Freie Universität Berlin

RIOT's Preliminary Network Stack Design

Martine Lenders
Hauke Petersen

Contents

- Motivation
- Overall Architecture
- Packet Buffer
- Netreg
- Netdev
- Netapi
- Examples / Use Cases
- Open Topics

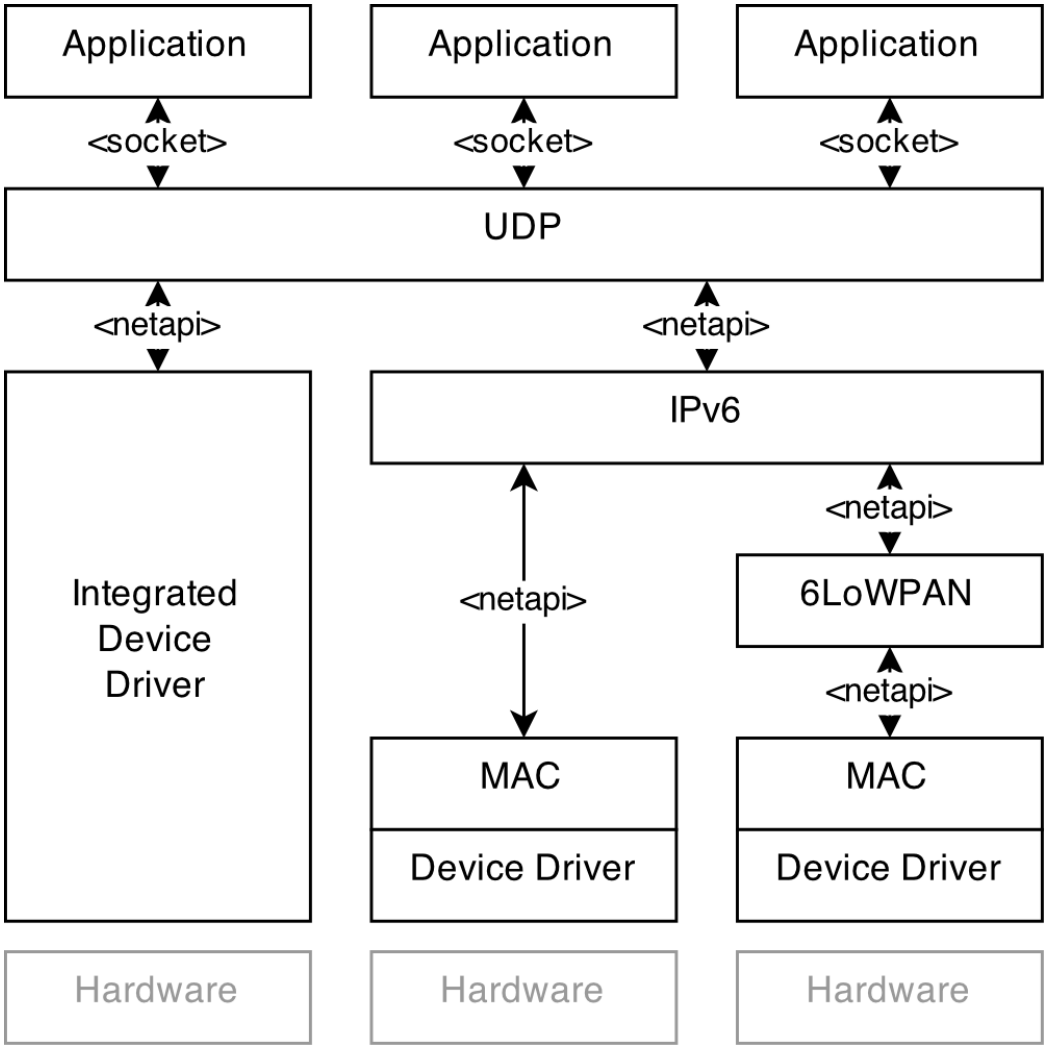
Motivation

- Problems with the current implementation:
  - Too big (buffers everywhere, multiple reimplementations of the same stuff)
  - Too inconsistant (every protocol has its own set of APIs)
  - Too monolithic
    - originally designed for just 6LoWPAN over cc110x
    - IEEE 802.15.4 support patched in with advent of at85rf231/cc2420 support
    - every new device type requires heavy patching in several layers
    - IPv6 without 6LoWPAN currently impossible
  - Transceiver API does not scale
    - (for every new device new #ifdef branch ⇒ >1000 loc for simple tasks)
  - Context (thread) of function calls not always clear
    - (ipv6_send_data() called from RPL, TCP/UDP, and Ipv6 thread itself)

Desing Principles

1) Modularity and extensibilty
2) Slim and well-defined interfaces
3) Memory efficiency (RAM and ROM)
4) Energy efficiency
5) Stability (→ testability per design, test-driven design)
6) Performance

Architecture



General Concepts

- Data and headers are stored in a central packet buffer
- Data is passed around the network stack by passing around pointers to elements in this buffer

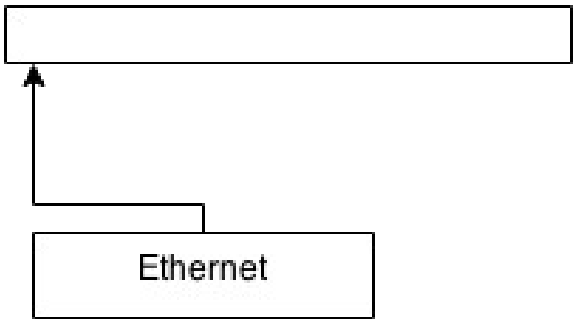
- Passing data up:
  - We always pass the complete packet (including all headers)
- Passing data down:
  - A module adds the header for the receiveing module before passing it on

Packet Buffer - Concept

- Goal:
  - Decrease size of .bss section / stack usage of network threads
  - Minimalize data movement between the layers
  - Minimalize data movement inside the buffer
  - Make overall used buffer size configurable at central location

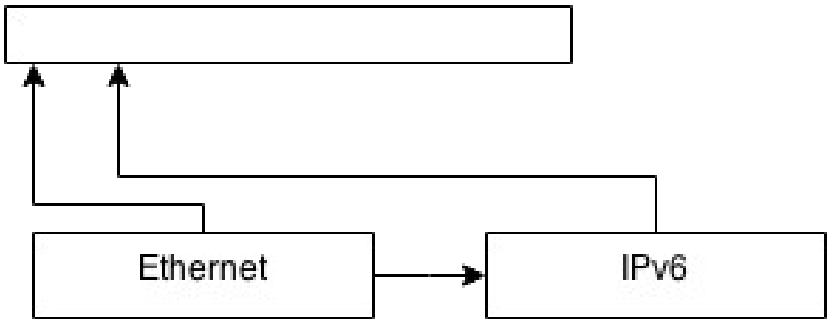
- Concept:
  - Centralized buffer (either static or dynamic, user's choice)
  - Common API for allocation in static buffer array or dynamic memory management
  - Packets are list of headers and payloads
  - Basic garbage collection (users)

Packet Buffer - Concept



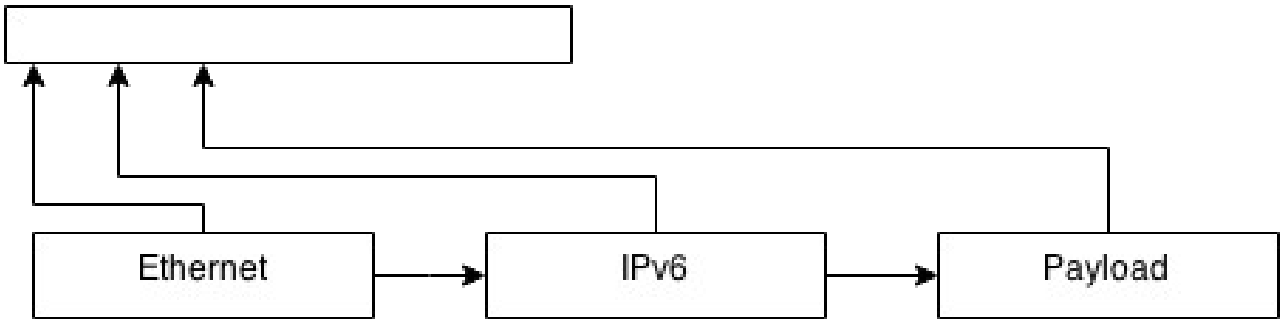
Receive packet

Packet Buffer - Concept



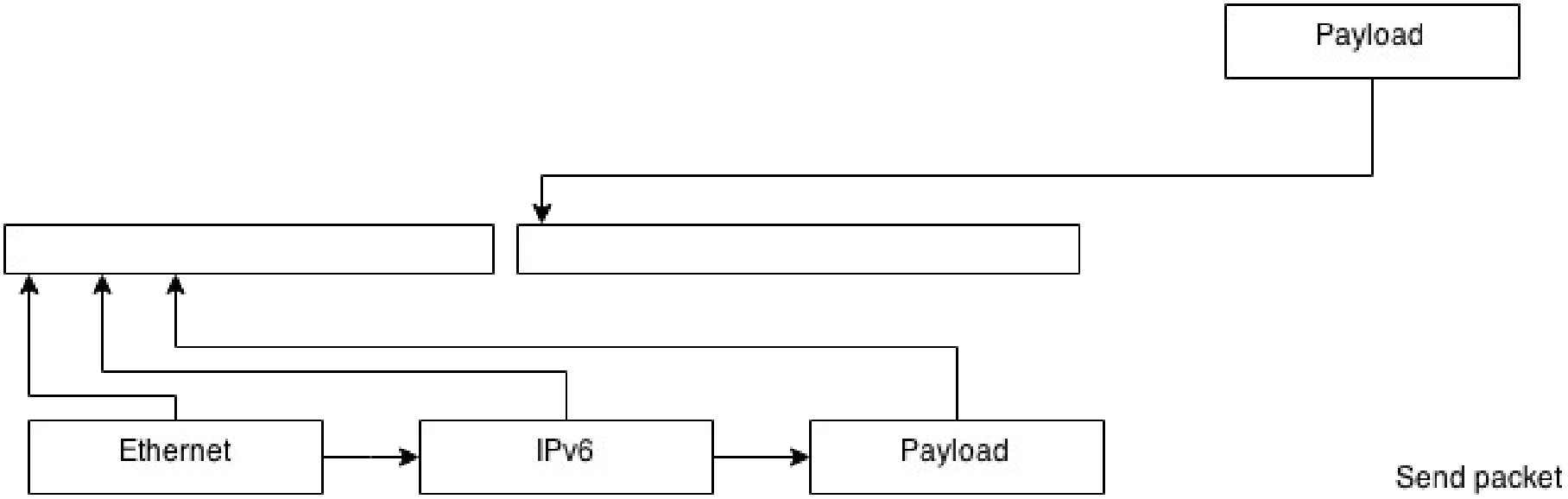
Receive packet

Packet Buffer - Concept

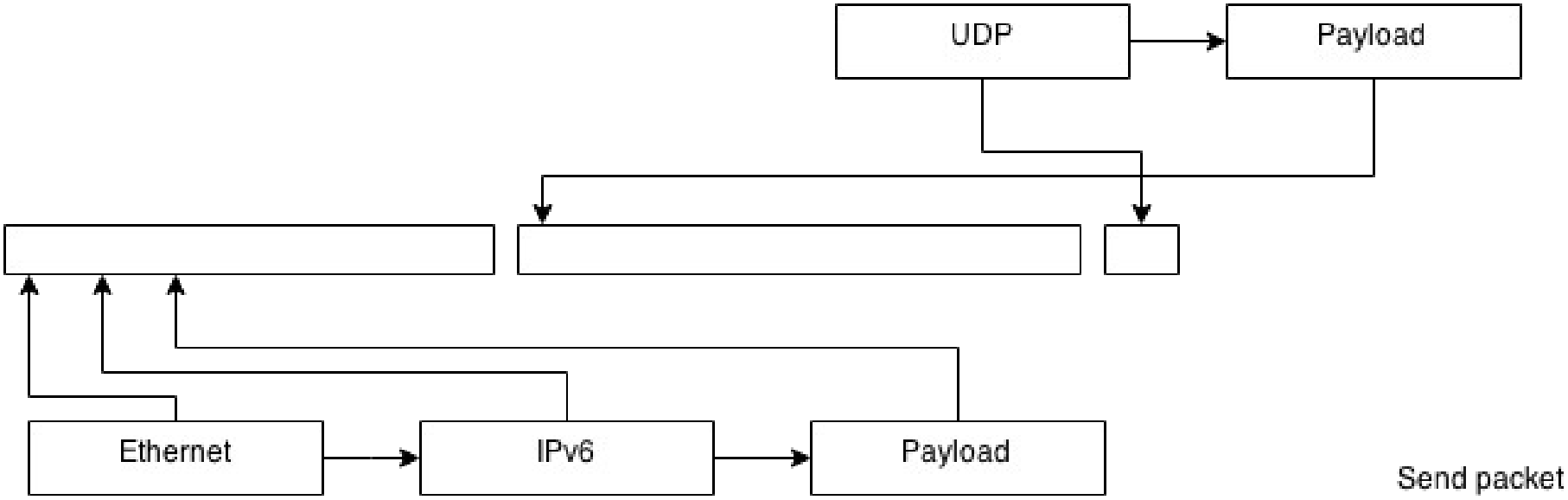


Receive packet

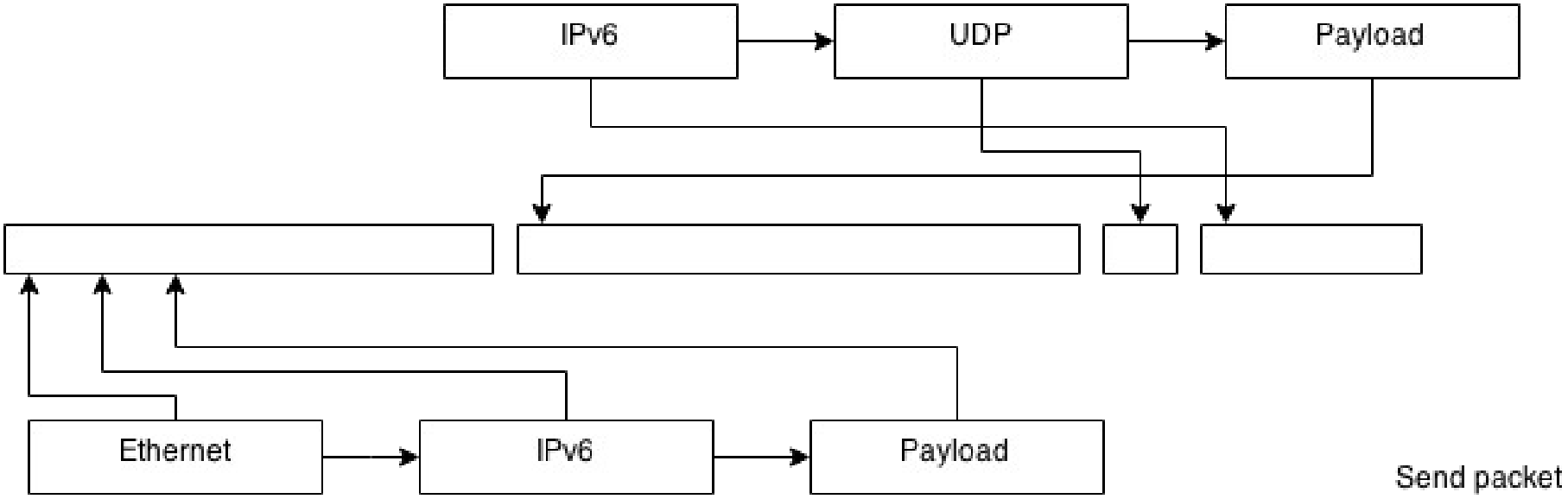
Packet Buffer - Concept



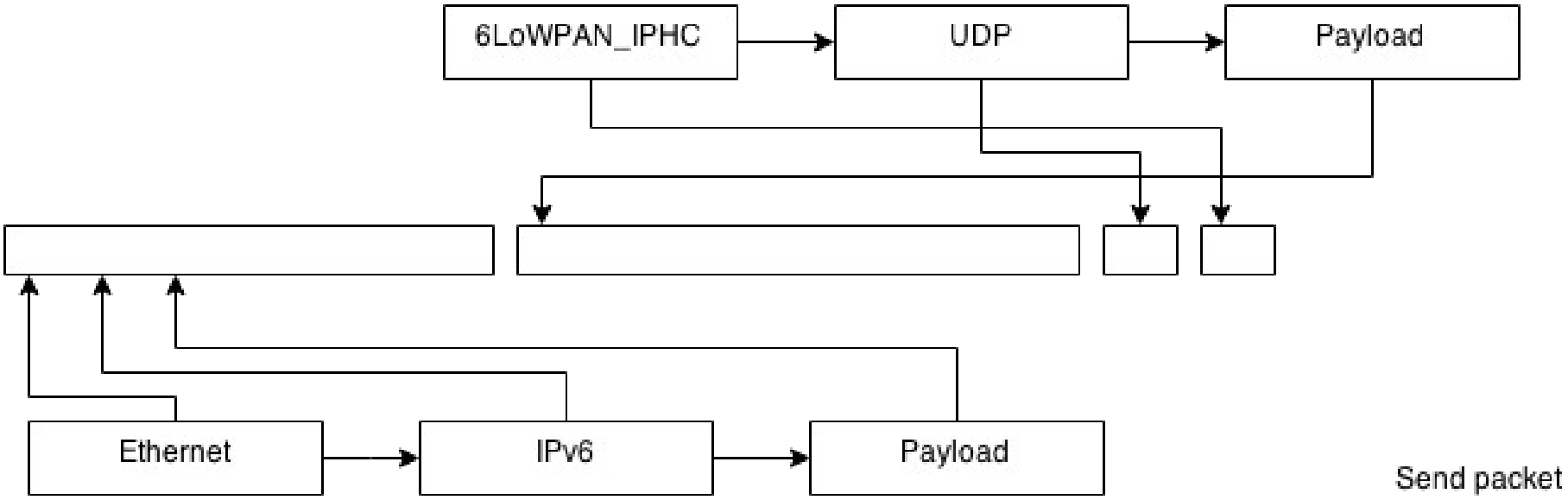
Packet Buffer - Concept



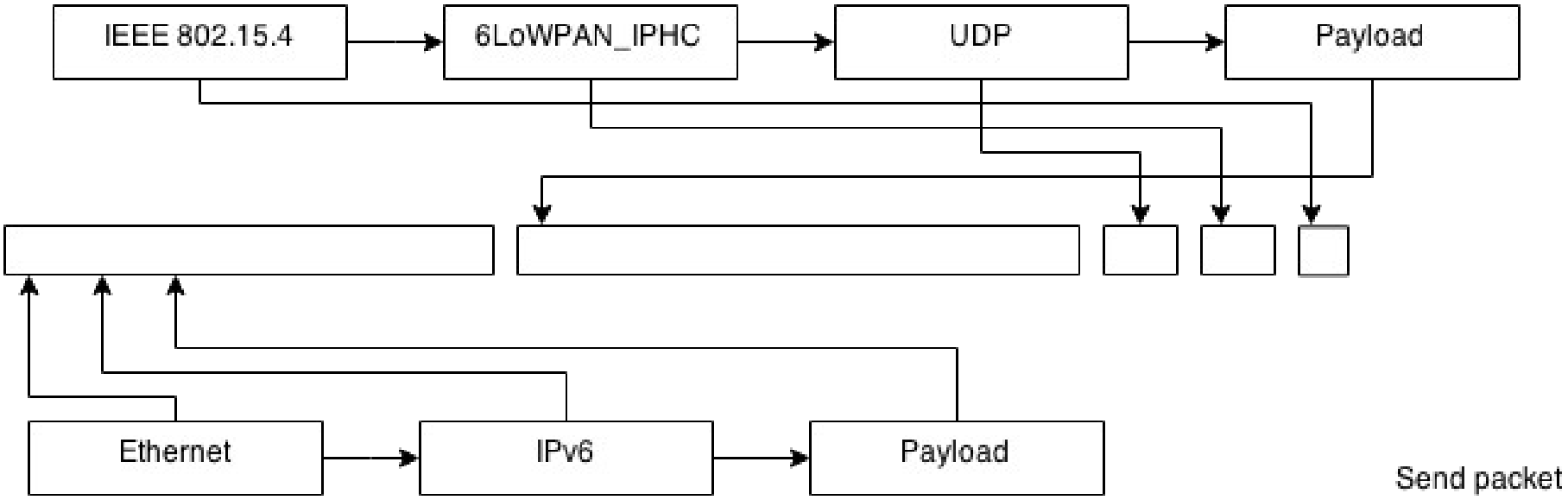
Packet Buffer - Concept



Packet Buffer - Concept



Packet Buffer - Concept



Packet Buffer - API

- Packet snip: **pktsnip_t**
  - next: **pktsnip_t *** // Next packet snip in packet
  - data: **void *** // pointer to data in buffer
  - size: **size_t** // Size of the data in buffer
  - type: **Integer** // Number identifying protocol type of data
  - users: **Integer** // Number of threads currently accessing packet snip

Packet Buffer - API

- Operations:
  - pktsnip_t ***pktbuf_add**(pktsnip_t *next, void *data, size_t size, netprot_t type)
    - allocates new packet in packet buffer
    - If size == 0: just return resulting pktsnip_t
    - If size != 0 and data == NULL and next == NULL: “malloc“ for result->data
    - If data != NULL and not in packet buffer: data will be copied into packet buffer
    - If data != NULL and in packet buffer and next != NULL and next->data == data: Header marked in data of next
      - next->data += size
  - void **pktbuf_hold**(pktsnip_t *pkt, uint8_t inc): Increment pkt->users atomically
  - void **pktbuf_release**(pktsnip_t *pkt):
    - Decrement pkt->users atomically and remove from pktbuf if pkt->users == 0
  - pktsnip_t ***pktbuf_start_write**(pktsnip_t *pkt):
    - Announce write operation
    - Duplicates packet in case of pkt->users > 1

Netreg

- Netreg is a global registry that connects the network stack modules based on their PIDs
- Netreg also keeps callback pointers for creating headers
- Number of interfaces and available protocols are set at compile time

- Example:
  - IP parses a packet and wants to hand it over to UDP
  - IP ask the netreg for all PIDs that are interested in UDP packets
  - IP sends the packet (pktsnip_t ptr) to each of these PIDs

Netreg: API

- netreg_register(netproto_t protocol, kernel_pid_t pid, create_header_cb)
- netreg_unregister(netproto_t protocol, kernel_pid_t pid)
- netreg_lookup(netreg_entry_t *entry, netproto_t protocol) : kernel_pid_t
- netreg_getnext(netreg_entry_t *entry) : kernel_pid_t

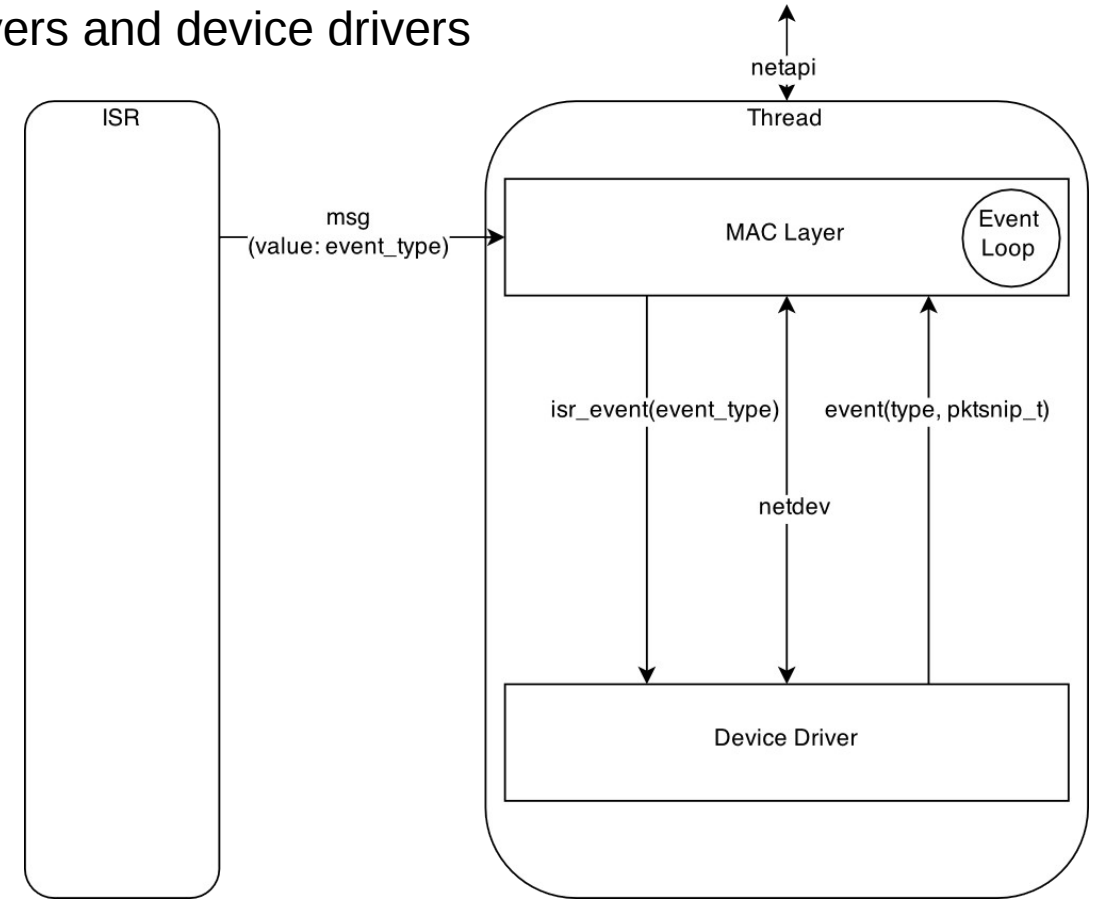
- … TBD

Netreg: Open for discussion

Is there a better way to connect network stack module?

Netdev

- Unified interface between device deriver and MAC layer
  → inter-link-layer interface
- Allows for exchangable MAC layers and device drivers



Netdev: API

- **netdev_t:**
  - driver: netdev_driver_t * // contains the driver's function interface
  - event_cb: netdev_event_cb_t // called by the driver to inform MAC layer
  - mac_pid: kernel_pid_t // the MAC layers PID (the driver's thread context)

- **netdev_event_cb_t**
  - event_cb(netdev_event_t type, void *arg) : void

- **netdev_driver_t:**
  - send_data(netdev_t *dev, pktsnip_t *pkt) : int
  - add_event_callback(netdev_t *dev, netdev_event_cb_t cb) : int
  - rem_event_callback(netdev_t *dev, netdev_event_cb_t cb) : int
  - get_option(netdev_t *dev, uint16_t scope, void *value, size_t value_len) : int
  - set_option(netdev_t *dev, uint16_t scope, void *value, size_t value_len) : int
  - isr_event(netdev_t *dev, uint16_t event_type) : void

Netdev: Device Descriptor

```
typedef struct {
    /* netdev interface */
    netdev_driver_t const * driver;
    netdev_event_cb_t event_cb;
    kernel_pid_t mac_pid;
    /* driver specific configuration */
    uint8_t rx_buf_next;                /**< pointer to free RX buffer */
    volatile uint8_t state;             /**< the current state of the device */
    uint8_t old_state;                  /**< saves the old state before sending
    uint16_t own_addr;                  /**< configured 16-bit RX address */
    uint16_t options;                   /**< bitfiels to save run-time options */
    nrf51prop_packet_t tx_buf;          /**< transmission buffer */
    nrf51prop_packet_t rx_buf[2];       /**< double buffered RX buffer */
    /* this would also include peripheral configuration (SPI, GPIO_INT, CS...) */
} nrf51prop_t;
```

Netdev: Operating modes

Different operating modes can be mapped onto this netdev:

- Promiscuous Mode:
  - set_option(dev, NETCONF_OPT_PROMISCUOUSMODE, 1)

- Preloading:
  - set_option(dev, NETCONF_OPT_PRELOADING, 1)
  - Sending data:
    - send_data(dev, pkt) ←this will preload the data (but not send)
    - set_option(dev, NETCONF_OPT_STATE, NETCONF_STATE_TX)

Netapi - Concept

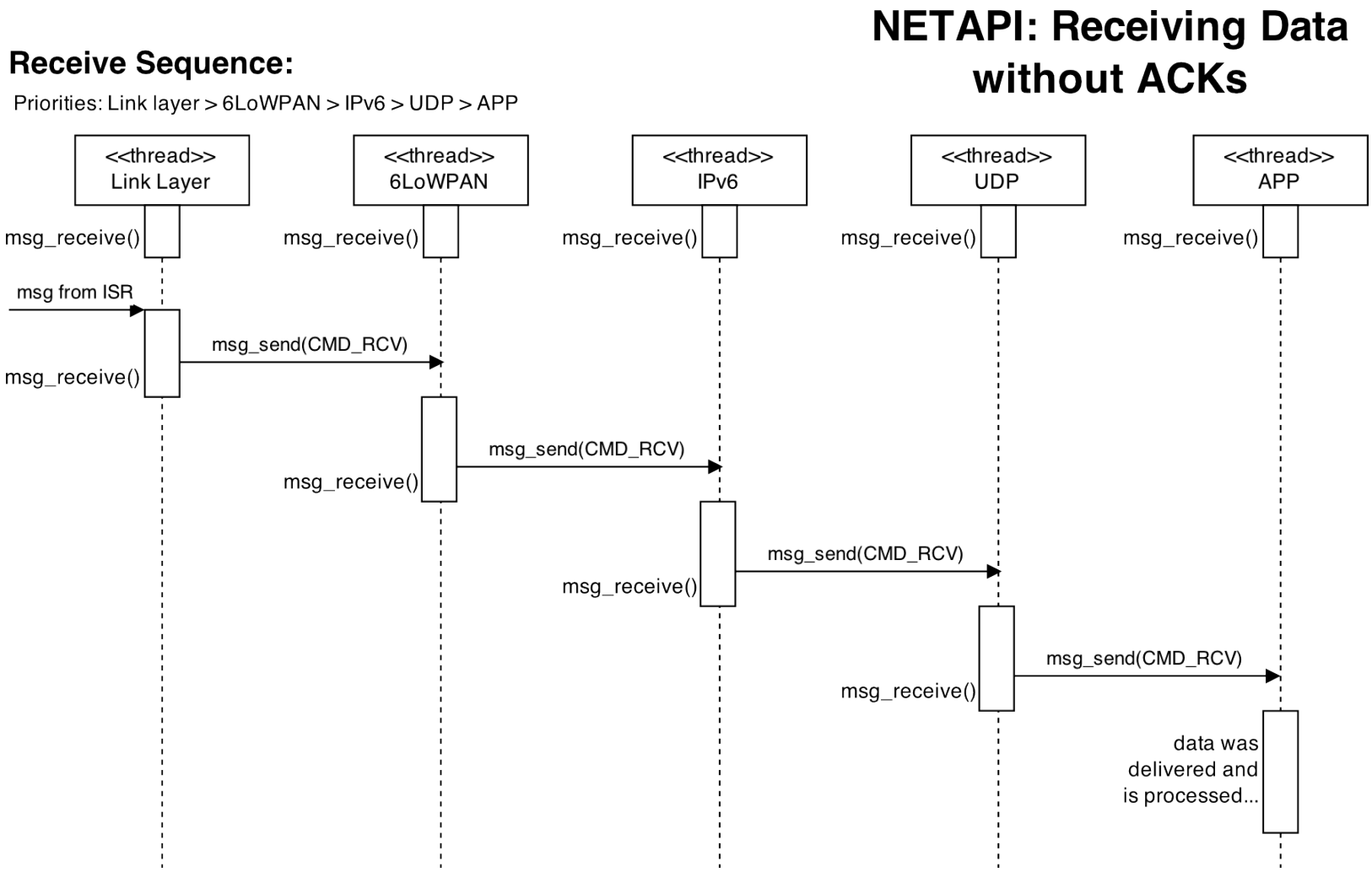
- Utilize IPC for sending/receiving of packets between layers
- Utilize IPC for option setting

Netapi - API

- 5 message types:
  - Send: msg.content.ptr on pktsnip_t of sending protocol, used with msg_send
  - Receive: msg.content.ptr on pktsnip_t of receiving protocol, used with msg_send
  - Get option: msg.content.ptr on netapi_opt_t; used with msg_send_receive
  - Set option: msg.content.ptr on netapi_opt_t; used with msg_send_receive
  - Acknowledgement: msg.content.value on result of get option or set option operation

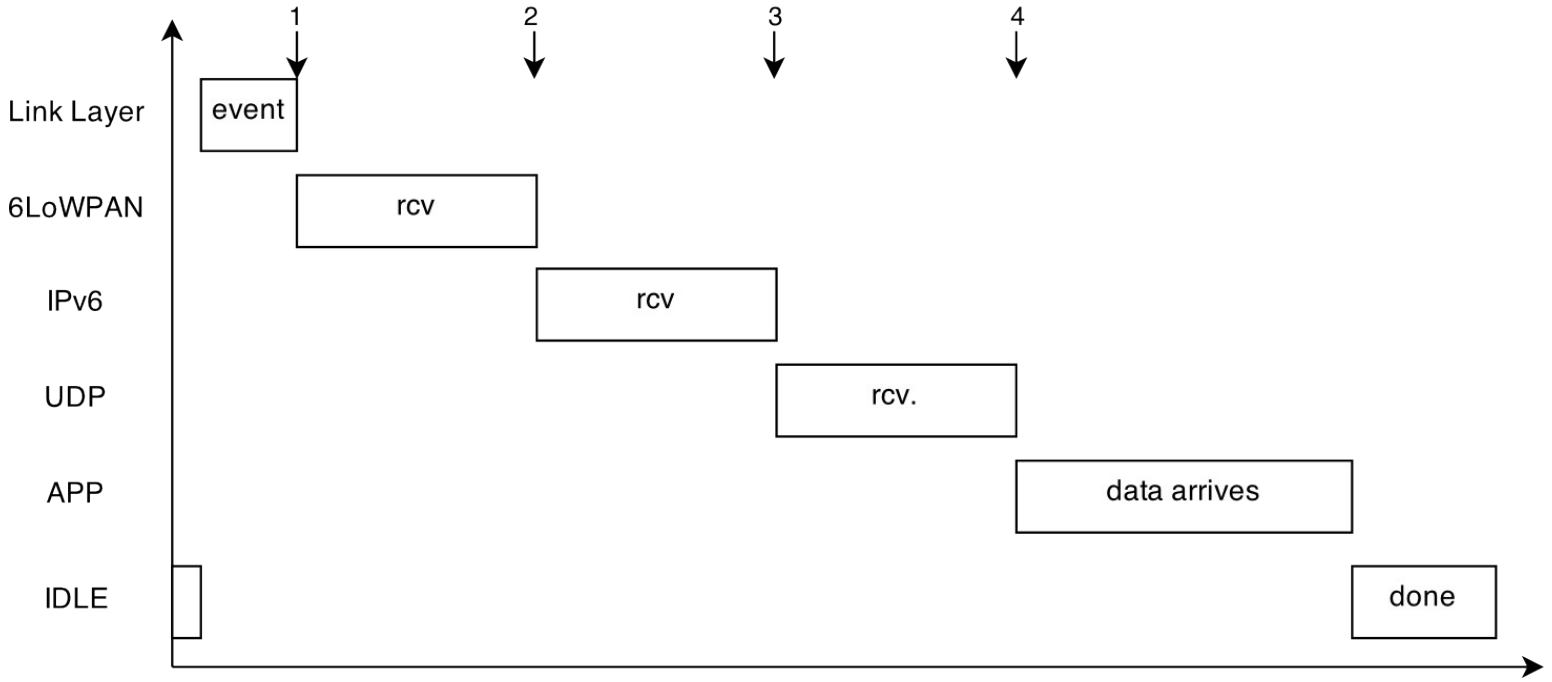
- **netapi_opt_t:**
  - Type: **Integer** // type of option. E.g. address, channel, etc
  - Param: **Integer** // optional parameter to identify possible internal interface/port
  - Value: **void***
  - Size: **size_t** // sizeof(typeof(value))

Use Cases: Netapi



Use Cases: Netapi

Context Switches:



Use Cases: Receiving UDP packet

Link Layer:

- Get message from interrupt
- pktbuf: allocate generic link layer header
- pktbuf: allocate space for data
- Set generik link layer header
- Copy payload from network device into pktbuf
- Get PIDs of interested modules from netreg
  - i.e. fixed by driver protocol as 6LoWPAN or by type as in Ethernet
- Pass pktsnip pointer up the stack (to IPv6 in this example)

Use Cases: Receiving UDP packet (cont)

IPv6:

- Get pointer to IPv6 header data
  - Found in the received pktsnip pointers next field
- Check if header is really IPv6 (e.g. by looking at the version field)
- Mark header as IPv6 or disregart packet
- Read IP destination address
- If address is me:
  - Read next header field
  - Get PID of target from netreg
  - Pass packet on (UDP in this example)
- Else if router:
  - As Forwarding table for next hop
  - Send packet to next hop
- Else:
  - Drop packet

Use Cases: Receiving UDP packet (cont)

UDP:

- Get pointer to UDP data
- Separate UDP header any payload
- Read destination PORT
- Lookup if socket is bound to this PORT
- Lookup PID for this socket
- Send Payload to socket

Socket:

- Get pointer to Payload data
- Copy payload data into application buffer
- Release packet

Open Topics

- ICMPv6
- Option Handling
  - IPv6 Extensions
  - NDP / ARP + Options
- FIB
- Routing
- Error Handling