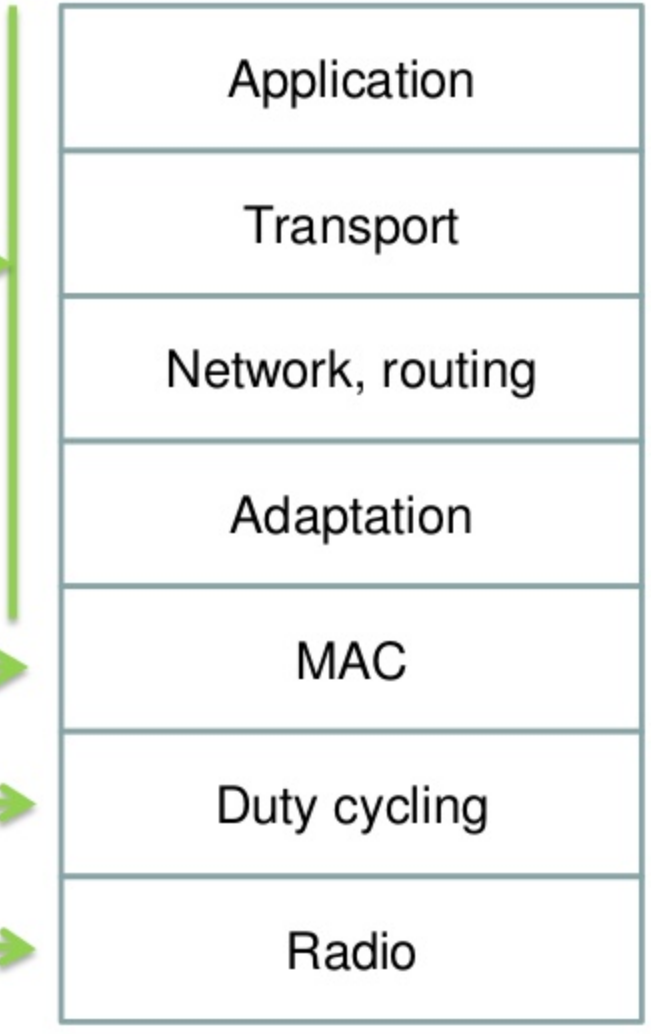
The Contiki netstack

The Contiki netstack

- Four layers
  - Network layer
  - MAC layer
  - RDC layer
  - Radio layer



Netstack concepts

- Four fixed layers
  - NETSTACK_RADIO
  - NETSTACK_RDC
  - NETSTACK_MAC
  - NETSTACK_NETWORK
- The packet buffer – packetbuf
- Queue buffers – queuebuf
- uIP packet buffer – uip_buf
- Framers
  - NETSTACK_FRAMER

The packetbuf

- One buffer, holds a single packet
- All layers of the netstack operate on the packetbuf
- Large enough to hold a single radio packet
  - PACKETBUF_CONF_LEN
- Packet attributes
  - Parsed header data: addresses
  - Packet meta data: RSSI

The packetbuf hdr and data

- hdr grows upwards
  - Only used on outbound path
- data typically don't grow
  - Contains data on outbound path
  - Contains header and data on inbound path

packetbuf

- `void packetbuf_clear(void);`
- `int packetbuf_copyfrom(const void *from, uint16_t len);`
- `void *packetbuf_dataptr(void);`
- `uint16_t packetbuf_datalen(void);`
- `void packetbuf_set_datalen(uint16_t len);`
- `void *packetbuf_hdrptr(void);`
- `uint8_t packetbuf_hdrlen(void);`
- `uint16_t packetbuf_totlen(void);`
- `void packetbuf_compact(void);`
- `int packetbuf_copyto(void *to);`

queuebuf

- The packetbuf only holds the current packet
- To store packets on queues, use a queuebuf
- `struct queuebuf *queuebuf_new_from_packetbuf(void);`
- `void queuebuf_to_packetbuf(struct queuebuf *b);`
- `void queuebuf_free(struct queuebuf *b);`
- Use a list to keep track of them

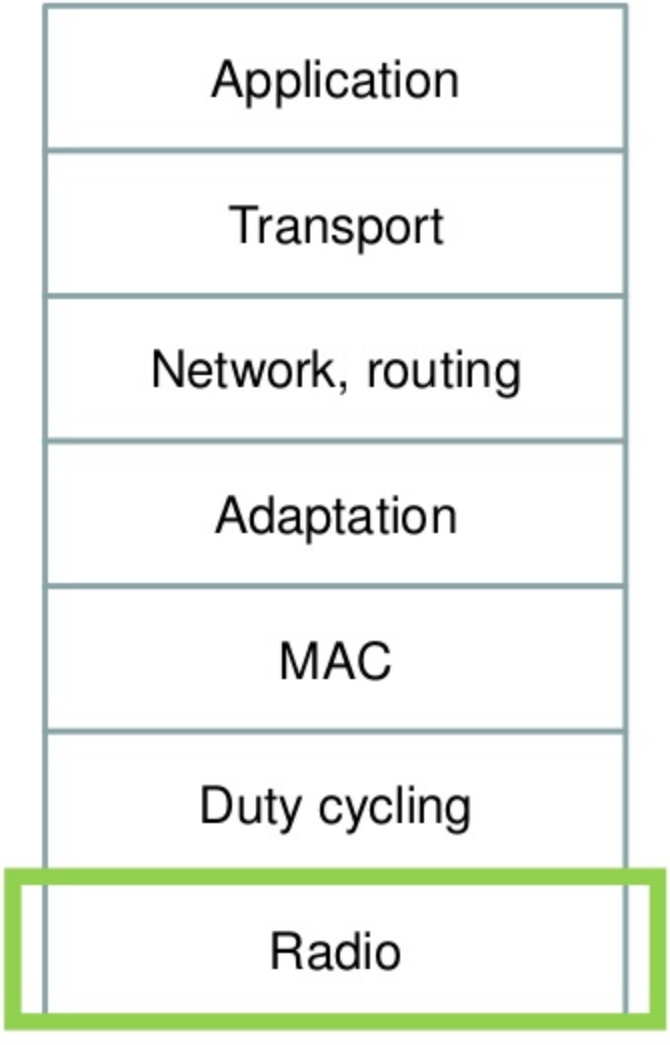
Framer

- The framer module converts link-layer headers to packet attributes
  - **`parse()`**
- And packet attributes to link-layer headers
  - **`create()`**

Walking up the netstack

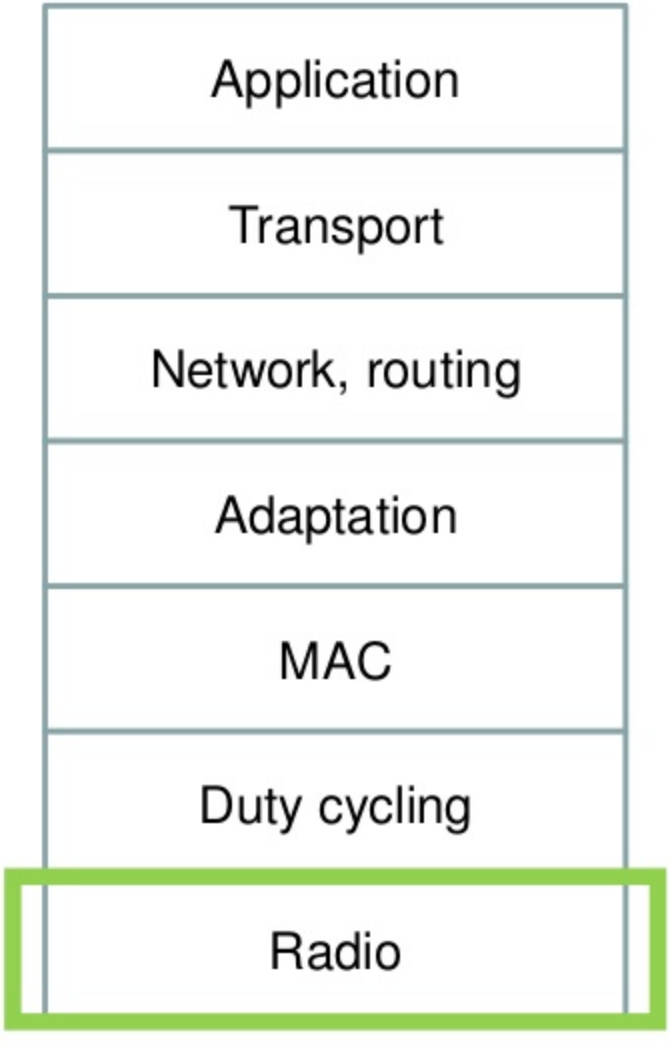
A packet through the netstack

- Step 1: the radio interrupt
  - Data arrives in bytes via an interrupt handler
    - Read byte by byte, put in buffer
    - Poll process on last byte
  - Or
  - Data arrives as a full packet via interrupt handler
    - Read out packet into buffer, poll process



A packet through the netstack

- Step 2: the radio process
  - Copy packet data from buffer into packetbuf
  - Read out RSSI, store as packetbuf attribute
  - Call NETSTACK_RDC.input();

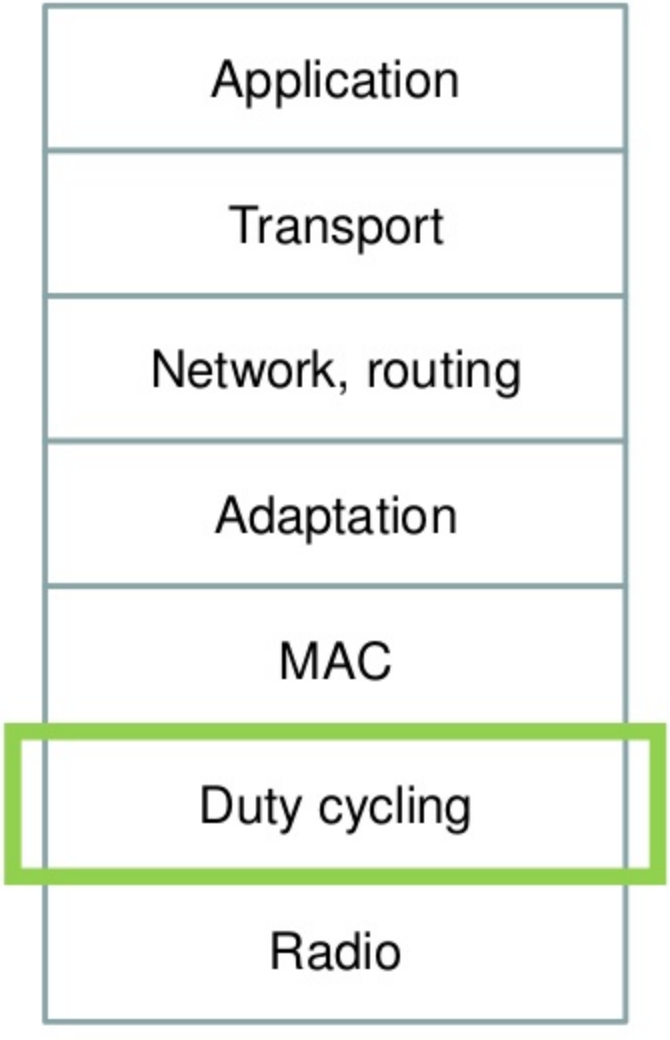


Interruption: Calling Contiki from an interrupt

- The golden rule: there is only one safe Contiki function to call
  - process_poll();
- process_poll(&process) will cause the process to be sent a special PROCESS_EVENT_POLL event
- Try to do as much as possible in a poll event handler
- Synchronize data with ringbuf

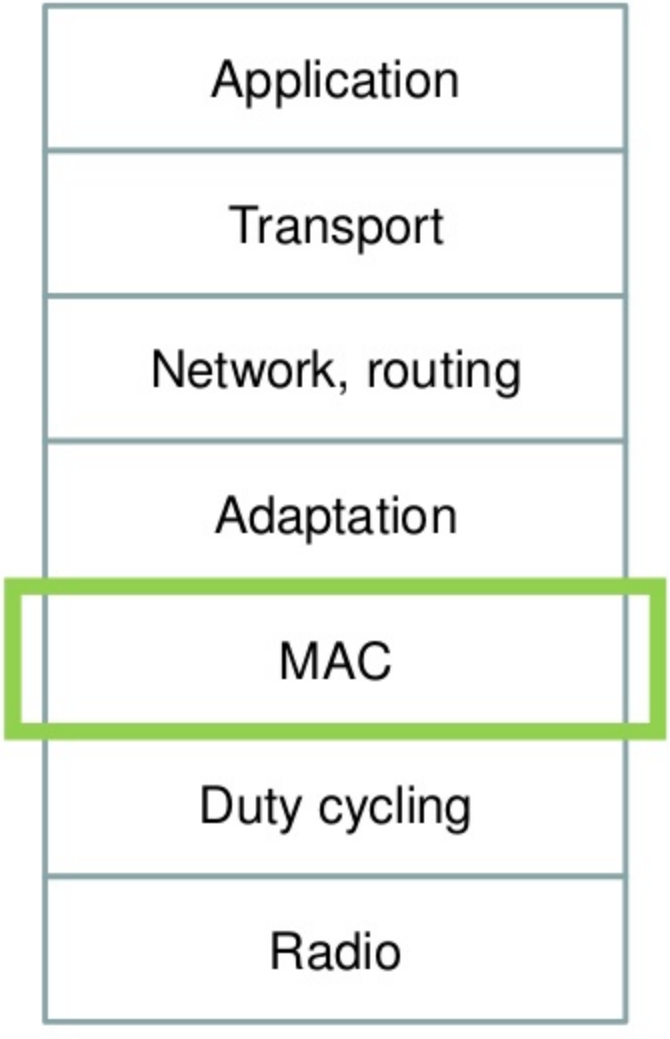
A packet through the netstack

- Step 3: the RDC input
  - Call framer to parse header
  - Check if pending bit set
    - Indicates a packet burst
    - Might need to have radio on for a while
  - Call NETSTACK_MAC.input()



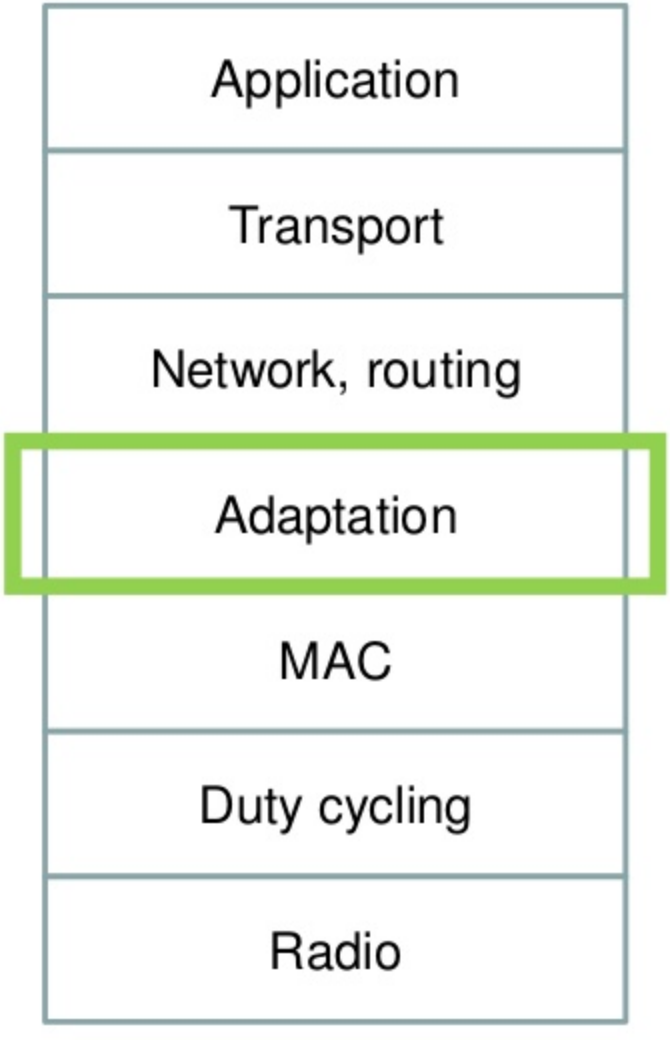
A packet through the netstack

- Step 3: the MAC input
  - Just call NETSTACK_NETWORK.input()



A packet through the netstack

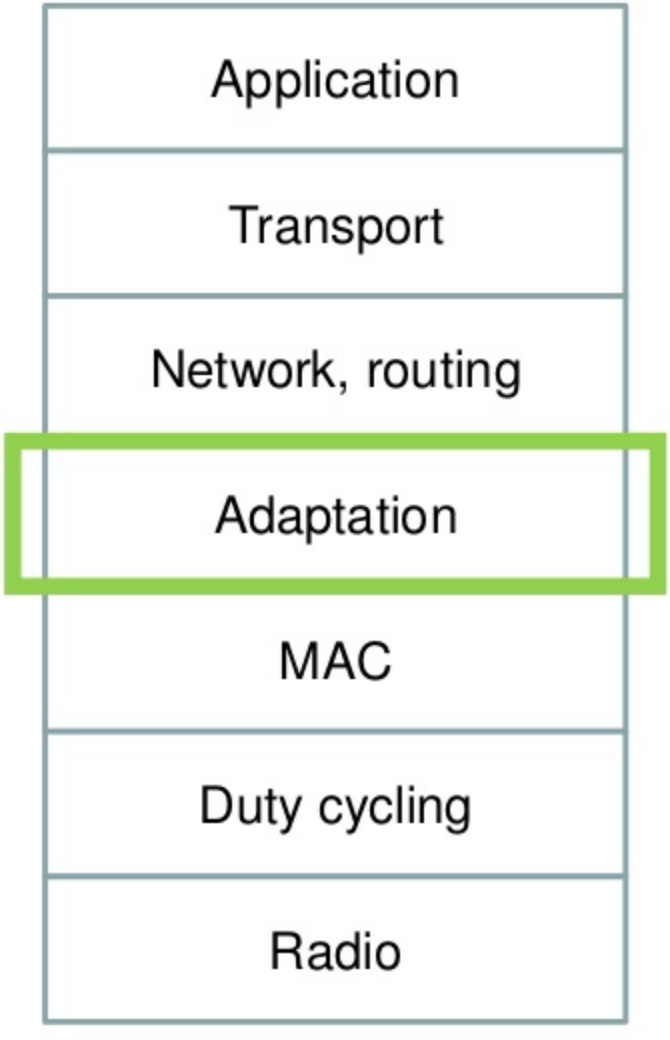
- Step 4: the 6lowpan input
  - Uncompress header
  - If part of a larger IPv6 packet, copy into fragmentation reassembly buffer and return to wait for the next packet
  - Copy packet into uip_buf
  - Call the IPv6 stack **`tcpip_input()`**



Walking down the netstack

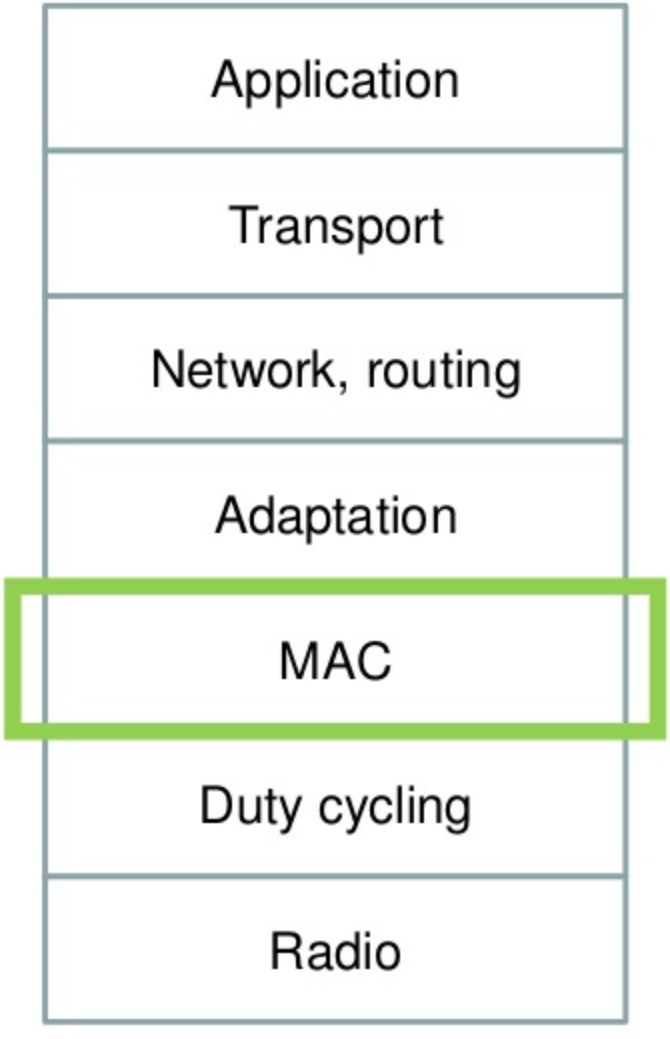
A packet through the netstack

- Step 1: the 6lowpan output
  - Clear packetbuf
  - Compress header from uip_buf to packetbuf
  - If uip_buf packet too large, split into several queuebufs
  - Call NETSTACK_MAC.send();



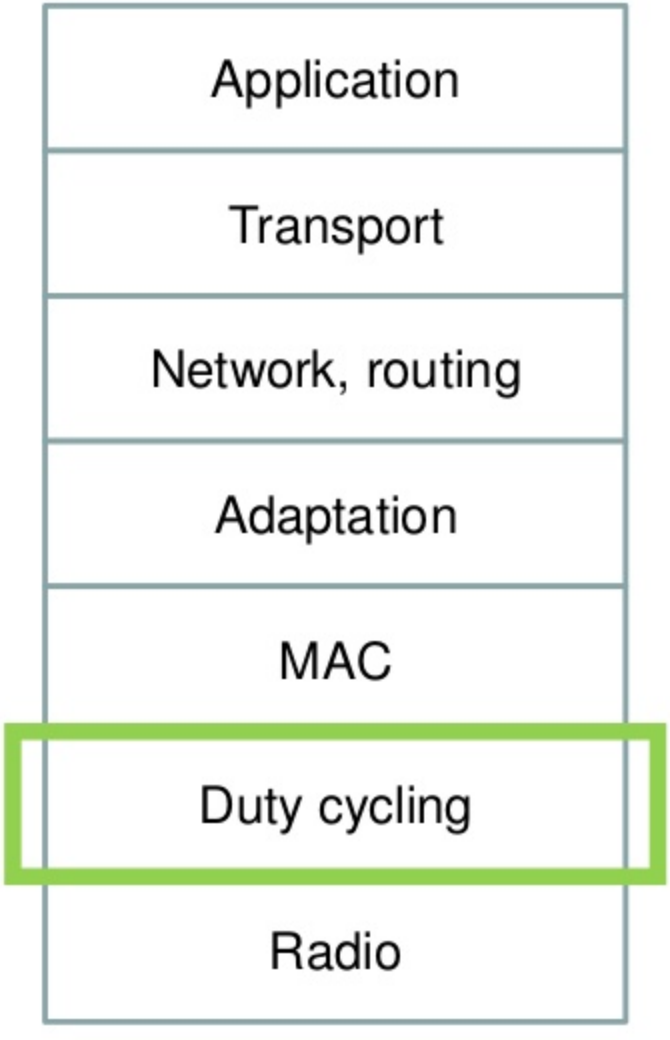
A packet through the netstack

- Step 2: the MAC output
  - Find the receiver in the neighbor list
  - Add the packet to the neighbor's output queue
  - If only packet on queue, call NETSTACK_RDC.send_list();



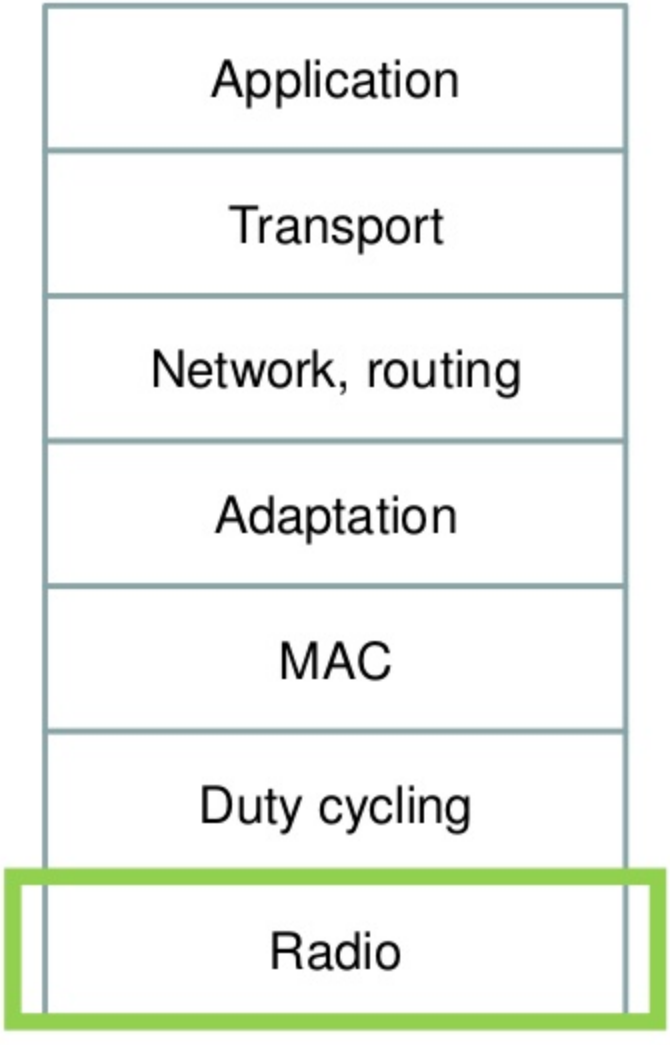
A packet through the netstack

- Step 3: the RDC output
  - Check for radio traffic
    - If so, return a collision to the MAC layer
  - Send all packets on list, wait for ACK between each



A packet through the netstack

- Step 4: the radio output
  - Push packet to radio hardware
  - Send packet



More



http://thingsquare.com