How to port a TCP/IP stack in your kernel

without an Ethernet driver

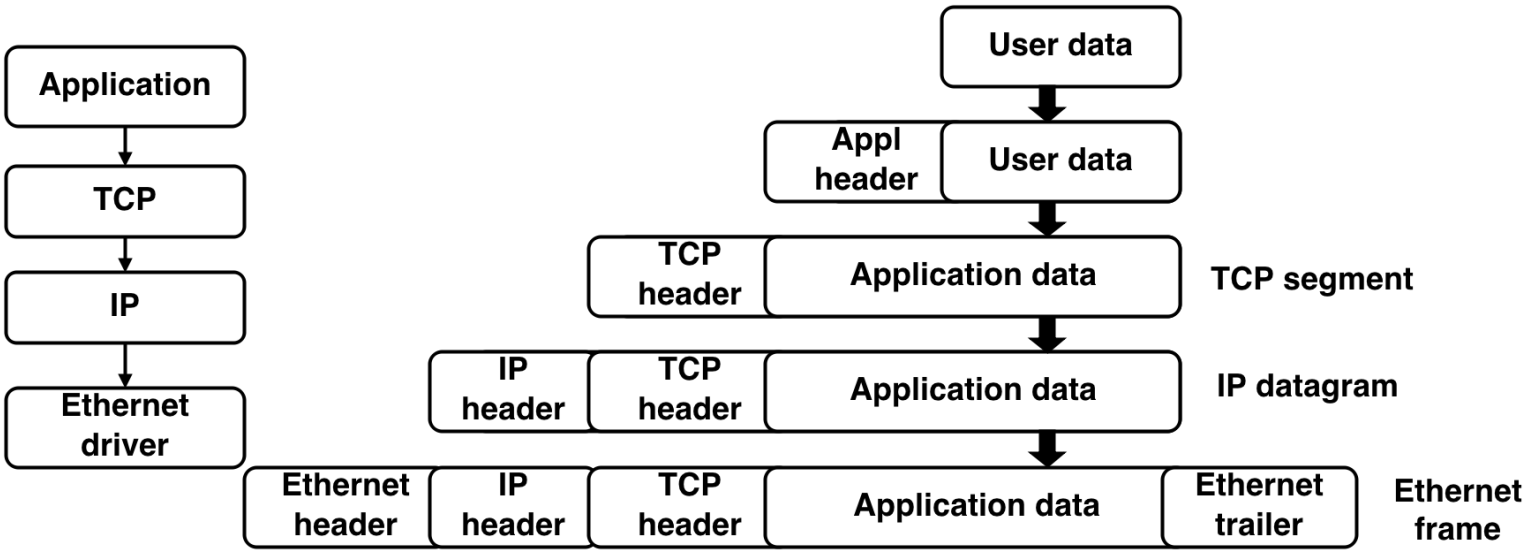
Nassim Eddequiouaq

July 20, 2014

Plan

1 Introduction

Generic TCP/IP stack



2 TCP/IP stacks

Which free open source stacks ?

- uIP
- lwIP
- tinytcp, wattcp..
- fNET
- Bentham's TCP/IP stack
- OpenBSD → not standalone

Focus on what you need

- embedded ?
- bare metal ? (no operating system)
- Keep It Simple, Stupid ?
- fast ?
- out of the box ?

Your choice.

- world's smallest TCP/IP Stack
- used mostly for 8/16 bits microcontrollers
- separates 32-bit arithmetic
- useful for embedded systems
- uses polling
- handles TCP, UDP (poorly), IPv4

uIP simple example

```
#define UIP_ACTIVE_OPEN 1
void setup() {
    connect_test();
}

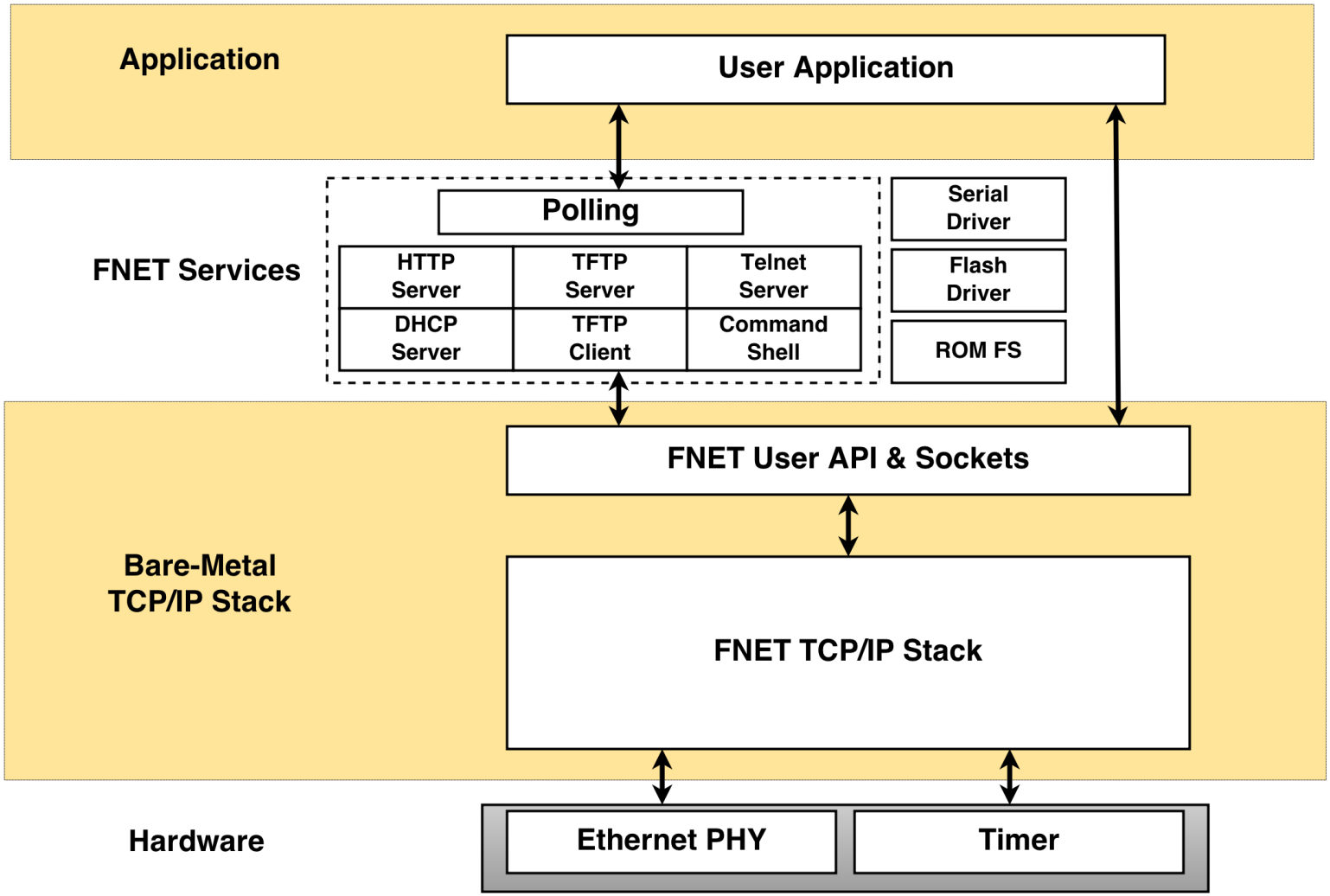
void loop() {
    uip_send("foo\n", 4);
}

void connect_test(void) {
    uip_ipaddr_t ipaddr;
    uip_ipaddr(&ipaddr, 192, 168, 1, 100);
    uip_connect(&ipaddr, HTONS(8080));
}

[...]
```

- works with IRQs
- which makes it VERY fast
- DHCP, AUTO-IP, ARP, UDP, PPP...
- works well even with few RAM
- several layouts of API

fNET structure



- mostly for 32bits microcontrollers
- supports ARM, Coldfire and Power Architecture
- TCP, UDP, IPv4, IPv6
- HTTP server, DHCP client, DNS...

Plan

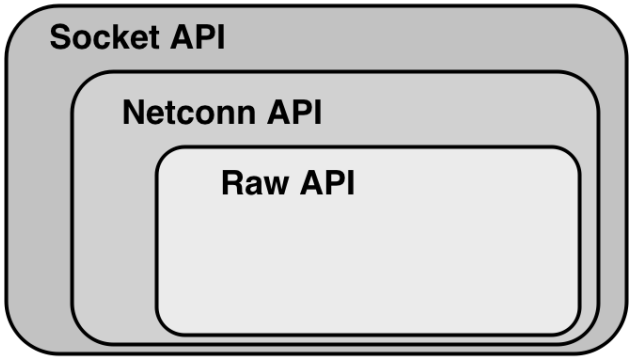
Plan

lwIP structure

lwIP offers 3 APIs:

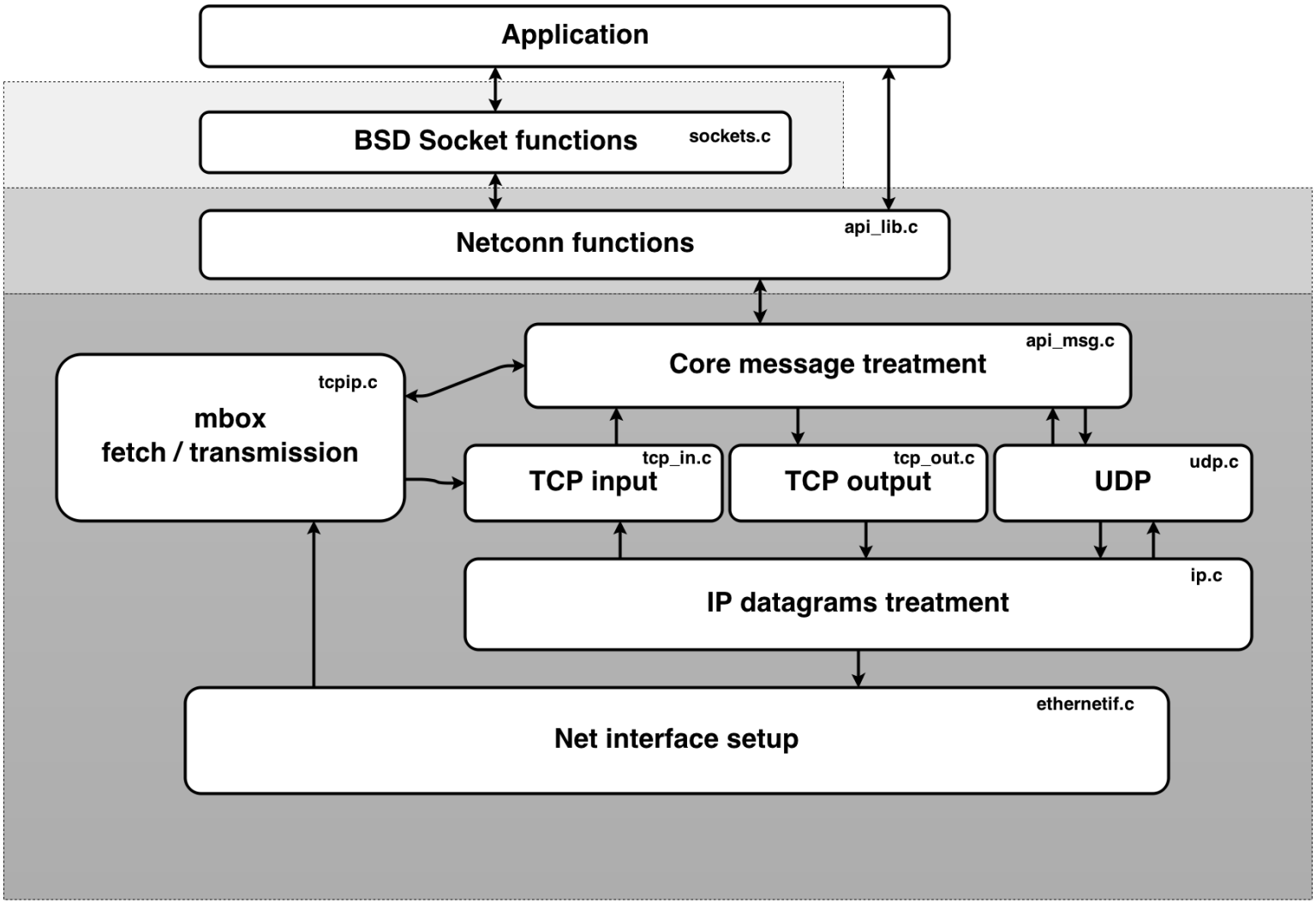
- Raw API
- Netconn API
- BSD Socket API

These APIs are layered in term of abstraction as follows:



Very useful!

lwIP structure



Raw API

This is the core API of lwIP

- best performances
- handles asynchronous events
- does not need an OS, bare metal works

Netconn API

This is the sequential API:

- allows multi-threading
- needs an OS
- lower performances than Raw API
- easier to use
- needs much more memory

BSD Socket API

A Berkeley-like Socket implementation:

- POSIX compliant
- very portable

Initialize the lwIP stack

Two ways:

- single-threaded:

  `lwip_init()`

- multi-threaded:

  `tcpip_init(...)`

The second one calls implicitly the first one

Your network interface

First, setup the compiler abstraction layer.
In ‘`cc.h`’, define macros describing your processor and your compiler for :

- types
- byte ordering
- structure packing

Your network interface

`lwIP` is able to use higher-level mechanisms and structures.

In '`sys_arch.c`', you may define a set of wrappers in order to make it access:

- semaphores
- mailboxes
- threads

This file is optional

Start your network interface

Add the interface:

- `struct *netif netif_add(...)`
- specify IP address, netmask, gateway address

Set the interface up:

- static IP address: `netif_set_{up,down}`
- DHCP: `dhcp_{start,stop}`
- AUTO-IP: `autoip_{start,stop}`

Initialize your device driver

`netif_add(...)` takes (among other parameters) the init and input functions.

You must provide these two functions with defined prototypes:

- `err_t foo_init(struct netif *netif)`
- ```
  err_t foo_input(struct pbuf *p,
                  struct netif *netif)
  ```
  ```

Set your net interface properly

Set your `struct netif` during initialization:

- state fields (i.e hardware address and length, MTU, name...)
- functions fields (output, link_output, input)
- flag field: OR options you want to activate (broadcast, PPP, ARP...)

Your driver's ouput functions

There are two function pointers that you need to set:

- netif->linkoutput
- netif->output

Implement the following functions:

- `err_t foo_linkoutput(...)`: called when a raw link packet is ready to be send and does the actual transmission
- `err_t foo_output(...)`: called when an IP packet is ready for transmission

`foo_output` should call `foo_linkoutput` when the treated packet is ready.

Your driver's input management

When you receive a packet, pass it to `foo_netif->input`.

This field has been set by your `netif_add` call, the last parameter being a function pointer:

- `err_t (*input)(struct pbuf *p, struct netif *netif)`

Sum up

1. Call `lwip_init()` / `tcpip_init(...)`
2. Ethernet init function to pass to `netif_add(...)`
3. Ethernet input function to pass to `netif_add(...)`
4. Setup `struct netif` to pass to `netif_add(...)`
5. Ethernet link_output function and add it to `struct netif`
6. Ethernet output function and add it to `struct netif`
7. Call `netif_add(...)`
8. Set default network interface with `netif_set_default(...)`
9. Set interface to 'UP' with `netif_set_up(...)`

Plan

How to test your network interface ?

Remember, we have no Ethernet driver yet!

Several solutions:

- init another lwIP inside your kernel as a standalone module
  - setup a protocol control block (PCB) and a TCP connection associated to this PCB
  - activate the TCP connection
  - make it `tcp_write(...)` data to your network interface
- create a TAP device before starting qemu and make it output/input data

Debugging your stack

There are some rough edges, you might want to tweak your stack:

- a debugger, a JTAG → what if you don't have one ?
- `printf` / built-in debug → quite nice actually!

1. `#define LWIP_DEBUG 1`
2. `#define IP_DEBUG LWIP_DBG_ON`
3. configure debug messages in '`opt.h`' and '`lwipopts.h`'

Some useful data

Figure: Lines of code in lwIP

Module	Lines of code	Relative size
TCP	1076	42%
Support functions	554	21%
API	523	20%
IP	189	7%
UDP	149	6%
ICMP	87	3%
Total	2578	100%

Linux networking stack had more than 25000 lines of code solely under net/ipv4 directory with `'tcp_*.c'` files in 2012...

Also, `lwIP` takes less than 14 kB in term of object code size. In comparison, Linux TCP/IP minimal stack size takes 153 kB.

Plan

4 Conclusion

Conclusion

If you want something scalable, `lwIP` is a very good solution!

Currently porting it myself to Stos.

Very active community around the project!

Tons of examples.

Questions ?

Feel free to ask at:

- nass@lse.epita.fr
- nass on #lse @irc.rezosup.org

Link

Adam Dunkels' (uIP and lwIP creator) thesis:
http://static2.wikia.nocookie.net/__cb20100724070440/mini6/images/0/0e/Lwip.pdf

lwIP Wiki:
http://lwip.wikia.com/wiki/LwIP_Wiki

lwIP -new- homepage, mailing list, forum:
http://savannah.nongnu.org/projects/lwip/