A glimpse into the Linux Wireless Core: From kernel to firmware

Francesco Gringoli
Nomadic Communication
University of Trento

Outline

- Linux Kernel Network Code
  - Modular architecture: follows layering
- Descent to (hell?) layer 2 and below
  - Why hacking layer 2
  - OpenFirmWare for WiFi networks
- OpenFWWF: RX & TX data paths
  - Hands on: examples
- OpenFWWF exploitations

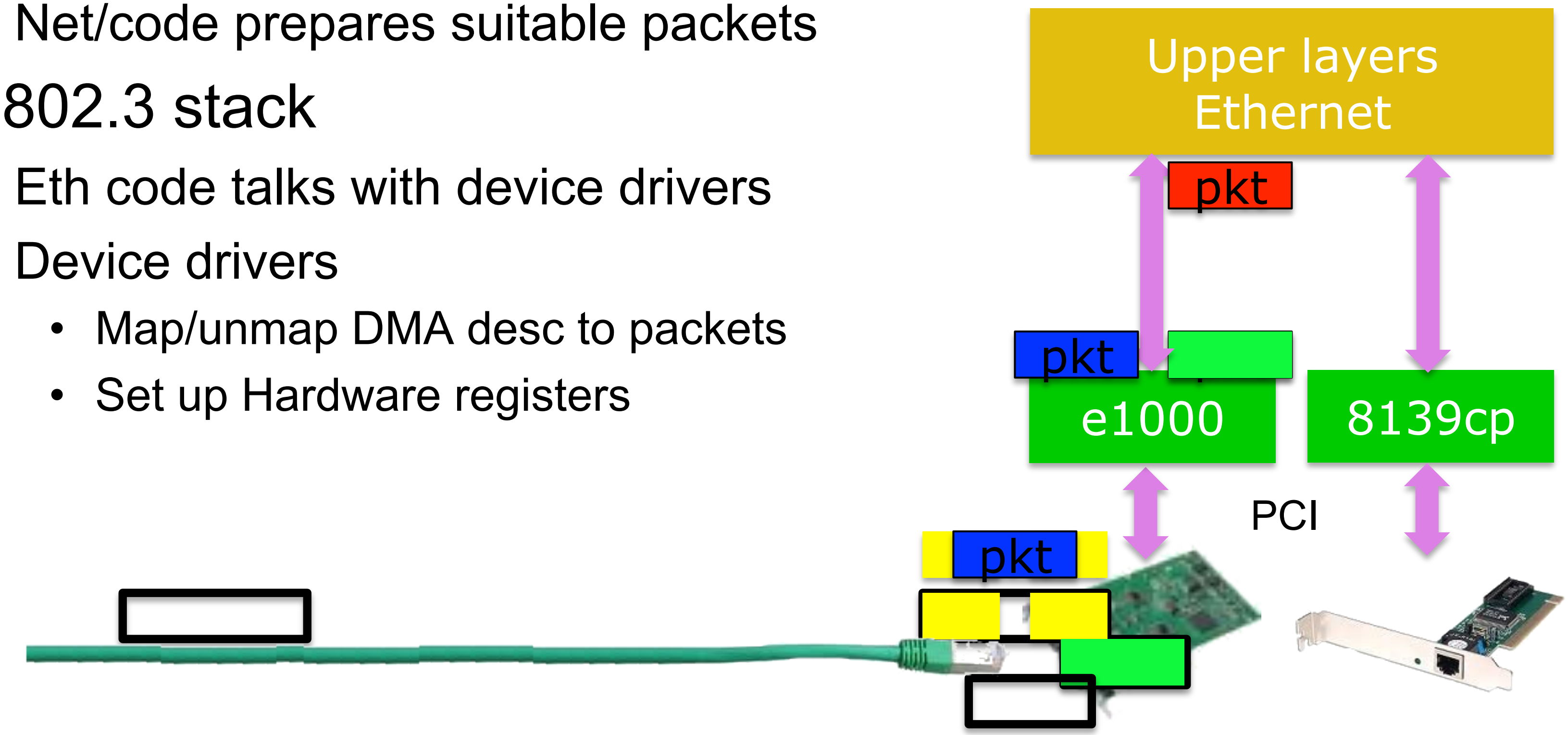
Linux Kernel Network Code

A glimpse into the
Linux Kernel Wireless Code

Part 1

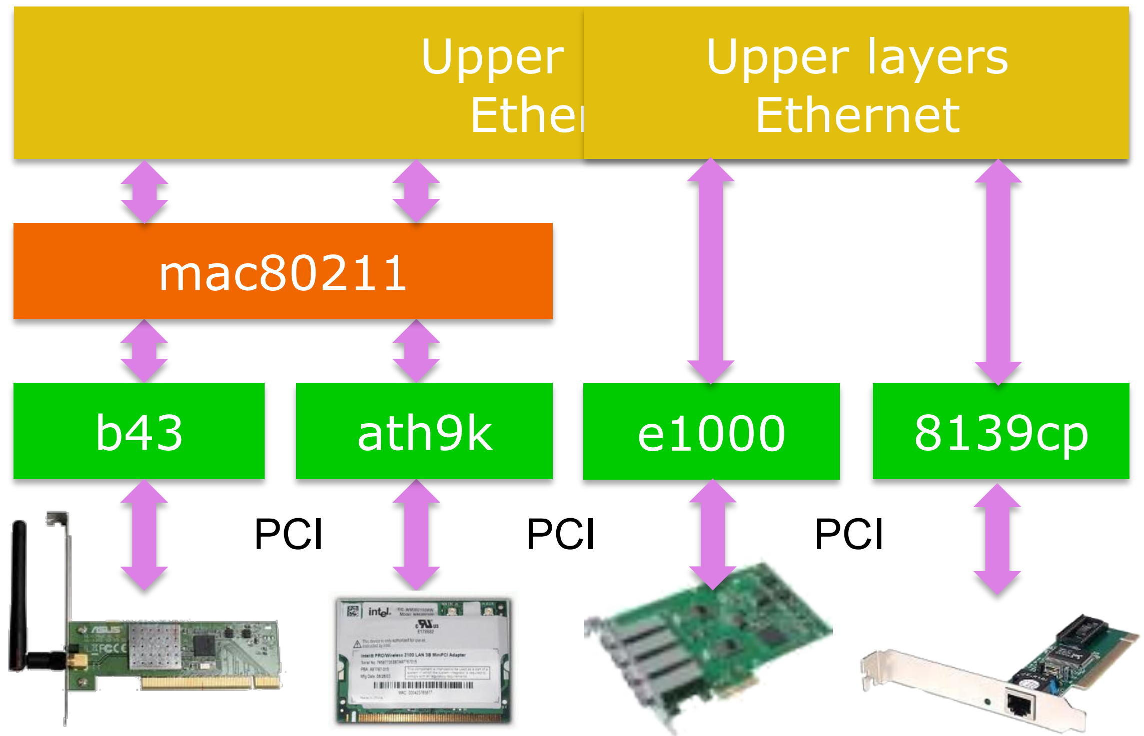
Linux Networking Stack
Modular architecture

- Layers down to MAC (included)
  - All operations above/including layer 2 done by kernel code
  - Network code device agnostic
  - Net/code prepares suitable packets
- In 802.3 stack
  - Eth code talks with device drivers
  - Device drivers
    - Map/unmap DMA desc to packets
    - Set up Hardware registers



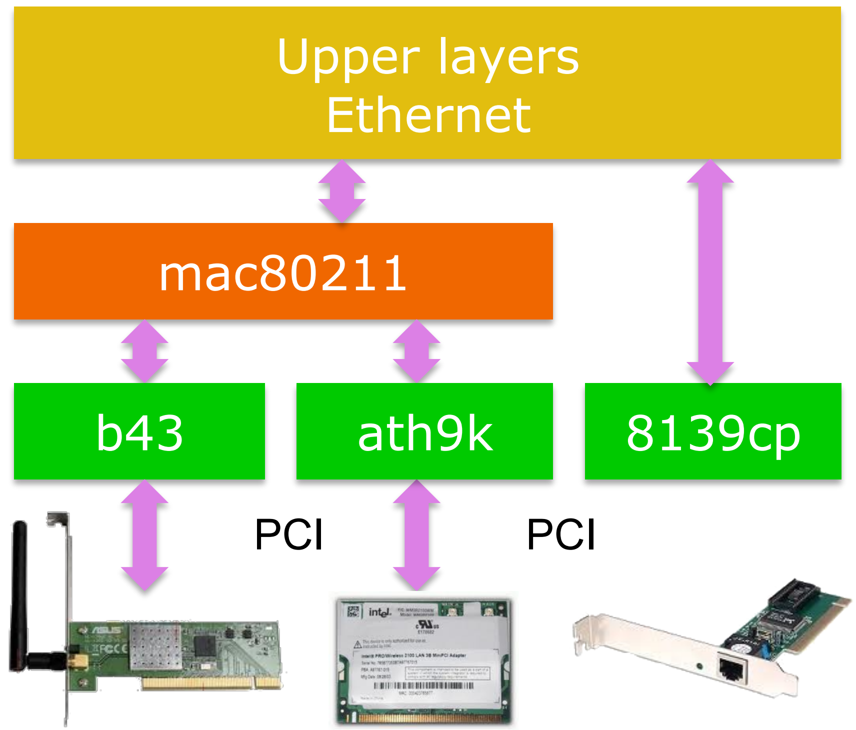
Linux Networking Stack Modular architecture

- What happens with 802.11?
  - New drivers to handle WiFi HW: how to link to net code?
  - A wrapper "mac80211" module is added



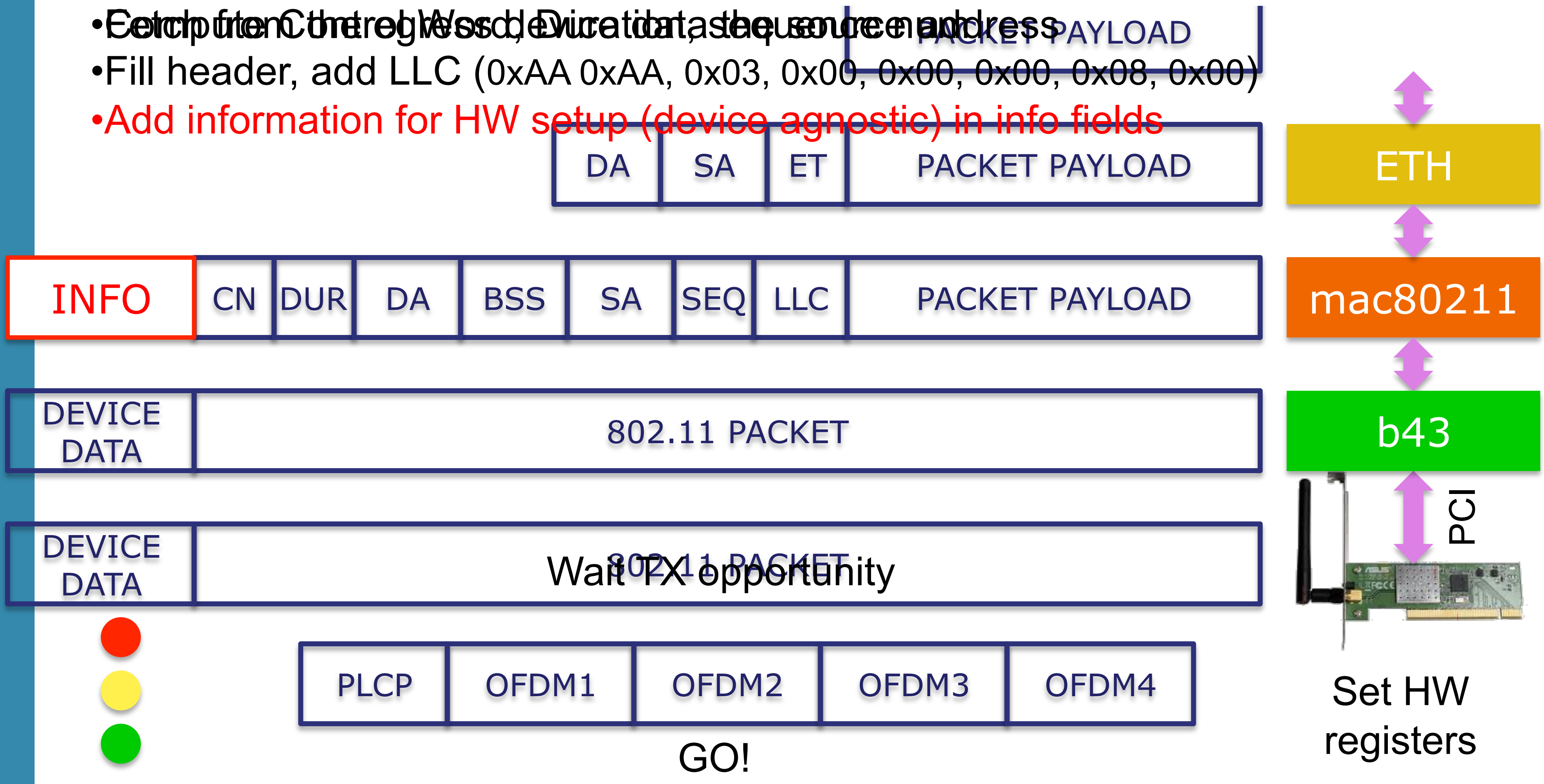
Linux & 802.11
Modular architecture

- Layers down to LLC (~mac) common with 802.3
  - All operations above/including layer 2 done by ETH/UP code
- Packets converted to 802.11 format for rx/tx
  - By wrapper "mac80211"
    - Manage packet conversion
    - Handle AAA operations
- Drivers: packets to devices
  - One dev type/one driver
    - Add data to "drive" the device



Linux & 802.11
Modular architecture/1

- Convert agnostic info into device dependent data
- Compute the egress device, the source address
- Fetch from Control Word, Duration, the sequence number
- Fill header, add LLC (0xAA 0xAA, 0x03, 0x00, 0x00, 0x00, 0x08, 0x00)
- Add information for HW setup (device agnostic) in info fields

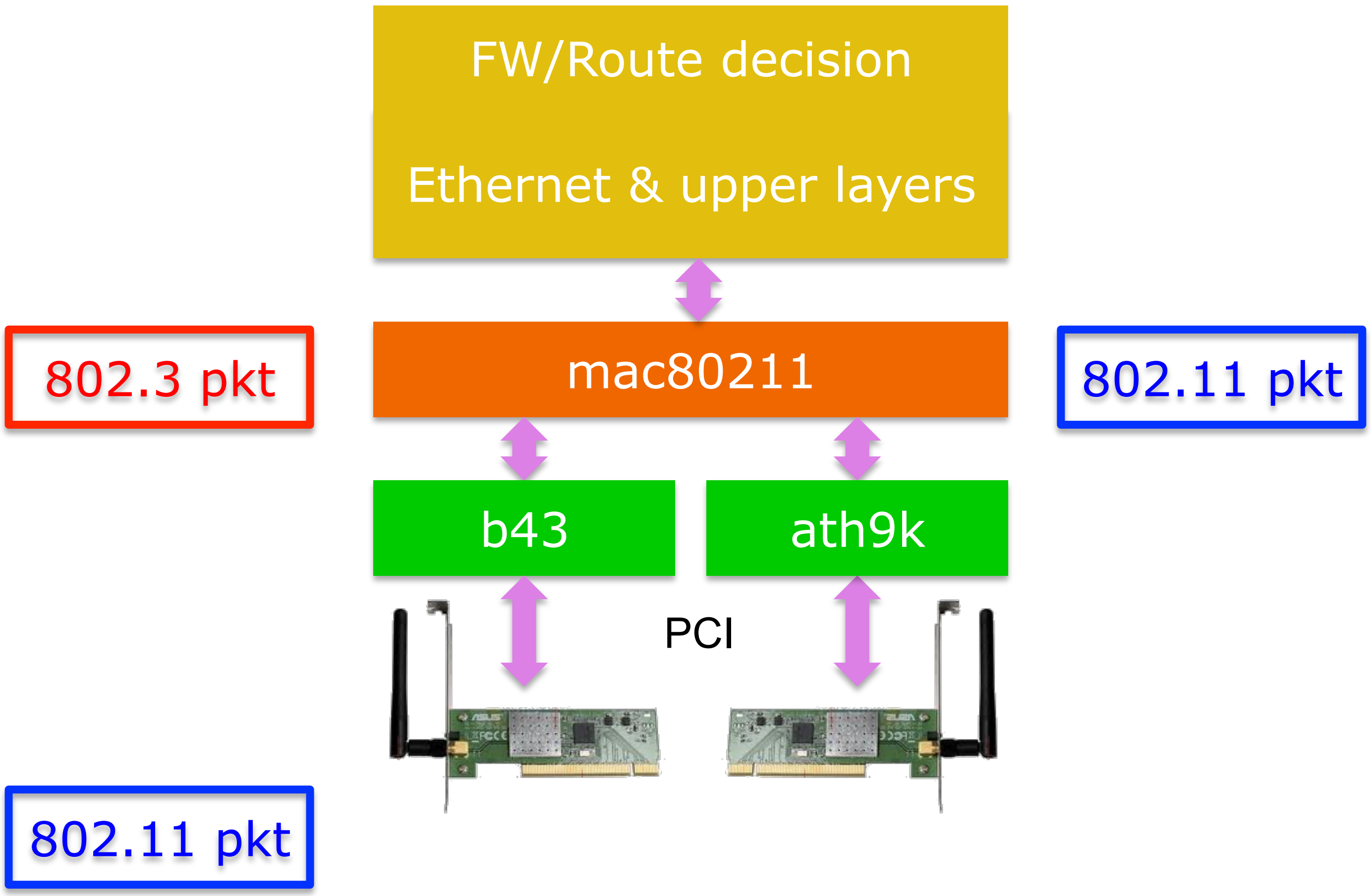


Linux & 802.11

- Opposite path: conversions reversed
- ☹ Several operations involved for each packet
- ☺ Multiple buffer copies (should be) avoided
  - E.g., original packet at layer 4 correctly allocated
    - Before L3 encapsulation output device already known
- ☹ Packets are queued twice/(3 times ☺)
  - Qdisc: before wrapper
  - Device queues: between wrapper and driver/(+DMA)
- Bottom line:
  - Clean design but can be resource exhausting

Linux & 802.11
Modular architecture

- Forwarding/routing packet on a double interface box



Linux & 802.11

- On CPU limited platform, fw performance too low
  – Need to accelerate/offload some operations
- Ralink was first to introduce SoC WiFi devices
  – A mini-pci card hosts an ARM CPU
  – Main host attaches a standard ethernet iface
  – The ARM CPU converts ETH packet to 802.11
  – Main host focuses on data forwarding
- Question: where can be profitably used?
  – Take a look to Andriod phones

Linux & 802.11: setup

- A simple BSS with Linux only nodes
  - One station runs hostapd (AP)
  - Others (STAs) join:
    - Once, with iw/iwconfig
    - Use a supplicant to join, e.g., use wpa_supplicant
  - Why using a supplicant?
    - management frame losses➔STA disconnection
    - Why? Kernel (STA) periodically checks if AP is alive
    - If management frames lost, kernel (STA) does not retransmit!
    - A supplicant (wpa_supplicant) is needed to re-join the BSS transparently

Linux & 802.11: kernel setup

- Check the device type with

```
$: lspci | grep -i net
```

- Load the driver for Broadcom devices and check is loaded

```
$: modprobe b43 qos=0
$: lsmod | grep b43
```

- Check kernel ring buffer with

```
$: dmesg | tail -30
```

- Bring net up and configure an IP address

```
$AP: ifconfig wlan0 172.16.0.1 up
$STA: ifconfig wlan0 172.16.0.10 up
```

- In following experiments we fix arp associations

```
$: ip neigh replace to PEERIP lladdr PEERMAC dev wlan0
```

  - Traffic not encrypted
  - QoS disabled

Linux & 802.11: hostapd setup

- Configuration of the AP in “hostapd.conf”

```
interface=wlan0
driver=nl80211
dump_file=/tmp/hostapd.dump
ctrl_interface=/tmp/hostapd
ssid=TESTTODAY
hw_mode=g
channel=14
beacon_int=100
auth_algs=3
wpa=0
```

Try to send SIGUSR1

PIPE used by

BSS properties

No encryption/ authentication

- Runs with

```
$: hostapd -B hostapd.conf    # -B: run in background
```

- Check dmesg!

Linux & 802.11: station setup

- Scan for networks

```
$: iwlist wlan0 scan
```

- Configuration of STAs in wpasupp.conf

```
ctrl_interface=/tmp/wpa_supplicant
network={
        ssid="TESTODAY"
        scan_ssid=1
        key_mgmt=NONE
}
```

PIPE used by wpa_cli

BSS to join

- Runs with

```
$: wpa_supplicant -B -i wlan0 -c wpasupp.conf
```

- Check dmesg!
- **Simple experiment: ping the AP**

```
$: ping 172.16.0.1
```

Linux & 802.11: run some traffic

- We use iperf in UDP mode
- On AP, server mode

```
$: iperf -s -u -p3000 -i1
```

- On STA, client mode

```
$: iperf -c172.16.0.1 -u -p3000 -i1 -t100 -b54M
```

- Channel 14 is usually free (by law)
  - Try another channel, e.g., 1 or 6 or 11
  - How to do it?
  - Reconfigure hostapd and reconnect, let's see how…

Linux & 802.11: check status

- There are some "debug" helpers, on AP:
  - Browse this folder

    `/sys/kernel/debug/ieee80211`
  - Learn what is `phy0`
  - Cd to `phy0/netdev:wlan0/stations`
  - Cd to the MAC address of the STA!!
    - Explore all the stats
    - Why rc_stats is almost empty?
- What on the STA?

Linux & 802.11: capturing packets

- On both AP and STA run “tcpdump”

```
$: tcpdump -i wlan0 -nn
```

- Is exactly what we expect?
  - What is missing?
  - Layer 2 acknowledgment?
- Display captured data

```
$: tcpdump -i wlan0 -nn -XXX
```

- What kind of layer 2 header?
- What have we captured?

Linux & 802.11: capturing packets

- Run “tcpdump” on another station set in monitor mode

```
$: ifconfig wlan0 down
$: iwconfig wlan0 mode monitor chan 4(?)
$: ifconfig wlan0 up
$: tcpdump -i wlan0 -nn
```

- What’s going on? What is that traffic?
  - Beacons (try to analyze the reported channel, what’s wrong?)
  - Probe requests/replies
  - Data frames
- Try to dump some packet’s payload
  - What kind of header?
  - Collect a trace with tcpdump and display with Wireshark

Linux & 802.11: capturing packets

- Exercise: try to capture only selected packets
- Play with matching expression in tcpdump

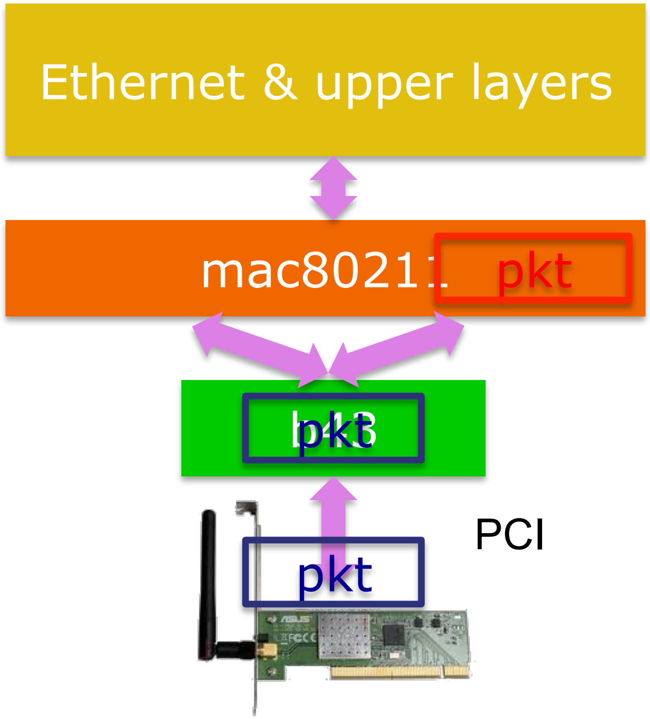
```
$: [cut] ether[N] ==|!= 0xAB
```

- Discard beacons and probes
- Display acknowledgments
- Display only AP and STA acknowledgments
- Question: is a third host needed?

Virtual Interfaces

- Wrapper/driver "may agree" on virtual packet path
  - Each received packet duplicated by the driver
  - mac80211 creates many interfaces "bound" to same HW
  - In this example
    - Monitor interface attached
    - Blue stream follow upper stack
    - Red stream hooked to pcap

```
$: iw dev wlan0 interface add \
          fish0 type monitor
```



  - Try capturing packets on the AP
    - What's missing?

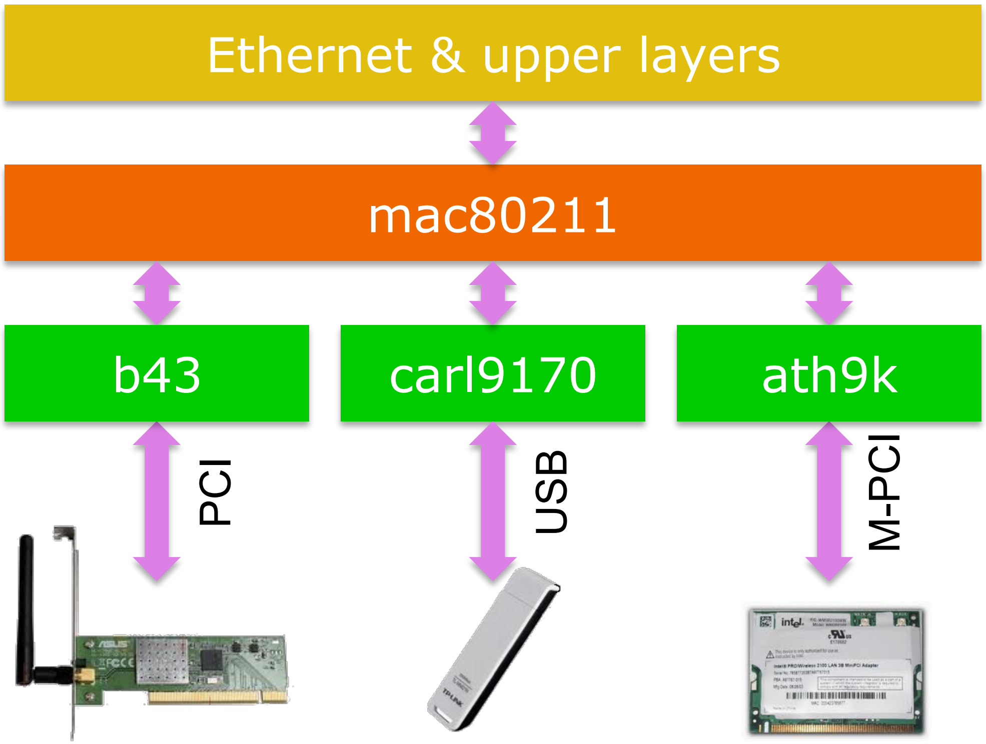
Descent to layer 2 and below
An open firmware

A glimpse into the
Linux Kernel Wireless Code

Part 2

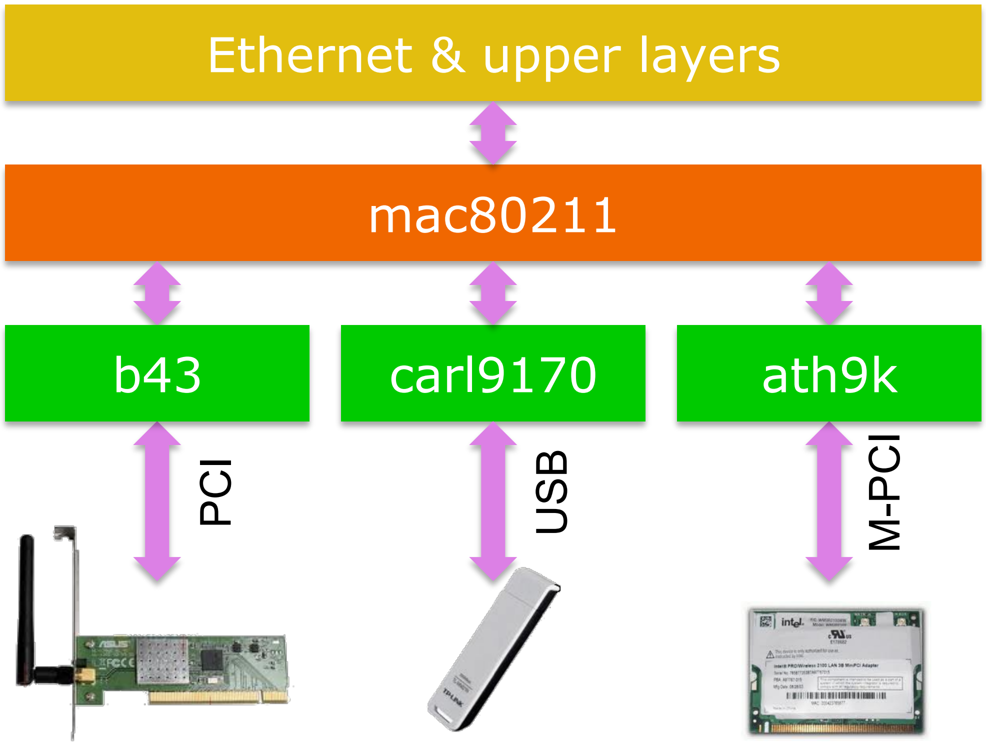
Linux & 802.11
Modular architecture

Wrapper for all hw
Find interface;
remove eth head;
add LLC&dot11 head;
fill (sa;da;ra;seq);
fill(control;duration);
set rate (from RC);
fill (rate;fallback);



Linux & 802.11
Modular architecture/2

Set up hw regs;
Fill hw private fields;
Send frame on DMA;
Get stats;
Reports to mac80211
Several MAC primitives missing!
Who takes care of ack?



Linux & 802.11
Modular architecture/3

Ethernet & upper layers

For sure

We will see the firmware in this course
but first…
Let’s check why we should do that ☺

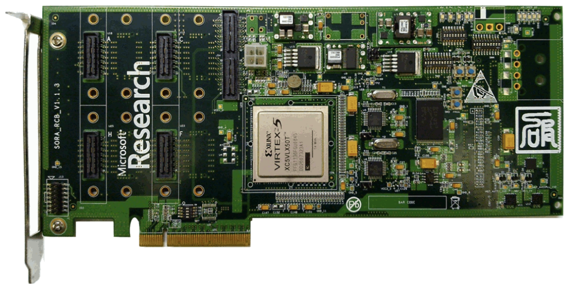
Firmware does

Why/how playing with 802.11

- Radio access protocols: issues
  - Some are unpredictable: noise & intf, competing stations
- Experimenting with simulators (e.g., ns-3)
  - Captures all “known” problems
    - Testing changes to back-off strategy is possible ☺
  - Unknown (not expected)?
    - Testing how noise affects packets not possible ☹
- **In the field testing is mandatory**
  - Problem: one station is not enough!

Programmable Boards

- Complete platforms like
  - WARP: Wireless open-Access Research Platform
  - Microsoft SORA
  - Based on FPGA
  - Everything can be changed
    - PHY (access to OFDM symbols!)
    - MAC
  - Two major drawbacks
    - More than very expensive
    - Complex deployment
  - **If PHY untouched: look for other solutions!**

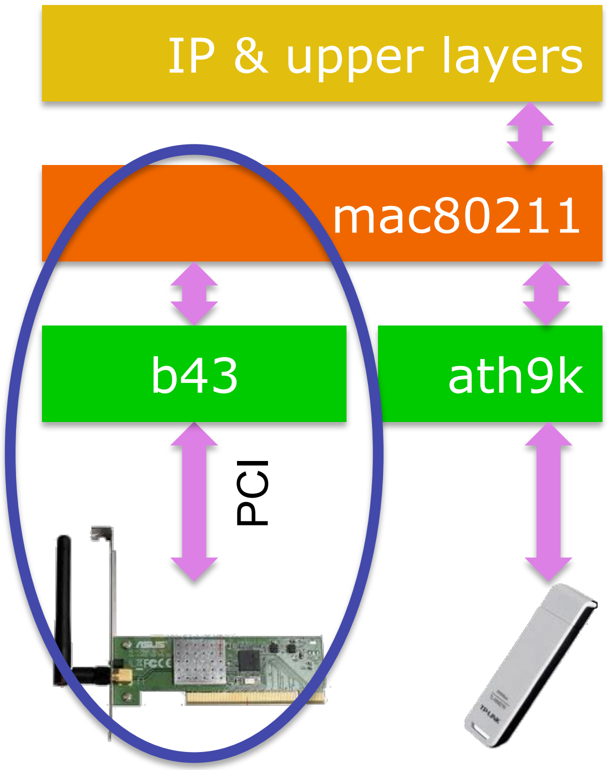


Off-the-shelf hardware

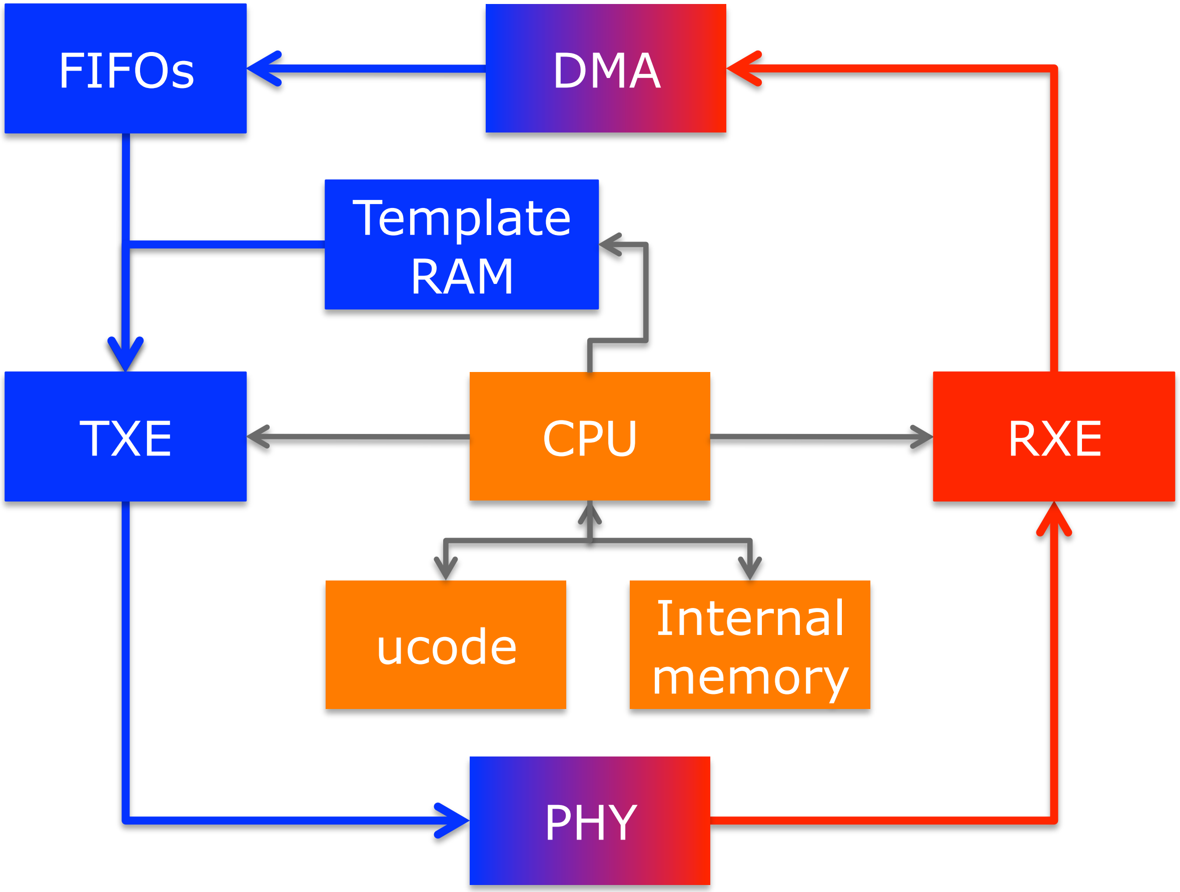
- Five/Six vendors develop cheap WiFi hw
  - Hundreds different boards
  - Almost all boards load a binary firmware
    - MAC primitives driven by a programmable CPU
  - Changing the firmware ➔ Changing the MAC!
- Target platform:
  - Linux & 802.11: modular architecture
  - Official support prefers closed-source drivers ☹
  - Open source drivers && Good documentation
    - Thanks to community! ☺

Linux & 802.11 Broadcom AirForce54g

- Architecture chosen because
  - Existing asm/dasm tools
    - A new firmware can be written!
  - Some info about hw regs
- We analyzed hw behavior
  - Internal state machine decoded
  - Got more details about hw regs
  - Found timers, tx&rx commands
  - Open source firmware for DCF possible
- We released OpenFWWF!
  - OpenFirmWare for WiFi networks



Broadcom AirForce54g Basic HW blocks



Description of the HW

- CPU/MAC processor capabilities
  - 88MHz CPU, 64 general purpose registers
- Data memory is 4KB, direct and indirect access
  - From here on it's called Shared Memory (SHM)
- Separate template memory (arrangeable > 2KB)
  - Where packets can be composed, e.g., ACKs & beacons
- Separate code memory is 32KB (4096 lines of code)
- Access to HW registers, e.g.:
  - Channel frequency and tx power
  - Access to channel transmission within N slots, etc…

TX side

- Interface from host/kernel
  - Six independent TX FIFOs
  - DMA transfers @ 32 or 64 bits
  - HOL packet from each FIFO
    - can be copied in data memory
      - Analysis of packet data before transmission
      - Kernel appends a header at head with rate, power etc
    - can be transmitted "as is"
    - can be modified and txed
      - Direct access to first 64 bytes

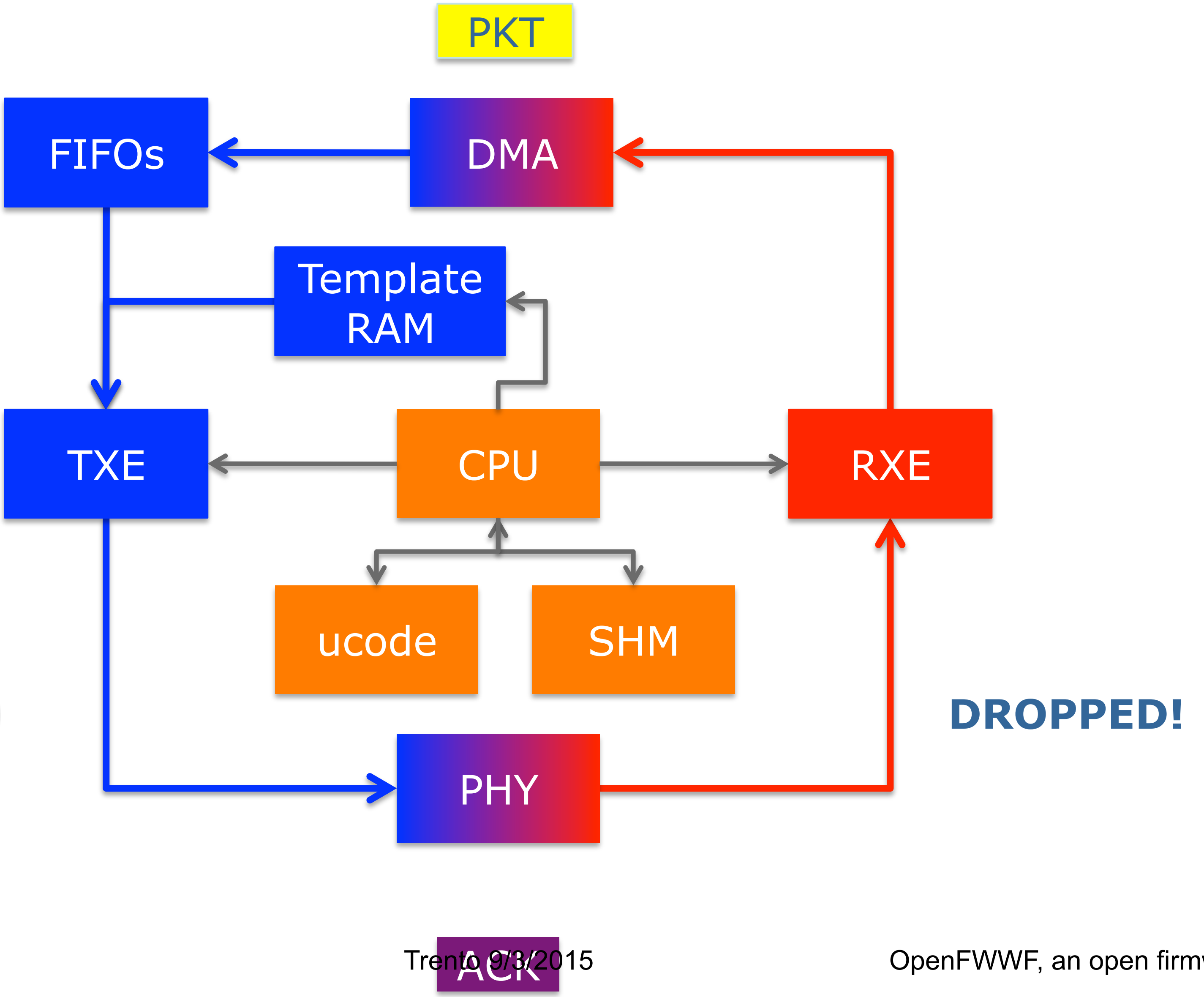
TX side/2

- Interface to air
  - Only 802.11 b/g supported, soon n
  - Full MTU packets can be transmitted (~2300bytes)
    - If full packet analysis is needed, analyze block-by-block
  - All 802.11 timings supported
    - Minimum distance between Txed frames is 0us
      - Note: channel can be completely captured!!
  - Backoff implemented in software (fw)
    - Simply count slots and ask the HW to transmit

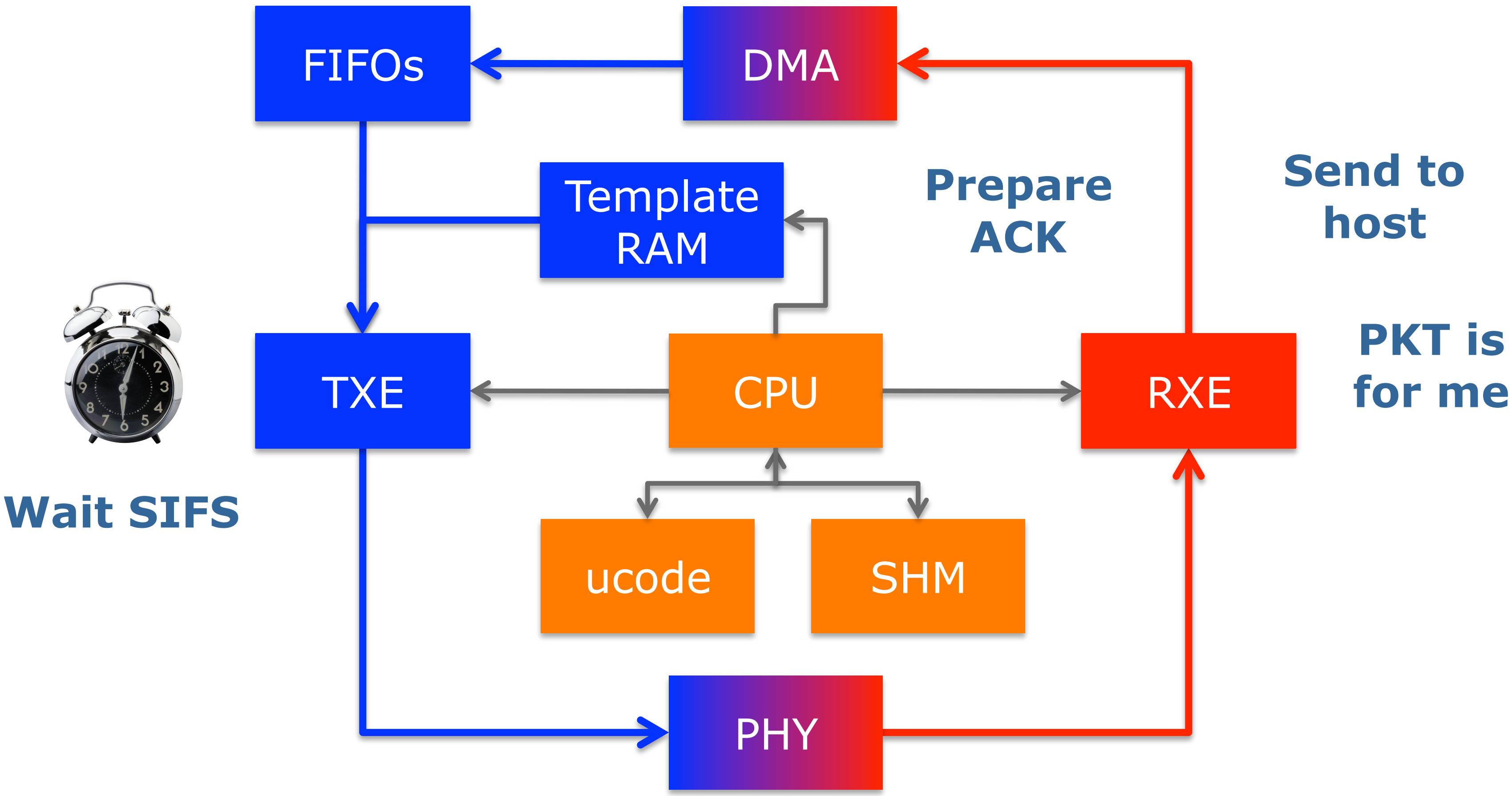
RX side

- Interface from AIR
  - HW acceleration for
    - PLCP and global packet FCS - Destination address matching
  - Packet can be copied to internal memory for analysis
    - Bytes buffered as soon as symbols is decoded
  - During reception and copying CPU is idle!
    - Can be used to offload other operations
    - Try to suggest something
  - Packets are pushed to host/kernel
    - If FW decides to go and through one FIFO ONLY
    - May drop! (e.g., corrupt packets, control…)

Example:
TX a packet, wait for the ACK



Example:
RX a packet, transmit an ACK



What lesson we learned

- From the previous slides
  - Time to wait ack (success/no success)
  - Dropping ack (rcvd data not dropped, goes up)
  - And much more
    - When to send beacon
    - Backoff exponential procedure and rate choice
  - Decided by MAC processor (by the firmware)
- Bottom line:

Hardware is (almost) general purpose

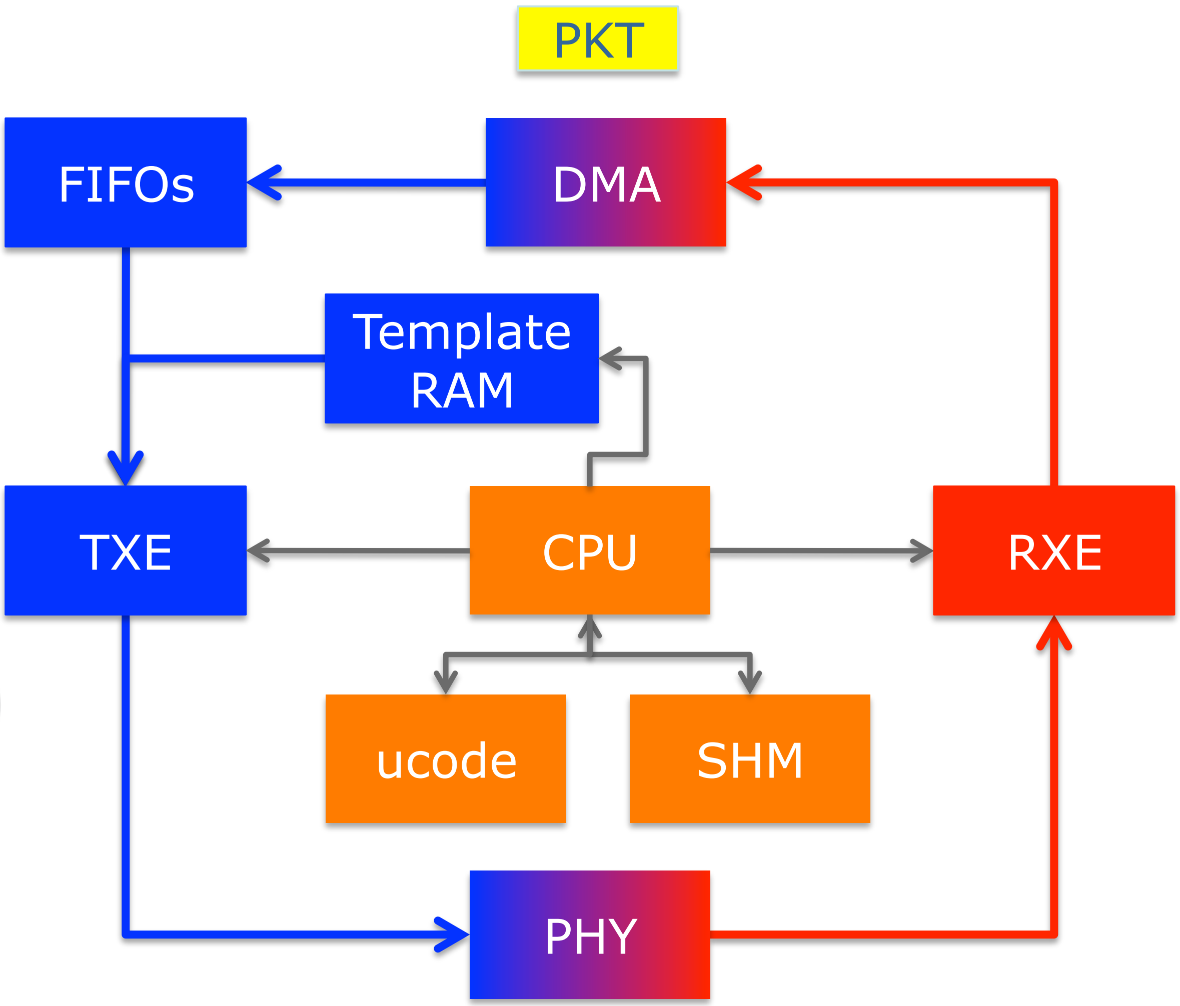
From lesson to OpenFWWF Description of the FW

- OpenFWWF
  - It's not a production firmware
  - It supports basic DCF
    - No RTS/CTS yet, No QoS, only one queue from Kernel
  - Full support for capturing broken frames
  - It takes 9KB for code, it uses < 200byte for data
    - **We have lot of space to add several features**
- Works with 4306, 4311, 4318 hw
  - Linksys Routers supported (e.g., WRT54GL)

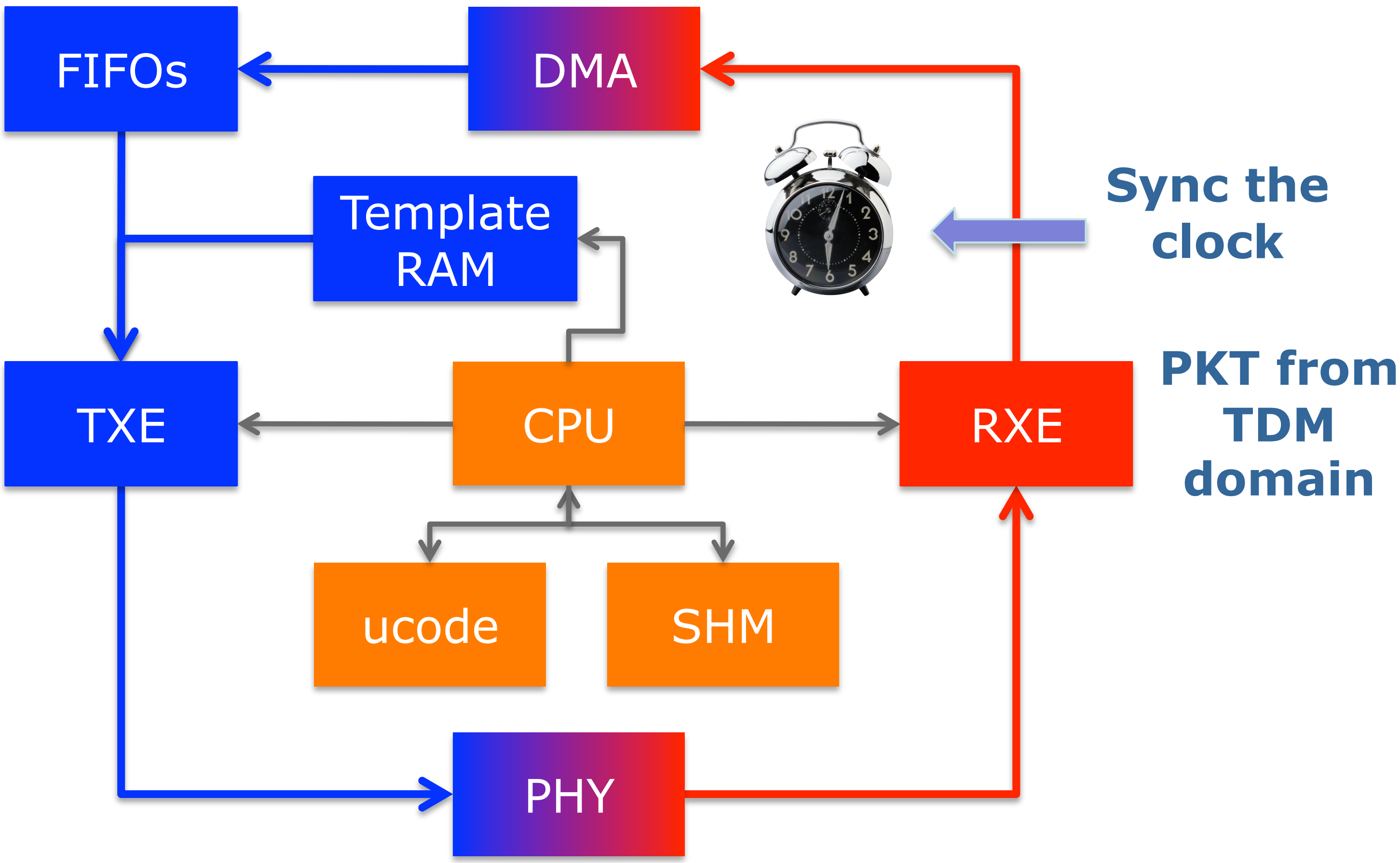
Broadcom AirForce54g Simple TDM

TDM needed! Waiting turn

GO!



Broadcom AirForce54g
Simple TDM/2



OpenFWWF
RX & TX data paths

A glimpse into the
Linux Kernel Wireless Code

Part 3

Firmware in brief

- Firmware seems really complex to understand ☹
  - Assembly language
    - CPU registers: 64 registers [r0, r1, …, r63]
    - SHM memory: 4KB of 16bits words addressable as [0x000] -> [0x7FF]
    - HW registers: spr000, spr001, …, spr1FF
  - Use `#define` macro to ease understanding
    - `#define CUR_CONTENTION_WIN r8`
    - `#define SPR_RXE_FRAMELEN spr00c`
    - `#define SHM_RXHDR SHM(0xA88)`
      - `SHM(.)` is a macro as well that divides by 2
  - Assignments:
    - Immediate `mov 0xABBA, r0; // load 0xABBA in r0`
    - Memory direct `mov [0x0013], r0; // load 16bit @ 0x0026 (LE!)`

Firmware in brief/2

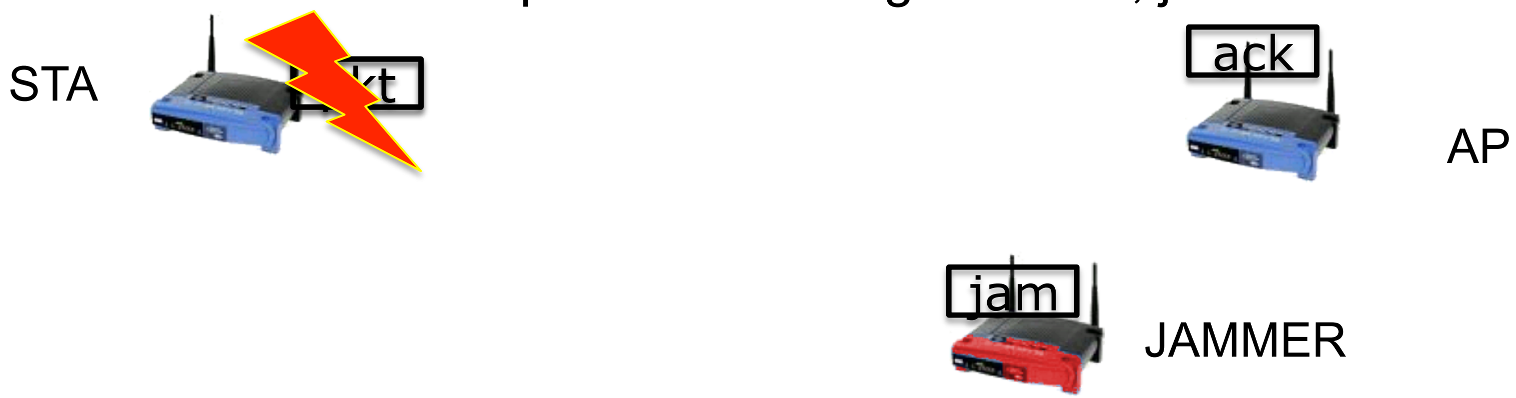
- Value manipulation:
  - Arithmetic:
    - Sum: `add r1, r2, r3;` // r3 = r1 + r2
    - Subtraction: `sub r2, r1, r3;` // r3 = r2 - r1
  - Logical:
    - Xor: `xor r1, r2, r3;` // r3 = r1 ^ r2
  - Shift:
    - Shift left: `sl r1, 0x3, r3;` // r3 = r1 << 3
- Pay attention:
  - In 3 operands instruction, immediate value in range [0..0x7FF]
  - Value is sign extended to 16bits

Firmware in brief/3

- Code flow execution controlled by using jumps
  - Simple jumps, comparisons
    - Jump if equal: `je r2, r5, loop;` // jump if r2 == r5
    - Jump if less: `jl r2, r5, exit;` // jump if r2 < r5 (unsigned)
  - Condition register jumps: jump on selected CR (condition registers)
    - on plcp end: `jext COND_RX_PLCP, rx_plcp;`
    - on rx end: `jext COND_RX_COMPLETE, rx_complete;`
    - on good frame: `jext COND_RX_FCS_GOOD, frame_ok;`
    - unconditionally: `jext COND_TRUE, loop;`
  - A check can also clean a condition, e.g.,
    - `jext EOI(COND_RX_PLCP), rx_plcp;` // clean CR bit before jump
  - Call a code subsection, save return value in link-registers (lr):
    - `call lr0, push_frame;` // return with `ret lr0, lr0;`

Firmware in brief/4

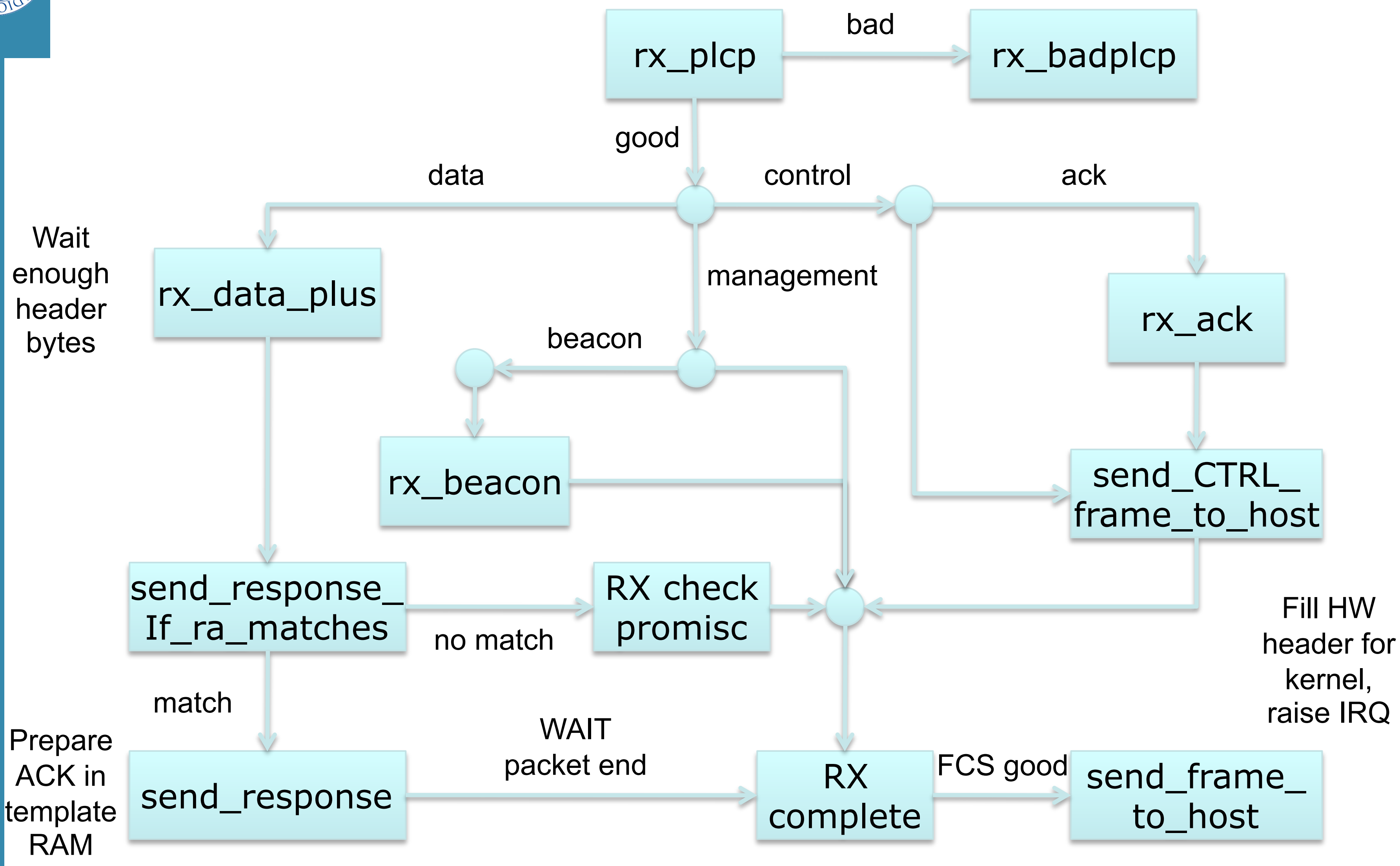
- OpenFWWF is today ~ 1000 lines of code
  - Not possible to analyze in a single lesson
  - We will analyze only some parts
- A simple exercise:
  - Analyze quickly the receiver section
  - Propose changes to implement a jammer
    - When receives packets from a given STA, jams noise!



RX code made easy

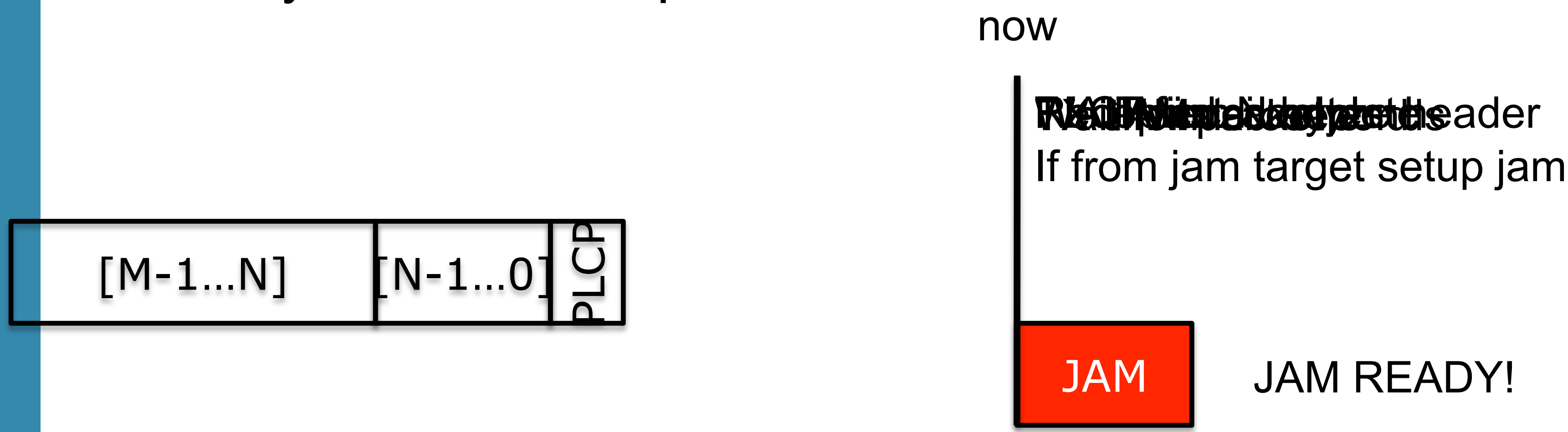
- During reception
  - CR RX_PLCP set when PLCP is completely received
  - CR COND_RX_BADPLCP set if PLCP CRC went bad
  - SPR_RXE_FRAMELEN hold the number of already received bytes
  - First 64B of packet are copied starting at `SHM_RXHEADER = SHM(0xA08)`
    - First 6B hold the PLCP
  - CR COND_RX_COMPLETE set when packet is ready
- We can have a look at the code flow for a data packet
  - rx_plcp: checks it's a data packet
  - rx_data_plus: checks packet is longer than 0x1C = 6(PLCP)B + 22(MAC)B
  - send_response: copy src mac address to ACK addr1, set state to TX_ACK
  - rx_complete: schedule ACK transmission

RX code path


rx_plcp
bad
rx_badplcp
good
data
control
ack
Wait enough header bytes
rx_data_plus
management
rx_ack
beacon
rx_beacon
send_CTRL_frame_to_host
send_response_If_ra_matches
RX check promisc
no match
Fill HW header for kernel, raise IRQ
match
WAIT packet end
Prepare ACK in template RAM
send_response
RX complete
FCS good
send_frame_to_host

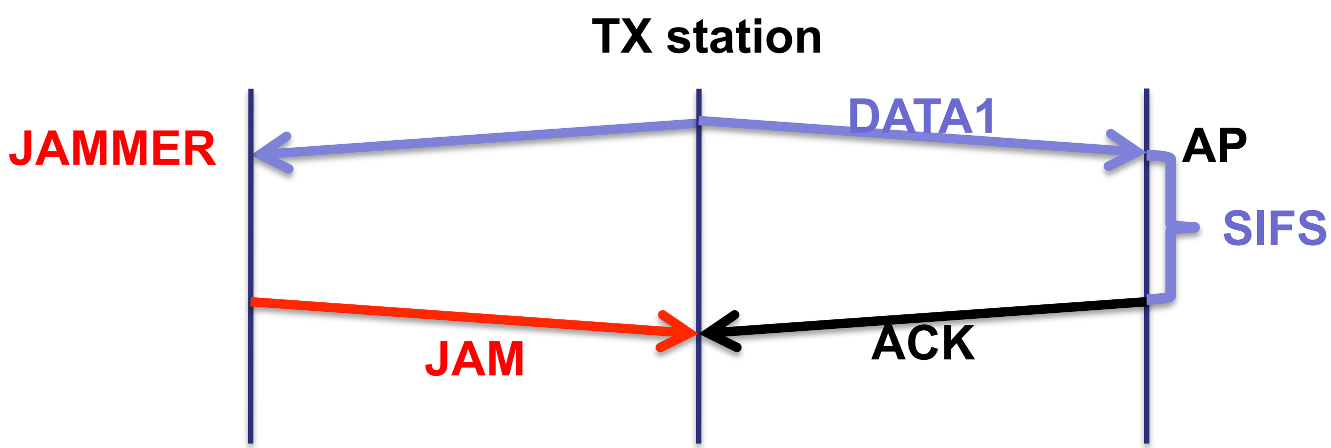
Let’s hack and do jamming

- During reception CPU keeps on running
  - Detect end of PLCP
  - May wait for a given number of bytes received
  - May prepare a response frame (ACK)
  - Wait for end of reception
  - May schedule response frame transmission after a while



Let’s hack and do jamming/2

- Disturbing a station when sending data
  - Jammer recognizes tx’ed data and sends fake ACK
- Maybe (for testing) jamming all packets is too much
  - Selected packets?



Let’s hack and do jamming/3

- If first byte of a packet are copied to SHM
- If we have ways of displaying SHM
  - Could we find evidence of received packets?
- Useful tool
  - `$: readshm`
  - Display shared memory
- Run this experiment: run traffic from the STA to AP
  - On AP dump the SHM: locate the UDP packet
  - Fix the rate on STA: how do the first 6 bytes change?

Let’s hack and do jamming/4

- Shared memory appears like this

```
0x0A00:  0000 0000 0000 0000 CCBF 0200 0000 0801   ................
0x0A10:  0400 0014 A442 958D 0014 A442 958D 0013   .....B.....B....
0x0A20:  D4BB 2CBF C006 AAAA 0300 0000 0800 4500   ..,...........E.
0x0A30:  05DA 3E7E 4000 4011 751B C0A8 0028 C0A8   ..>~@.@.u....(..
0x0A40:  0001 CB86 0BB8 05C6 0F6E 0000 459E 531C   .........n..E.S.
0x0A50:  ADA9 0000 84FD 0000 0000 0000 0001 0000   ................
0x0A60:  0BB8 0000 0000 0337 F980 FFFE 7960 3637   .......7....y`67
0x0A70:  3839 3031 3233 3435 3637 3839 3031 3233   8901234567890123
0x0A80:  3435 3637 3839 3031 5100 0000 0600 2A50   45678901Q.....*P
0x0A90:  E54F 0000 0000 0000 B4FB A202 0000 0000   .O..............
```

Let's hack and do jamming/4

- Shared memory appears like this

```
0x0A00:  0000 0000 0000 0000 CCBF 0200 0000 0801   ................
0x0A10:  0400 0014 A442 958D 0014 A442 958D 0013   .....B.....B....
0x0A20:  D4BB 2CBF C006 AAAA 0300 0000 0800 4500   ..,...........E.
0x0A30:  05DA 3E7E 4000 4011 751B C0A8 0028 C0A8   ..>~@.@.u....(..
0x0A40:  0001 CB86 0BB8 05C6 0F6E 0000 459E 531C   .........n..E.S.
0x0A50:  ADA9 0000 84FD 0000 0000 0000 0001 0000   ................
0x0A60:  0BB8 0000 0000 0337 F980 FFFE 7960 3637   .......7....y`67
0x0A70:  3839 3031 3233 3435 3637 3839 3031 3233   8901234567890123
0x0A80:  3435 3637 3839 3031 5100 0000 0600 2A50   45678901Q.....*P
0x0A90:  E54F 0000 0000 0000 B4FB A202 0000 0000   .O..............
```

- What should we check if we want to jam only UDP frame to port 3000?
- We have also to wait for at least …. Bytes have been received, right?

Let’s hack and do jamming/5

- Legacy rx_data_plus:

```
rx_data_plus:
        jext    COND_RX_COMPLETE, end_rx_data_plus
        jl      SPR_RXE_FRAMELEN, 0x01C,rx_data_plus
end_rx_data_plus:
        jl      SPR_RXE_FRAMELEN, 0x01C, rx_check_promisc
        jnext   COND_RX_RAMATCH, rx_ra_dont_match
        jext    COND_TRUE, send_response
```

- What we change?
  - Change the frame length
  - Add filter
  - If frame match filter, then “send_response” and remember somewhere!

Let's hack and do jamming/6

- Legacy rx_complete

```
rx_complete:
   [cut]
frame_successfully_received:
        jext    COND_RX_FIFOFULL, rx_fifo_overflow
        jnext   COND_NEED_RESPONSEFR, check_frame_subtype
need_regular_ack:
        je      [SHM_CURMOD], 0x001, ofdm_modulation
```

- What we change?
  - If we had remembered somewhere this is to jam
    - JAM IT!, schedule the frame anyway

JAM code

- To switch to a different firmware
  - Look at /lib/firmware
  - Link the desired firmware release as “b43”
  - Remove b43 module, reload and bring back the network up

```
$: rmmod b43 . . .
```

- How to test JAM code? “iperf” performance tool
- On AP run in server mode (receiver)

```
$: iperf -s -u -p 3000 -i 1
```

- On STA run in client mode (transmit)

```
$: iperf -c IP_OF_AP -u -p 3000 -i 1 -t 10
```

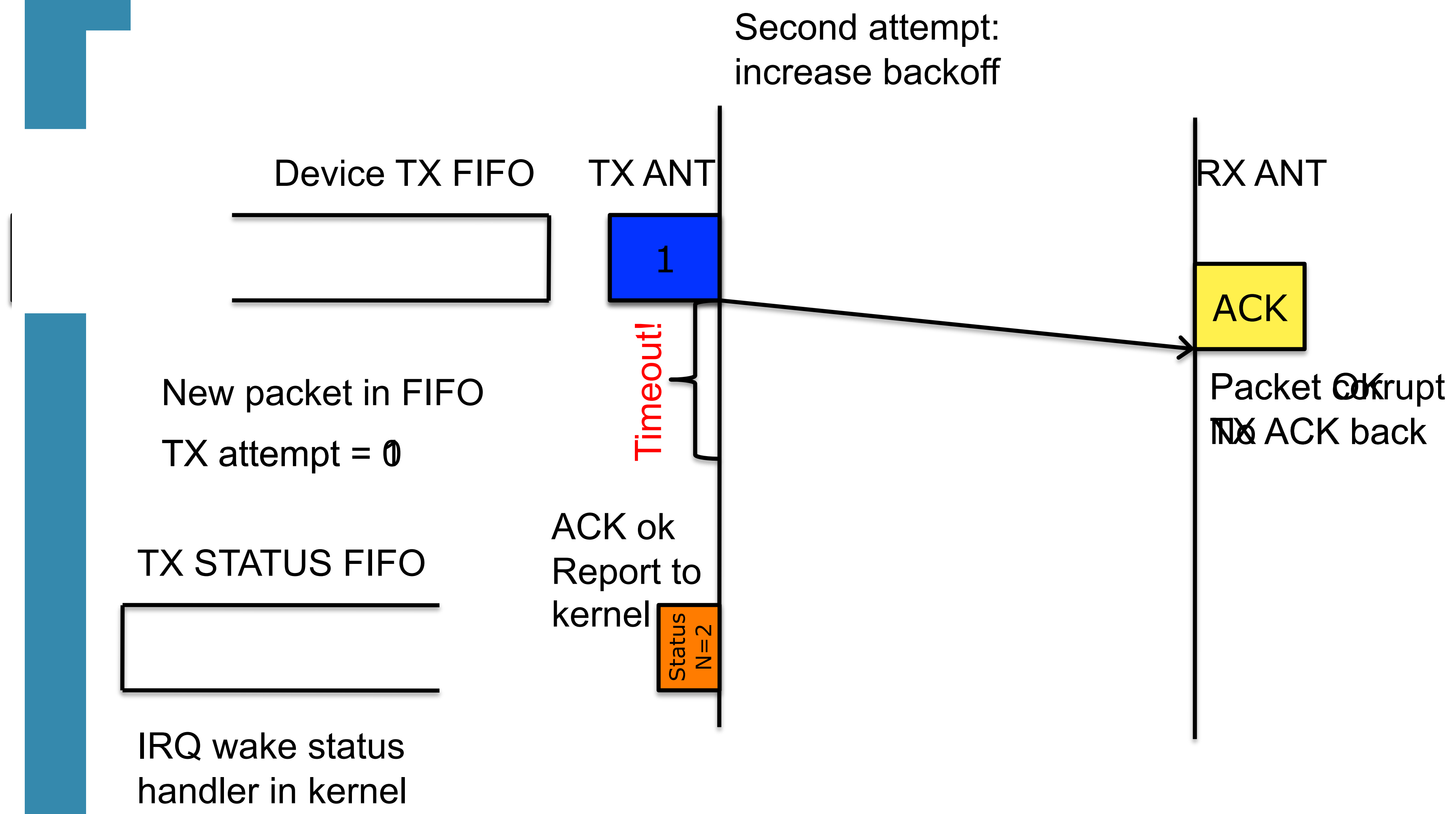
TX made easy

- Packets are prepared by the kernel
  - Fill all packet bytes (e.g., 802.11 header)
  - Choose hw agnostic device properties
    - Tx power to avoid energy wasting
    - Packet rate: rate control algorithm (minstrel)
  - A driver translates everything into hw specific
    - b43: rate encoded in PLCP (first 6B)
    - b43: append a fw-header at packet head
      - Firmware will setup hw according to these values

TX made easy/2

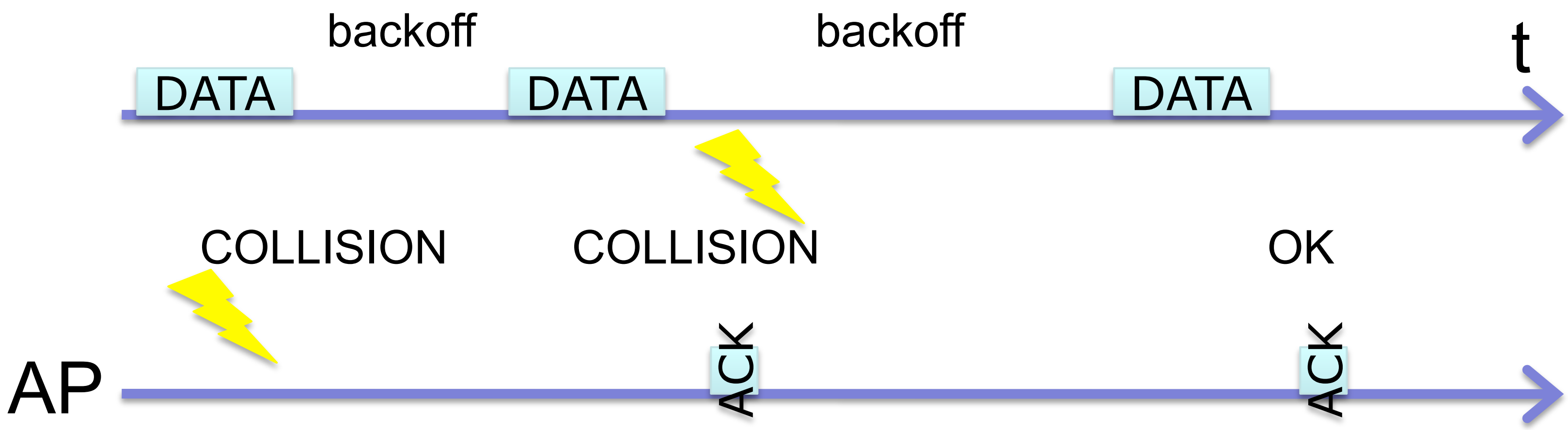
- Kernel (follows)
  - b43: send packet data (+hw info) through DMA
- firmware:
  - Continuous loop, when no receiving
    - If IDLE, check if packet in FIFO (comes from DMA)
    - If packet does not need ACK, TX,report and exit
    - If packet needs ACK, wait ACK timeout
    - If ACK timeout expired:
      - if ACK RXed, report to kernel, exit
      - If ACK not RXed, setup backoff, try again
      - If too much TX attempts
        - » remove packet from FIFO, report to kernel, exit

TX made easy/3



TX made easy/4

- Summary



- FW reports to kernel the number of attemps
  - Kernel feeds the rate control algo
  - A rate for the next packet is chosen

TX made easy/5

- Currently “minstrel” is the default RC algo
  – At random intervals tries all rates
  – Builds a tables with success “rate” for each “rate”
  – In the short term it selects the best rate
  – How to checks this table from userspace?
    - DEBUGFS ☺
    - Take a look at folder

      `/sys/kernel/debug/ieee80211/phyN/`

TX made easy: exercise

- Firmware: backoff entered if ack is not rx
  - Simple experiment
    - Two STAs joined to the same BSS
    - iperf on both STAs to the AP
    - They should share the channel
  - What happen if we hack one station fw?
  - Let's try…
    - TX path really complex, skip
    - But at source top we have a few "_CW" values

OpenFWWF Exploitations

A glimpse into the
Linux Kernel Wireless Code

Part 4

OpenFWWF Exploitation: Partial Packet Recovery

In collaboration with

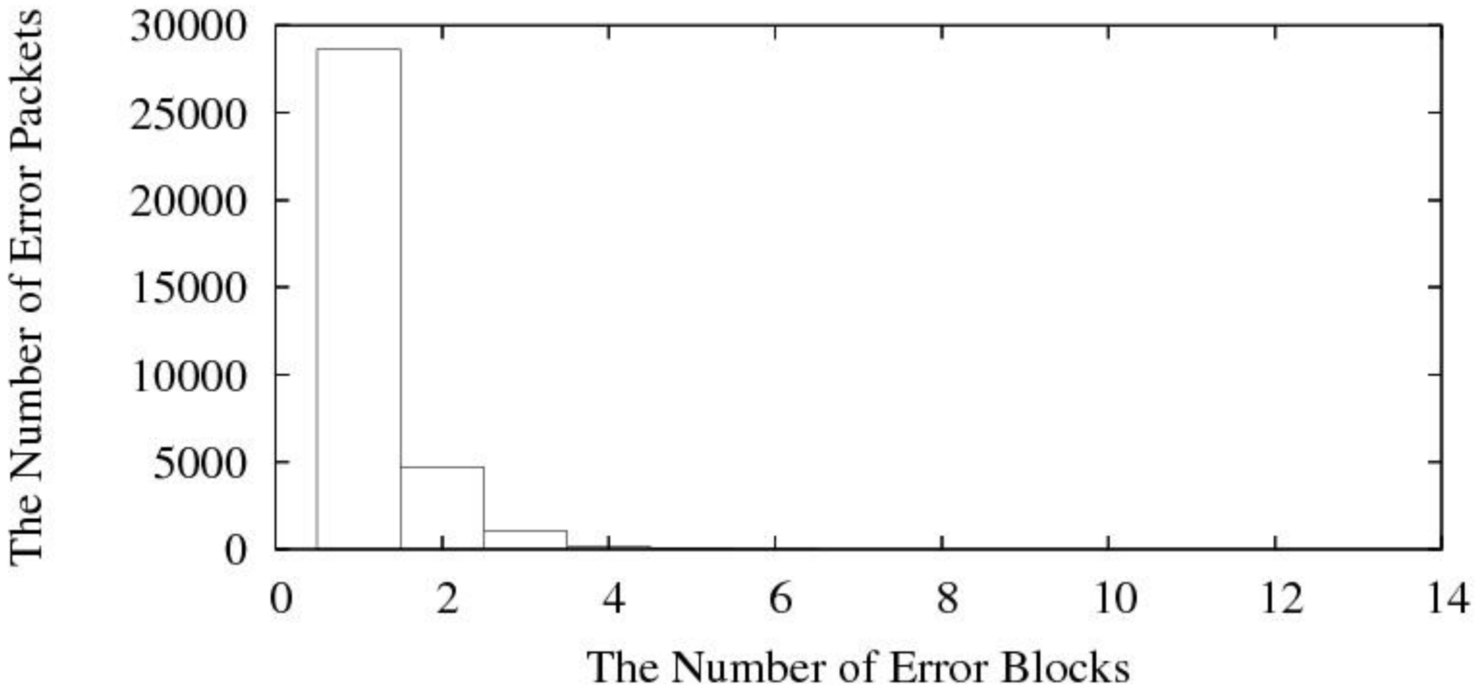
Errors & noise in WiFi

- Packet Error Rate of 802.11 networks is high[1]
  - Random noise can affect only a few bits
    - One or multiple blocks of corrupted bits inside a packet
  - Corrupted frames are discarded
    - Even if only 1 bit is wrong!
  - 802.11 retransmits after ACK timeout
  - Correctly received bits are completely wasted

[1] Bo Han, Lusheng Ji, Seungjoon Lee, Bobby Bhattacharjee, and Robert R. Miller. All Bits Are Not Equal. A Study of IEEE 802.11 Communication Bit Errors. INFOCOM 2009, pp. 1602-1610, Apr. 2009.

Errors & noise in WiFi/2

- Suppose we divide packets into 64bytes block
  - Typical packet trace of a managed station



Recent Approaches

- Forward Error Correction (FEC) based
  - ZipTx [2] sends RS redundant bits for recovery
  - Two-round coding scheme
  - Educated guess of BER and high recovery delay
    - Implemented(?) in kernel-space on Atheros devices
    - Evaluated in 11a, outdoor tests (low interference)

[2] K. C.-J. Lin, N. Kushman, and D. Katabi. ZipTx: Harnessing Partial Packets in 802.11 Networks. ACM MOBICOM 2008, pag. 351–362, Sept. 2008.

Recent Approaches

- Based on Automatic Repeat reQuest (ARQ)
  - PPR [3] relies on the confidence of each bit's correctness
  - Retransmit only corrupted bits
  - Not available in commercial hardware
    - implemented and evaluated on 802.15.4 protocol stack

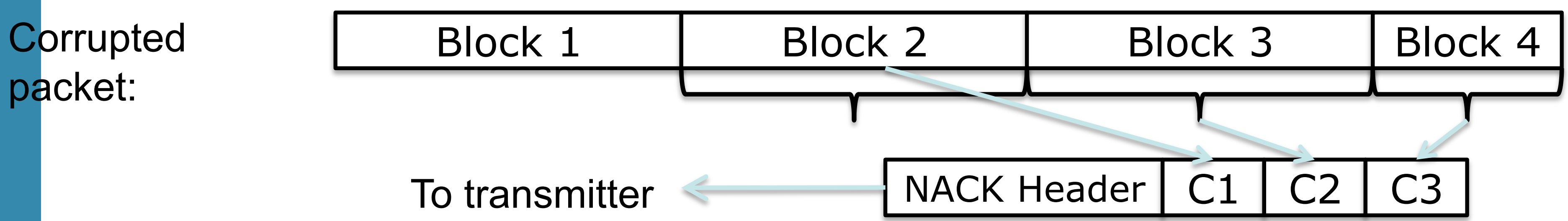
[3] K. Jamieson and H. Balakrishnan. PPR: Partial Packet Recovery for Wireless Networks. ACM SIGCOMM 2007, pag. 409–420, Aug. 2007

Our approach

- Similar to PPR
  - No access to confidence information
- Use checksum coefficient embedded in packets
- We implemented everything from scratch
  - Changes to Linux kernel
  - Changes to OpenFWWF
- We designed MARANELLO and BOLOGNA
  - AKAS Practical Partial Packet Recovery P^3R!

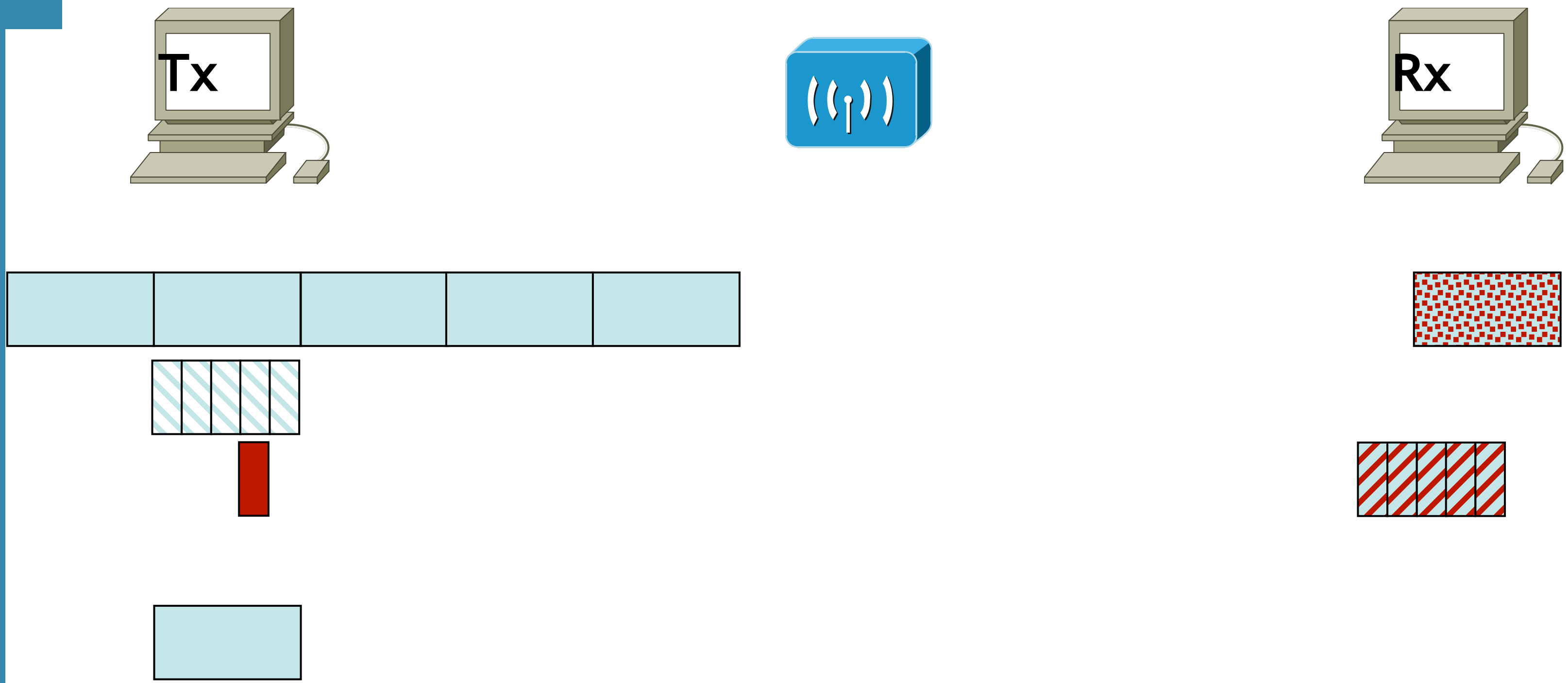
Maranello: P³R

- At rx corrupted packet is divided into blocks
  - Blocks are equally sized (apart the last one)
  - For each block apart the first compute a checksum
  - Checksums sent back to the transmitter in a N-ACK



  - Transmitter retransmits only corrupted blocks
  - First block can't be protected
    - It must always be retransmitted, contains the header!

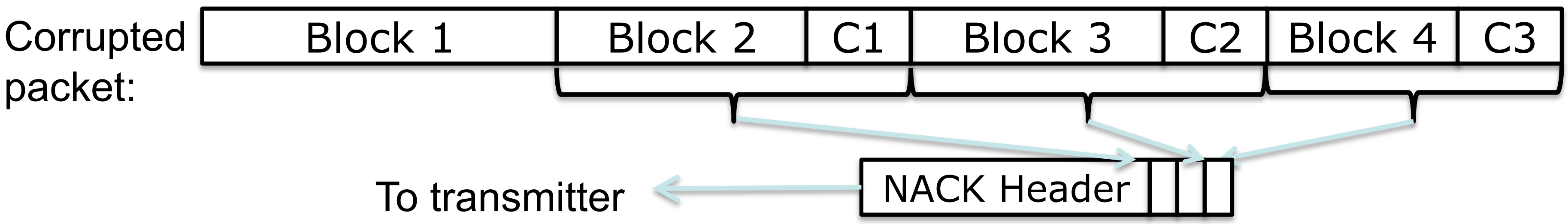
Maranello: handling retransmission



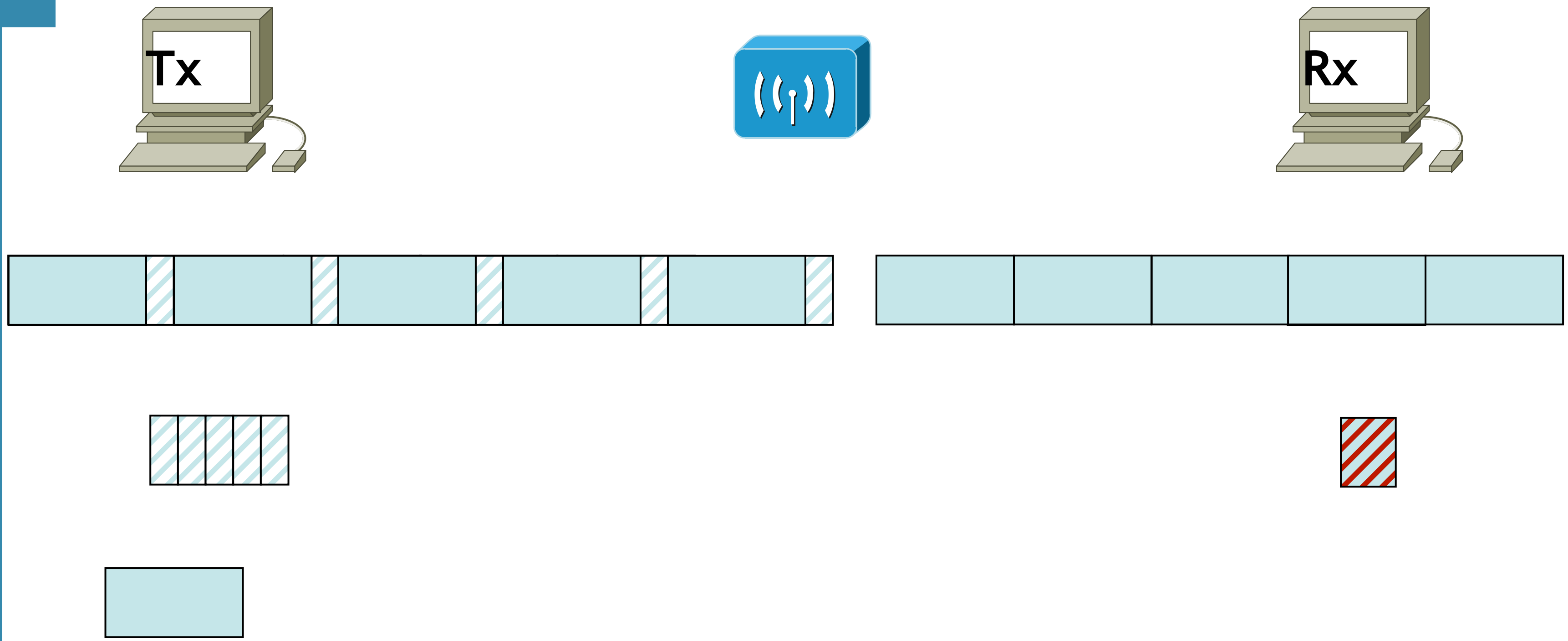
Block checks[illegible] computation ([illegible]fletcher32), packet retransmission

Bologna: P^3R

- Like Maranello but…
- At tx packet is expanded
  - In each block a checksum is embedded
- Rx checks all blocks:
  - If packet fails, send back a NACK
  - NACK is the bitmap of corrupt blocks



Bologna: handling retransmission



Advantages of P^3R

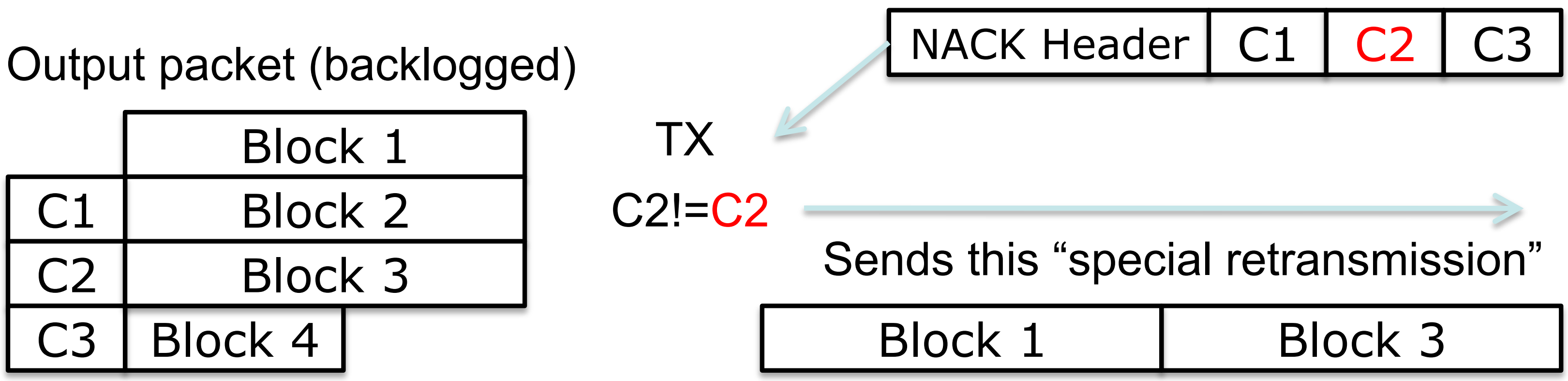
- Receiver-controlled recovery
- Utilizing the airtime reserved for ACKs
  - No additional overhead for correct packets
- Faster packet recovery
  - Recovery immediately after a transmission fails
  - Shorter recovery frames

Implementation Architecture

- Time-critical operations should be implemented in firmware space
  - RX: block checksum calculation, NACK generation
  - TX: block checksum calc., block retransmissions
- Why not in driver space
  - High bus transfer delay + interrupt latency (>70 us)
- ACK, and NACK:
  - must start within 10us after receiving a frame

Implementation: Transmitter

- Kernel=>Maranello operations:
  - precompute checksums for each output packet
  - send packet and checksums to the firmware
- Firmware=>Maranello operations:
  - receive NACK: compares checksums to those precomputed
  - rebuild “special retransmission” putting pieces together



Implementation: receiver

- Firmware=>Maranello operations:
  - compute checksums on packet reception
  - if frame is corrupted
    - send NACK instead of ACK, same timings
    - send corrupted packet up to kernel
- Kernel=>Maranello operations:
  - stores corrupted packet
  - when receives a special retransmission
    - rebuild the original packet

Other details

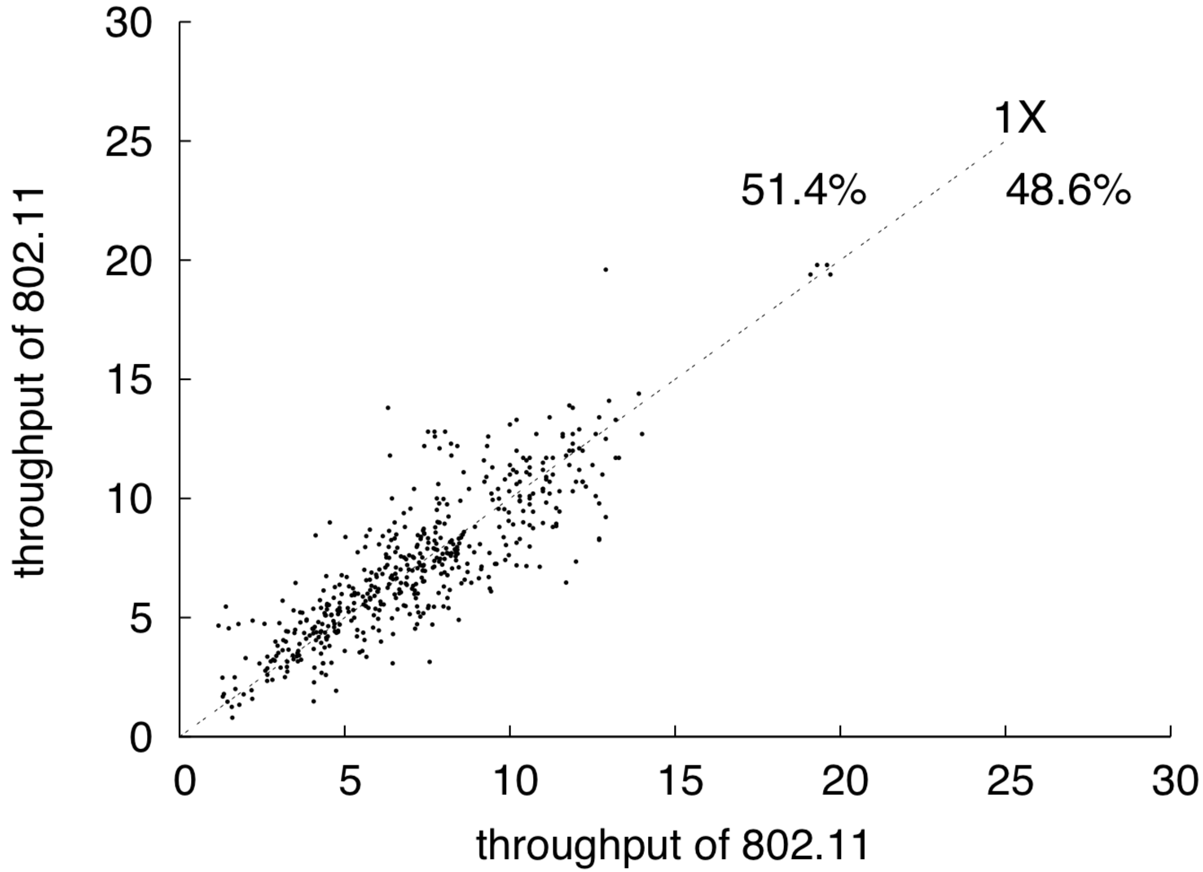
- Maranello & Bologna
  - We used 64-byte blocks
  - Checksum:
    - CRC16 is desiderata
    - OpenFWWF has not access to CRC engine
    - We compute Fletcher-16/32 checksums on the fly
  - Recovered packets protected by an additional CRC32 checksum

Throughput tests

- Repeat this experiment
  - 60s UDP traffic, sta to AP (iperf), legacy => ϑ_1
  - 60s UDP traffic, sta to AP (iperf), Maranello => ϑ_2
  - Plot (ϑ_1, ϑ_2)
- Each run follows sta initialization
- Three environments
  - ATT lab
  - Maryland campus
  - Bo's home
- Linux sta
  - Fixed channels (1, 6, 11)
  - Minstrel as RC

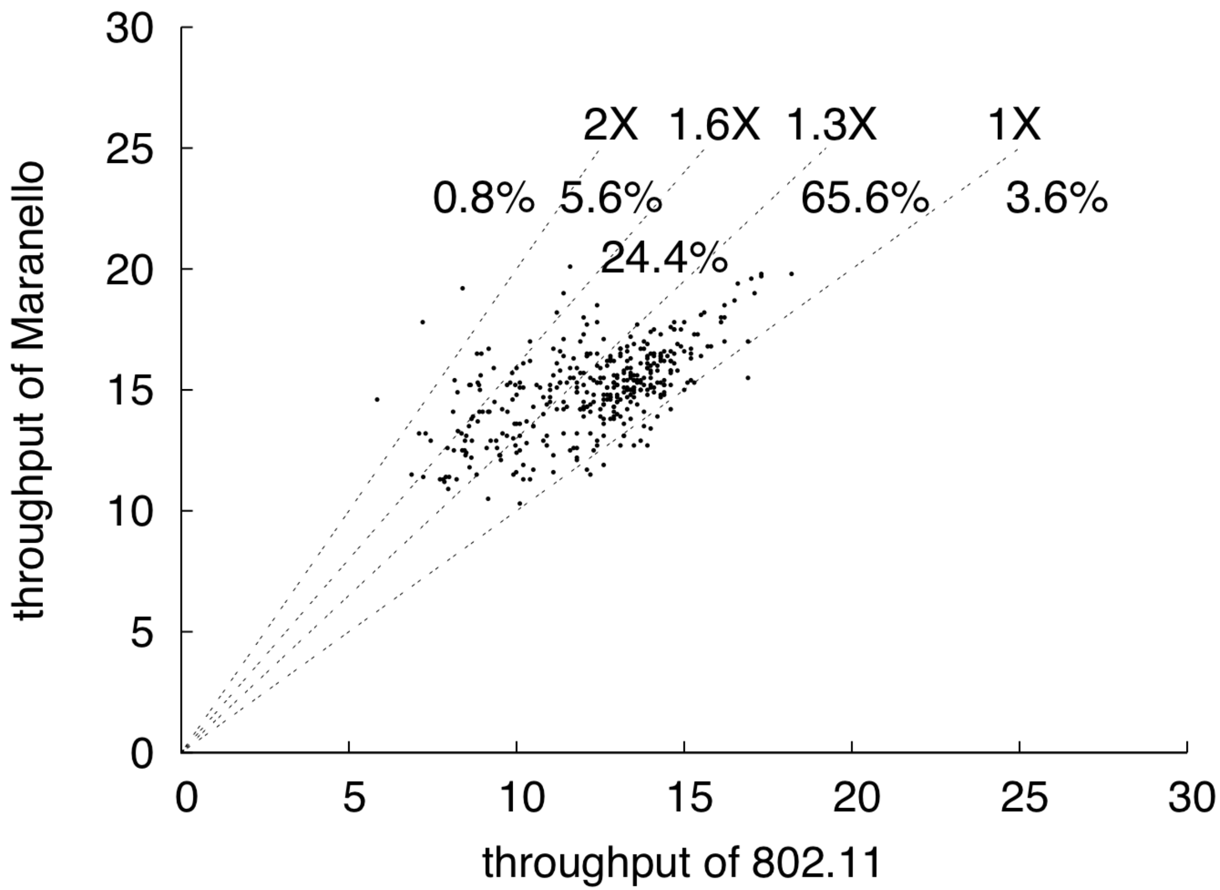
Throughput tests

- Reliable test?



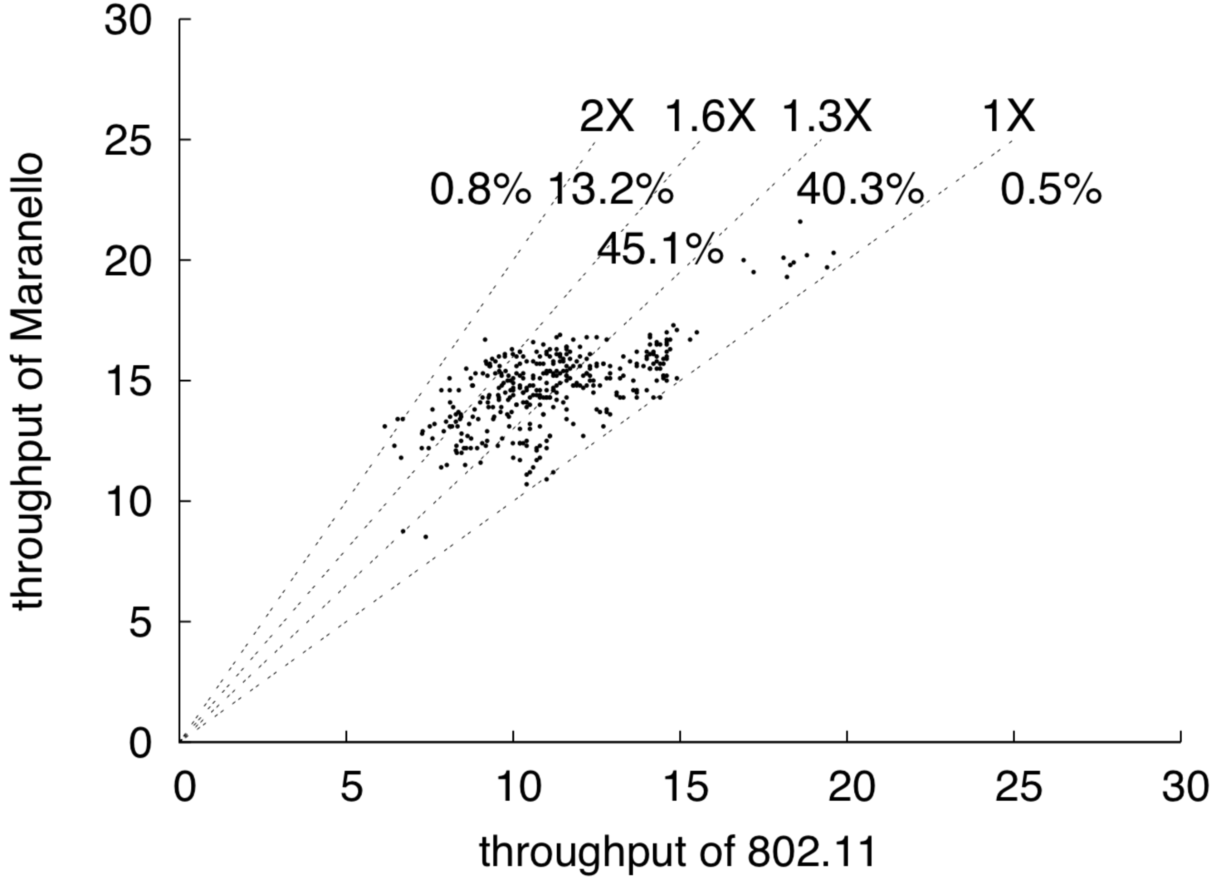
Throughput tests

- Bo’s home



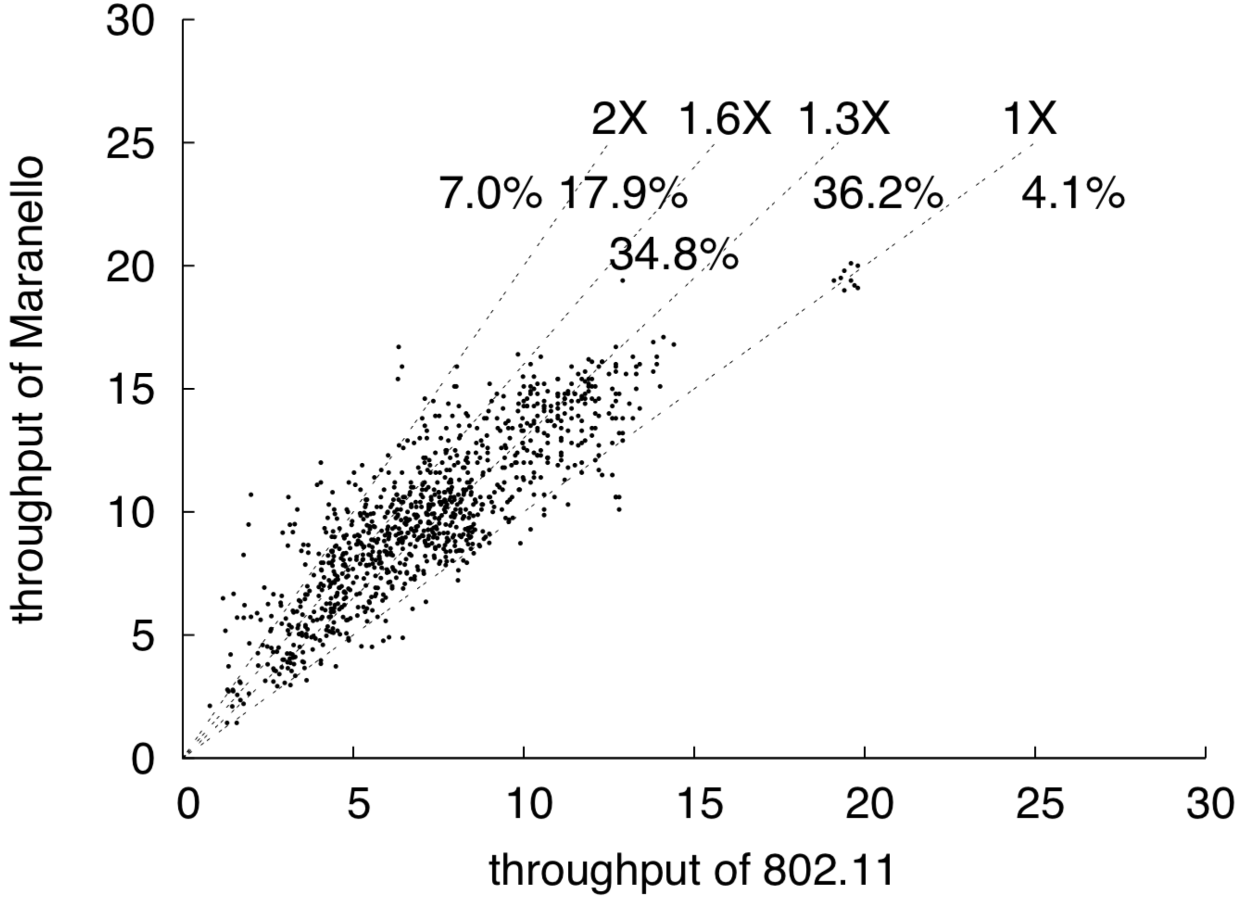
Throughput tests

- ATT lab



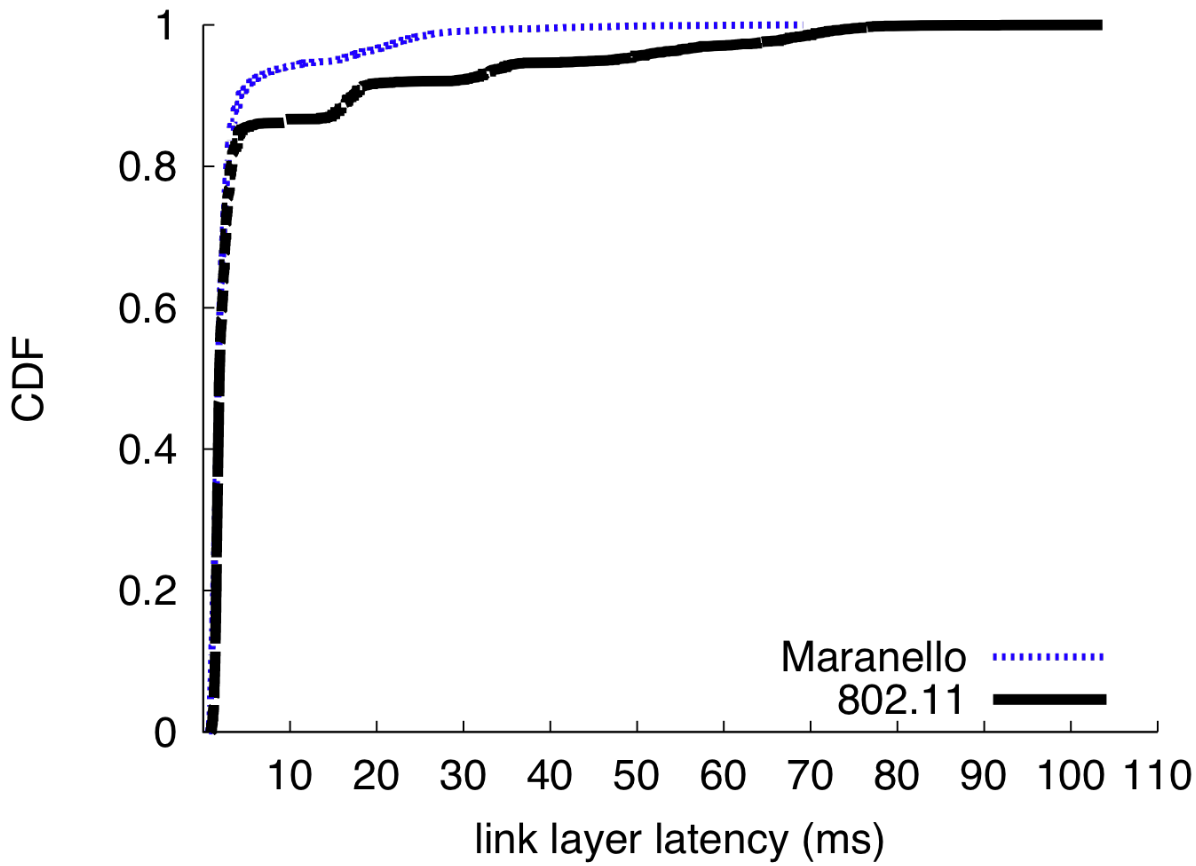
Throughput tests

- Maryland campus



Throughput tests

- Link layer latency is reduced (shorter retr)



MARANELLO vs BOLOGNA

Maranello

PRO
- *Partial Packet Recovery*
- Backward comp. 802.11
- Link latency--
- No extra-bits in reg. packets

ISSUES
- NACK very long

BBR

PRO
- *Partial Packet Recovery*
- Backward comp. 802.11
- Link latency--
- NACK minimized

ISSUES
- Packet expansion

OpenFWWF Exploitation: Implementation of 802.11aa

In collaboration with

Universidad Carlos III de Madrid

Dept. Ingeniería Telemática

Overview

- Uni/Multicast support in IEEE 802.11
- New amendment IEEE 802.11aa
- Implementation description
- Performance tests
- Conclusions

802.11 Channel access techniques
Unicast traffic

- Distributed Coordination Function
  - Based on CSMA/CA with binary exponential back-off
    - Waits for channel to be idle
    - Transmits a frame and wait for acknowledgement
    - If collision, inflates contention window and retransmits
  - **Reliability** through feedback

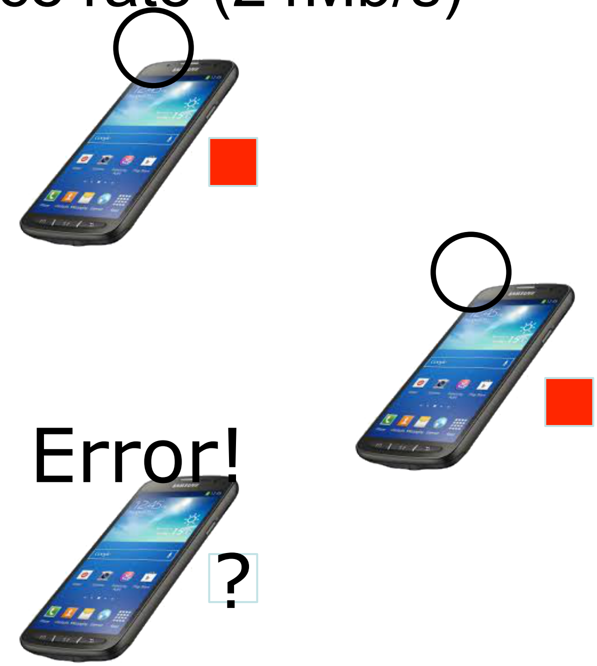
802.11 Channel access techniques/2 Unicast traffic

- DCF access and **unicast** frames
- Evolutions since release of first standard (1997)
  - QoS:
    - Many queues at single node competing for access
  - Block-Ack:
    - Transmits many frame and waits for single ack frame
  - AMPDU (Aggregated MPDU)
    - Transmits a single physical header + many frames, use a single HT-ACK
- Majority of 11N and 11 AC chipsets already support!

What about **multicast** access?

802.11 Channel access techniques/3 Multicast traffic (as in 1997 802.11)

- **Multicast access**
  - Frames sent with default (minimum) contention
  - No ACK, no retransmission
  - Transmission rate up to basic service rate (24Mb/s)
    - **Not reliable!**



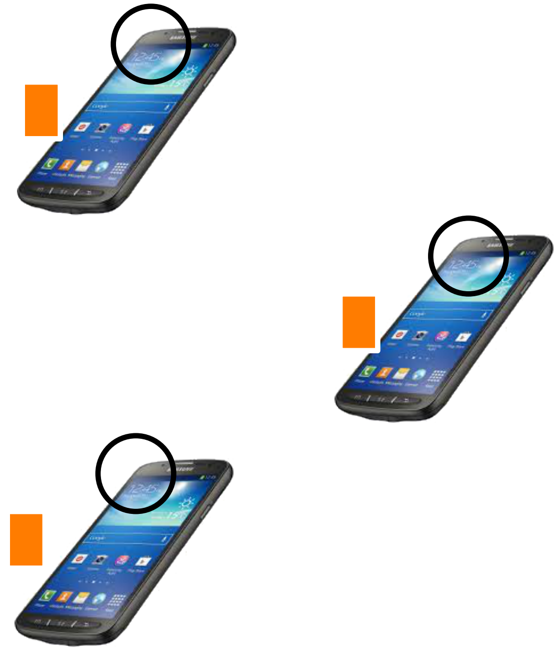
- Stuck to 1997…?

802.11 Channel access techniques/4 Multicast traffic: some news!

- 802.11aa and Group Address Transmission Service (GATS)
  - Removes 24Mb/s limit in MCS selection
  - Defines GroupCast Concealment Address as multicast target
- Access mechanisms
  - DMS – Directed Multicast Service
    - Delivers multicast frames with many unicast streams
  - GCR UR – GroupCast with Retries – Unsolicited Retries
    - Preemptively transmits frames 1 + **R** times
  - GCR BA – GCR with Block-Acknowledgment
    - Transmits burst of **M** frames and polls stations for collecting info
- Problem: do they work? No implementation yet… no real test!
  - We built the first working prototype and measured performance

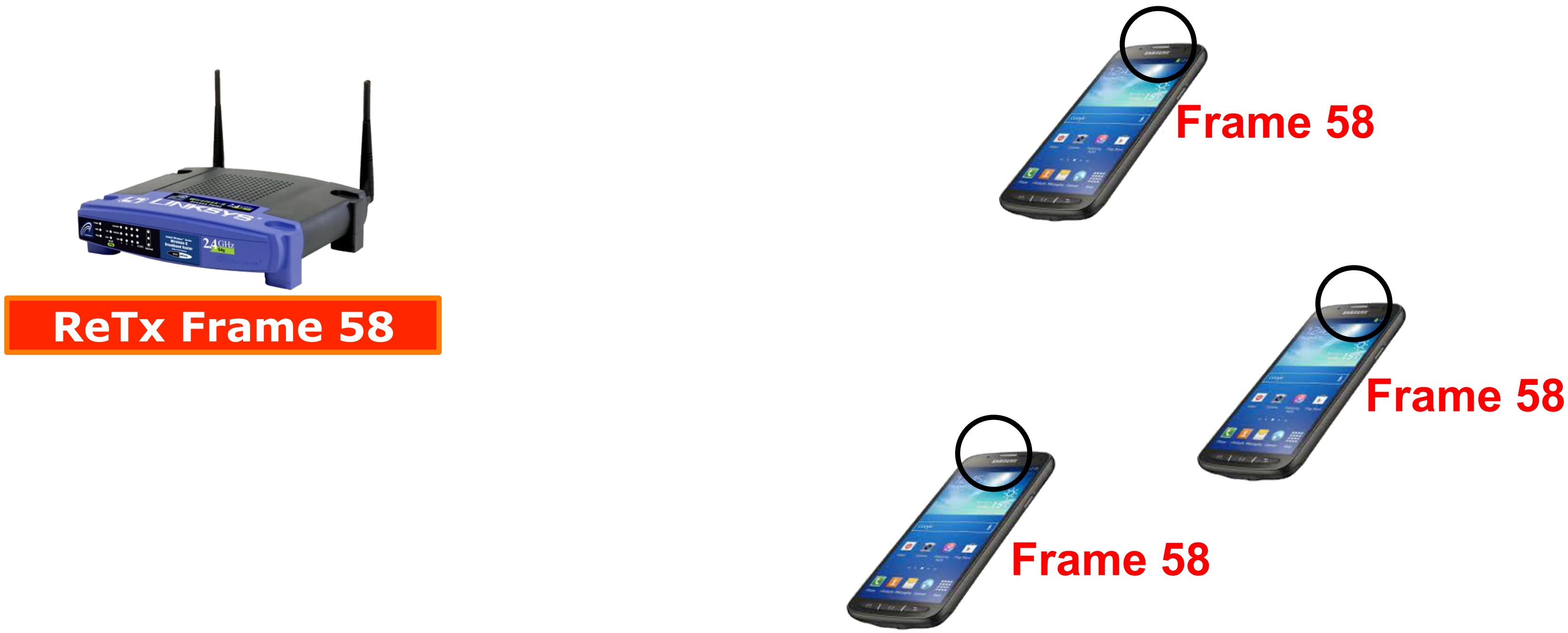
802.11aa: Directed Multicast Service

- Use DCF for unicast delivery to each destination
  - From a single stream (multicast) to many (unicast)
- Standard access: exponential backoff!
  - No prioritization over other traffic
- **Reliability** builds on DCF!



802.11aa: GCR Unsolicited Retry

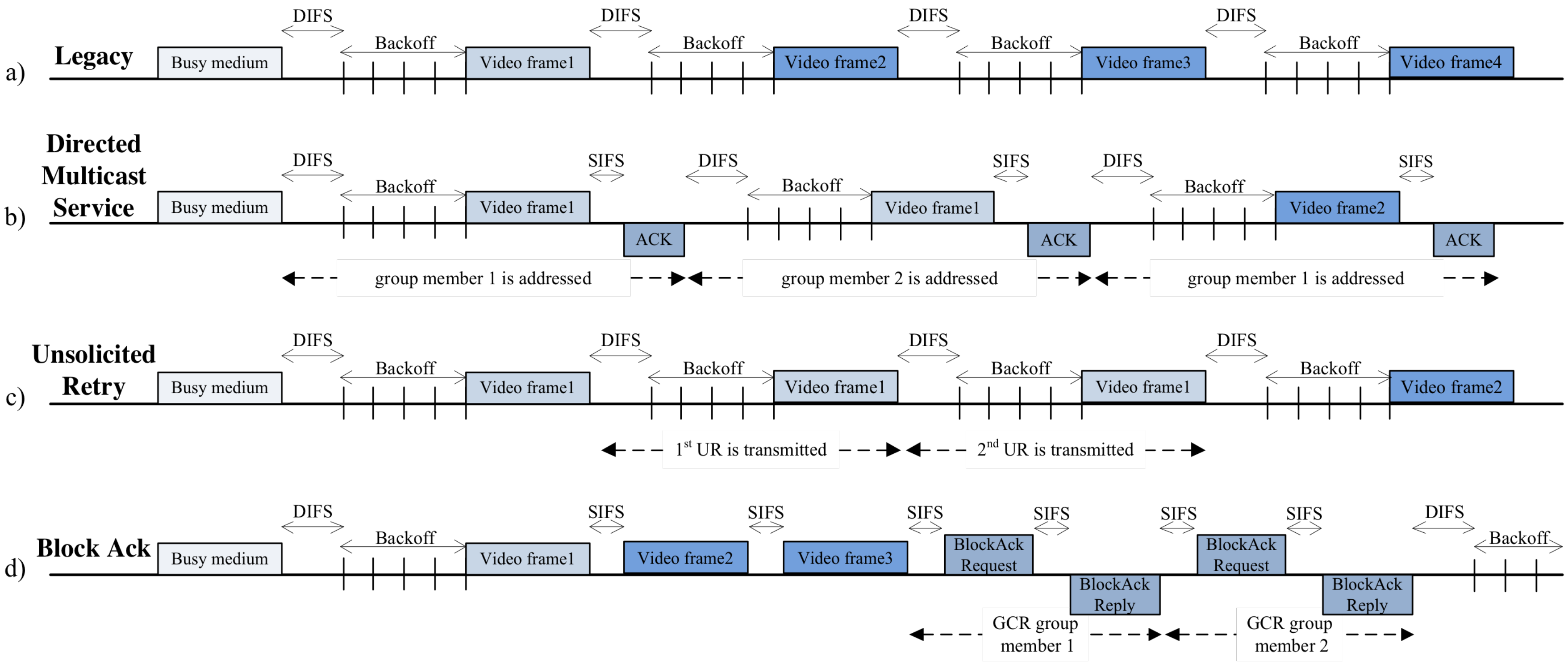
- Similar to legacy service without MCS limit
- **Reliability** builds on preemptive **R** (re)transmission
  - Open loop, does not use feedback from receivers



802.11aa: GCR Block-Ack

- Frames sent in (configurable length **M**) bursts
  - Really multicast delivery to Groupcast address
- Feedback (Block-Ack) collected with unicast polls
  - Block-Ack-Request(BAR) followed by Block-Ack(BA)

802.11aa: synoptic table of GATS



802.11aa: summary

• w

– Complexity – Overhead

DMS

- − Highly dependent on the number of stations
- + Excellent reliability
- + **Simple**
- − **Lot of overhead**

GCR UR

- + Independent on the number of stations
- / Discrete reliability
- + **Simple**
- − **Lot of overhead**

GCR Block-Ack

- / Depends on the number of stations
- + Good reliability
- − **Complex**
- + **Some overhead for poll procedure**
  - + Retransmit only what is missing

Implementation of GATS

- DMS & GCR-UR: can be implemented at kernel
- But GCR-BA: many time-critical operations
  - Need to change the firmware at the NIC
  - Broadcom 4318 consumer chipset mandatory choice
    - Supported by Opensource firmware OpenFWWF
  - New functionalities at NIC:
    - Keep delivery statistics at receiver in real-time
    - Collect delivery statistics at sender by BAR-BA procedure
      - Polling mechanism: forging BAR and BA
    - Immediate retransmission of lost frames
- **Platform: Linux + b43 kernel driver + OpenFWWF**

Performance evaluation

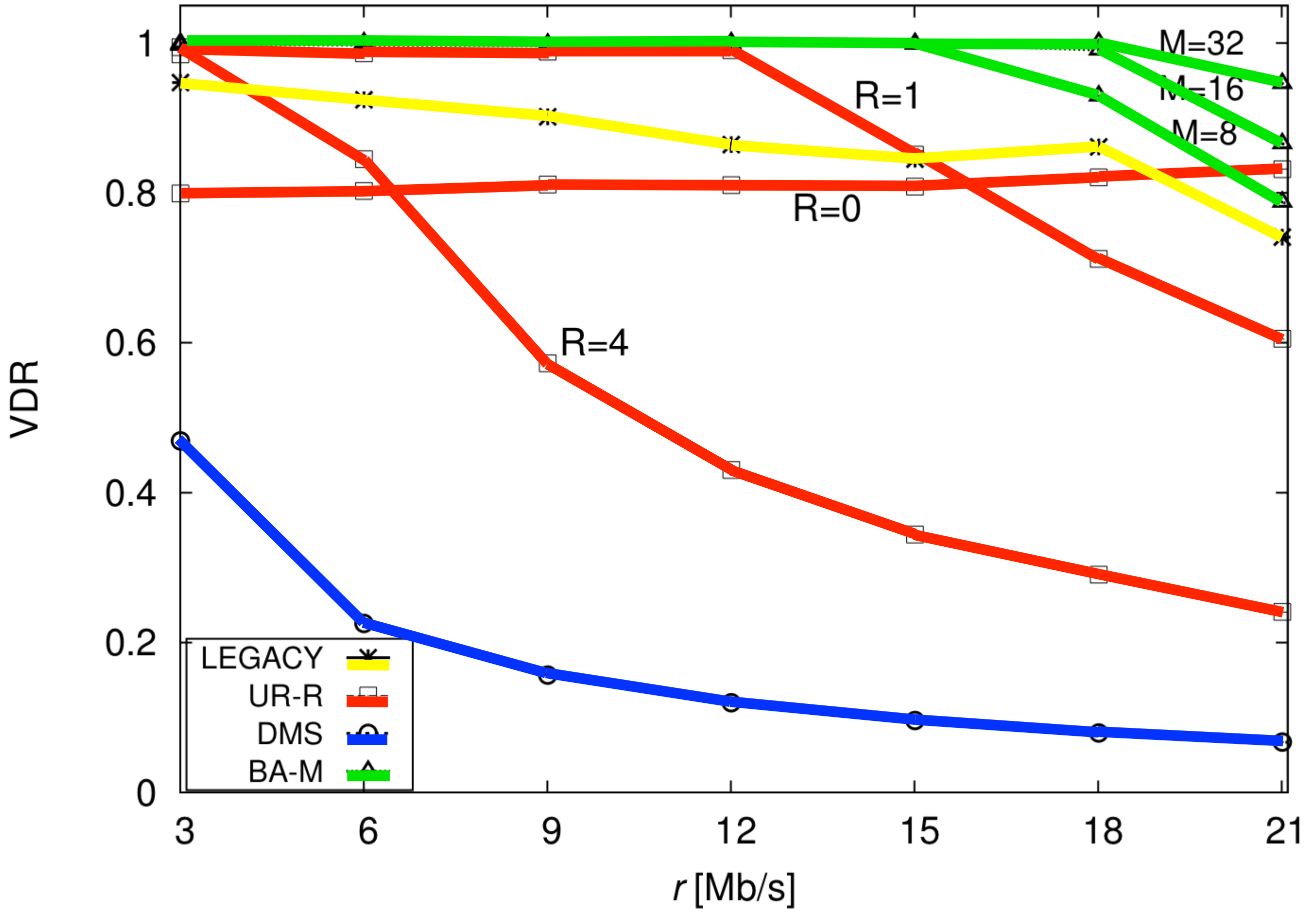
- Compare GATS mechanisms, different input load
  - Multicast video is CBR, generated at fixed rate *r*
  - Nv = 10 multicast receiver
  - Nd = 10 data stations sending backlogged UDP to AP
  - All frames are 1400 bytes
- Two performance figures
  - Video Delivery Rate (VDR)
    - Average percentage of throughput received by Nv
  - Aggregated Data Throughput (ADT)
    - Sum of data throughput at AP
    - Reveal how many wireless resources are left

Testbed

- AP is a PC
- Both Video and Data stations are Alix 2d2 nodes
  - All tests done at UC3M
- Tests on channel 14 (interference free) and 11
  - On channel 11 also with (Emulated) massive video loss
- For GCR-BA
  - Explored M=[8, 16, 32]
- For GCR-UR
  - Explored R= [0, 1, 4]
- MCS choice
  - GATS: fixed to 54Mb/s
  - Legacy: fixed to 24Mb/s

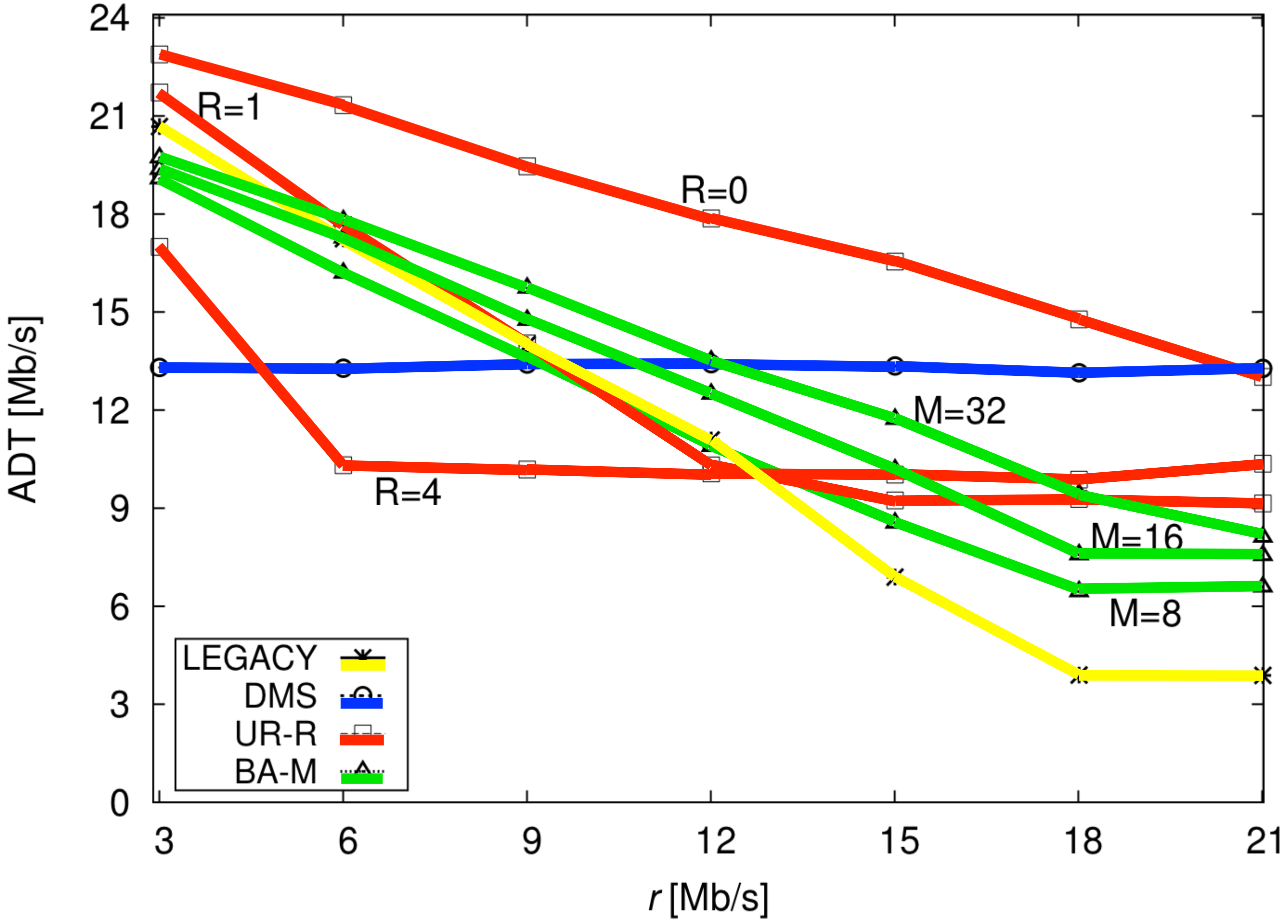
Results

- Video Delivery Ratio channel 14



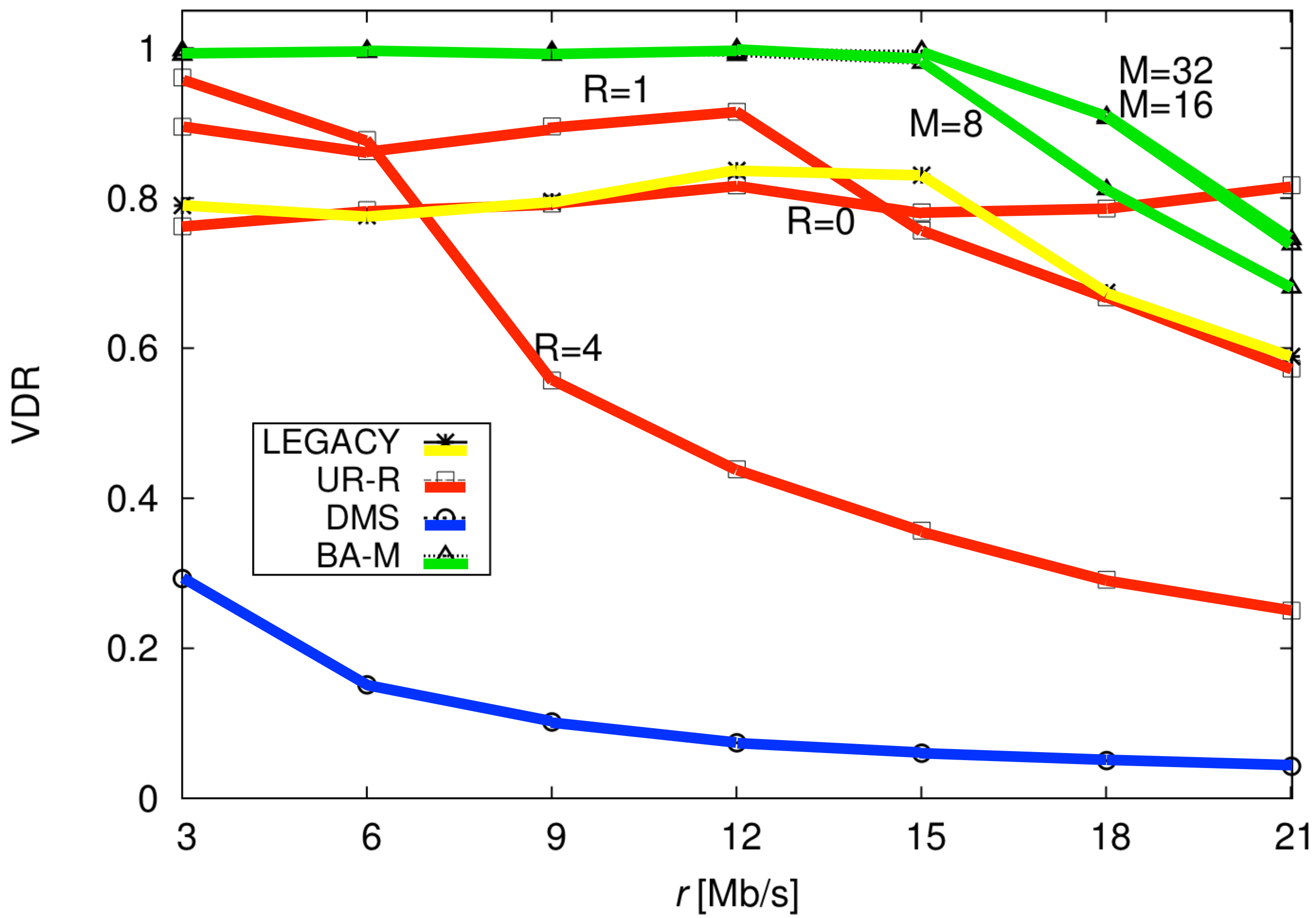
Results/2

- Aggregated data throughput channel 14



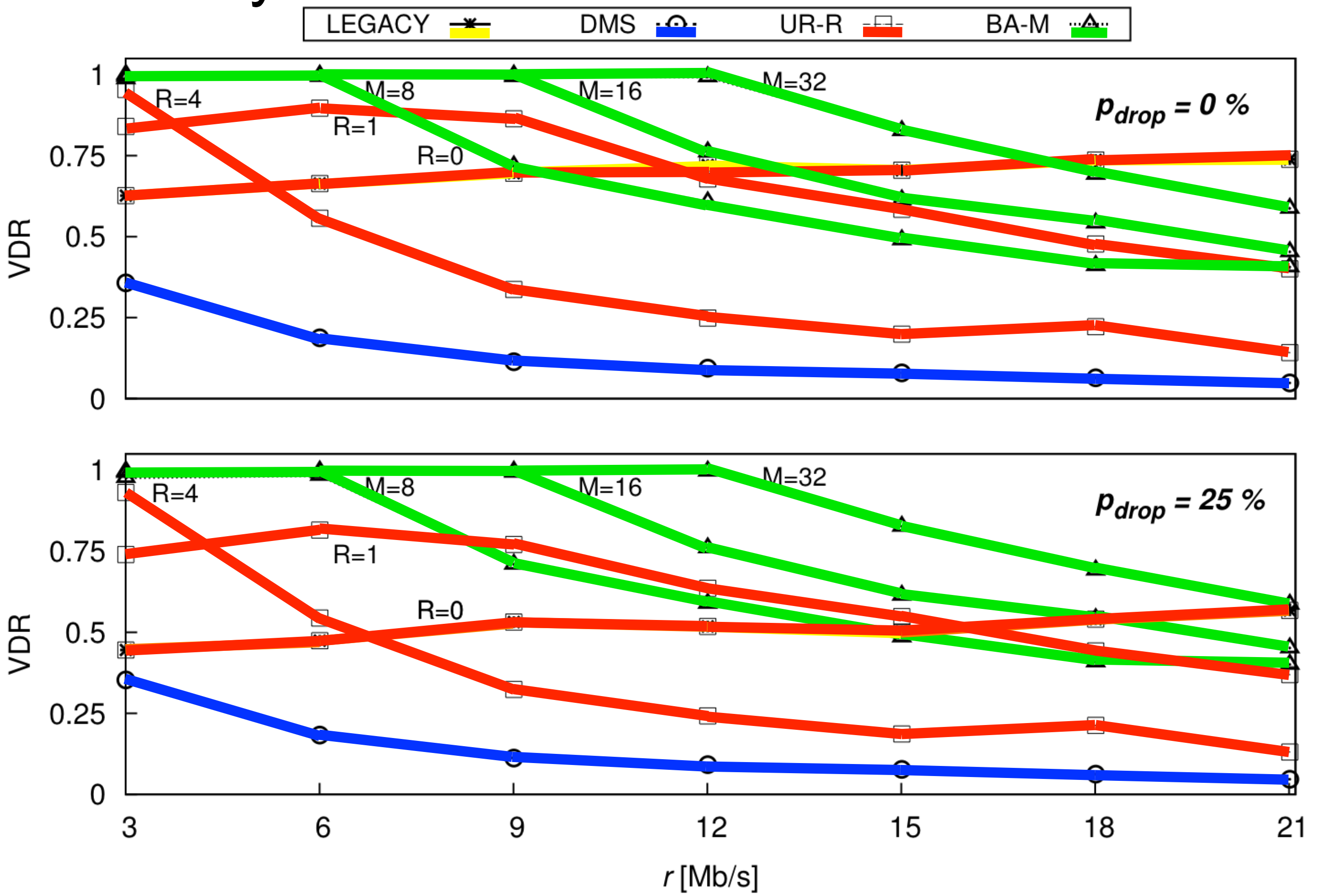
Results/3

- Video Delivery Ratio channel 11



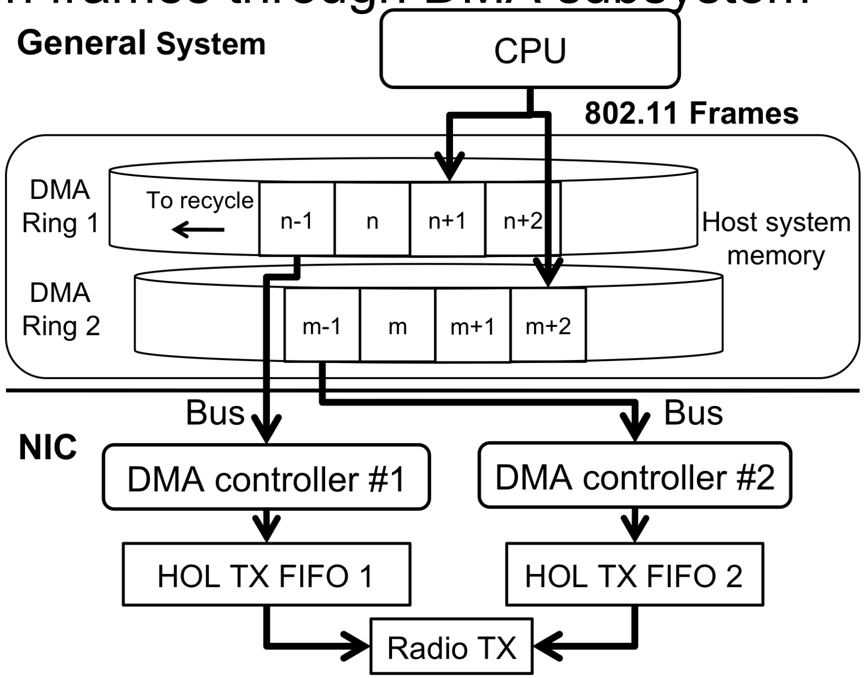
Results/4

- Video Delivery Ratio: 25% loss at one video receiver



Analysis of current devices About supporting GCR-BA

- GCR-BA: after BAR-BA polling
  - AP retransmits frames (if needed) within milliseconds
  - Problem: NICs do not have large buffers
    - Hosts push frames through DMA subsystem



Analysis of current devices About supporting GCR-BA/2

- For 802.11aa with GCR-BA
  - We replicate each M-frame burst filling all 5 queues
  - Original transmission empties queue #1
  - For first retransmission, NIC scans queue #2
    - Only lost frame are transmitted
  - When all frames are retransmitted, frames left in other queues are simply dropped (queue flush)
- Problem: given these “queues” are DMA FIFO
  - Scanning/flushing a queue requires time for transferring frame from host memory to the NIC
    - Limited bandwidth (it’s a PCI bus)

Analysis of current devices About supporting GCR-BA/3

- We found this can be an issue, example:
  - All frames received at first attempt
  - Need to flush the remaining four queues: takes time
  - We can not cope with maximum throughput!
- Bottom line:
  - If 802.11aa implemented like we did (no other possibilities actually) current NIC generation can't cope with 802.11aa at full speed!
- We examined most recent devices
  - E.g., 11ac chipset from Broadcom exhibit same architecture, meaning same problems!

Conclusions

- First experimental evaluation of 802.11aa standard
  - Each GATS mechanism offers specific improvement WRT legacy multicast
- We release all sources as open-source
  - Simple starting-block for developing new multicast access delivery protocols
- Future works
  - Add support for rate control
    - Especially for GCR-BA: transmitter knows channel joint probability of reception, can estimate best rate
  - Find optimal configuration for R and M

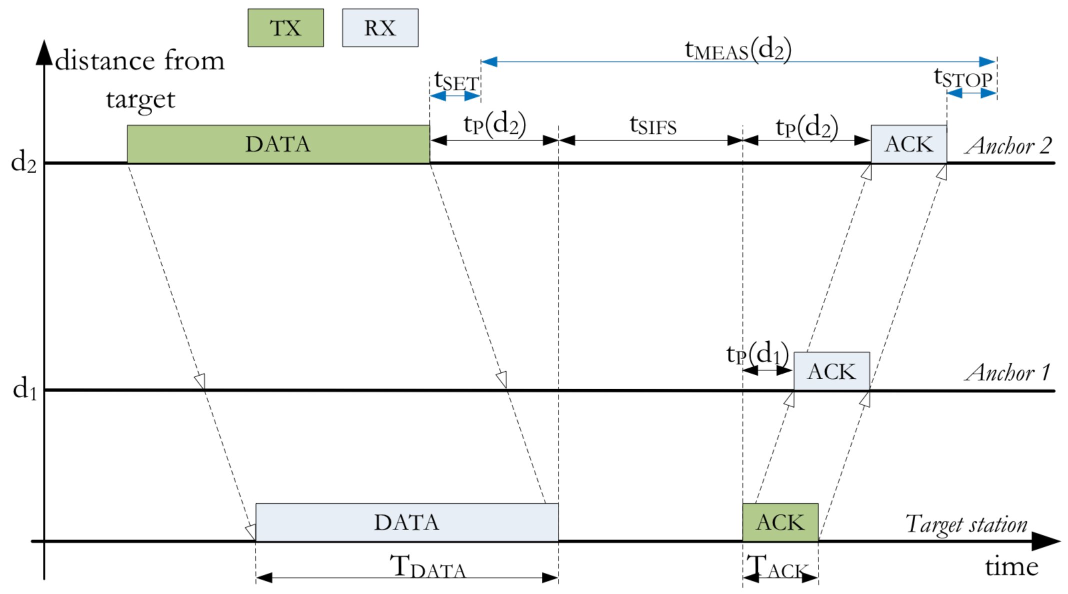
OpenFWWF Exploitation: Node localization

In collaboration with

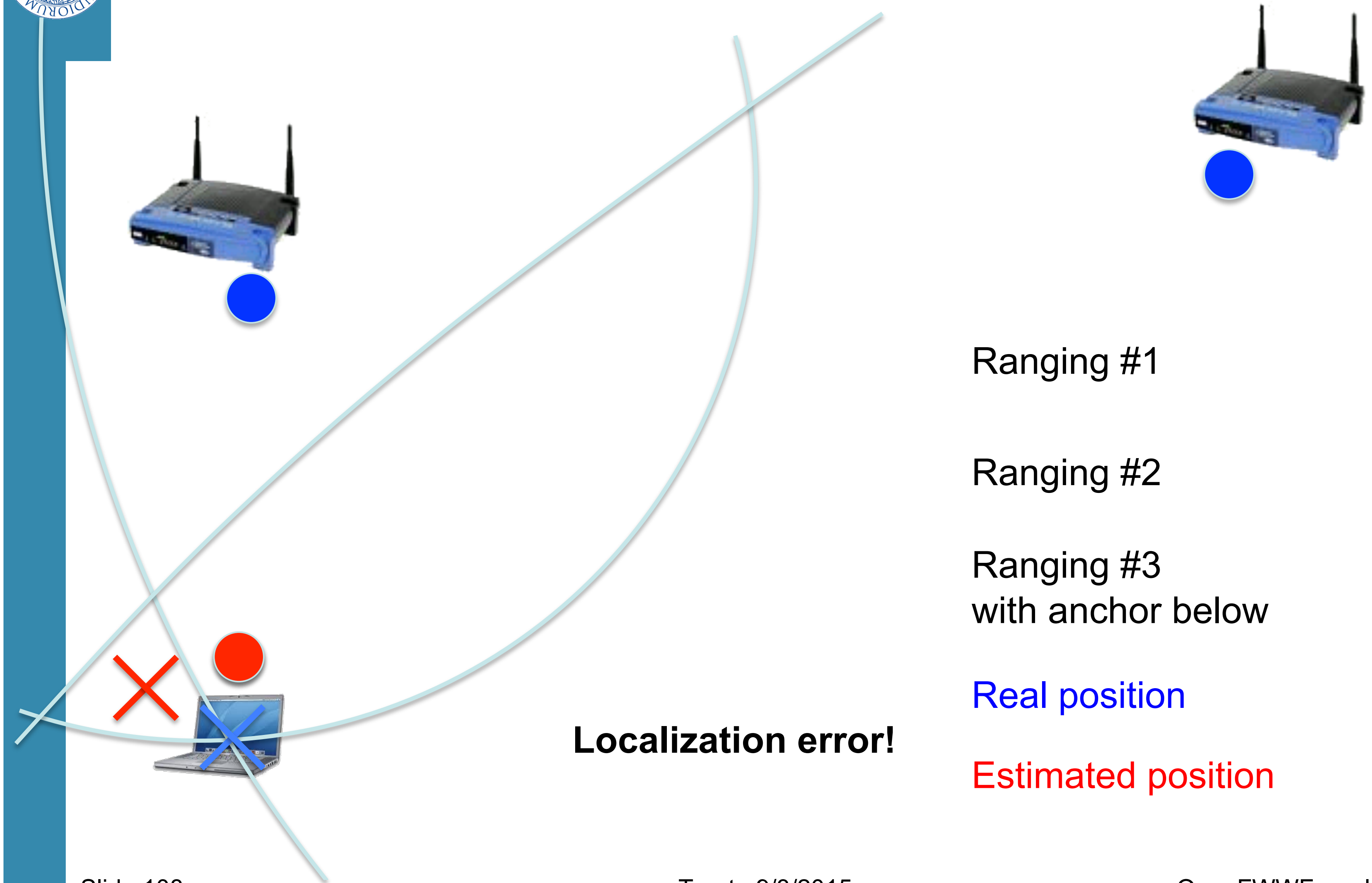
Too many…

Localization with 802.11

- Find position of a node
  - Ranging problem: measure distances from known anchors
- Ingredients:
  - Fast clock: Broadcom cards have 88MHz, ☺
  - Easy to trigger conditions: TX_END and RX_COMPLETE, ☺



Localization with 802.11: how to/1



Ranging #1

Ranging #2

Ranging #3
with anchor below

Real position

Estimated position

Localization error!

Localization with 802.11: how to/2

- Many anchors send probes to target to localize
  - When probe tx'ed, start a clock
  - When ACK rx'ed, stop the clock, compute delay DTn
- It's based on Time-of-flight (TOF)
- Positions of anchors is known (e.g, museum, store…)
  - Correlates DTn from all anchors
  - May use Bancroft algorithm (GPS), or bounding box…
- It's easy… Cisco and Fraunhofer sell this system today!
  - Q: so what? (BTW, they also use power estimation)
  - A: we want to check if it works ☺