An MP-capable network stack for DragonFlyBSD with minimal use of locks

Aggelos Economopoulos
<aoiko@cc.ece.ntua.gr>

What is DragonFly all about again?

- SSI clustering (we'll do it next weekend, promise!)

What is DragonFly all about again?

- SSI clustering (we'll do it next weekend, promise!)
- HAMMER

What is DragonFly all about again?

- SSI clustering (we'll do it next weekend, promise!)
- HAMMER
- Vkernel

What is DragonFly all about again?

- SSI clustering (we'll do it next weekend, promise!)
- HAMMER
- Vkernel
- MP-scaling (Ah. Now we're getting somewhere)

Multiprocessor Support

- One CPU, one process in the kernel -> Many CPUs, one process in the kernel

Multiprocessor Support

- One CPU, one process in the kernel -> Many CPUs, one process in the kernel
- No scalability except for number crunching

Multiprocessor Support

- One CPU, one process in the kernel -> Many CPUs, one process in the kernel
- No scalability except for number crunching
- The traditional next step: breaking up the locks
- And again and again

Fine-grained locking

- Proven effective *many* times in the past
- Performs very well for current MP systems
  - –

Fine-grained locking

- Proven effective *many* times in the past
- Performs very well for current MP systems
- But:
  - Hard to ensure correctness
  -

Fine-grained locking

- Proven effective *many* times in the past
- Performs very well for current MP systems
- But:
  - Hard to ensure correctness
  - Massive locking is pure overhead on small MP
  -

Disclaimer

I am not now, nor have I ever been a network guru

Organization of the DragonFlyBSD network stack

The BSD Roots

What did we start with?

- BSD net stack
  - Reasonably fast
  - Reliable
  - Well documented (TCP/IP Illustrated Vol.2 is still useful as a reference)

What did we start with?

- BSD net stack
  - Reasonably fast
  - Reliable
  - Well documented (TCP/IP Illustrated Vol.2 is still useful as a reference)
  - 20 years of history (and it shows!)

tcp_input()

- Back in Net/2: 1100 lines

tcp_input()

- Back in Net/2: 1100 lines
- DragonFly 1.0: 1900+ lines

tcp_input()

- Back in Net/2: 1100 lines
- DragonFly 1.0: 1900+ lines
-
- Even worse, ranks second in cyclomatic complexity in the whole kernel

tcp_input()

- Back in Net/2: 1100 lines
- DragonFly 1.0: 1900+ lines
-
- Even worse, ranks second in cyclomatic complexity in the whole kernel
-
- You don’t know how bad the situation is until you try to change something

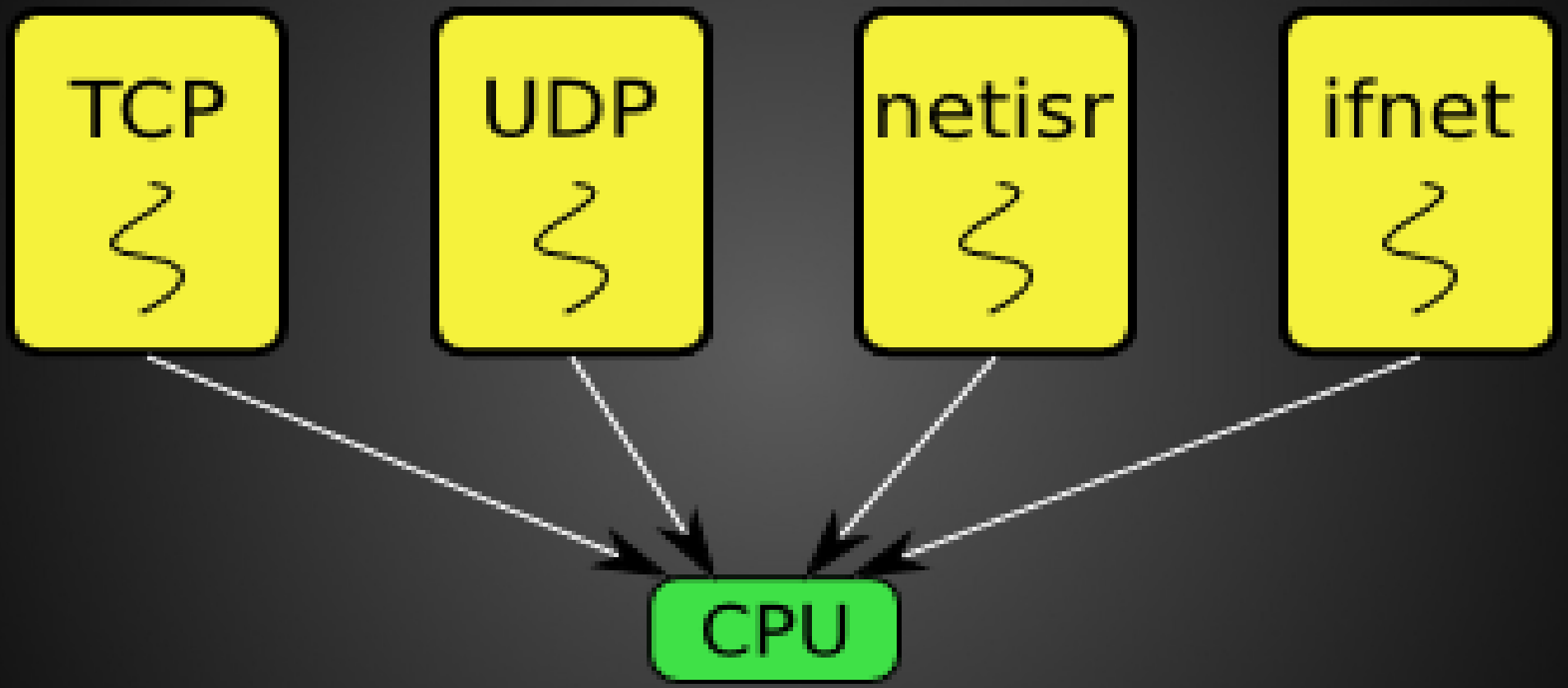
Mbufs

- There is NO abstraction
- People typically open-code things instead of using the API
- For all intents and purposes the mbuf structure is set in stone

Organization of the DragonFlyBSD network stack

Changes made in DragonFly

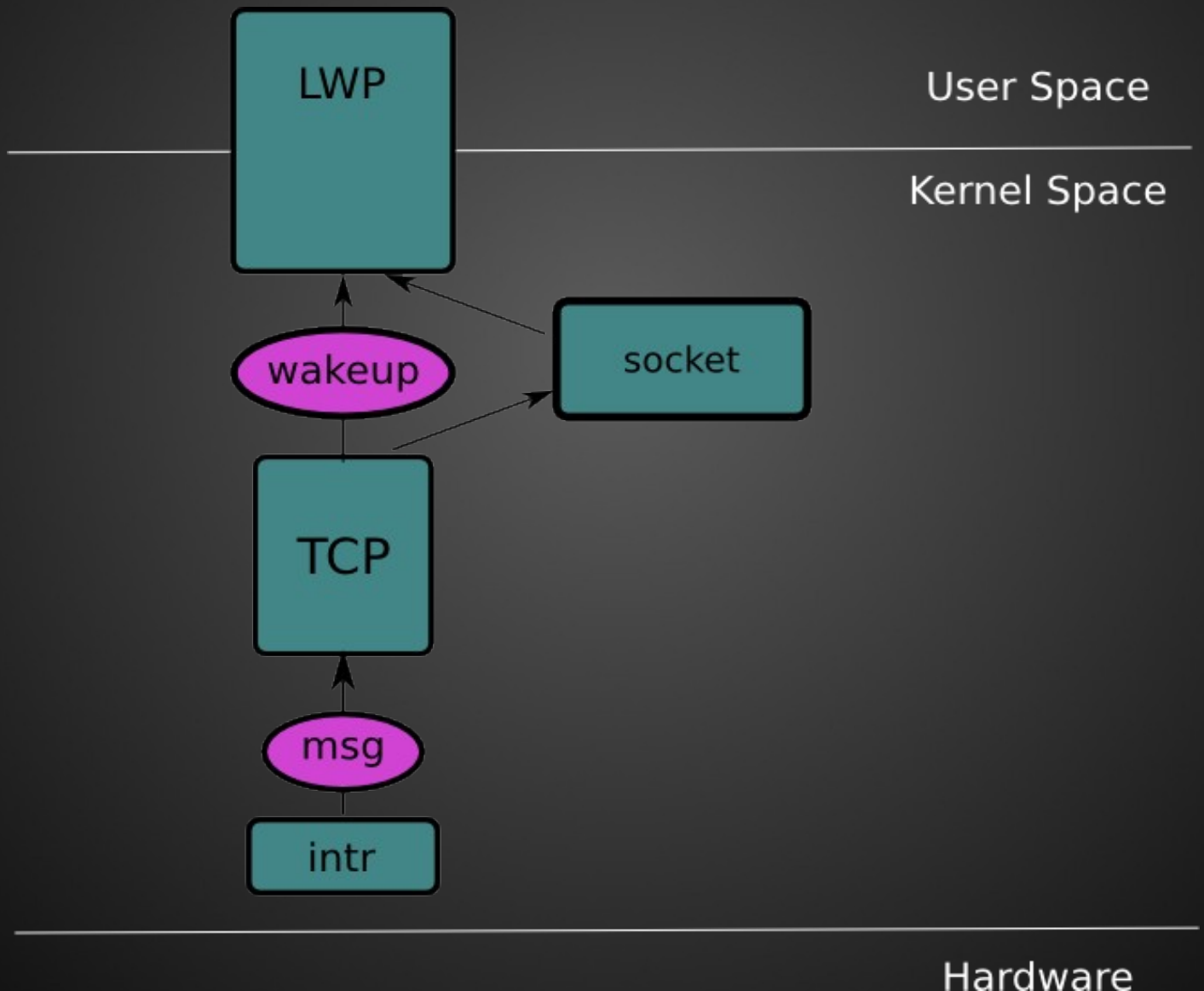
Meet the threads



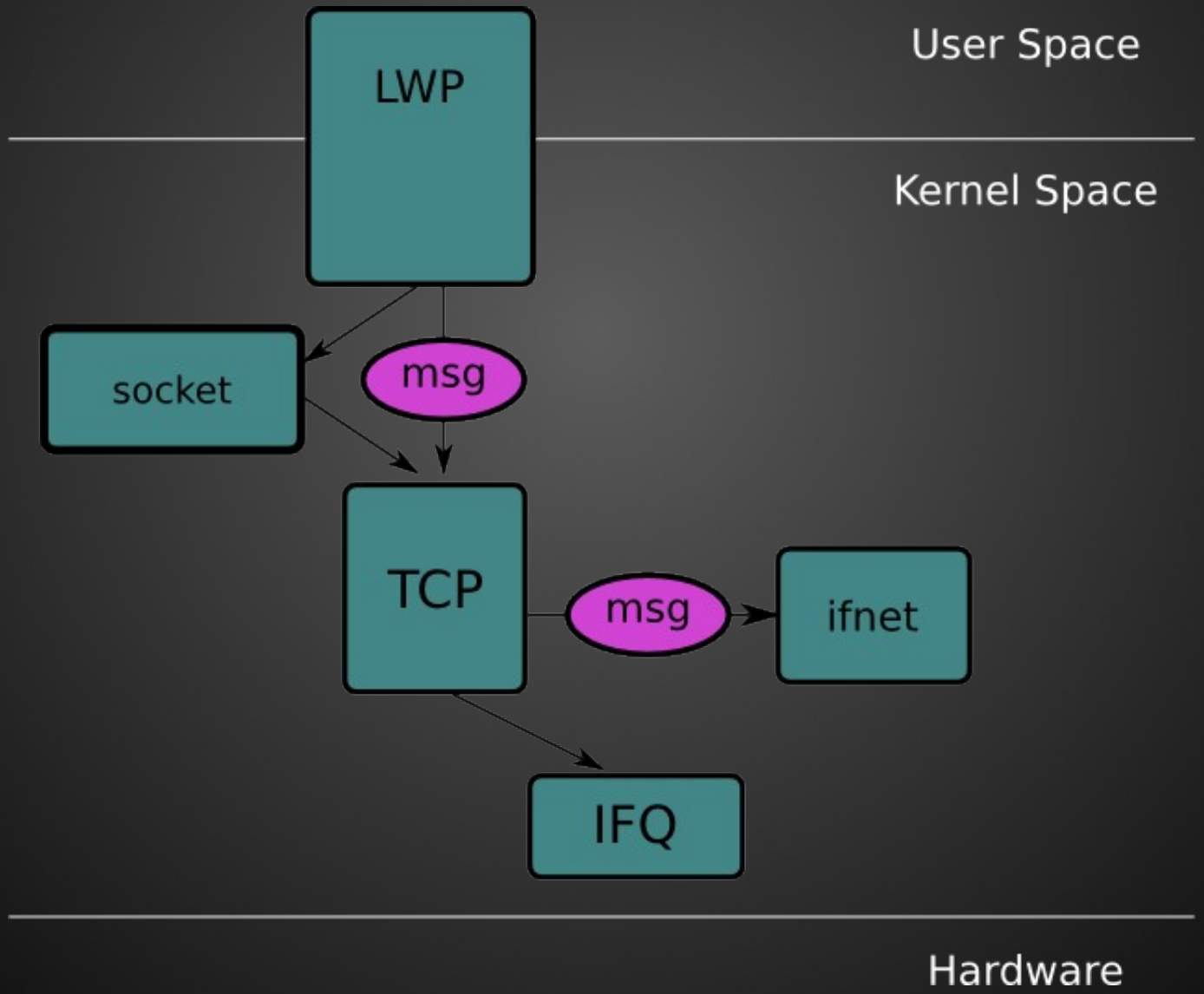
Communicating with the protocol threads

- abort
- accept
- attach
- bind
- connect
- connect2
- control
- detach
- disconnect
- listen
- peeraddr
- recvd
- recvoob
- send
- etc

Receive Path

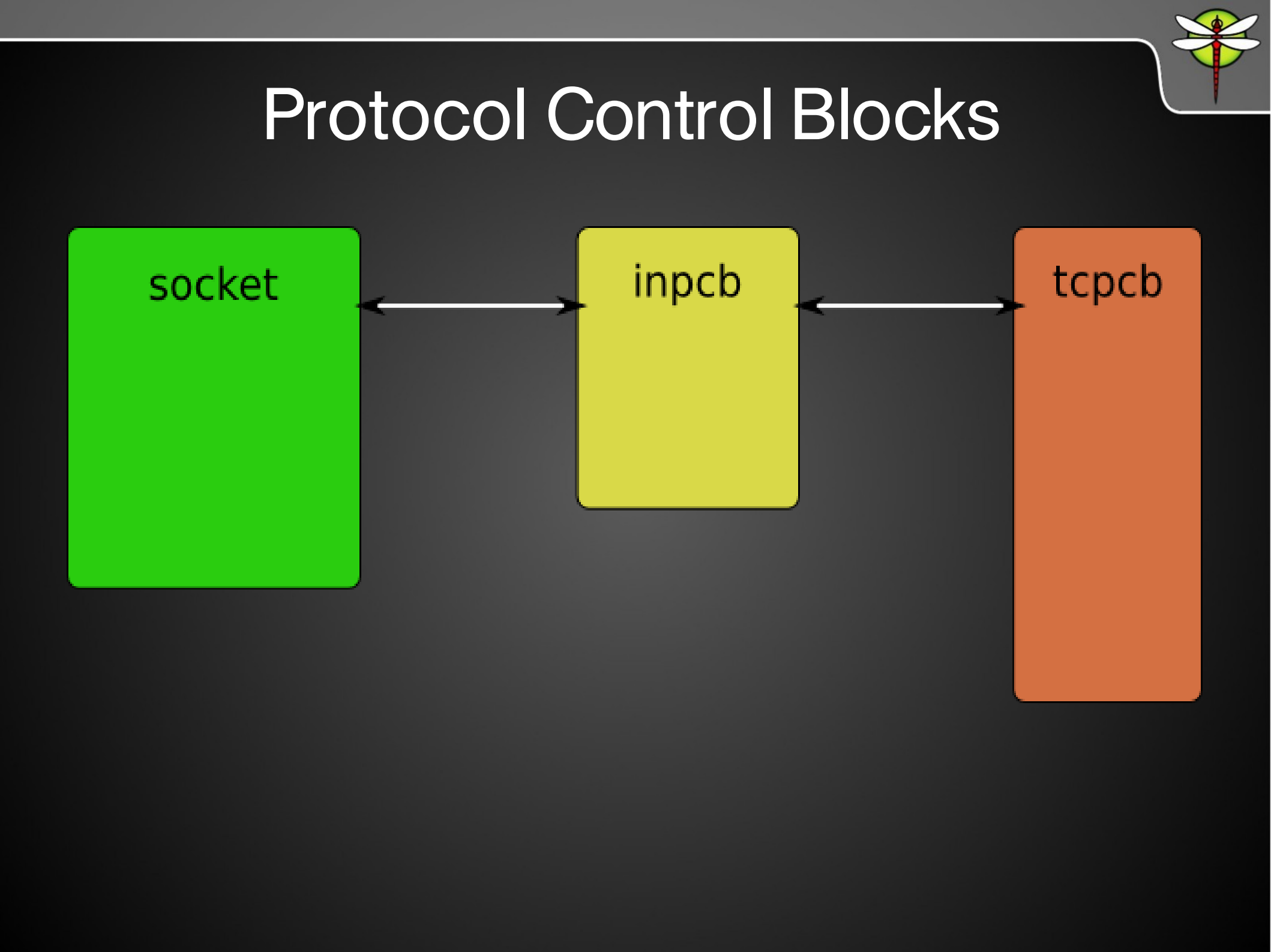


Transmit Path



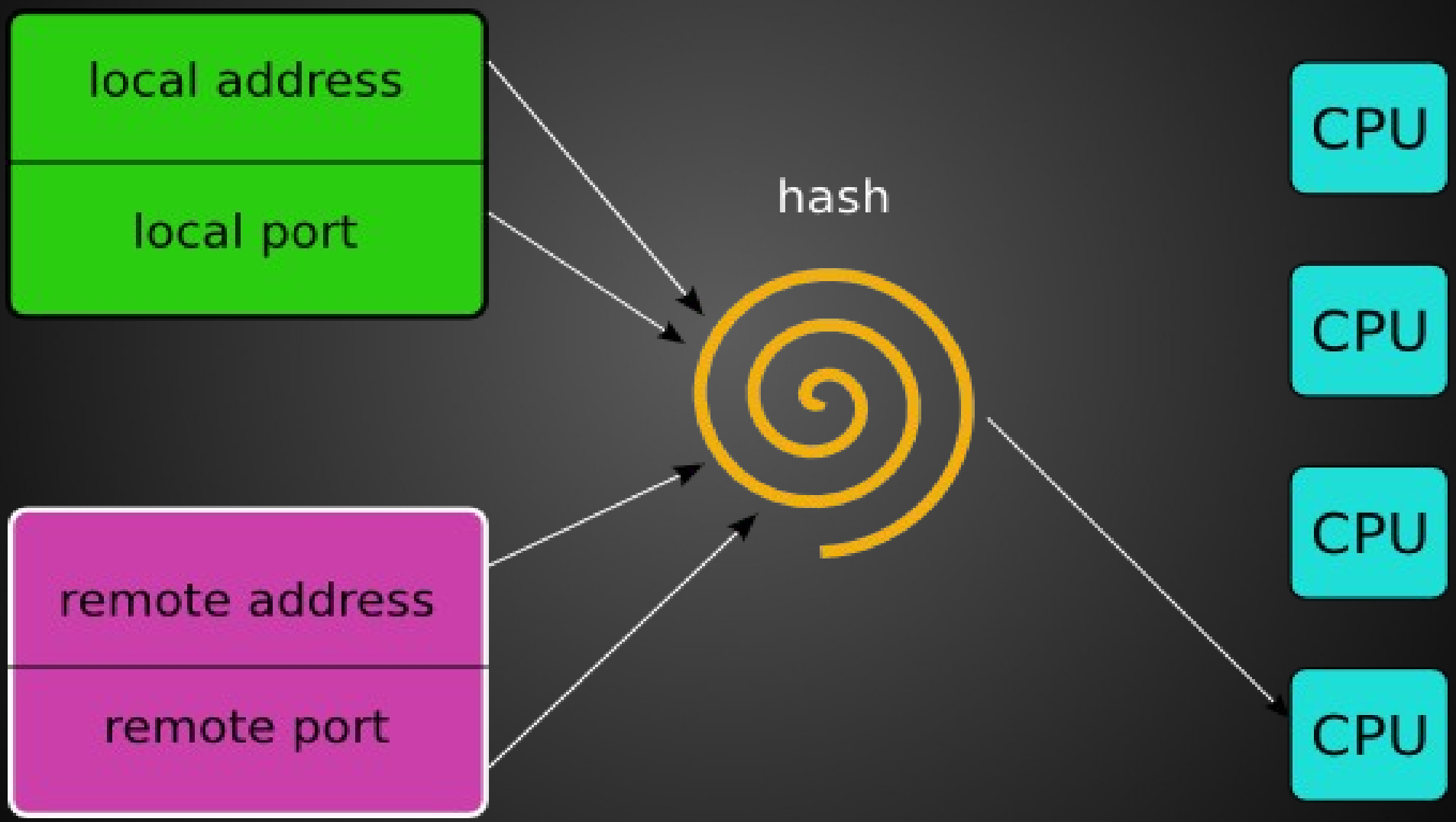
Issues we've had to deal with (1)

The Protocol Control Blocks

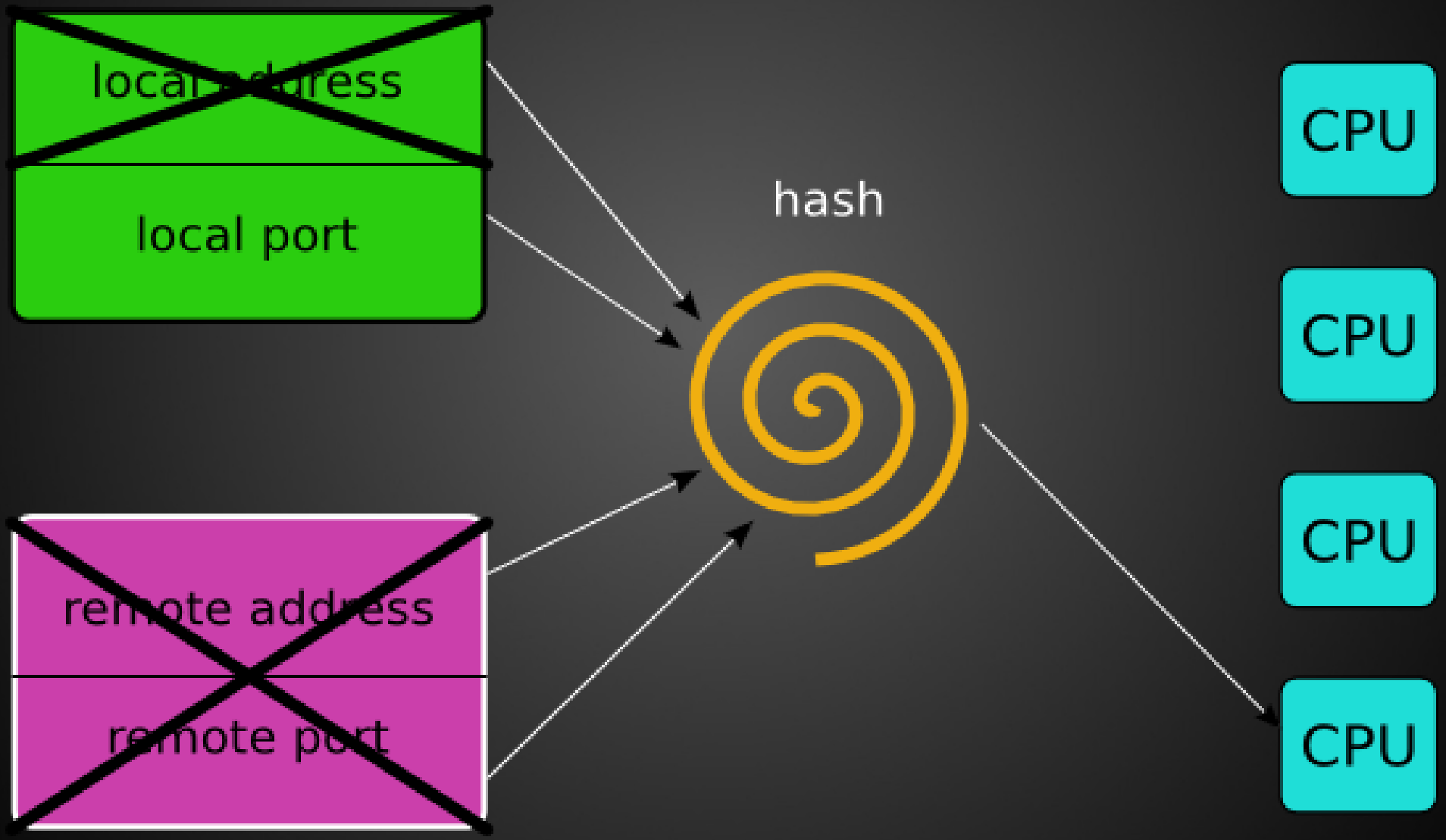


Protocol Control Blocks

TCP Hash


local address
local port
remote address
remote port
hash
CPU
CPU
CPU
CPU

UDP Hash


local address
local port
remote address
remote port
hash
CPU
CPU
CPU
CPU

What to do about UDP (1)

- Use local port only as input (suboptimal)

What to do about UDP (1)

- Use local port only as input (suboptimal)
- Use tombstones when one of the hash inputs changes and we have to migrate the inpcb

What to do about UDP (1)

- Use local port only as input (suboptimal)
- Use tombstones when one of the hash inputs changes and we have to migrate the inpcb
- Replicate inpcbs with wildcards (that would be most of them)

What to do about UDP (2)

- Then, why not go all the way and use N sockbufs for N CPUs?

What to do about UDP (2)

- Then, why not go all the way and use N sockbufs for N CPUs?
- It’s UDP! Everything goes!
- The process side can pull data in a mostly-fair manner (best effort is good enough)
- This means we will routinely deliver datagrams out-of-order

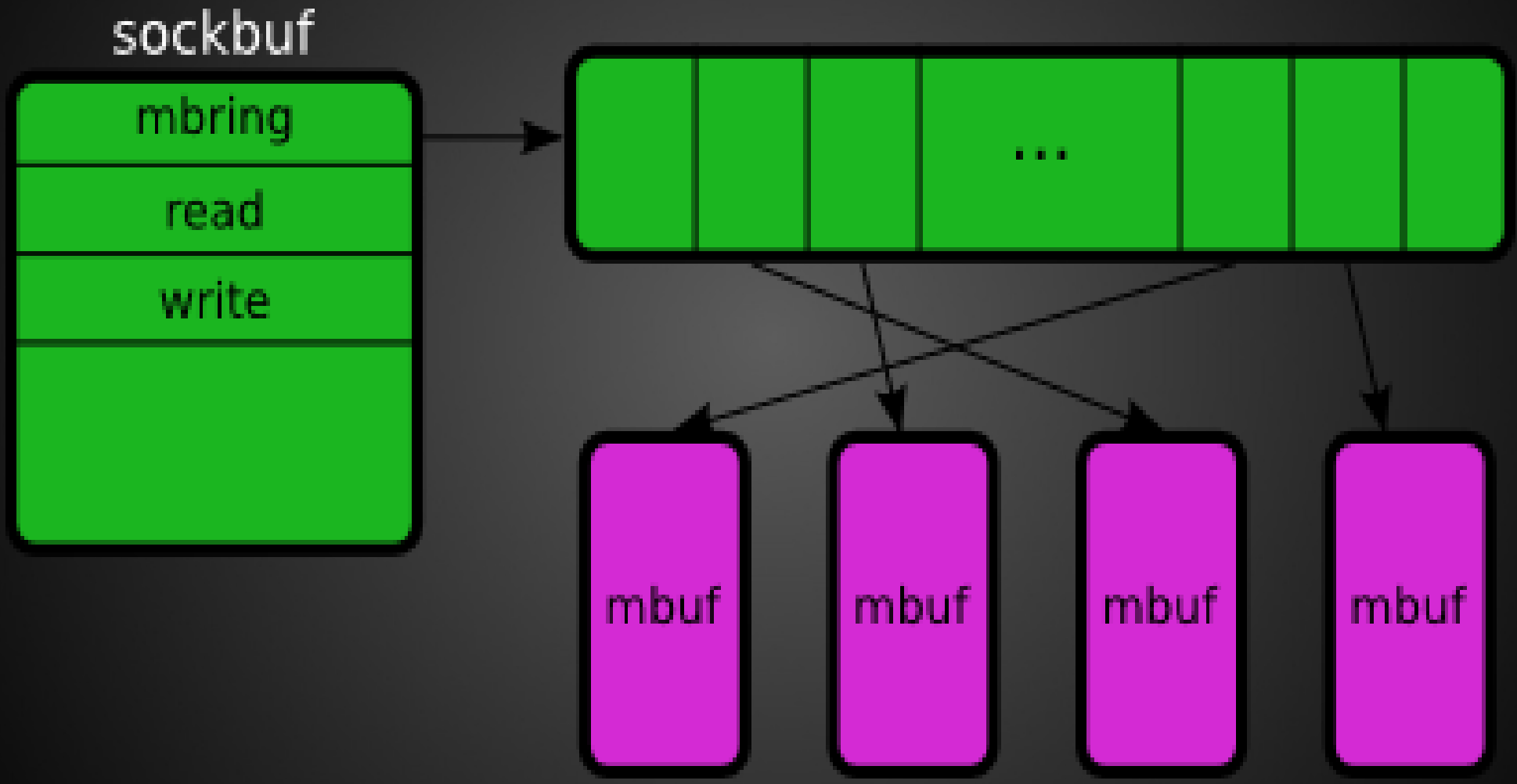
What to do about UDP (2)

- Then, why not go all the way and use N sockbufs for N CPUs?
- It’s UDP! Everything goes!
- The process side can pull data in a mostly-fair manner (best effort is good enough)
- This means we will routinely deliver datagrams out-of-order
- Can the apps handle it?

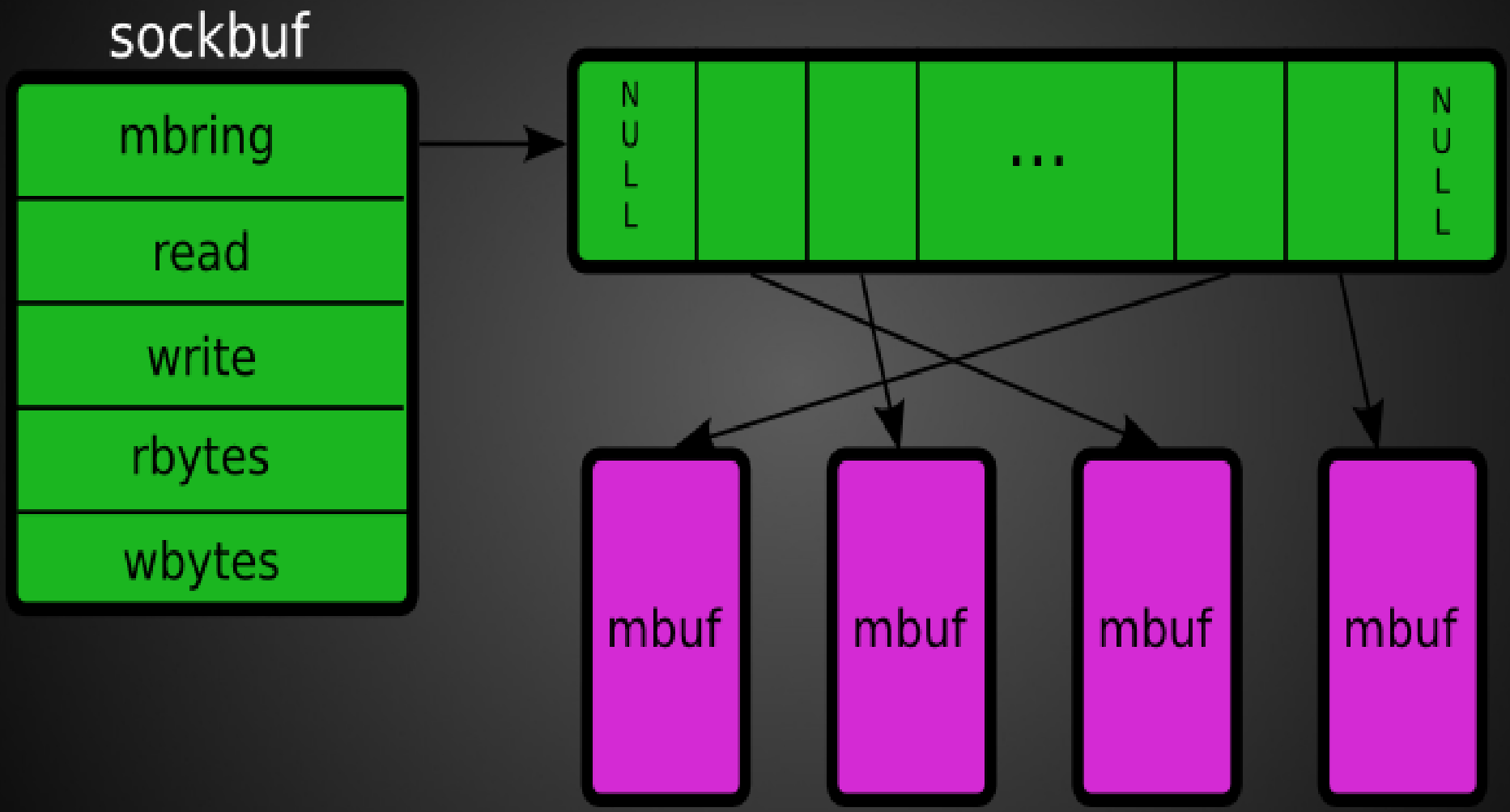
Issues we've had to deal with (2)

- Sockbuf
  - Send buffer (MPSC)
  - Receive buffer (SPMC)
- User side is always the "many" side

Ring Buffer (Lamport)



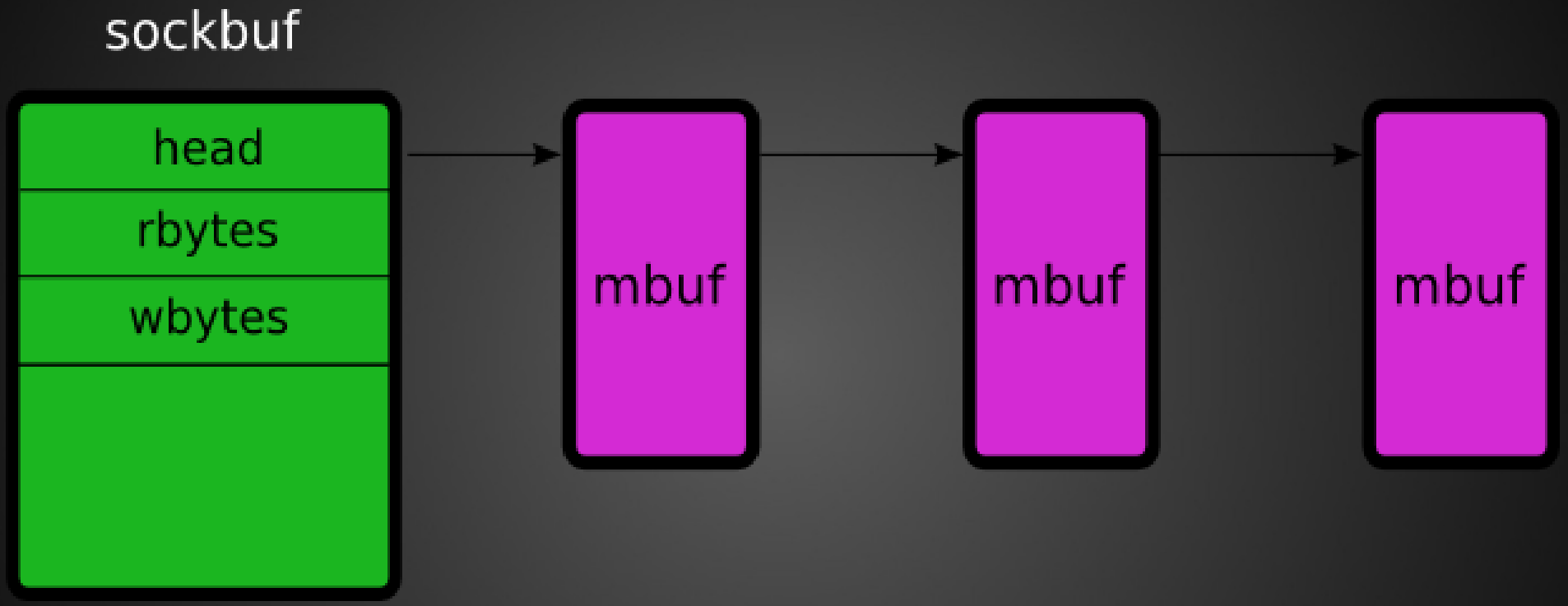
Ring Buffer (FastForward)



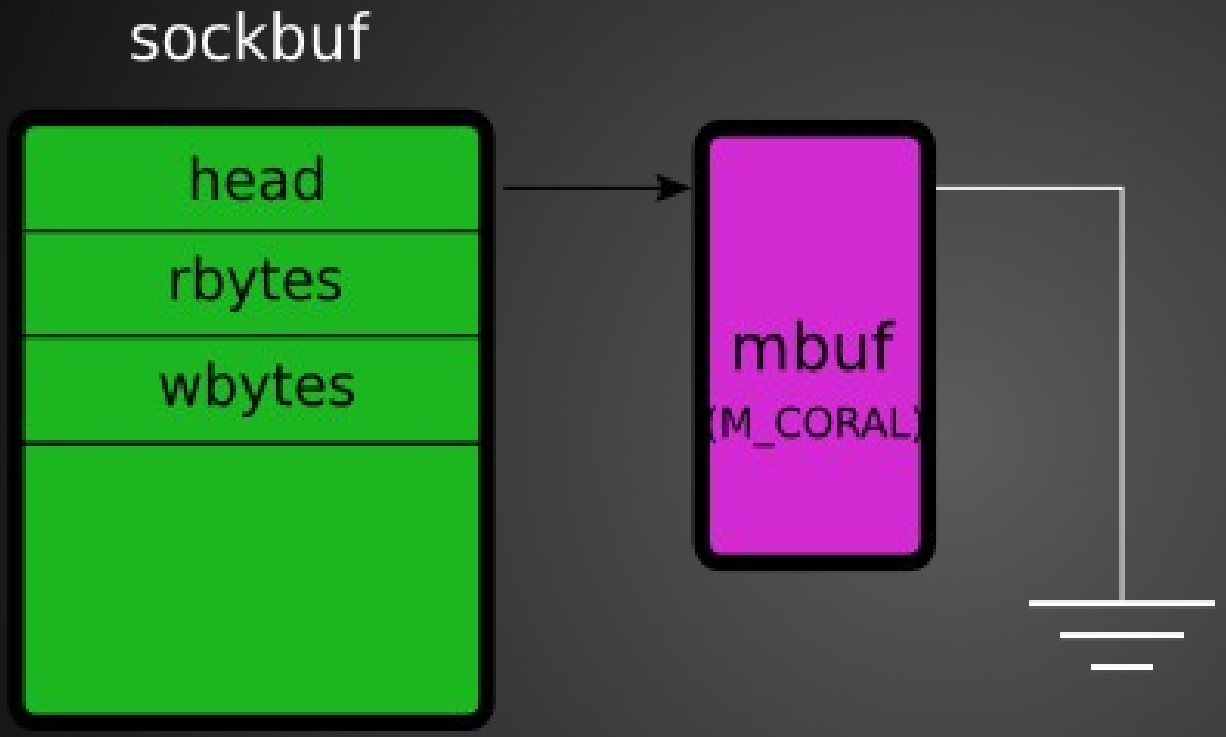
Ring Buffer Summary

- Pros
  - Minimal cacheline thrashing
  - Very straightforward code
- Cons
  - Large memory overhead (always)
- Future
  - Only if dynamic-sized

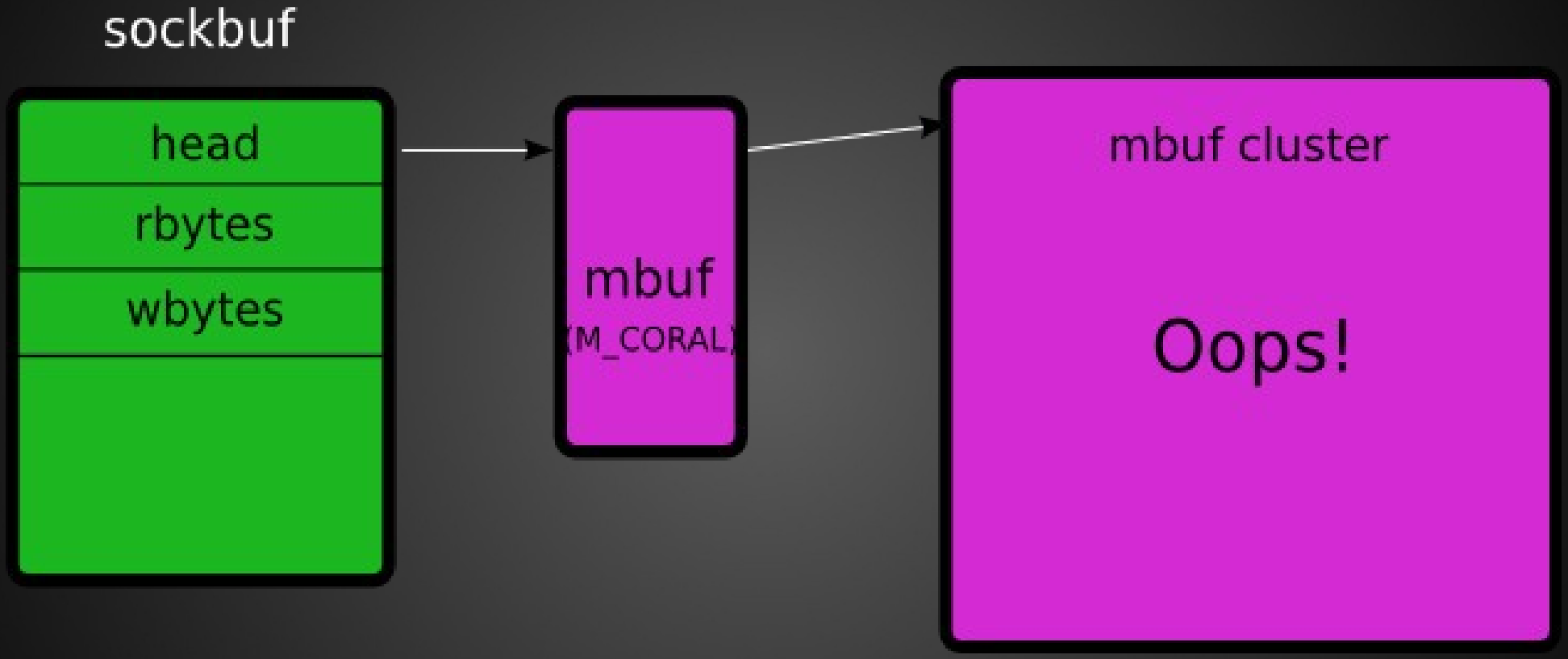
M_CORAL (1)



M_CORAL (2)



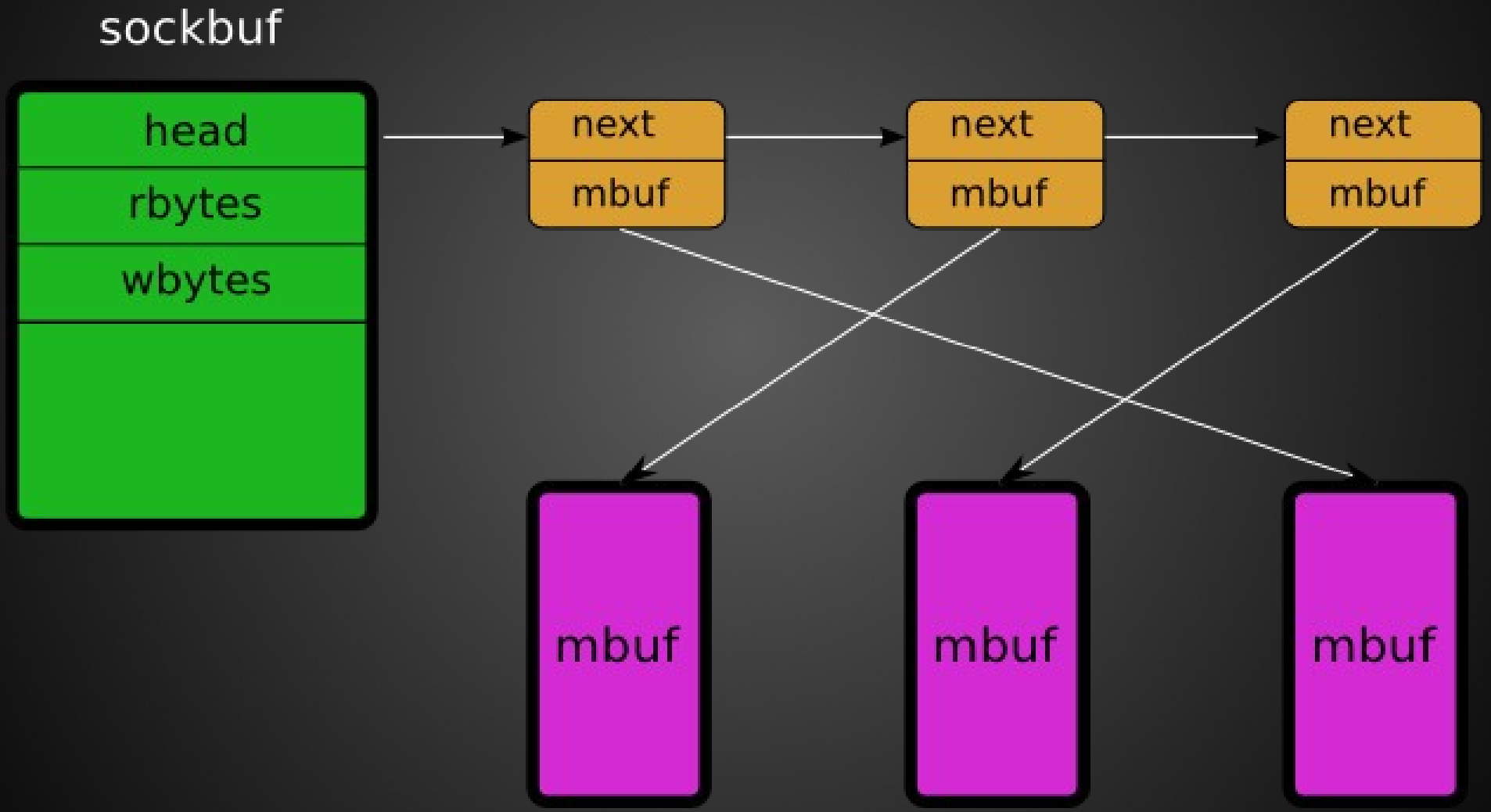
M_CORAL (3)



M_CORAL Summary

- Pros
  - Dynamic size
  - Memory overhead only for used sockbufs
  - Cache-friendly
- Cons
  - Mbuf clusters are large (2K)
  - Select-happy code
- Future
  - Just need a smart idea to solve the cluster deallocation problem

Cupholders


sockbuf
head
rbytes
wbytes
next
mbuf
next
mbuf
next
mbuf
mbuf
mbuf
mbuf

Cupholders Summary

- Pros:
  - Works now
  - Dynamic size
  - Simple code
- Cons
  - Larger cache footprint
  - Frequent trips to objcache
- Future
  - Will serve until we come up with a better solution

What about the character count?

- There ain’t no such thing
- BUT, we can provide lower XOR upper bounds and that turns out to be enough (To be continued)

Issues we've had to deal with (3)

Races

Races

- *Some* races are OK
  - Socket options
- Some are not, e.g.
  - Connection state
  - SS_CANTRCVMORE

Lost wakeup

```
get_mplock();
[…]
again:
if (have_data)
    break;
if (exception)
    break;
sleep_on_sockbuf();
```

```
again:
if (have_data)
    break;
if (exception)
    break;
syncmsg(notify_me);
```

Status

- Known issues
- Testing
- Performance measurements
- Code available in the netmp git repo
- Should be ready for 2.2

Fin

Questions?