and after each experiment. To prevent any possible interference with other frames from affecting results, network traffic was logged at runtime by the measurement system (see Section IV-B3).

2) Preventing interference with beacons: The above countermeasure alone does not solve the problem of interference. In fact, the AP used in our testbed periodically broadcasts beacon frames (every T_{beac}), which may interfere with the frames sent by the measurement task. An effective approach to prevent this from happening is to synchronize operations of the measurement task to the AP, so as to evenly interleave measurement frames and beacons. In particular, the measurement task was instructed to wait for $T_{wait} = 20\,\text{ms} \simeq T_{beac}/2$ following every beacon event before invoking `SDMAC_DATA_req()`. Function `SDMAC_beacon_ind()`, which relies on the character device, was purposely defined to notify the user application of the beacon arrival (detected in the driver). After obtaining the transmission outcome through `SDMAC_DATA_con()`, the measurement task invokes `get_stats()`, which relies on a separate character device (*logger*), to transfer timestamps $t^{2\ldots8}$ (acquired in kernel space) to user space, where they are logged to memory.

3) Preventing interference with other frames: In spite of the previous two countermeasures, the presence of sporadic transmissions on the selected channel can not be ruled out for sure. Think of, e.g., probe frames sent by mobile phones of people walking around near our testbed. This means that some samples may be affected by unexpected access delays. For them, T_{acc} is not null and unknown. In order to prevent these events from affecting statistics of SDMAC latencies, such samples have to be identified and discarded. To this purpose, a specific concurrent thread (*network analyzer*) was run on the PC, which took a timestamp on every frame received on air, with the exclusion of beacons and measurement frames generated by the testbed. This was accomplished by configuring the network interface in promiscuous mode and using the `libpcap` library to detect any such frames.

Timestamps of unexpected interfering frames, the j-th of which is denoted with t_j^{int}, were logged and used in the post analysis phase to filter out samples possibly affected by access delays. In particular, measured samples where timestamps t^4 or t^5 fell in any interval $t_j^{int} \pm 10\,\mathrm{ms}$ were discarded. It is worth pointing out that only $0.1\,\%$ of the samples were actually cast away, which confirms the effectiveness of our technique.

V. Results

Relevant settings for the experimental campaign are reported in Table III. In particular, the bit rate was set to $54\,\mathrm{Mb/s}$ (fixed) to disable the rate adaptation algorithm, hence making the quantities T_{DATA} and T_{ACK} constant. Instead, the beacon interval was shortened from the default value ($102.4\,\mathrm{ms}$) to $51.2\,\mathrm{ms}$, to double the number of acquired samples. Unless otherwise specified, the payload size of measurement frames was set to $50\,\mathrm{B}$, which is realistic for process data in industrial scenarios. Statistical indices on latency samples, which include mean value ($\overline{T}$), standard deviation (s_T), minimum ($T_{\min}$), maximum ($T_{\max}$), and the 99-, 99.9- and 99.99-percentiles (T_{p99}, $T_{p99.9}$ and $T_{p99.99}$, respectively), were computed offline from the timestamps acquired in the SDMAC testbed.

A. Interfering load

The first experimental campaign analyzes to which extent interfering tasks inside the PC affect SDMAC latencies. Among the kinds of load considered in [6], we selected the most aggressive one, that is *I/O load*: the `dd` Linux utility was invoked to transfer, through the hard disk controller, huge amounts of data on the system bus ($\sim$80 MB/s), which sensibly increases interrupt generation rate ($\sim$215 interrupts per second). On the contrary, the *no load* condition refers to an idle system, where only the tasks of a typical Linux distribution (including the graphical user interface and the `ssh` server daemon) and the measurement application are running. Each experiment lasted one day (i.e., 1 728 000 samples), and two configurations were analyzed, *baseline* and *optimized*.

TABLE IV
Experimental results for successful frame transmissions (standard Linux kernel without and with optimizations).

	Cond.	Latency	$\overline{T}$	s_T	$T_{\min}$	$T_{p99.9}$	$T_{p99.99}$	$T_{\max}$
			(μs)					
Baseline	*No load*	T_{sw_req}	**2.676**	0.361	1.745	6.274	11.720	36.519
		T_{hw}	**20.171**	0.472	18.130	21.313	21.543	22.978
		T_{sw_con}	**21.421**	2.102	19.338	27.438	30.986	59.321
		T_{SDMAC}	**44.268**	**2.095**	40.274	51.829	**61.025**	**92.331**
	I/O load	T_{sw_req}	6.034	3.158	1.527	21.215	32.441	91.535
		T_{hw}	19.940	0.612	17.878	22.133	23.012	61.665
		T_{sw_con}	21.657	2.581	18.593	50.698	76.128	373.083
		T_{SDMAC}	**47.631**	5.230	39.376	82.635	**108.314**	**398.039**
Optimized	*No load*	T_{sw_req}	1.733	0.171	1.470	3.067	7.531	16.972
		T_{hw}	19.888	0.467	17.920	21.076	21.427	44.773
		T_{sw_con}	19.545	0.443	18.564	22.849	31.383	40.592
		T_{SDMAC}	41.166	**0.676**	38.904	46.016	59.049	81.450
	I/O load	T_{sw_req}	4.178	2.014	1.557	13.487	20.306	32.148
		T_{hw}	19.904	0.610	17.813	22.271	23.206	53.511
		T_{sw_con}	22.431	2.235	18.449	36.047	50.947	63.792
		T_{SDMAC}	46.513	4.053	38.882	69.571	**90.077**	**107.014**

1) Baseline: Standard settings of the Linux distribution were left unchanged, energy management and frequency scaling were disabled [6], and the Linux kernel was optimized and recompiled for Intel® i3 CPUs (*Core 2/newer Xeon* option in *Processor family* kernel parameter). Experimental results for this configuration are reported in the upper part of Table IV.

Regarding SDMAC overhead (T_{SDMAC}), mean latency in *no load* conditions is $44.268\,\mu$s, while real-time statistical indices, $T_{p99.99}$ and $T_{\max}$, are $61.025\,\mu$s and $92.331\,\mu$s, respectively. All values are bounded and reasonably low. In presence of *I/O load*, the average latency is similar, that is $47.631\,\mu$s. Unfortunately, $T_{p99.99}$ and $T_{\max}$ grow to $108.314\,\mu$s and $398.039\,\mu$s, respectively. In this case, the 99.99-percentile is more than twice the average value, and the maximum, as expected, grew out of control. This is no surprise, because SDMAC implementation is not hard real-time.

Considering the overall path of confirmed one-shot transmissions, the biggest contributions to the latency are due to the software confirmation path ($\overline{T}_{sw_con} = 21.421\,\mu$s) and the hardware ($\overline{T}_{hw} = 20.171\,\mu$s), which are much higher than the software request path ($\overline{T}_{sw_req} = 2.676\,\mu$s). This is good news because, from the application viewpoint, the ability of a node to timely inject a frame on air is more important than the notification latency of transmission outcomes. Think, e.g., to scheduled transmissions, including TDMA schemes.

2) Optimized: In this case, some strategies have been devised and applied to increase determinism and reduce transmission latencies. We only considered *general* optimization strategies, which are not targeted to specific devices, brands, or operating system releases. All the proposed settings can be applied to conventional PCs and Wi-Fi adapters, on any multi-core CPU running a mainstream Linux distribution. Although we re-compiled the Linux kernel, performance only slightly worsens if this part of optimization is skipped. Among a relatively high number of strategies we tested, the following ones have been selected.

As for the *baseline* configuration, the kernel was compiled with the specific CPU optimizations, but disabling all kernel debugging features. Regarding optimizations that do not

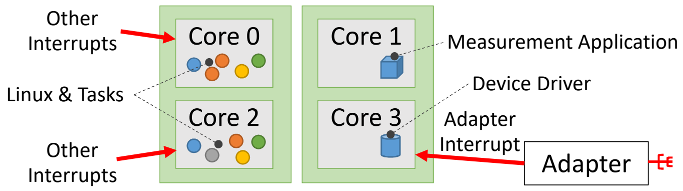


Fig. 4. Isolation between the execution of SDMAC and other tasks.

require kernel re-compilation, we partitioned the execution environment (i.e., CPU cores) so as to dedicate one core for the execution of the "industrial" application (i.e., the measurement application in the context of this paper), another for the driver, and the remaining ones for the operating system and other tasks (see Fig. 4). The CPU used in our testbed has two physical cores and four logical cores. As logical cores are not physically separate entities, results are sub-optimal. Our proposed allocation schema can be enforced by setting the default affinity of the operating system and all other tasks to logical cores 0 and 2, which reside in the same physical core (by running the kernel with boot parameter `isolcpus=1,3`). Then, the affinity of the kernel thread that executes the device driver was set to core 3 (with the `taskset` command), and the priority of the driver was set to the highest-but-one real-time priority with first in, first out scheduling policy (with the `chrt -f -p 98 <pid>` command). To better isolate tasks, we also modified interrupt affinity: all interrupts were scheduled on cores 0 and 2, with the only exception of those related to the Wi-Fi adapter, which were scheduled on core 3 (the same execution core as the device driver). Finally, the measurement application was scheduled on the reserved core 1 by means of the `taskset` command.

Results for the *optimized* configuration are reported in the lower part of Table IV. The most significant improvements brought by optimizations regard real-time T_{SDMAC} indices. In particular, for the aggressive *I/O load* condition, $T_{p99.99} = 90.077\,\mu\mathrm{s}$ and $T_{\max} = 107.014\,\mu\mathrm{s}$. All other indices improve as well, including minimum and average values. Effectiveness of optimizations is also confirmed in *no load* conditions, where determinism is remarkably better. For example, standard deviation s_T decreased from $2.095\,\mu\mathrm{s}$ to $0.676\,\mu\mathrm{s}$.

B. *Latency vs. packet size*

This experiment was aimed at evaluating the effect of the payload size on latency T_{SDMAC}. In practice, the experiment in *no load* conditions with the *optimized* configuration was repeated for different payload sizes (from 50 to 1500 B in 50 B steps). Each experiment lasted 1 hour. In Fig. 5, statistical indices of T_{SDMAC} are plotted. The shape of curves is piecewise linear. For frames up to ~ 500 B, dependence is linear and T_{SDMAC} increased by $\sim 1\,\mu\mathrm{s}$ every 150 B. When payload size is larger than 500 B, T_{SDMAC} was stable and practically stayed between $T_{\min} \simeq 42\,\mu\mathrm{s}$ and $T_{p99.9} \simeq 50\,\mu\mathrm{s}$ (as can be seen, the limited duration of experiments made, by necessity, $T_{p99.99}$ not as reliable as other statistics).

Above behavior is due to the fact that, in order to start transmission, the adapter does not wait for the whole packet to be fetched in DMA from the PC main memory. Instead, it just acquires a prefix of the packet in advance and loads the remaining part while transmitting on air.

C. *Latency vs. transmission outcome*

The last experiment, carried out in typical, realistic operating conditions (*no load*, *optimized* configuration, 50 B payload), had two main purposes. Firstly, we wanted to analyze SDMAC over a longer period of time, to assess its real-time performance with higher confidence. For this reason, the experiment lasted 7 days. Secondly, we wished to determine dependence of timings on transmission outcomes. To this purpose, we split acquired samples into three sets:

- $\mathcal{A}$: DATA frame correctly delivered and acked;
- $\mathcal{L}_D$: DATA frame lost and consequently no ACK frame;
- $\mathcal{L}_A$: DATA frame correctly delivered but ACK frame lost.

A sample is assigned to set $\mathcal{A}$ when `SDMAC_DATA_con()` notifies a successful transmission. Instead, when it returns an *ACKTimeout* event, the frame is added to either set $\mathcal{L}_A$, if correctly received by the recipient, or set $\mathcal{L}_D$ otherwise. To assign non-acked frames to sets $\mathcal{L}_A$ and $\mathcal{L}_D$, sequence numbers were included in the payload of measurement frames, and a purposely-developed program was run on the recipient side (a virtual AP running on a PC) to log sequence numbers of received frames. Statistics were evaluated on both the overall latency and individual contributions to the latency.

1) Acked frames: Results for acked frames in set $\mathcal{A}$ are reported in the leftmost part of Table V. They confirm the good real-time properties of SDMAC, even over wider time spans. As can be seen, only 1 frame out of 10000 suffered from an internal delay (due to SDMAC overhead) longer than $54.346\,\mu\mathrm{s}$ (see $T_{p99.99}$ referred to T_{SDMAC}), and the maximum is bounded to a quite low value ($T_{\max} = 77.512\,\mu\mathrm{s}$), despite the system is not hard real-time.

When individual contributions to the SDMAC latency are concerned, most of the time is spent in the T_{hw} and T_{sw_con} components. In particular, the largest part of T_{sw_con} was caused by the ISR ($T^{ISR}_{sw\_con}$) and the device dependent part of the driver ($T^{DRV\_dep}_{sw\_con}$), whose mean values were $11.911\,\mu\mathrm{s}$ and $6.364\,\mu\mathrm{s}$, respectively. This means that the time taken by SDMAC to transfer the transmission outcome from the device driver in kernel space to the application in user space ($T^{API}_{sw\_con}$) is negligible ($1.368\,\mu\mathrm{s}$, on average). Therefore, optimizations aimed at reducing such time further are practically worthless.

2) ACK timeout: Exact duration of frame transmission on air depends on its outcome. In theory, the difference between

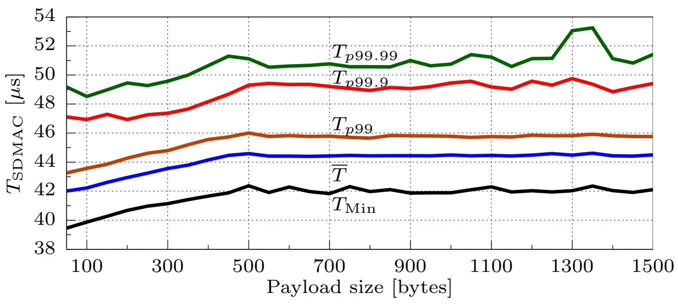


Fig. 5. Statistical indices of SDMAC latency vs. payload size.

TABLE V
EXPERIMENTAL RESULTS FOR SUCCESSFUL/FAILED TRANSMISSIONS (STANDARD LINUX KERNEL WITH OPTIMIZATIONS — ONE-WEEK RUN).

		Acked frames ($\mathcal{A}$ set) Number of samples $\lvert\mathcal{A}\rvert = 12091189$							*DATA frame is lost* ($\mathcal{L}_D$ set) Number of samples $\lvert\mathcal{L}_D\rvert = 529$					*ACK frame is lost* ($\mathcal{L}_A$ set) Number of samples $\lvert\mathcal{L}_A\rvert = 1938$				
	Latency	$\overline{T}$	s_T	$T_{\min}$	T_{p99} (μs)	$T_{p99.9}$	$T_{p99.99}$	$T_{\max}$	$\overline{T}^*$	s_{T^*}	$T^*_{\min}$ (μs)	T^*_{p99}	$T^*_{\max}$	$\overline{T}^*$	s_{T^*}	$T^*_{\min}$ (μs)	T^*_{p99}	$T^*_{\max}$
Request	$T^{API}_{sw\_req}$	0.886	0.105	0.652	1.104	1.959	3.054	9.914	0.876	0.088	0.722	1.115	1.268	0.884	0.107	0.711	1.104	2.254
	$T^{DRV\_ind}_{sw\_req}$	0.727	0.114	0.467	1.014	1.560	2.385	7.002	0.723	0.111	0.506	0.910	1.795	0.720	0.112	0.502	1.018	1.823
	$T^{DRV\_dep}_{sw\_req}$	0.797	0.082	0.598	0.974	1.749	2.735	5.968	0.799	0.099	0.647	1.010	2.025	0.794	0.081	0.647	0.977	2.367
	T_{sw_req}	2.410	0.248	1.786	2.897	3.797	7.923	21.967	2.398	0.235	1.940	2.899	3.819	2.398	0.244	1.913	2.920	6.007
Hw	T_{hw}	20.311	0.453	18.051	21.287	21.455	21.599	51.322	13.518	0.445	12.211	14.424	14.673	14.180	4.692	12.027	54.605	56.207
	$T_{hw} + T_{toa}$	**116.311**	0.453	114.051	**117.287**	117.455	117.599	147.322	**74.518**	0.445	73.211	**75.424**	75.673	75.180	4.692	73.027	115.605	117.207
Confirmation	$T^{ISR}_{sw\_con}$	**11.911**	0.236	11.118	12.321	12.520	13.635	23.597	11.882	0.212	11.626	12.341	13.315	11.890	0.213	11.604	12.325	13.173
	$T^{DRV\_dep}_{sw\_con}$	**6.364**	0.184	5.850	6.863	7.257	9.885	19.401	0.522	0.045	0.426	0.638	0.763	0.573	0.447	0.411	4.726	4.984
	$T^{DRV\_ind}_{sw\_con}$	0.119	0.049	0.050	0.334	0.428	0.464	1.498	0.092	0.033	0.045	0.171	0.188	0.092	0.032	0.045	0.170	0.299
	$T^{API}_{sw\_con}$	**1.368**	0.213	0.758	1.608	6.261	7.028	9.503	1.216	0.042	1.111	1.283	1.852	1.221	0.165	1.101	1.291	6.629
	T_{sw_con}	19.762	0.386	18.768	20.565	24.726	25.899	42.093	13.712	0.213	13.376	14.272	15.175	13.775	0.530	13.363	17.894	19.086
	T_{SDMAC}	42.483	0.662	39.268	43.943	47.534	**54.346**	**77.512**	29.628	0.528	28.157	30.894	31.585	30.353	5.125	27.858	75.011	77.050
	T_{path}	138.483	0.662	135.268	139.943	143.534	150.346	173.512	90.628	0.528	89.157	91.894	92.585	91.353	5.125	88.858	136.011	138.050

Statistics related to $T_{p99.9}$ and $T_{p99.99}$ were not reported for sets $\mathcal{L}_D$ and $\mathcal{L}_A$ because, due to the limited number of samples in such sets, they are mostly the same as $T_{\max}$.

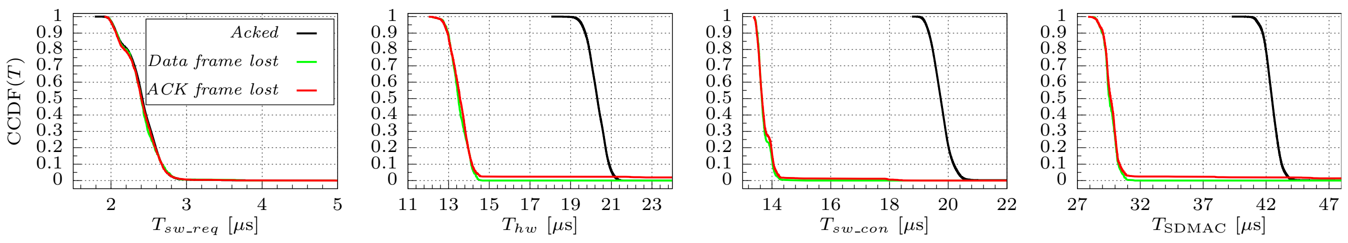


Fig. 6. Complementary cumulative distribution functions of the different contributions to the SDMAC delay for sets $\mathcal{A}$, $\mathcal{L}_D$, and $\mathcal{L}_A$.

durations of correctly performed and failed transmissions can be evaluated from equations (4) and (6), using values in Table III and setting $T_{\mathrm{ACKTimeout}}$ to the default value $50\,\mu$s reported in the IEEE 802.11 specification [1], i.e., $T_{toa} - T^*_{toa} = 10\,\mu$s. The same difference is expected when considering the time elapsing between timestamps t^4 and t^5, which corresponds to $T_{hw} + T_{toa}$ and $T^*_{hw} + T^*_{toa}$ values in Table V, for acked and non-acked frames, respectively. By comparing set $\mathcal{A}$ to set $\mathcal{L}_D$, one can see that the measured difference is, on average, sensibly larger, i.e., $116.311 - 74.518 = 41.793\,\mu$s. Very similar differences can be obtained for 99-percentiles, for which we have $117.287 - 75.424 = 41.863\,\mu$s.

The main reason of this discrepancy is that, the actual *ACKTimeout* value set in the driver was lower than the default IEEE 802.11 value. We verified that, for Wi-Fi adapters based on the `ath9k` driver, $T_{\mathrm{ACKTimeout}} = 25\,\mu$s. This is possibly due to the fact that such adapters have a short transmission range and are able to quickly detect the beginning of received frames. For this reason, when computing the latency caused by the hardware on non-acked frames, $T^*_{hw} = t^5 - t^4 - T^*_{toa}$, we set $T^*_{toa} = 61\,\mu$s (while $T_{toa} = 96\,\mu$s for acked ones). From results in the table, the hardware appears to be slightly faster when no ACKs were received, i.e., $T_{hw} - T^*_{hw} \simeq 7\,\mu$s.

Differences were slightly larger for the overall path, where, on average, $T_{path} - T^*_{path} = 138.483 - 90.628 = 47.855\,\mu$s. This is because the device-dependent code in the driver, which manages confirmations, is also slightly faster in case *ACKTimeout* expired. In particular, $T^{DRV_dep}_{sw_con} - T^{*DRV_dep}_{sw_con} \simeq 6\,\mu$s. Above results imply that, upon detection of a transmission failure, the application in the originator was notified, on average, $\sim 48\,\mu$s in advance compared to successful transmissions. This behavior can be profitably exploited by real-time applications running above SDMAC. In fact, whenever a packet is lost, more time is given to the software to react to the event. Such additional time can be used, e.g., to execute complex scheduling algorithms or heuristics aimed at counteracting the problem (e.g., by selecting another frame to be transmitted, changing the transmission rate, and so on).

Latencies for frames belonging to set $\mathcal{L}_A$ are similar, on average, to those of set $\mathcal{L}_D$. Differences are described below.

3) DATA frame loss vs. ACK frame loss: Results for non-acked frames are reported in the two rightmost parts in Table V, for sets $\mathcal{L}_D$ and $\mathcal{L}_A$, respectively. Statistics are, on average, similar. In fact, from the point of view of the originator, the loss of either a DATA frame or the related ACK are almost indistinguishable, since both conditions are detected following the lack of the ACK frame. The most significant differences regard standard deviation, higher-order percentiles, and maximum. In the case of set $\mathcal{L}_D$, when the DATA frame is lost (because of either corruption or collision), the related ACK frame is not returned by the recipient and, upon *ACKTimeout* expiry, the originator detects that the transmission has failed.

Most of the transmissions in set $\mathcal{L}_A$ experienced an *ACKTimeout* event, as for $\mathcal{L}_D$. According to the IEEE 802.11 specification, this occurs when ACK frame reception does not start within $T_{\mathrm{ACKTimeout}}$, e.g., because the PHY preamble is corrupted. However, in a few cases the ACK frame did indeed arrive to the originator, but it was corrupted. In such event, the interrupt is raised by the Wi-Fi adapter after the ACK frame has been completely read and the relevant frame check sequence (FCS) verified. Hence, failure is notified later, more or less at the same time as it would be in case of transmission success (see, e.g., the worst-case latencies $T_{\max}$ for $\mathcal{A}$ and $\mathcal{L}_A$, which are similar). From the point of view of applications, these events (which actually depend on what takes place on air) can be suitably modeled as jitters introduced by SDMAC. As a consequence, the distribution of latencies for $\mathcal{L}_A$ becomes bimodal, and higher-order percentiles and maximum are higher than $\mathcal{L}_D$. For the same reason, standard deviation of $\mathcal{L}_A$ is noticeably larger than both $\mathcal{A}$ and $\mathcal{L}_D$.

4) Validation of results: Starting from the sample standard deviation and the number of samples included in each set, confidence intervals for mean values can be easily computed. Concerning the mean latency $\overline{T}_{\mathrm{SDMAC}}$, the confidence interval for set $\mathcal{A}$, which includes more than 12 million samples, is $42.483 \pm 0.0004\,\mu$s. Reliability for $\mathcal{L}$ sets is not as good. In particular, for $\mathcal{L}_D$, which includes 529 samples, the confidence interval is $29.628 \pm 0.045\,\mu$s, while for $\mathcal{L}_A$, which includes 1938 samples but is characterized by a noticeably larger variance, it is $30.353 \pm 0.228\,\mu$s. Such intervals are narrow enough for the purposes of our analysis.

In addition, the complementary cumulative distribution functions (CCDFs) of the most important contributions to the measured latency T_{SDMAC} are reported in Fig. 6 for the three sets of samples. They provide a further confirmation of actual SDMAC determinism, and clearly highlight the differences between latencies of set $\mathcal{A}$ and those of sets $\mathcal{L}_D$ and $\mathcal{L}_A$. In particular, the leftmost plot confirms that latencies in the request path do not depend in any way on the transmission outcome. Similarly, the slight difference between $\mathcal{L}_D$ and $\mathcal{L}_A$ in the rightmost plot corroborates the hypothesis that the probability density function for the latter is bimodal.

D. Applicability of SDMAC

Time-sensitive application scenarios listed in Section II-C can be analyzed in the light of the above results.

1) Time Division Multiple Access: Safety margins width is directly related to SDMAC determinism and synchronization quality between clocks of nodes. As shown in Fig. 7, if only jitters due to SDMAC were taken into account, the frame release jitter on air would be $J_{\max} = \max(T_{\mathrm{SDMAC}\_req}) - \min(T_{\mathrm{SDMAC}\_req})$. Since T_{SDMAC_req} is not made available by our testbed, its upper bound $\hat{T} = T_{sw\_req} + T_{hw}$ can be used, which realistically implies $J_{\max} \leq \hat{T}_{\max} - \hat{T}_{\min}$.

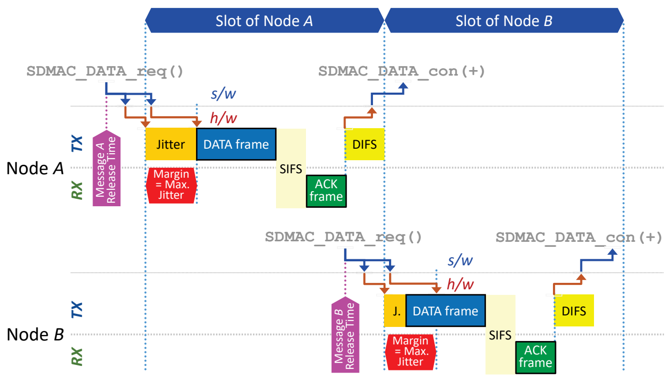


Fig. 7. Example of TDMA management using SDMAC.

In the following we refer to percentiles, as they are more statistically reliable than worst-case values. From measured delays in Table V for acked frames, a margin equal to $\hat{T}_{p99.9} - \hat{T}_{\min} = (3.797 + 21.455) - (1.786 + 18.051) = 5.415\,\mu$s is large enough to guarantee that, reasonably, only one frame out of 1000 falls out of time slot boundaries. Concerning clock synchronization over Wi-Fi, the 99.9-percentile of the measured synchronization error for an implementation based on commercial PCs [14] is $\epsilon_{p99.9} = 7.110\,\mu$s, also including scheduling jitters (which represent the main contribution to uncertainties). As a very rough approximation, setting the safety margin to $5.415 + 7.110 = 12.525\,\mu$s should be enough to ensure a similar probability that slot boundaries are not exceeded at runtime.

2) Deadline-driven traffic scheduling: SDMAC overhead is, on average, $42.483\,\mu$s for successful transmission attempts and $\sim 30\,\mu$s for failed ones, which are comparable to the typical CSMA/CA timings (i.e., $T_{DIFS} = 34\,\mu$s and $T_{slot} = 9\,\mu$s when operating in the 5 GHz band). Faster notification of failures (compared to the latencies experienced for successful attempts) and the ability to provide user space applications with statistics about the quality of the communication channel represent further advantages that justify SDMAC adoption in these application contexts.

3) Seamless redundancy: Above considerations still apply, as reasoning can be easily extended to cases where the same packet is concurrently sent on multiple adapters.

4) Additional application contexts: SDMAC applicability to other scenarios is part of our future work. Its extensive performance characterization on a real prototype is, probably, the most important milestone of this paper, and proves that a variety of time-sensitive wireless systems can be implemented with this paradigm.

VI. Conclusions

The SDMAC paradigm consists of a software overlay aimed at providing researchers, developers, and final users with full control on Wi-Fi adapters from user space applications. By relying on few and well-defined modifications to drivers, it enables effortless integration in existing commercial equipment, simple updating to newer driver releases, and supports easy customization to the features required by applications.

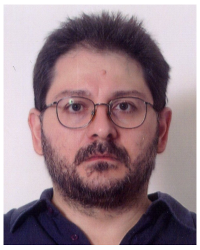
A number of guidelines are introduced and discussed in the paper, aimed at optimizing SDMAC. Results obtained this way are quite promising: for example, an experimental campaign which lasted 7 days shows that latencies in our testbed, due to the software and hardware components of SDMAC, are bounded to reasonably low values, with statistically low jitters. Despite the resulting system can not be considered hard real-time, all measured values for the SDMAC delay ranged from $39\,\mu$s to $77.5\,\mu$s, and the 99.99-percentile is about $54\,\mu$s.

The thorough performance evaluation we carried out evidences that SDMAC behavior is deterministic enough to make it an attractive enabling software component for many soft real-time applications, not only in industrial contexts.

References

[1] "IEEE Standard for Information technology–Telecommunications and information exchange between systems Local and metropolitan area

networks–Specific requirements - Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications," *IEEE Std 802.11-2016 (Rev. of IEEE Std 802.11-2012)*, pp. 1–3534, Dec 2016.
[2] C. Xu, W. Jin, G. Zhao, H. Tianfield, S. Yu, and Y. Qu, "A Novel Multipath-Transmission Supported Software Defined Wireless Network Architecture," *IEEE Access*, vol. 5, pp. 2111–2125, 2017.
[3] A.-S. K. Pathan, *Security of Self-Organizing Networks: MANET, WSN, WMN, VANET*, 1st ed. Boston, MA, USA: Auerbach Publications (Taylor & Francis Group), 2010.
[4] A. Sharma, V. Gelara, S. R. Singh, T. Korakis, P. Liu, and S. Panwar, "Implementation of a cooperative MAC protocol using a software defined radio platform," in *IEEE Workshop on Local and Metropolitan Area Networks (LANMAN)*, Sept 2008, pp. 96–101.
[5] K. Kang, Z. Zhu, D. Liu, W. Zhang, and H. Qian, "A software defined open Wi-Fi platform," *China Commun.*, vol. 14, no. 7, pp. 1–15, 2017.
[6] G. Cena, S. Scanzio, and A. Valenzano, "A software-defined MAC architecture for Wi-Fi operating in user space on conventional PCs," in *IEEE Int. Workshop on Factory Communication Systems (WFCS)*, May 2017, pp. 1–10.
[7] ——, "Seamless Link-Level Redundancy to Improve Reliability of Industrial Wi-Fi Networks," *IEEE Trans. on Ind. Informat.*, vol. 12, no. 2, pp. 608–620, April 2016.
[8] M. Cereia, I. C. Bertolotti, and S. Scanzio, "Performance of a Real-Time EtherCAT Master Under Linux," *IEEE Tran. on Ind. Informat.*, vol. 7, no. 4, pp. 679–687, Nov 2011.
[9] Y. H. Wei, Q. Leng, S. Han, A. K. Mok, W. Zhang, and M. Tomizuka, "RT-WiFi: Real-Time High-Speed Communication Protocol for Wireless Cyber-Physical Control Applications," in *IEEE Real-Time Systems Symposium (RTSS)*, Dec 2013, pp. 140–149.
[10] F. Santos, L. Almeida, and L. S. Lopes, "Self-configuration of an adaptive TDMA wireless communication protocol for teams of mobile robots," in *IEEE Int. Conf. on Emerging Technologies and Factory Automation (ETFA)*, Sept 2008, pp. 1197–1204.
[11] A. Vesco and R. Scopigno, "Advances on Time-Division Unbalanced Carrier Sense Multiple Access," in *Int. Conf. on Computer Communications and Networks (ICCCN)*, July 2011, pp. 1–6.
[12] V. Sevani, B. Raman, and P. Joshi, "Implementation-Based Evaluation of a Full-Fledged Multihop TDMA-MAC for WiFi Mesh Networks," *IEEE Trans. Mobile Comput.*, vol. 13, no. 2, pp. 392–406, Feb 2014.
[13] A. Mahmood, R. Exel, H. Trsek, and T. Sauter, "Clock Synchronization Over IEEE 802.11 - A Survey of Methodologies and Protocols," *IEEE Trans. on Ind. Informat.*, vol. 13, no. 2, pp. 907–922, April 2017.
[14] G. Cena, S. Scanzio, A. Valenzano, and C. Zunino, "Implementation and Evaluation of the Reference Broadcast Infrastructure Synchronization Protocol," *IEEE Trans. Ind. Informat.*, vol. 11, no. 3, pp. 801–811, 2015.
[15] L. Seno, G. Cena, S. Scanzio, A. Valenzano, and C. Zunino, "Enhancing Communication Determinism in Wi-Fi Networks for Soft Real-Time Industrial Applications," *IEEE Tran. on Ind. Informat.*, vol. 13, no. 2, pp. 866–876, April 2017.
[16] G. Patti, G. Alderisi, and L. L. Bello, "SchedWiFi: An innovative approach to support scheduled traffic in ad-hoc industrial IEEE 802.11 networks," in *IEEE Int. Conf. on Emerging Technologies Factory Automation (ETFA)*, Sept 2015, pp. 1–9.
[17] G. Cena, S. Scanzio, and A. Valenzano, "Experimental Evaluation of Seamless Redundancy Applied to Industrial Wi-Fi Networks," *IEEE Tran. on Ind. Informat.*, vol. 13, no. 2, pp. 856–865, April 2017.
[18] ——, "A Prototype Implementation of Wi-Fi Seamless Redundancy with Reactive Duplication Avoidance," in *IEEE Int. Conf. on Emerging Technologies and Factory Automation (ETFA)*, Sept 2018, pp. 179–186.
[19] D. Xia, J. Hart, and Q. Fu, "Evaluation of the Minstrel rate adaptation algorithm in IEEE 802.11g WLANs," in *IEEE Int. Conf. on Communications (ICC)*, June 2013, pp. 2223–2228.
[20] F. Tramarin, S. Vitturi, and M. Luvisotto, "A Dynamic Rate Selection Algorithm for IEEE 802.11 Industrial Wireless LAN," *IEEE Trans. on Ind. Informat.*, vol. 13, no. 2, pp. 846–855, April 2017.
[21] A. Mahmood, R. Exel, and T. Sauter, "Delay and Jitter Characterization for Software-Based Clock Synchronization Over WLAN Using PTP," *IEEE Trans. on Ind. Informat.*, vol. 10, no. 2, pp. 1198–1206, 2014.
[22] M. Cereia and S. Scanzio, "A user space EtherCAT master architecture for hard real-time control systems," in *IEEE Int. Conf. on Emerging Technologies Factory Automation (ETFA)*, Sept 2012, pp. 1–8.
[23] H. Trsek, S. Schwalowsky, B. Czybik, and J. Jasperneite, "Implementation of an advanced IEEE 802.11 WLAN AP for real-time wireless communications," in *IEEE Int. Conf. on Emerging Technologies and Factory Automation (ETFA)*, Sept 2011, pp. 1–4.
[24] A. Hoban, "Designing real-time solutions on embedded Intel architecture processors," *Intel Technology Journal*, vol. 16, no. 1, pp. 100–113, 2012.



Gianluca Cena (SM'09) received the Laurea degree in electronic engineering and the Ph.D. degree in information and system engineering from the Politecnico di Torino, Italy, in 1991 and 1996, respectively. Since 2005 he has been a Director of Research with the Institute of Electronics, Computer and Telecommunication Engineering, National Research Council of Italy (CNR–IEIIT), Torino.

His research interests include wired and wireless industrial communication systems, real-time protocols, and automotive networks. In these areas he has coauthored about 130 technical papers, three of which awarded as Best Papers of the 2004, 2010, and 2017 editions of the IEEE Workshop on Factory Communication Systems, and one as 2017 Best Paper for the IEEE TRANSACTIONS ON INDUSTRIAL INFORMATICS, plus one international patent.

Dr. Cena served as a Program Co-Chairman for the 2006 and 2008 editions of the IEEE International Workshop on Factory Communication Systems, and as a Track Co-Chairman in six editions of the IEEE International Conference on Emerging Technologies and Factory Automation. Since 2009 he has been an Associate Editor of the IEEE TRANSACTIONS ON INDUSTRIAL INFORMATICS.

Stefano Scanzio (S'06-M'12) received the Laurea and Ph.D. degrees in Computer Science from Politecnico di Torino, Torino, Italy, in 2004 and 2008, respectively. He was with the Department of Computer Engineering, Politecnico di Torino, from 2004 to 2009, where he was involved in research on speech recognition and, in particular, he has been active in classification methods and algorithms. Since 2009, he has been with the National Research Council of Italy (CNR), where he is a tenured Researcher with the Institute of Electronics, Computer and Telecommunication Engineering (IEIIT), Torino.

Dr. Scanzio served as a Work-in-Progress Co-Chairs in the 2018 edition of the IEEE International Workshop on Factory Communication Systems (WFCS 2018). He teaches several courses on Computer Science at Politecnico di Torino. He has authored and co-authored of more than 50 papers in international journals and conferences, in the area of industrial communication systems, real-time networks, wireless networks and clock synchronization protocols. He received the award for the best paper published in the IEEE TRANSACTIONS ON INDUSTRIAL INFORMATICS during 2016, and the Best Paper Awards for the papers he presented at the 8th and 13th IEEE Workshops on Factory Communication Systems (WFCS 2010 and WFCS 2017).

Adriano Valenzano (SM'09) received the Laurea degree magna cum laude in electronic engineering from Politecnico di Torino, Torino, Italy, in 1980. He is Director of Research with the National Research Council of Italy (CNR). He is currently with the Institute of Electronics, Computer and Telecommunication Engineering (IEIIT), Torino, Italy, where he is responsible for research concerning distributed computer systems, local area networks, and communication protocols. He has coauthored approximately 200 refereed journal and conference papers in the area of computer engineering.

Dr. Valenzano is the recipient of the 2013 IEEE IES and ABB Lifetime Contribution to Factory Automation Award. He was also awarded for the best paper published in the IEEE TRANSACTIONS ON INDUSTRIAL INFORMATICS during 2016, and received the Best Paper Awards for the papers presented at the 5th, 8th and 13th IEEE Workshops on Factory Communication Systems (WFCS 2004, WFCS 2010 and WFCS 2017).

Adriano Valenzano has served as a technical referee for several international journals and conferences, also taking part in the program committees of international events of primary importance. Since 2007, he has been serving as an Associate Editor for the IEEE TRANSACTIONS ON INDUSTRIAL INFORMATICS.