Network Coding (NC)

CITHN2002 – Summer 2024

Prof. Dr.-Ing. Stephan Günther

Chair of Distributed Systems and Security
School of Computation, Information and Technology
Technical University of Munich

Chapter 2: libmoep

What is `libmoep`?

How is it implemented?

moep 80211 frame format

Blocking interfaces

Handling other file descriptors

`libmoepcommon`

- The Linux kernel's list
- Timeouts
- Working with `struct timespec`
- Logging and debugging

Chapter 2: libmoep

What is `libmoep`?

How is it implemented?

moep 80211 frame format

Blocking interfaces

Handling other file descriptors

`libmoepcommon`

What is `libmoep`?

`libmoep`[1] is a library written in C that allows to inject

- cooked IEEE 802.11 frames (native mode),
- frames based on a proprietary, extensible frame format (moep 802.11) to develop and evaluate custom link-layer protocols and
- various other frametypes, and
- it supports various interfaces (WLAN, Ethernet, TAP, Unix sockets).

[1] It has been primarily developed by Maurice Leclaire, a former staff member of the chair.

What is `libmoep`?

`libmoep`[1] is a library written in C that allows to inject

- cooked IEEE 802.11 frames (native mode),
- frames based on a proprietary, extensible frame format (moep 802.11) to develop and evaluate custom link-layer protocols and
- various other frametypes, and
- it supports various interfaces (WLAN, Ethernet, TAP, Unix sockets).

Why not opening raw sockets?

[1] It has been primarily developed by Maurice Leclaire, a former staff member of the chair.

What is libmoep?

libmoep[1] is a library written in C that allows to inject

- cooked IEEE 802.11 frames (native mode),
- frames based on a proprietary, extensible frame format (moep 802.11) to develop and evaluate custom link-layer protocols and
- various other frametypes, and
- it supports various interfaces (WLAN, Ethernet, TAP, Unix sockets).

Why not opening raw sockets? ...libmoep uses raw sockets but:

- it hides most of the complexity of
  - creating monitor mode interfaces,
  - setting interface parameters,
  - parsing radiotap headers, etc.
- and allows a convenient way to *pair* a monitor interface with a TAP interface.

[1] It has been primarily developed by Maurice Leclaire, a former staff member of the chair.

What is `libmoep`?

Example: `ptm`

The `ptm` (PTM stands for packet transfer module) is

- the most simple kind of module using `libmoep` to
- relay packets by
  - accepting IEEE 802.3 frames over a virtual Ethernet interface (tap0),
  - converting those frames to a custom format suitable for wireless transmission,
  - sending those frames over a monitor interface and
  - translating incoming frames from the monitor interface back to valid IEEE 802.3 frames.

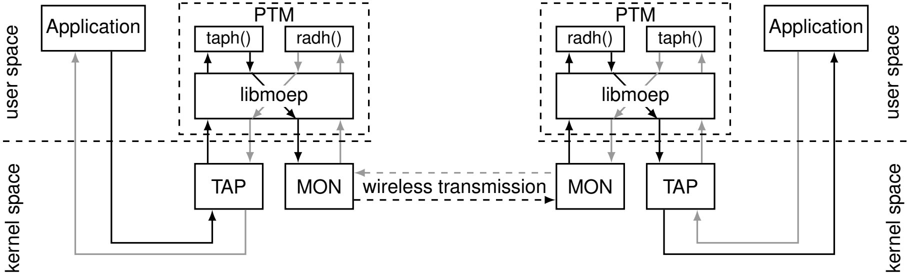


Figure 1: PTM block diagram

What is `libmoep`?

Example: `ptm`

The TAP interface presents itself like a physical Ethernet device, i.e.,

- it has a MAC address and
- can be assigned IP(v6) addresses:

```
tap0: <BROADCAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast ...
        link/ether 06:36:10:3e:a8:b0 brd ff:ff:ff:ff:ff:ff
        inet 10.0.0.1/24 brd 10.0.0.255 scope global tap0
           valid_lft forever preferred_lft forever
        inet6 fe80::436:10ff:fe3e:a8b0/64 scope link
```

What is libmoep?

Example: ptm

The TAP interface presents itself like a physical Ethernet device, i.e.,

- it has a MAC address and
- can be assigned IP(v6) addresses:

```
tap0: <BROADCAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast ...
        link/ether 06:36:10:3e:a8:b0 brd ff:ff:ff:ff:ff:ff
        inet 10.0.0.1/24 brd 10.0.0.255 scope global tap0
           valid_lft forever preferred_lft forever
        inet6 fe80::436:10ff:fe3e:a8b0/64 scope link
```

Advantages:

- Applications are completely unaware of the translation.
- It works with any kind of traffic (even ARP).
- We have any control about the radio interface we can ever have without writing custom device drivers.

How is it implemented?

1. Create tap and monitor devices:

```
if (!(tap = moep_dev_ieee8023_tap_open(args.addr, &args.ip, 24,
                        args.mtu + sizeof(struct ether_header)))) {
        fprintf(stderr, "ptm: error: %s\n", strerror(errno));
        return -1;
}

if (!(rad = moep_dev_moep80211_open(args.rad, args.freq,
                        MOEP80211_CHAN_WIDTH_20,
                        0, 0, args.mtu + radiotap_len(-1) +
                        sizeof(struct moep80211_hdr) +
                        sizeof(struct moep_hdr_pctrl)))) {
        fprintf(stderr, "ptm: error: %s\n", strerror(errno));
        moep_dev_close(tap);
        return -1;
}
```

2. Set `rx_handler` for both devices that will be used as callbacks upon frame arrival:

```
moep_dev_set_rx_handler(tap, taph);
moep_dev_set_rx_handler(rad, radh);
```

3. *Pair* both devices and turn control to `libmoep`:

```
moep_dev_pair(tap, rad);
moep_run(sigh, NULL);
```

How is it implemented?

- The call to `moep_run()` turns control to `libmoep`.
- The internal event loop is essentially a wrapper for `epoll`.
- Depending on which interface a frame is received, the appropriate handler is called:
  - If a frame arrives at the TAP interface, `taph()` is called and the received frame is passed to this handler.
  - The handler can translate the frame to a suitable format and schedule it for transmission on the radio interface.

How is it implemented?

- The call to `moep_run()` turns control to `libmoep`.
- The internal event loop is essentially a wrapper for `epoll`.
- Depending on which interface a frame is received, the appropriate handler is called:
  - If a frame arrives at the TAP interface, `taph()` is called and the received frame is passed to this handler.
  - The handler can translate the frame to a suitable format and schedule it for transmission on the radio interface.

Do we have to turn complete control over to `libmoep`?

- Of course not.
- There is `moep_wait()`, which works just like `epoll_wait()` but still supports `rx_handlers`.
- You can even configure `libmoep` to use another custom `epoll_wait()` compatible function internally.

However, you will probably never need that.

Chapter 2: libmoep

What is `libmoep`?

How is it implemented?

moep 80211 frame format

Blocking interfaces

Handling other file descriptors

`libmoepcommon`

There are two different ways to create radio interfaces:

- `moep_dev_ieee80211_open()`
  - Frames passed to the `rx_handler` will be ordinary IEEE 802.11 frames, including their link-layer headers.
  - The radiotap header will be a `moep80211_radiotap` since `ieee80211_radiotap` sucks.[1]
- `moep_dev_moep80211_open()`
  - Frames passed to the `rx_handler` will be in a custom format that is based on the generic IEEE 802.11 header for data frames.
  - The radiotap header is again `moep80211_radiotap`.

There are more functions to open / create TAP and Ethernet interfaces (and even Unix sockets).

In all cases, a frame is represented by the type `moep_frame_t`, which is a typedef of a pointer to a `struct moep_frame`.

- The members of this struct are an implementation detail and not accessible.
- You cannot modify a `moep_frame_t` directly, use the interface functions.

[1] The difference between `moep80211_radiotap` and `ieee80211_radiotap` is basicalle that the former one is fully expanded, i. e., all options allocate memory even if the present mask does not specify them.

moep 80211 frame format

Getting the headers of a `moep_frame_t`:

```
// Returns the radiotap header
struct moep80211_radiotap *moep_frame_radiotap(moep_frame_t frame);
// Returns the IEEE80211 header (generic format, your have to parse it)
struct ieee80211_hdr_gen *moep_frame_ieee80211_hdr(moep_frame_t frame);
// Returns the moep80211_hdr common to all our custom frames
struct moep80211_hdr *moep_frame_moep80211_hdr(moep_frame_t frame);
// Returns the IEEE802.3 header (in case of Ethernet frames)
struct ether_header *moep_frame_ieee8023_hdr(moep_frame_t frame);
```

Transmit a frame:

```
int moep_dev_tx(moep_dev_t dev, moep_frame_t frame);
```

If you want to convert a `moep_frame_t` to a buffer or create a `moep_frame_t` from a buffer, you can use the following functions:

```
int moep_frame_encode(moep_frame_t frame, u8 **buf, size_t buflen);
int moep_frame_decode(moep_frame_t frame, u8 *buf, size_t buflen);
```

If you want to transmit a manually created frame from a buffer, you may also use:

```
int moep_dev_tx_raw(moep_dev_t dev, u8 *buf, size_t buflen);
```

moep 80211 frame format

The generic moep 80211 header

When not operating in native mode, **all** radio frames will have this common header:

```
struct moep80211_hdr {
        u16 frame_control;
        u16 duration_id;
        u8 ra[IEEE80211_ALEN];
        u8 ta[IEEE80211_ALEN];
        u32 disc;
        u16 txseq;
        u16 seq_ctrl; /* unused/reserverd */
} __attribute__((packed));
```

moep 80211 frame format

The generic moep 80211 header

When not operating in native mode, **all** radio frames will have this common header:

```
struct moep80211_hdr {
        u16 frame_control;
        u16 duration_id;
        u8 ra[IEEE80211_ALEN];
        u8 ta[IEEE80211_ALEN];
        u32 disc;
        u16 txseq;
        u16 seq_ctrl; /* unused/reserverd */
} __attribute__((packed));
```

- `frame_control` has the same meaning as for ordinary IEEE 802.11 frames.
- We set it to `FTYPE_DATA | STYPE_DATA` for **all** of our frames to avoid unexpected behavior of hardware.

moep 80211 frame format

The generic moep 80211 header

When not operating in native mode, **all** radio frames will have this common header:

```
struct moep80211_hdr {
        u16 frame_control;
        u16 duration_id;
        u8 ra[IEEE80211_ALEN];
        u8 ta[IEEE80211_ALEN];
        u32 disc;
        u16 txseq;
        u16 seq_ctrl; /* unused/reserverd */
} __attribute__((packed));
```

- `duration_id` may be interpreted by other STAs.
- We set it to zero for now.

moep 80211 frame format

The generic moep 80211 header

When not operating in native mode, **all** radio frames will have this common header:

```
struct moep80211_hdr {
        u16 frame_control;
        u16 duration_id;
        u8 ra[IEEE80211_ALEN];
        u8 ta[IEEE80211_ALEN];
        u32 disc;
        u16 txseq;
        u16 seq_ctrl; /* unused/reserverd */
} __attribute__((packed));
```

- `ra` is the 6 B receiver address of this frame.
- If we exploit the wireless broadcast advantage, we set it to the MAC broadcast address.

moep 80211 frame format

The generic moep 80211 header

When not operating in native mode, **all** radio frames will have this common header:

```
struct moep80211_hdr {
        u16 frame_control;
        u16 duration_id;
        u8 ra[IEEE80211_ALEN];
        u8 ta[IEEE80211_ALEN];
        u32 disc;
        u16 txseq;
        u16 seq_ctrl; /* unused/reserverd */
} __attribute__((packed));
```

- `ta` is the 6 B transmitter address of this frame.
- This is **not** the MAC of our wireless interface but of the tap interface. Think about it!

moep 80211 frame format

The generic moep 80211 header

When not operating in native mode, **all** radio frames will have this common header:

```
struct moep80211_hdr {
        u16 frame_control;
        u16 duration_id;
        u8 ra[IEEE80211_ALEN];
        u8 ta[IEEE80211_ALEN];
        u32 disc;
        u16 txseq;
        u16 seq_ctrl; /* unused/reserverd */
} __attribute__((packed));
```

- `disc` is a 4 B field that we call frame discriminator.
- In IEEE 802.11 data frames this would be the third MAC address.
- We choose a value that should be invalid as MAC address.
- This way we can differentiate our own frames from normal IEEE 802.11 traffic.

moep 80211 frame format

The generic moep 80211 header

When not operating in native mode, **all** radio frames will have this common header:

```c
struct moep80211_hdr {
        u16 frame_control;
        u16 duration_id;
        u8 ra[IEEE80211_ALEN];
        u8 ta[IEEE80211_ALEN];
        u32 disc;
        u16 txseq;
        u16 seq_ctrl; /* unused/reserverd */
} __attribute__((packed));
```

- `txseq` are the latter 2 B of the third MAC address in IEEE 802.11 data frames.
- We use it as per-node TX sequence number, e.g. to estimate erasure probabilities.

moep 80211 frame format

The generic moep 80211 header

When not operating in native mode, **all** radio frames will have this common header:

```
struct moep80211_hdr {
        u16 frame_control;
        u16 duration_id;
        u8 ra[IEEE80211_ALEN];
        u8 ta[IEEE80211_ALEN];
        u32 disc;
        u16 txseq;
        u16 seq_ctrl; /* unused/reserverd */
} __attribute__((packed));
```

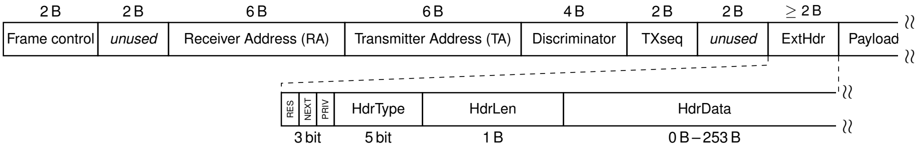
- `seq_ctrl` is fragment number / sequence number of the normal IEEE 802.11 data frame header.
- Problem with this field is that the NIC's driver may play with it.
- It is safer to set it to zero and to ignore it on reception.

moep 80211 frame format

Extension headers

We use extension headers to resemble different frame types, e. g. the packet control header:

- After the `moep_hdr` at **least one** extension header must follow.
- Bit 7 in the extension header's type field indicates whether another extension header follows.
- Type and length field precisely specify the extension header, and allow anyone to skip unknown extension headers.




```
struct moep_hdr_pctrl {
        struct moep_hdr_ext hdr;
        u16 type;  // corresponding to the Ethertype
        u16 len;   // explicit length of the frame's payload
} __attribute__((packed));

struct moep_hdr_ext {
        u8 type;   // type of the extension header, e.g. MOEP_HDR_PCTRL
        u8 len; // total length of the extension header
} __attribute__((packed));
```

moep 80211 frame format

How to add extension headers?

- Extension headers are part of the `l2_header` in the private `struct moep_frame`.
- How exactly extension headers are stored within a typedeffed `moep_frame_t` is not your business.

moep 80211 frame format

How to add extension headers?

- Extension headers are part of the `l2_header` in the private `struct moep_frame`.
- How exactly extension headers are stored within a typedeffed `moep_frame_t` is not your business.

Just let `libmoep` do it for you:

- `moep_frame_add_moep_hdr_ext()`
  Add a new extension header to an existing frame.
- `moep_frame_set_moep_hdr_ext()`
  Replace an existing extension header by a new one
- `moep_frame_del_moep_hdr_ext()`
  Delete an extension header.
- `moep_frame_moep_hdr_ext()`
  Get a pointer to a specific extension header (or `NULL` if it does not exist).

Blocking interfaces

As handlers must never perform blocking operations, we have to deal with the possibility that

- we read a frame from the TAP interface (`taph()`) but
- cannot write it to the monitor interface.

(The reverse way may occur in theory but is of no practical interest.)

Possible solutions:

1. Discard the packet
2. Buffer the packet
3. Do not read from the TAP interface in the first place, eventually leading to frame drops in the kernel

If we use random linear network coding, we have buffers and thus

- read from the TAP interface until our internal buffers are filled and
- block the TAP interface until we have free space again.

Blocking interfaces

`libmoep` offers the possibility to “block” and “unblock” an interface with `moep_dev_set_rx_status()`.

Normally, an interface should be “blocked” (receiving) in case another interface is blocking transmissions.
This event is signalled by `libmoep` through a callback handler, which can be installed with
`moep_dev_set_tx_status_callback()`.

In the `ptm` example this combination is automatically installed with `moep_dev_pair()`.
In more complex scenarios involving buffers, we have to write our own callbacks handling the buffer status.

Note: This does not guarantee that no frames are dropped.

- The TAP interface buffers frames on its own, and will eventually drop frames under load.
- Linux does not guarantee that frames scheduled for transmission are indeed transmitted.

However: Once a frame is read using `libmoep`, we can guarantee that it reaches its destination. But we have to ensure it ourselves.

Handling other file descriptors

Example: tranmit beacon frames in regular time intervals

```
if ((bcn_timer = timerfd_create(CLOCK_MONOTONIC, TFD_NONBLOCK)) < 0) {
        fprintf(stderr, "ptm: error: %s\n", strerror(errno));
        return -1;
}

interval.it_interval.tv_sec = args.beacon / 1000;
interval.it_interval.tv_nsec = (args.beacon % 1000) * 1000000;
interval.it_value.tv_sec = interval.it_interval.tv_sec;
interval.it_value.tv_nsec = interval.it_interval.tv_nsec;
if (timerfd_settime(bcn_timer, 0, &interval, NULL)) {
        fprintf(stderr, "ptm: error: %s\n", strerror(errno));
        close(bcn_timer);
        return -1;
}

if (!(bcn_callback = moep_callback_create(bcn_timer, send_beacon, NULL, EPOLLIN))) {
        fprintf(stderr, "ptm: error: %s\n", strerror(errno));
        close(bcn_timer);
        return -1;
}
```

- Create a `timer_fd` (file descriptor) bound to a monotonic clock (nonsettable monotonically increasing clock that measures time from some unspecified point, not affected by leap seconds)
- Prepare a `struct timespec` (two 64 bit integers) representing the interval
- Set the timer’s interval
- Register the the timer and its callback in `libmoep`

What is `libmoep`?

How is it implemented?

moep 80211 frame format

Blocking interfaces

Handling other file descriptors

`libmoepcommon`

- The Linux kernel's list
- Timeouts
- Working with `struct timespec`
- Logging and debugging

libmoepcommon

`libmoepcommon` is not part of `libmoep`, but a separate header-only library for common tasks:

- Modified version of the Linux kernel's list implementation
- Implementation of timeouts using event file descriptors
- Macros to calculate sum / difference between `struct timespec` instances
- Various small functions, e. g. to compare / test MAC addresses, hexdumps, logging, common math operations, etc.

It is part of the network coding module (NCM):

```
.
|-- benchmark.h
|-- list.h
|-- timeout.h
|-- types.h
|-- util
|   |-- alignment.h
|   |-- assertion.h
|   |-- hexdump.h
|   |-- log.h
|   |-- mac.h
|   |-- maths.h
|   |-- timespec.h
|-- util.h
```

The Linux kernel's list

`list.h` contains a (slightly modified) version of the Linux kernel's list.

- Your list elements are structs:

```c
struct neighbor {
    struct list_head list;
    u8 hwaddr[IEEE80211_ALEN];
};
```

- Create a new list and a new element:

```c
LIST_HEAD(nblist);
struct neighbor *nb = calloc(1, sizeof(struct neighbor));
list_add(&nb->list, &nblist);
```

- Search for an element in the list:

```c
struct neighbor *cur;
list_for_each_entry(cur, &nblist, list) {
    if (0 == memcmp(cur->hwaddr, hwaddr, IEEE80211_ALEN))
        return cur;
}
return NULL;
```

- Remove a given element from the list:

```
list_del(&nb->list);
free(nb);
```

- Search for an element that shall be removed or remove multiple elements:

```
struct neighbor *cur, *tmp;
list_for_each_entry_safe(cur, tmp, &nblist, list) {
    if (is_to_be_removed(cur)) {
        list_del(&cur->list);
        free(cur);
    }
}
```

Warning: removing a list element while iterating with `list_for_each_entry` invalidates list pointers. Use `list_for_each_entry_safe` instead.

Timeouts

`timeout.h` allows you to register a callback that is executed when the timeout times out.

- A timeout is internally represented by:

```
struct timeout {
    timer_t timerid;        // internal timer id
    struct sigevent sevp;   // eventfd for notification
    timeout_cb_t cb;        // callback when the timeout times out
    void *data;             // private data for the callback
};
```

- The callback must be given as function pointer of the following type:

```
typedef int (*timeout_cb_t)(timeout_t, u32, void *);
```

- Create a new timeout:

```
if (0 > timeout_create(CLOCK_MONOTONIC, &logt, state_log_cb, NULL))
    DIE("timeout_create() failed: %s", strerror(errno));
timeout_settime(logt, 0, timeout_msec(LOG_INTERVAL,LOG_INTERVAL));
```

- To make sure timeouts are signalled, you need to pass a signal handler to `moep_run`.
- Inside that handler you have to handle `SIGRTMIN`.
- `timeout_exec((void *)siginfo.ssi_ptr, siginfo.ssi_overrun);`

Looks complicated? Maybe, but we can

- create arbitrary timeouts (both one-shot and interva-based),
- perform (almost) arbitrary tasks in the timeout's callback,
- and even have private data for a timeout readily available.

And it is much easier than creating and setting a `timer_fd` and registering the callback manually.

The behaviour of `timeout_settime()` differs depending on flags:

- With all flags cleared and
  - `NULL` as new timeout value the timeout is cleared, or
  - unconditionally set to the new timeout value.
- If `TIMEOUT_FLAG_SHORTEN` is given, the new value takes effect only if it is smaller than the remaining time.
- If `TIMEOUT_FLAG_INACTIVE` is given, the new value takes effect only if the timeout is currently inactive.

Note: Timeouts always use relative time values, i. e., the time to the next event is given, not the absolute point in time when the event should be triggered. We must keep that in mind since a timeout may not be handled immediately (there is some processing time). Therefore, we might end up with a clock skew when repeating time intervals.

Working with `struct timespec`

Time values are represented by:

```
struct timespec {
    long tv_sec;    /* seconds */
    long tv_nsec;   /* nanoseconds */
};
```

Adding or subtracting time values is extremely prown to errors. Instead, you may use:

```
/* Add a and b, storing result in a */
timespecadd(&a, &b);

/* Subtract a and b, storing result in a */
timespecsub(&a, &b);

/* Get maxmimum of a and b, storing result in c */
timespecmax(&c, &a, &b);

/* Create timespec of x milliseconds */
timespecmset(&a, x);

/* Create timespec of x microseconds */
timespecuset(&a, x);
```

Logging and debugging

There are two macros for logging and debugging:

```
LOG(loglevel, const char *format, ...);
```

- Writes the format string to `stderr`, including filename and line number.
- Only messages whose log level is larger or equal to loglevel are printed, i. e., specify verbosity at compile time.

```
DIE(const char *format, ...);
```

- Primarly used as assertion.
- Prints the specified format string to `stderr`, including filename and line number.
- Immediately terminates the application.

Logging can be redirected to syslog if `MOEP80211_LOG_USE_SYSLOG` is set.