Wireless MAC layer Reconfigurability *from an SDN perspective*

Giuseppe Bianchi, University of Roma Tor Vergata

Credits to: I. Tinnirello, P. Gallo, D. Garlisi, F. Giuliano, F. Gringoli

Software-Defined Radio
from wikipedia

➔ A software-defined radio system, or SDR, is a radio communication system where **components** that have been typically implemented in hardware (e.g. mixers, filters, amplifiers, modulators/demodulators, detectors, etc.) are instead **implemented by means of software** on a personal computer or embedded system.

➔ **20+ years long research path**
- ⇨ AirBlue, CalRadio, GNURadio, RUNIC, SORA, USRP, WARP, ...

➔ **Niche commercial exploitation**
- ⇨ Military, etc

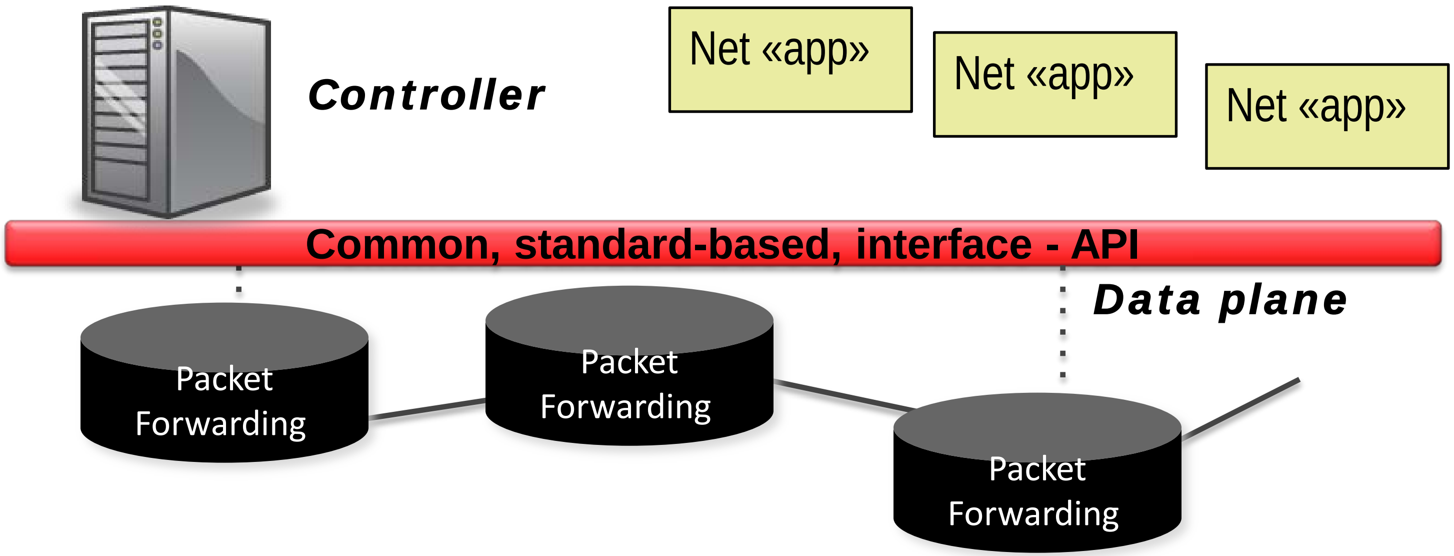
Software-Defined Networking
from wikipedia

- ➔ Software defined networking (SDN) is an approach to building computer networks that **separates and abstracts elements of these systems** [...] SDN allows network administrators to have programmable central control of network traffic without requiring physical access to the network's hardware devices.

- ➔ **5 years long research path**
  - ⇨ Pioneered by 2008 OpenFlow paper

- ➔ **almost 2B$ company acquisitions in 2012**
  - ⇨ Mainly Nicira, but also Contrail, Big Switch, Cariden, Vyatta, …

Why SDN == $$$



- ➔ **Business: provisioning and control of network services**
  - ⇨ Fostering easy deployment → fast innovation
- ➔ **Technical enabler: open configuration APIs**
  - ⇨ e.g. OpenFlow
  - ⇨ *but SDN is NOT (just) OpenFlow*

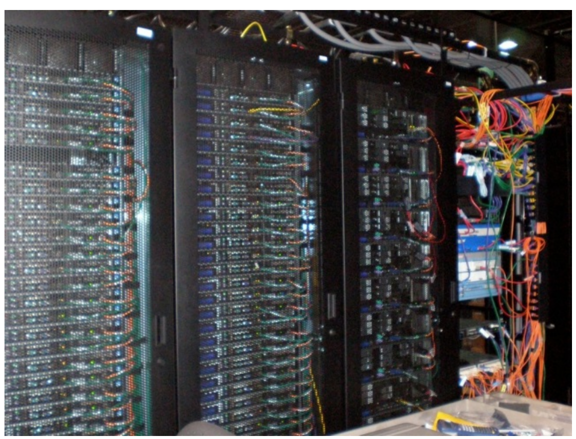
SDN: it's all about abstractions

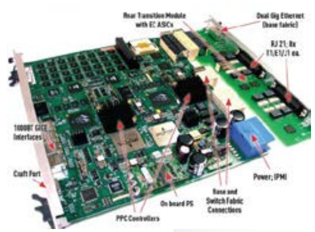
So far mostly dealt with in wired networks

Node behavior Description (formal)

Network Entity

e.g. Switch





Any vendor, any size, any HW/SW platform...

OpenFlow: a compromise
[original quotes: from OF 2008 paper]

- **Best approach**: "persuade commercial name-brand equipment vendors to provide an open, programmable, virtualized platform on their switches and routers"
  - Plainly speaking: *open the box!! No way…*

- **Viable approach**: "compromise on generality and seek a degree of switch flexibility that is
  - High performance and low cost
  - Capable of supporting a broad range of research
  - **Consistent with vendors' need for closed platforms**.

A successful compromise, indeed… ask Nicira …

OpenFlow: just *one* abstraction
good for switches, not for «all»

Matching Rule	Action

1. FORWARD TO PORT
2. ENCAPSULATE&FORWARD
3. DROP
4. …

Extensible

Switch Port	MAC src	MAC dst	Eth type	VLAN ID	IP Src	IP Dst	IP Prot	TCP sport	TCP dport

What about SDN in wireless?

- ➔ **Wireless Openflow…**
  - ⇨ Wireless specific actions: very helpful…
  - ⇨ … but match/action API way too skinny
    - ➔ We all agree now: SDN >> OpenFlow

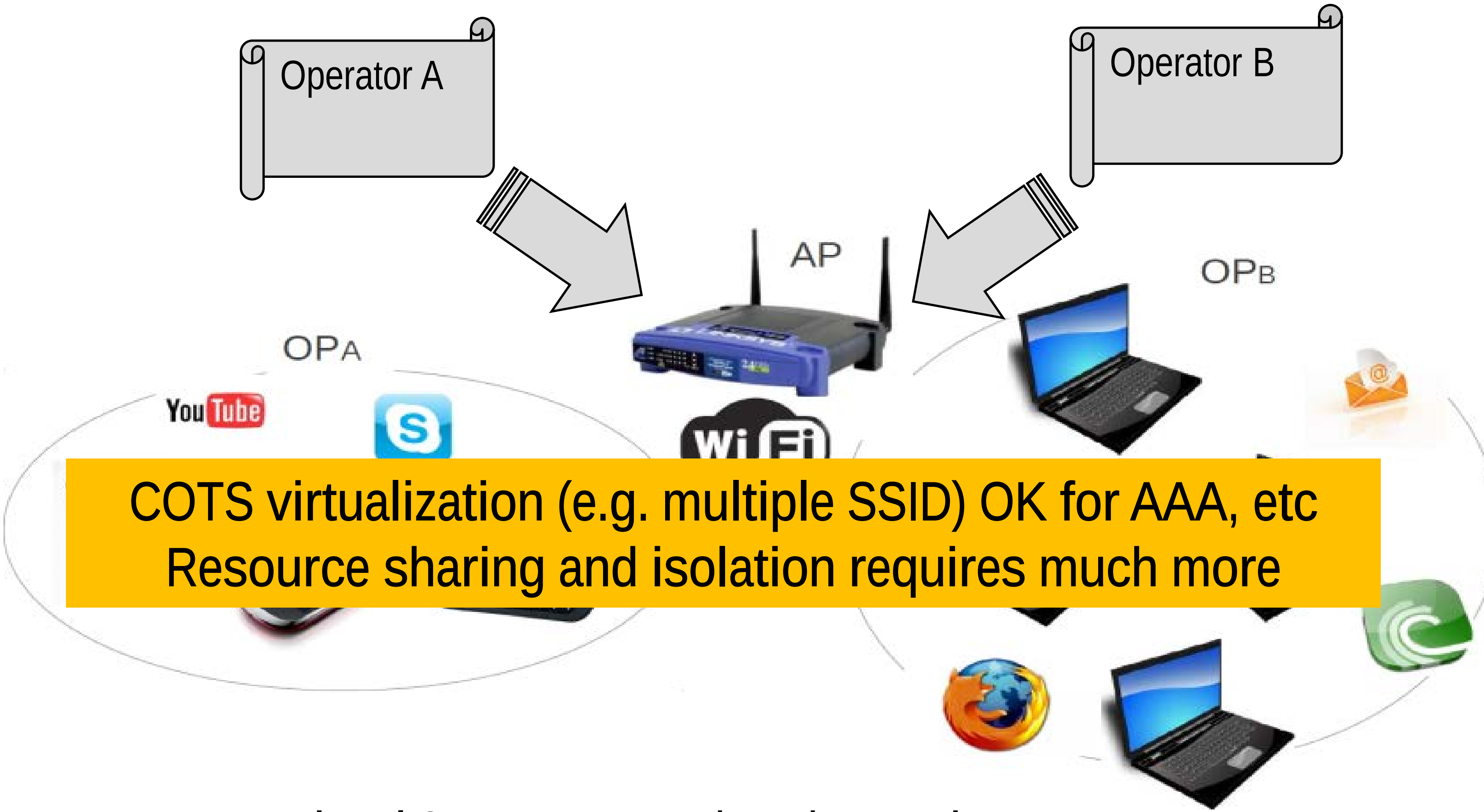
- ➔ **Challenge: which programming abstractions for wireless terminals and nodes?**
  - ⇨ Without requiring to «open the box»

Beneficial to multiple scenarios

- ➔**Dynamic spectrum access**
- ➔**Cognitive**
- ➔**Performance optimization in niche environments**
  - →home, industrial, …
  - →Adaptation to specific context or applications
- ➔**Improved support for new PHY**
- ➔**Virtualization and access network sharing**

- ➔**And many more…**

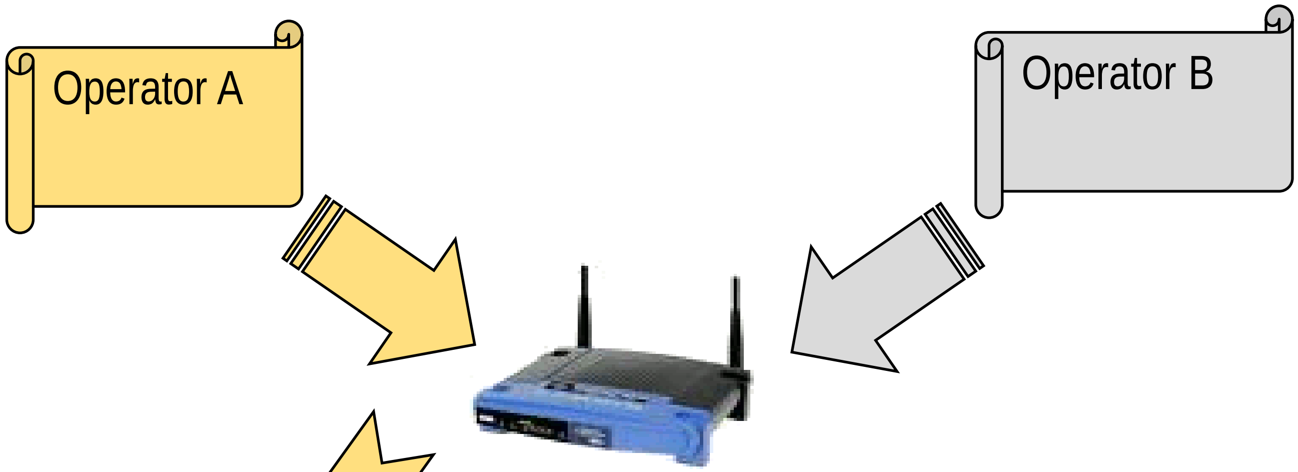
A basic (but compelling) use case: multi-tenant WLAN sharing



COTS virtualization (e.g. multiple SSID) OK for AAA, etc
Resource sharing and isolation requires much more

Virtual Operators over shared WLAN instractructure
e.g., airport, hotel, enterprise, etc

Well, we might «hack» this



← Send suitable 802.11e parameters in OP_A Beacon
… must be carefully chosen to coexist with OP_B

← Configure via 802.11v params

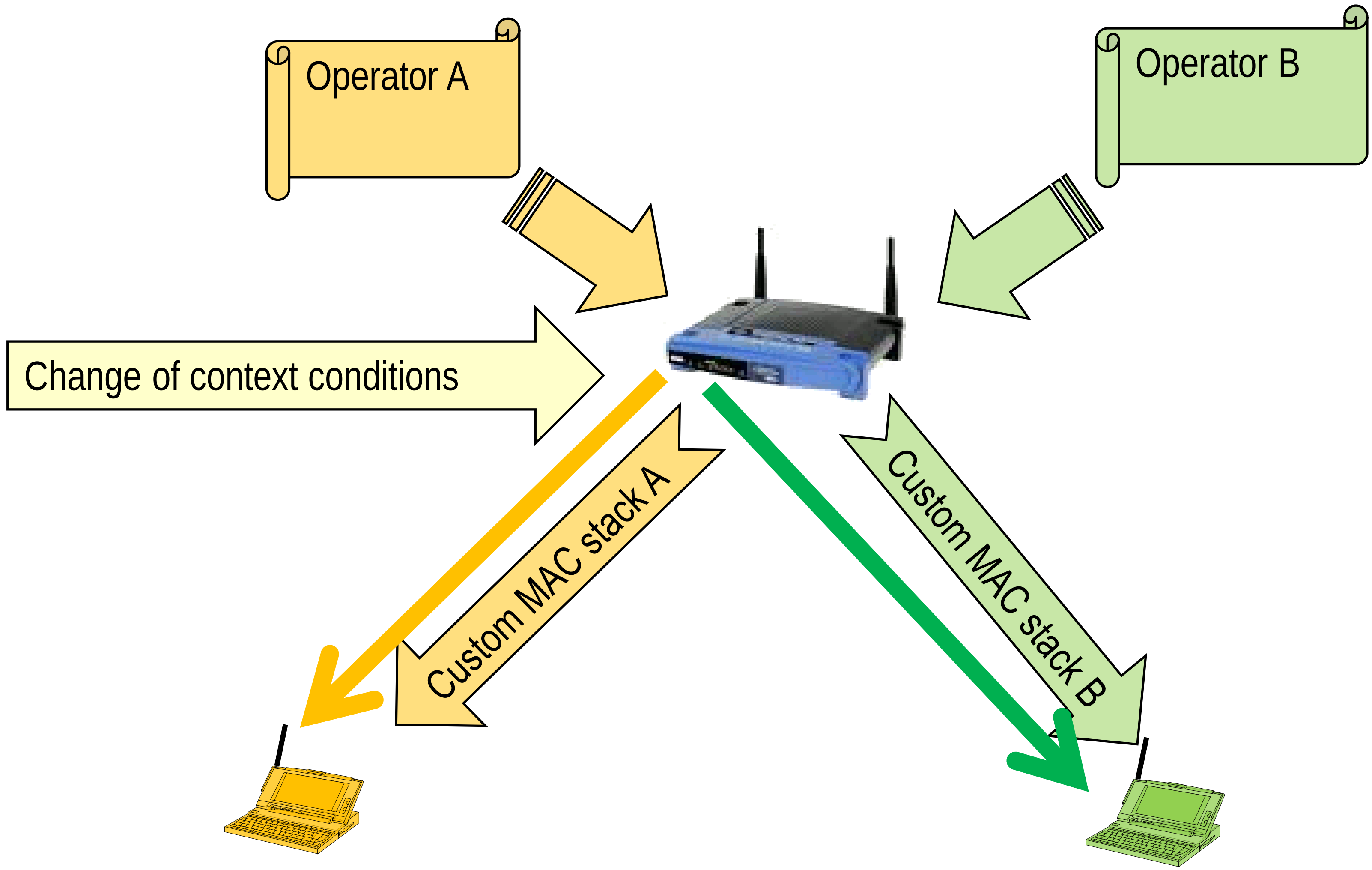
← Custom operator's protocol
… if further differentiation among terminals needed

The point is another

- ➔ **All-in-one MAC protocol, e.g. 802.11**
  - ⇨ We can probably stretch it to fit our context
    - → Creative parameter configs, overlay tricks, …
    - → We are good at mastering complexity
      - » and brings to accepted papers
  - ⇨ When impossible? Just promote an amendment!

- ➔ **But what if… we could change the MAC protocol for each and every context?**
  - ⇨ And we could trivially program our MAC operation?
  - ⇨ Much simpler!
  - ⇨ No anymore amendments, unless HW changes

Vision: Software-Defined MAC…



<u>Whole</u> MAC protocol stack as a sort of JAVA applet

… but…

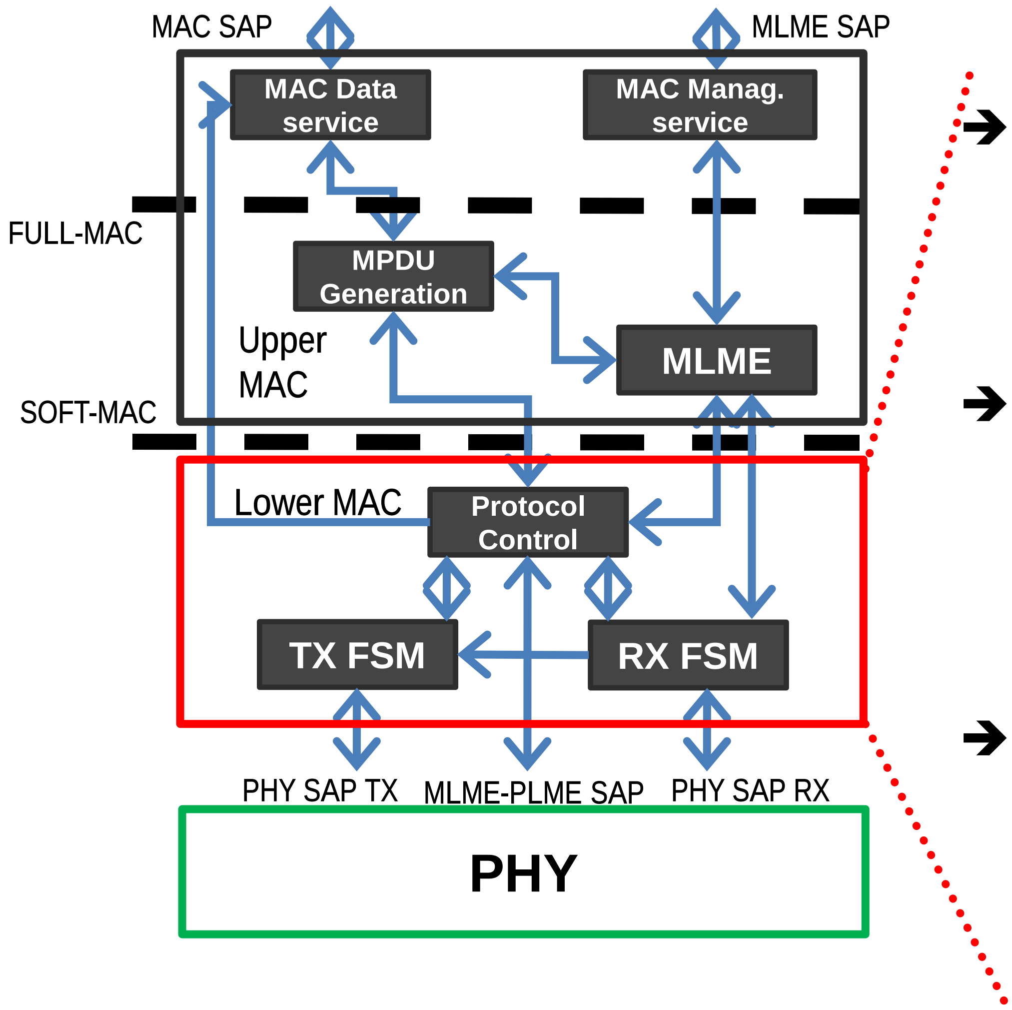
- ➔ **Best approach:** "persuade commercial name-brand equipment vendors to provide an open programmable platform on their Wireless NICs"
  - ⇨ Plainly speaking: *let me hack your NIC!! No way…*

- ➔ **Viable approach:** "compromise on generality and seek a degree of Wireless NIC flexibility that is
  - ⇨ High performance and low cost
  - ⇨ Capable of supporting a broad range of research
  - ⇨ **Consistent with vendors' need for closed platforms.**

Compromise in Wireless MAC cannot be just a rule-action table!

Current SW coding is wrong answer

[even assuming Boxes are opened]



- **DSP/FPGA SDR boards**
  - Cost, performance: just for research
  - «open» box approach: must convince vendors
- **Open firmware**
  - Probably only openFWWF, sneaked out…
  - not "much" (?!) vendor support
- **BUT in both cases…**
  - Huge skills/experience, low level languages, inter-module dependencies
    - Assembly, VHDL, low level C, …
  - Complexity! Slow deployment time

Right answer

- ➔ **Find the right abstractions!**
  - ⇨ Must yield simple programming models
  - ⇨ Must not impair performance
  - ⇨ Sufficient flexibility to support most customization needs
  - ⇨ Must be «vendor-friendly» ☺

- ➔ **Our own attempt at this:**
  - ⇨ Wireless MAC processor: Computing environment and abstractions for programming MAC protocols
  - ⇨ MAClets: from offline programming to online, dynamic, MAC stack injection and ultra fast reconfiguration, << 1 micro second

Learn from computing systems?

➔ **1: Instruction sets**
perform elementary tasks on the platform

→A-priori given by the platform

→Can be VERY rich in special purpose computing platforms

» Crypto accelerators, GPUs, DSPs, etc

➔ **2: Programming languages**
sequence of such instructions + conditions

⇨ Convey desired platform's operation or algorithm

➔ **3: Central Processing Unit (CPU)**
execute program over the platform

⇨ Unaware of what the program specifically does

⇨ Fetch/invoke instructions, update registers, etc

Clear decoupling between:
- platform's vendor → implements (closed source!) instruction set & CPU
- programmer → produces SW code in given language

Learn from computing systems?

➔ **1: Instruction sets**
perform elementary ta... ...n the platform

- →A-priori given by the ...
- →Can be VERY rich i... ...ose computing platforms
  - » Crypto accelerat... ...tc

Let's MIMIC all this!

➔ **2: Prog...**
sequence ... ...ions

- ⇨ Convey desired ... ...hm

➔ **3: Central Pr...**
execute pro... ...r the pl...

- ⇨ Unaware o... ...he program specifically d...
- ⇨ Fetch/inv...ke instructions, update registers, etc

Clear decoupling between:
- platform's vendor ➔ implements (closed source!) instruction set & CPU
- programmer ➔ produces SW code in given language

1: Which elementary MAC tasks?
("our" instruction set!)

- ➔ **ACTIONS**
  - ⇨ frame management, radio control, time scheduling
    - →TX frame, set PHY params, RX frame, set timer, freeze counter, build header, forge frame, switch channel, etc
- ➔ **EVENTS**
  - ⇨ available HW/SW signals/interrupts
    - →Busy channel signal, RX indication, inqueued frame, end timer, etc
- ➔ **CONDITIONS**
  - ⇨ boolean/arithmetic tests on available registers/info
    - →Frame address == X, queue length >0, ACK received, power level < P, etc

Actually implemented API

Platform: Broadcom Airforce54g commodity card

events	*actions*	*conditions*
CH_UP	set/get(reg, value)	dstaddr
CH_DOWN	switch_RX()	myaddr
RCV_ACK	tx_ACK()	queue_length
RCV_DATA	tx_beacon()	queue_type
RCV_PLCP	tx_data()	cw
RCV_RTS	tx_RTS()	cwmin
RCV_CTS	tx_CTS()	cwmax
RCV_BEACON	switch_TX()	backoff
HEADER_END	set_timer(value)	RTS_thr
COLLISION	set_bk()	ACK_on
MED_DATA_CONF	freeze_bk()	srcaddr
MED_DATA_START	update_retry()	frame_type
MED_DATA_END	more_frag()	fragment
QUEUE_OUT_UP	prepare_header()	channel
QUEUE_IN_OVER		tx_power
END_TIMER		

Actually implemented API

Platform: Broadcom Airforce54g commodity card

events	*actions*	*conditions*
CH_UP	set/get(reg, value)	dstaddr
CH_DOWN	switch_RX()	myaddr

Just "one" possible API

convenient on our commodity platform

"others" possible as well

improved/extended

tailored to more capable radio HW

Our point:

<u>have a specified set</u> of actions/events/conditions,

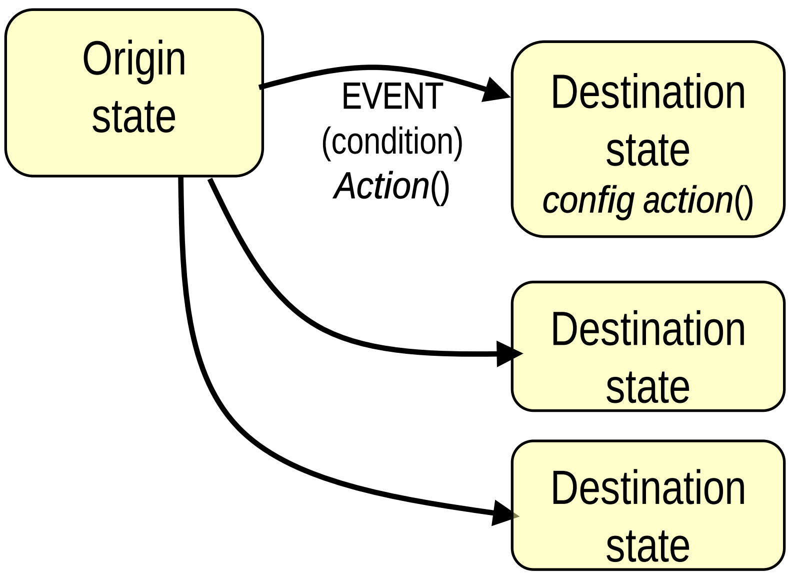
not "<u>which</u>" specific one)

2: How to compose MAC tasks?

(“our” programming language!)

➔ **Convenient “language”: XFSM**
eXtended Finite State Machines

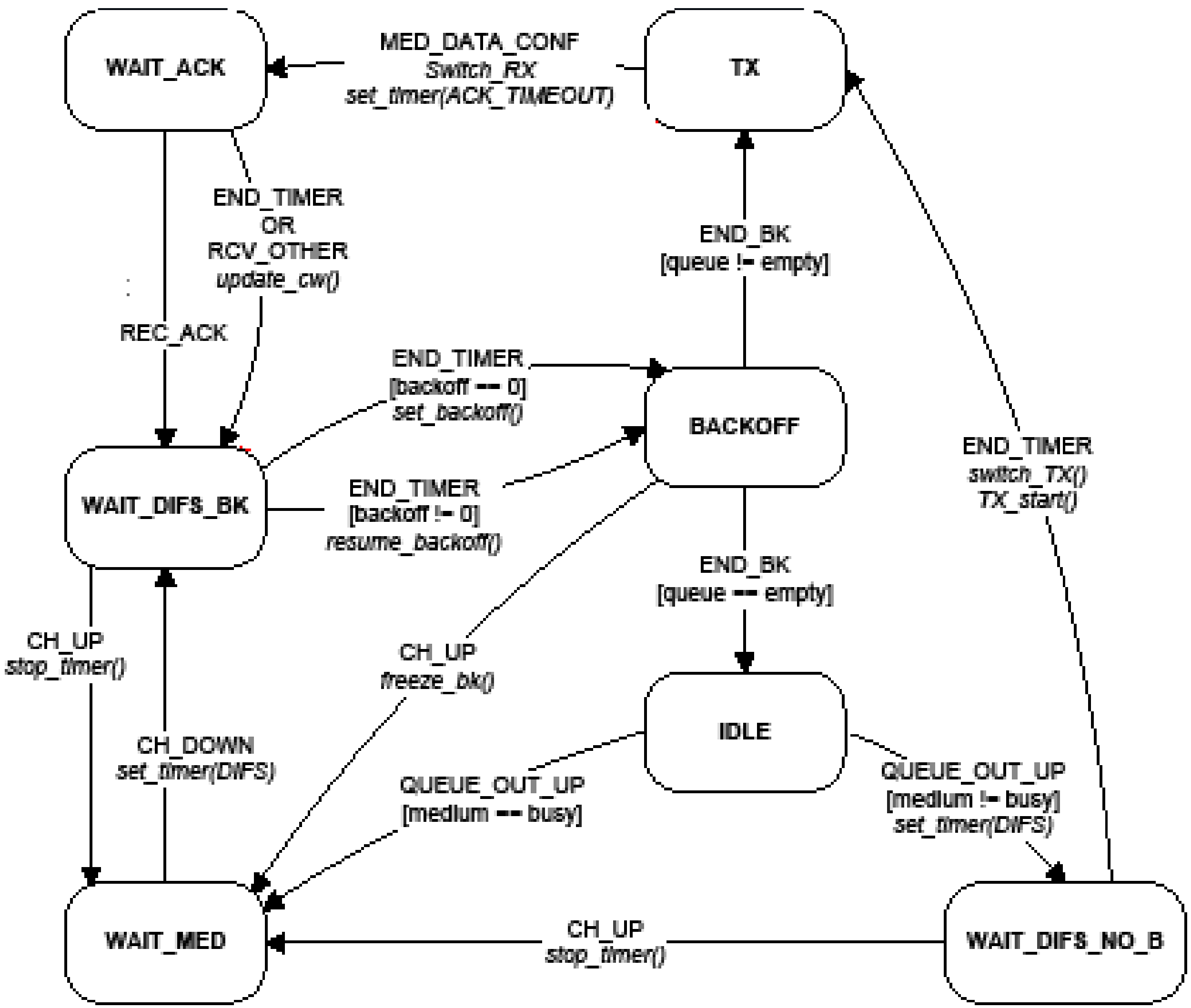
⇨ Compact way for composing available acts/ev/cond to form a custom MAC protocol logic



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

XFSM example: legacy DCF

simplified for graphical convenience



Actions:
set_timer, stop_timer, set_backoff, resume_backoff, update_cw, switch_TX, TX_start

Events:
END_TIMER, QUEUE_OUT_UP, CH_DOWN, CH_UP, END_BK, MED_DATA_CONF

Conditions:
medium, backoff, queue

3: How to run a MAC program?

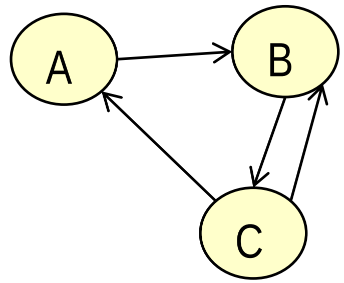
(MAC engine – XFSM onboard executor - our CPU!)

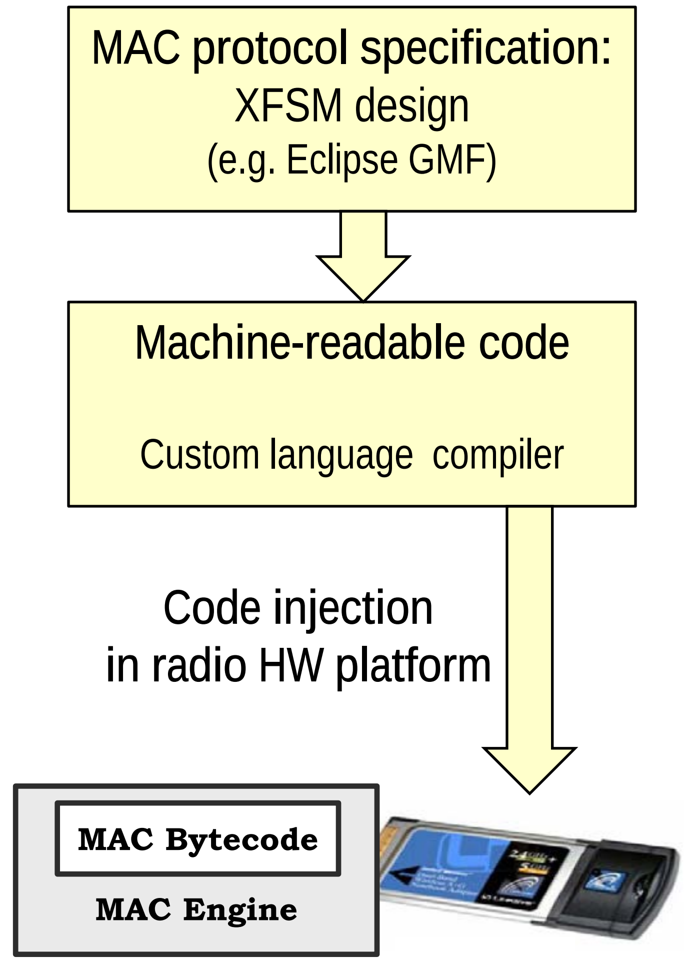
- ➔ **MAC engine: specialized XFSM executor** *(unaware of MAC logic)*
  - ⇨ Fetch state
  - ⇨ Receive events
  - ⇨ Verify conditions
  - ⇨ Perform actions and state transition
- ➔ **Once-for-all "vendor"-implemented in NIC** *(no need for open source)*
  - ⇨ "close" to radio resources = straightforward real-time handling

MAC Programs

➔ **MAC description:**

- ⇨ XFSM





➔ **XFSM ➔ tables**

	A	B	C
A		T(A,B)	
B			T(B,C)
C	T(C,A)	T(C,B)	

➔ **Transitions**

- ⇨ «byte»-code event, condition, action
  - ➔**Portable over different vendors' devices, as long as API is the same!!**
- ⇨ Pack & optimize in WMP «machine-language» bytecode

A	T(A,B)	
B	T(B,C)	
C	T(C,A)	T(C,B)

Machine Language Example

(DCF, 544 bytes)

Memory address | **Memory** | **Description**

Initial State Descriptor

Memory address								
0x0BC0:	**0100**	FFFF	0B00	0014	A5FF	6ADA	0014	A5FF
0x0BD0:	6ADA	6C00	80A4	FF00	FF00	3600	80EE	FF00
0x0BE0:	0000	0000	0000	0000	0000	0000	0000	0000

Coded state machine

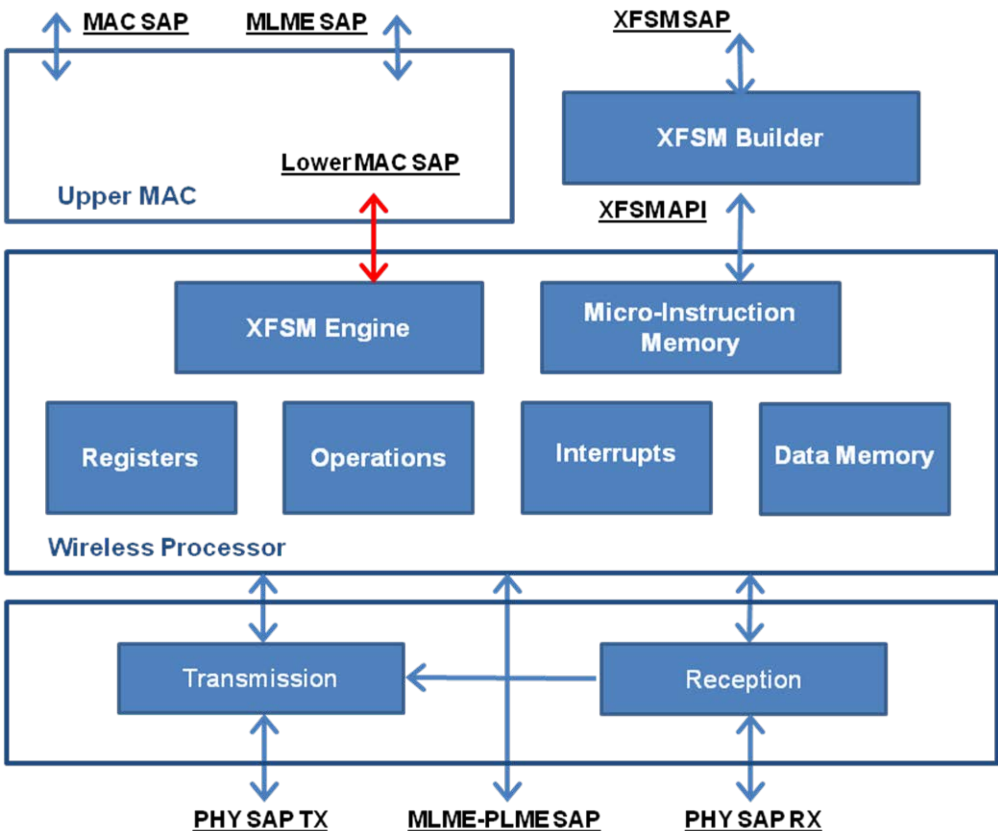
Memory address	00	01	02	03				
0x0C00:	0100	0100	0100	**0401**	**0108**	**0508**	**1C01**	**010B**
0x0C10:	**010B**	**3001**	**010D**	**0200**	**FFFF**	5101	010E	030D
0x0C20:	0000	0100	010F	C100	0102	0602	E100	0106
0x0C30:	0106	0401	0108	0508	1C01	010B	030B	FFFF
0x0C40:	CD00	0104	0E0C	0000	0100	0D00	FFFF	0E01
0x0C50:	0109	0909	1C01	010B	0D0B	FFFF	C700	0103
0x0C60:	0C03	E100	0106	0106	FFFF	6601	0110	1600
0x0C70:	0000	0100	0100	FFFF	0E01	0109	0109	1C01
0x0C80:	010B	010B	FFFF	5F01	010F	0A00	0000	0100
0x0C90:	0D00	FFFF	C100	0102	0A02	C700	0103	0B03
0x0CA0:	E100	0106	0D06	FFFF	D300	0105	0D05	E100
0x0CB0:	0106	0D06	FFFF	D300	0105	0705	E100	0106
0x0CC0:	0106	FFFF	6D01	0111	1800	0000	0100	0100
0x0CD0:	0000	0100	0D10	7401	0112	1512	0000	0100
0x0CE0:	1111	9601	0113	0513	0000	0100	0500	0000
0x0CF0:	0100	0304	E100	0106	1206	0401	0108	0508
0x0D00:	1C01	010B	120B	FFFF	A901	0115	0100	B401
0x0D10:	0117	1200	0000	0100	0100	0000	0100	1214
0x0D20:	B901	0118	0310	0000	0100	0300	0401	0108
0x0D30:	1708	1501	010A	010A	1C01	010B	010B	C501
0x0D40:	0119	0800	0000	0100	0500	3001	010D	0200
0x0D50:	0401	0108	0508	1C01	010B	180B	CB01	011A
0x0D60:	0200	0000	0100	0100	0000	0000	0000	0000
0x0D70:	0000	0000	0000	0000	0000	0000	0000	0000
0x0D80:	0000	0000	0000	0000	0000	0000	0000	0000
	00	01						
0x0D90:	00F0	**03FE**	0DF2	13FE	20FE	27FE	2EFE	35FE
0x0DA0:	0000	0000	0000	0000	0000	0000	0000	0000

```
Outgoing transitions
for state 01
0401 0108 0508 = trans. 1
1C01 010B 010B = trans. 2
3001 010D 0200 = trans. 3
FFFF           = delimiter
```

```
Transition 1
0401 = event pointer
01   = event parameter
08   = event index
05   = target state
08   = action
```

State **01**
03 = transitions offset (9 bits)
E = FFFF delimiter

Wireless MAC Processor: Overall architecture



- ➔ <u>**MAC Engine:**</u> **XFSM executor**
- ➔ <u>**Memory blocks:**</u> **data, prog**
- ➔ <u>**Registers:**</u> **save system state (*conditions*);**
- ➔ <u>**Interrupts block**</u> **passing HW signals to Engine *(events)*;**
- ➔ <u>**Operations**</u> **invoked by the engine for driving the hardware (*actions*)**

From MAC Programs to MACLets

➔Upload MAC program on NIC from remote

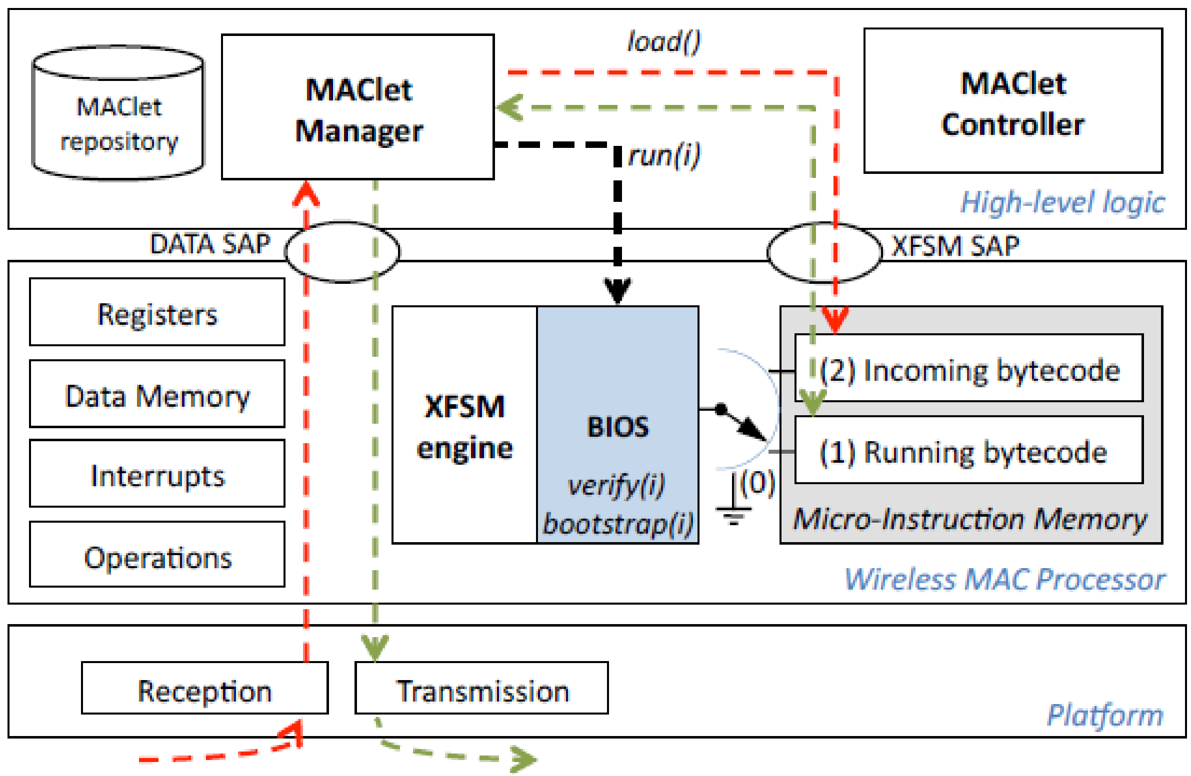
- ⇨While another MAC is running
- ⇨Embed code in ordinary packets

➔ WMP Control Primitives

- ⇨ load(XFSM)
- ⇨ run(XFSM)
- ⇨ verify(XFSM)
- ⇨ switch(XFSM1, XFSM2, ev, cond)

➔ Further primitives

- ⇨ Synchro support for distributed start of same MAC operation
- ⇨ Distribution protocol



"Bios" state machine: DEFAULT protocol (e.g. wifi) which all terminals understand

From theory to practice

- ➔ **Obviously, instruction set and MAC Engine can be "easily" implemented in a software-defined radio…**
  - ⇨ e.g., FPGA, WARP, …
- ➔ **But… can this be done on commodity HW?**
  - ⇨ e.g., ultra-cheap ordinary WLAN NIC
- ➔ **Yes!!!**
  - ⇨ Reference platform: broadcom Airforce54g 4311/4318
    - →Hands-on experience on card's assembly language FW
    - →general purpose processor (88 MHz), 64 registers, 4KB data memory, 32 KB code memory
  - ⇨ Partly leveraging existing card HW facilities
    - →HW configuration registers for radio resource and event handling
    - →Frequency, power, channel sensing, frame forging facilities, etc
    - →Available HW events (packet queued, plcp end, rx end, rx correct frame, crc failure, timer expiration, carrier sense, etc)

Implementation at a glance

- ➔ **Delete 802.11 firmware**
  - ⇨ Both Broadcom and openFWWF
    we do NOT want yet another firmware MAC to hack!
- ➔ **Replace it with** *[once for all developed]***:**
  - ⇨ Implementation of actions, events, conditions
    - → in part reusing existing HW facilities
  - ⇨ MAC engine: XFSM executor
- ➔ **Develop "machine language" for MAC engine**
  - ⇨ Custom made "bytecode" specified and implemented
    - → 6 bytes instructions, state transition table (sparseness exploited)
- ➔ **Address several annoying technical hurdles**
  - ⇨ NO direct HW interrupts control available in Broadcom
  - ⇨ State and state transition optimizations, …

Public-domain

- **Supported by the FLAVIA EU FP7 project**
  - http://www.ict-flavia.eu/
    - **general coordinator:** giuseppe.bianchi@uniroma2.it
    - **Technical coordinator:** ilenia.tinnirello@tti.unipa.it
- **Public domain release in alpha version**
  - https://github.com/ict-flavia/Wireless-MAC-Processor.git
  - Developer team:
    - **ilenia.tinnirello@tti.unipa.it**
    - **domenico.garlisi@dieet.unipa.it**
    - **fabrizio.giuliano@dieet.unipa.it**
    - **francesco.gringoli@ing.unibs.it**
- **Released distribution:**
  - Binary image for WMP
  - You DO NOT need it open source!
    Remember the "hard-coded" device philosophy…
    - Conveniently mounted and run on Linksis or Alix
  - Source code for everything else
  - Manual & documentation, sample programs

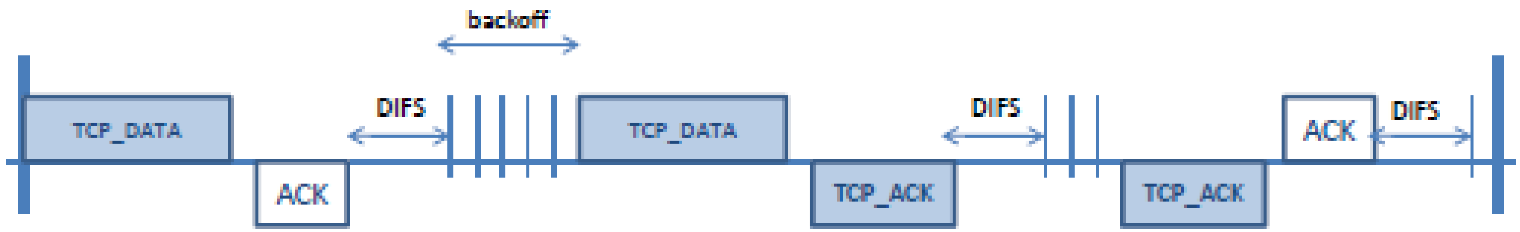
WMP Functional validation «static» MAC programs

Success IF WMP permits very easy/fast Lower MAC modifications or re-design (vs months or hands-on experience with openFWWF/assembly

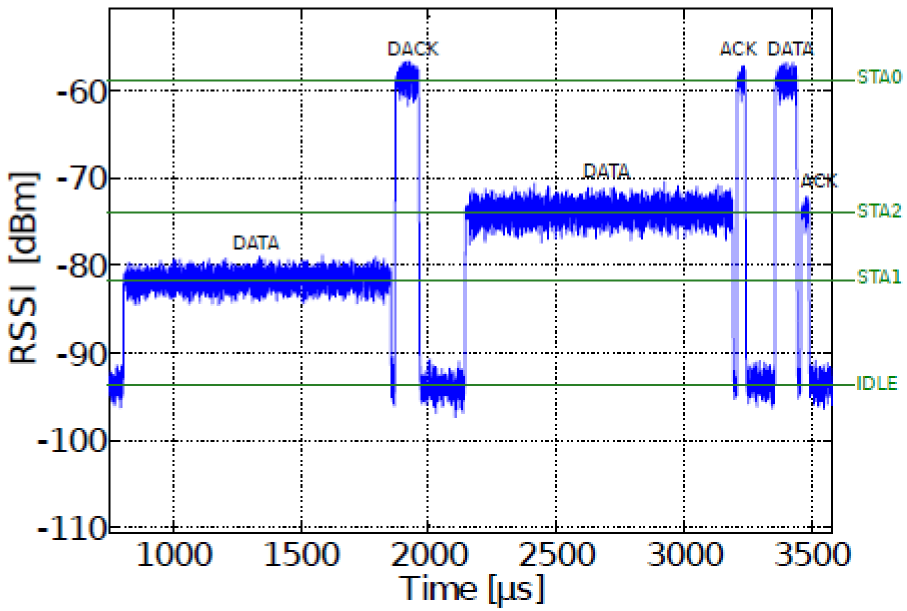
- **"scientifically trivial" use cases, tackling distinct MAC aspects recurring in literature proposals**
  - Piggybacked ACK
    - Programmable management of frame replies
  - Pseudo-TDMA
    - Precise scheduling of the medium access times
  - Randomized multi-channel access
    - Fine-grained radio channels control
  - Multi-tenant access network sharing, with different protocols
    - virtualization
- **Development time: O(days)**
  - Including bug fixing in engine/API, otherwise hours

Piggybacked ACK

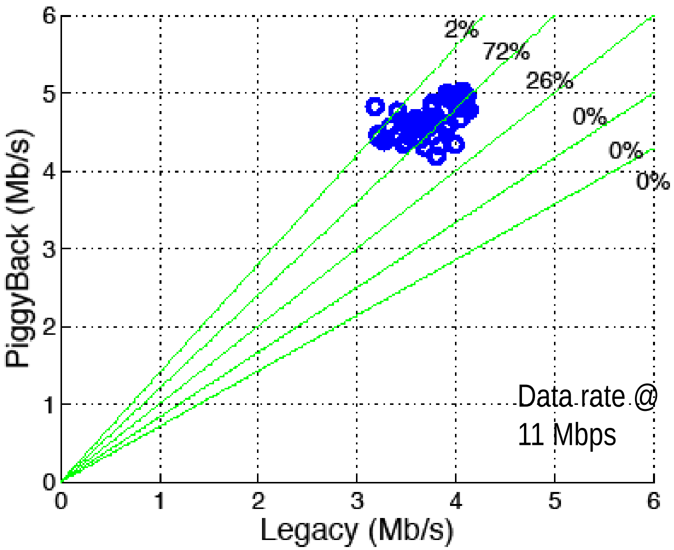
If available, send TCP ACK instead of MAC ACK, otherwise send normal ACK



Channel activity trace



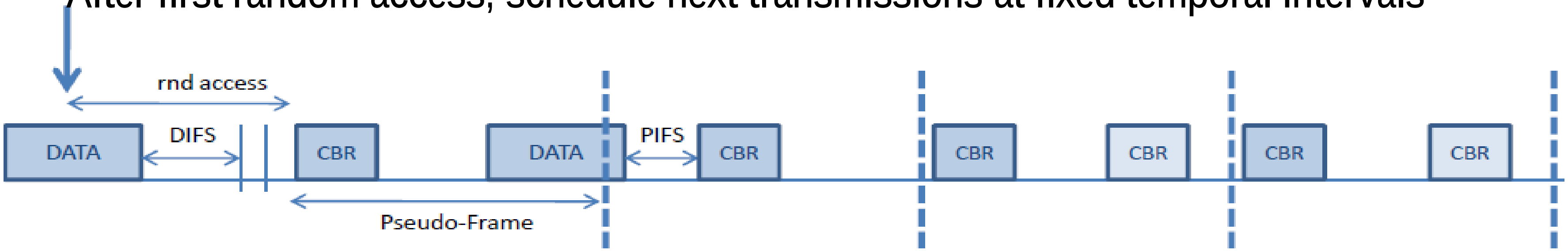
Performance gain

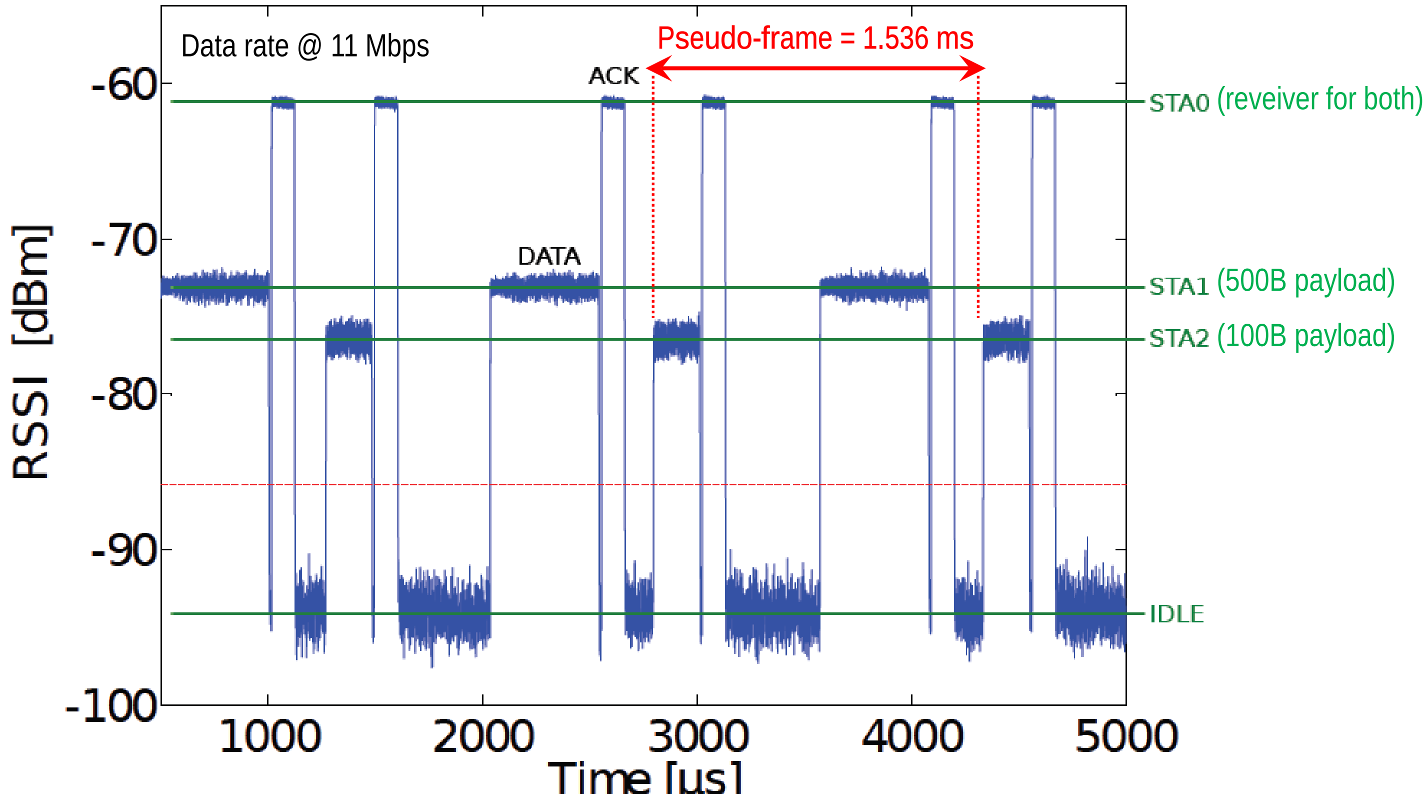


Pseudo-TDMA

[literature proposal]

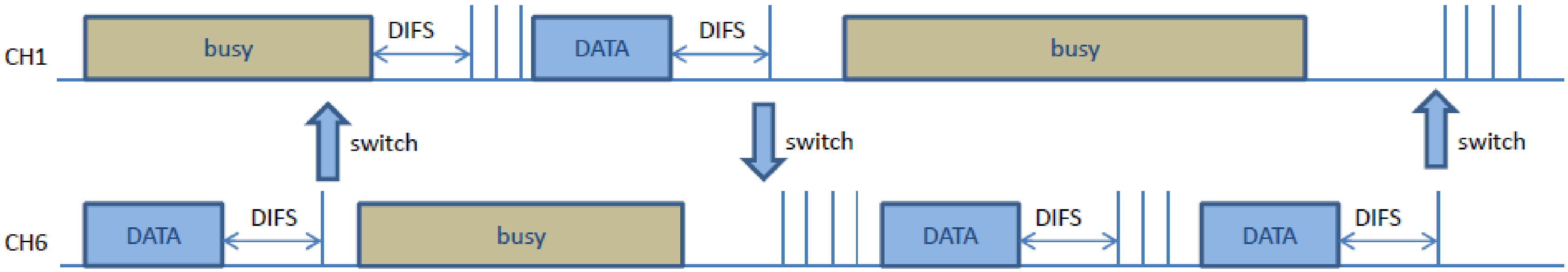
After first random access, schedule next transmissions at fixed temporal intervals



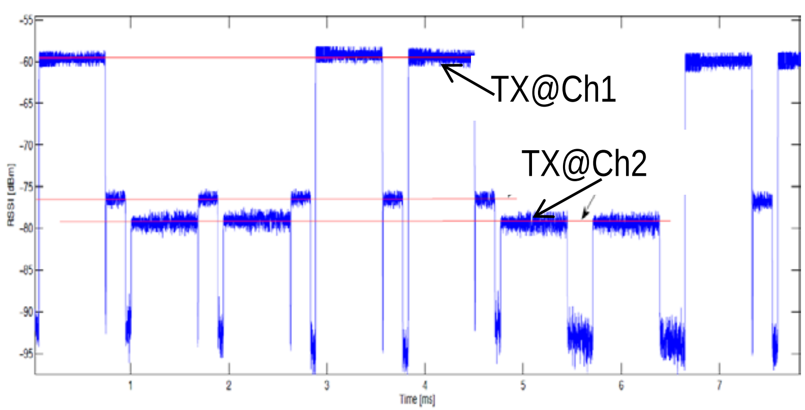


Randomized multichannel access

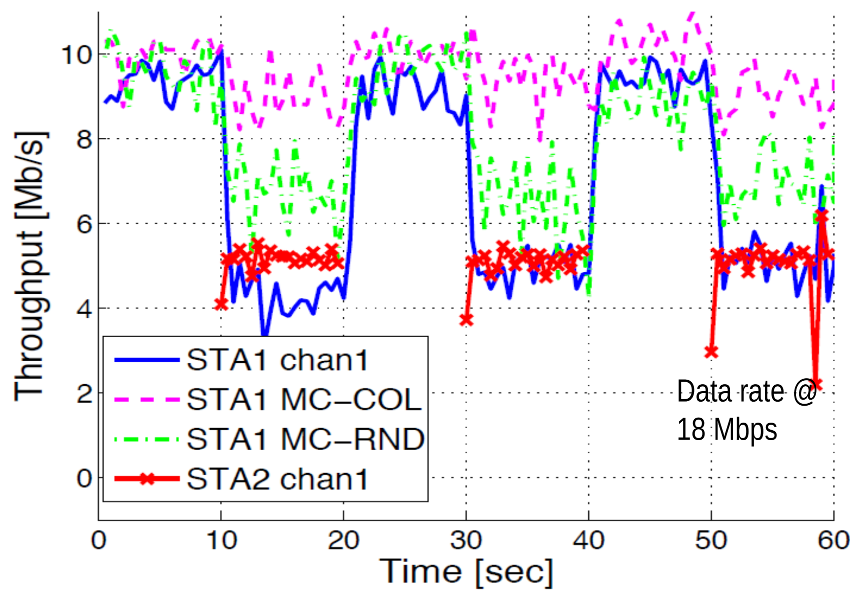
Per EACH frame, randomly select backoff AND channel (switch on as little as per frame basis)



Channel activity trace



Performance gain

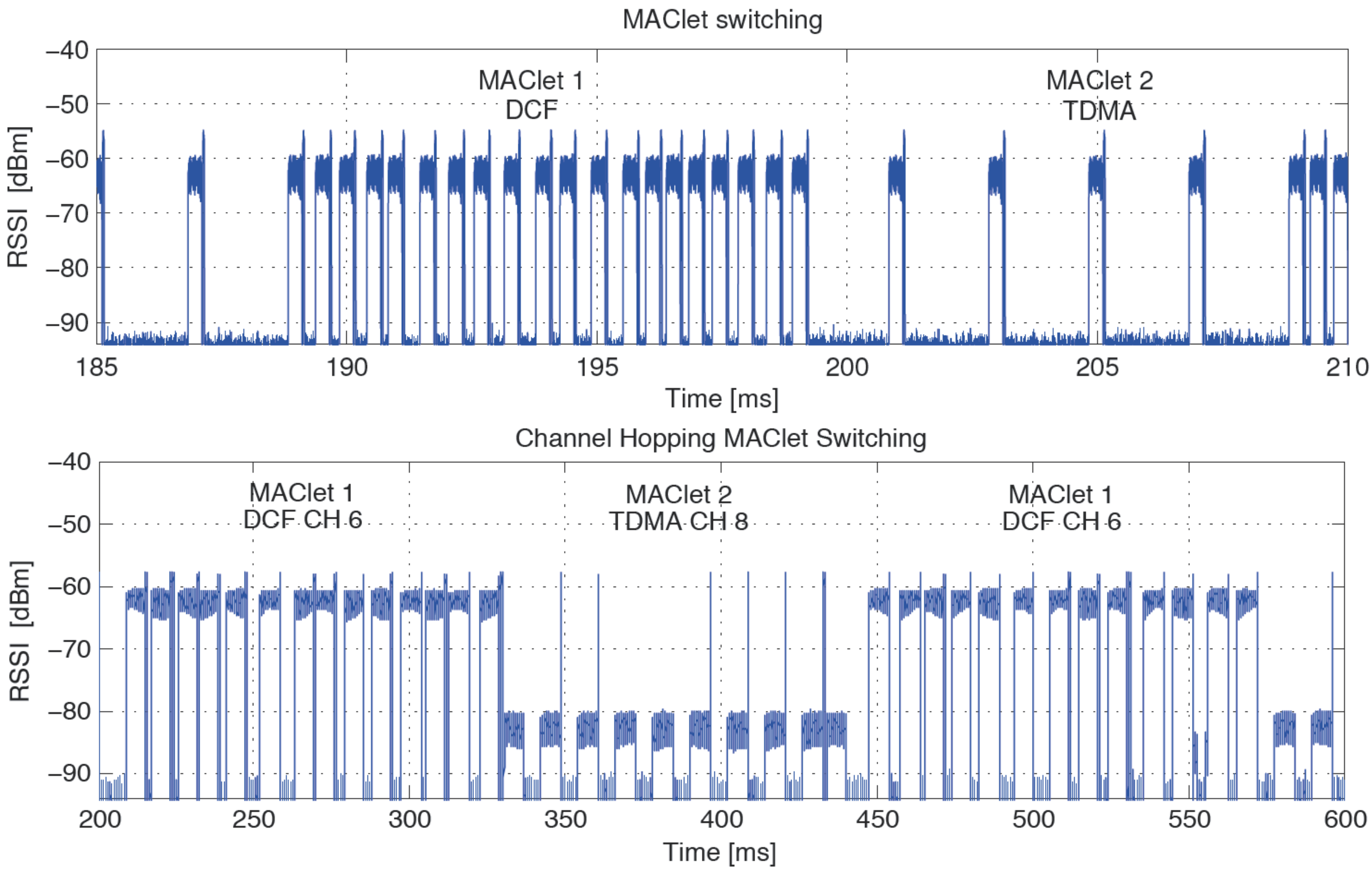


Multi-threaded MAC

Success IF seamless switch from one MAC program to another in negligible time

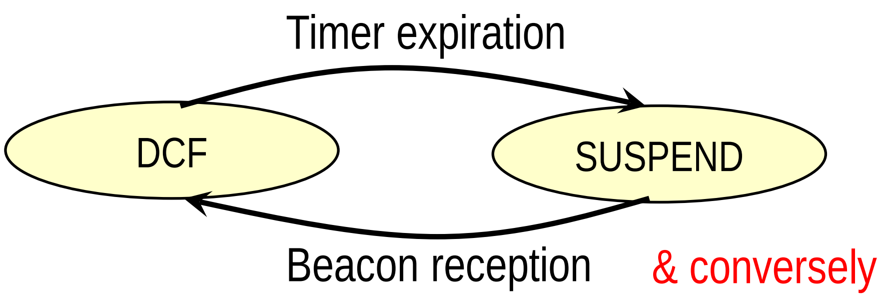
Result: less than 0.2 us over such cheap hardware!

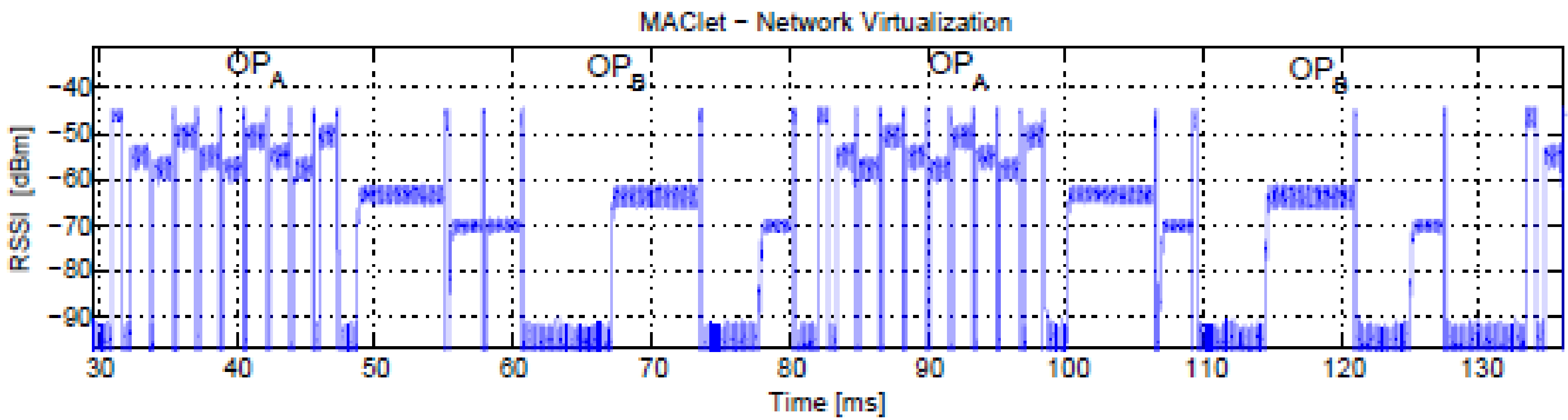
(plus channel switching time if required)



AP Virtualization with MAClets

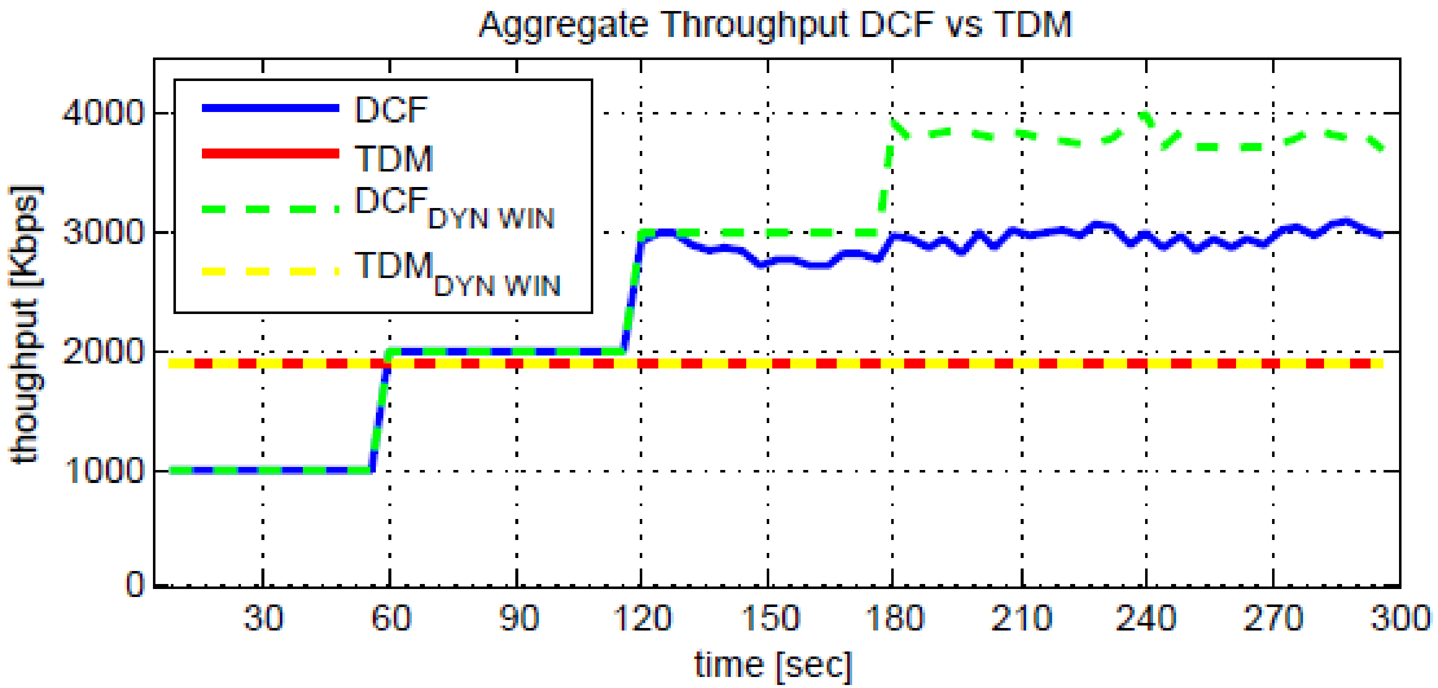
- ➔ **Two operators on same AP/infrastructure**
  - ⇨ A: wants TDM, fixed rate
  - ⇨ B: wants best effort DCF
- ➔ **Trivial with MAClets!**
  - ⇨ Customers of A/B download respective TDM/DCF MAClets!
- ➔ **Isolation via MAClet design**
  - ⇨ Time slicing DESIGNED INTO the MAClets! (static or dynamic)





An Example of Throughput Performance

3 FIXED stations @ 0.63 Mbps vs. 5 BEST stations @ 1Mbps



Conclusions

- ➔ **New vision:**
  - ⇨ MAC no more an all-size-fits-all protocol
  - ⇨ Can be made context-dependent
  - ⇨ Complex scenarios (e.g. virtualization) become trivial!
- ➔ **Very simple and viable model**
  - ⇨ Byte-coded XFSM injection
  - ⇨ Does NOT require open source NICs!

- ➔ **Next steps**
  - ⇨ We focused on the «act» phase; what about the decision and cognitive plane using such new weapons?
  - ⇨ can we think to networks which «self-program» themselves?
    - →Not too far, as it just suffices to generate and inject a state machine…