**ADVANCED OPERATING SYSTEMS**
**Milestone 7: Networking**
**Fall Term 2016**

Assigned on: **25.11.2016** Due by: **16.12.2016**

# Overview

In this project, you will develop a simple network stack for your operating system. It will allow applications to communicate with the outside world using the UDP protocol. To simplify things you will not use the on-board Ethernet adapter, which is USB attached and would depend on a working USB stack. To receive and send packets from the PandaBoard we will provide you with an additional serial to USB converter that you can use to communicate using the SLIP protocol. On your host computer you will execute a small utility `tunslip` that bridges the serial port and a tunnel interface. We will provide you with a driver for the UART that you can run in userspace on your operating system.

We require that the serial driver is running outside the `init` process. Later, also applications using the network should run outside your network stack process. Otherwise, you are free in the design of the network stack. For full points, multiple applications should be able to use the network at the same time.

**The work consists of:**

- Setting up the additional UART
- De- and encoding the SLIP protocol
- Implementing ICMP and reply to ping requests
- Implementing UDP and a simple echo server
- Passing UDP packets over your RPC implementation
- Enabling multiple applications to use the network

# 1 Using the additional UART

If your PandaBoard does not have an extra serial yet, one of the assistants will help you to connect one. The extra serial adapter will use the UART4 of the OMAP44xx SoC. As you now have two

/dev/ttyUSB* on your host, you might want to assign them unique names to avoid confusion. See the page on the Barrelfish wiki for hints[1].

We will provide you with a driver for the extra UART. The idea is, that this driver is executed in userspace. To do so, you will need to access memory mapped device and receive its interrupts. The (physical) memory location of the device frame is specified in aos/include/omap44xx_map.h in the constant OMAP44XX_MAP_L4_PER_UART4. Your driver must get a capability to this memory region and map into its address space. As with the interrupt table capability, you can either pass it in the CSpace of the network-stack process on creation or request it using an RPC. Once you can access the device, you can use the netutil library to instantiate the driver. Note that once you link against this library, your code must contain an implementation of the function serial_input as defined in include/netutil/user_serial.h, otherwise it will not link.

You will need to handle interrupts from the UART device. This requires setting up the interrupt descriptor table and register an endpoint with an interrupt number. In order to register for interrupts, you need to invoke a special capability (IRQ_CAP). This capability is passed to init and is located in the task CNode in TASKCN_SLOT_IRQ. You need to acquire it from init, either throught an RPC or by modifying the domain spawning. As soon as you have this capability in your CSpace, the function inthandler_setup_arm in the driver will succeed. The interrupt number for the UARTs can be found in include/netutil/user_serial.h in the defines UARTx_IRQ.

Receiving from the serial line is done on interrupt. The interrupt handler will only get called if you dispatch events from the default waitset. Make sure your application does so.

At the end of this step, you should be able to print and receive a message using the new serial interface.

# 2 Implement the SLIP protocol

To transmit packets over a serial line we use the *serial line internet protocol (SLIP)*. Essentially, it writes the content of the packet over the serial line followed by a special character, SLIP_END, to signal the end of a packet. If the byte appears in the packet, it gets replaced with a two byte escape sequence: SLIP_ESC SLIP_ESC_END. Additionally to the standard SLIP escaping, we also escape null bytes to avoid problems with the serial adapter. The tunslip already implements the (de-)escaping of the null byte.

| | Octal | Hex |
|---|---|---|
| SLIP_END | 0300 | 0xc0 |
| SLIP_ESC | 0333 | 0xdb |
| SLIP_ESC_END | 0334 | 0xdc |
| SLIP_ESC_ESC | 0335 | 0xdd |
| SLIP_ESC_NUL | 0336 | 0xde |

Table 1: SLIP escape characters

On the host side, we provide the utility tunslip. It creates a Linux tunnel device, reads and writes SLIP encoded packets from the serial line and connects these two. The following commands compile and start tunslip to set up a tunnel device, perform the point-to-point configuration, assign the IP *10.0.2.2* to the host side and to the PandaBoard the IP *10.0.2.1*. The last command assumes you have set up your system

[1] http://wiki.barrelfish.org/PandaBoard/ExtraSerial

to make the extra UART available at `/dev/panda-uart2`. Otherwise you have to change the name to your current `ttyUSBx`.

```
cd build
make tools/bin/tunslip
sudo ./tunslip -s panda-uart2 -H 10.0.2.2 10.0.2.1 255.255.255.0
```

The UART has a 64 byte FIFO buffer. If you produce too much (debug) output, it will slow down the handling and characters will be dropped. If you need debug information, make sure your packet fits in the buffer. To generate a small echo request packet, you can use ping with a zero byte payload:

```
ping 10.0.2.1 -c 1 -s 0
```

### Hints

- The source code of `tunslip` is located in `tools/tunslip/tunslip.c`. It may help you debugging your implementation by printing additional information. You can also start it in verbose mode by adding `-v` as an argument.
- You may want to consider writing parts of your IP stack to be portable, so that you can debug it on your host.

You should now be able to perform the SLIP decoding and print the (SLIP decoded) contents of a small ping packet.

## 3 Reply to echo requests (ping)

The next step is to decode the IP/ICMP packet, and write back a correct response. Wikipedia[2] has a good description how the request and reply have to look.

Packets are always encoded in the network byte order, which is big-endian. Your system uses little-endian, so you have to convert between host and network byte order. To do so, you can use the `htons` family of functions. In the function `htons`, `h` is for host, `n` for network and `s` for short. Hence you should use this to convert a 16bit value from your host to to the network byte order. We provide an implementation of these functions in `netutil/htons.h`. This is all you need to read and write shorts. If you encounter a bitmask however, you must also take into account the bit order. For instance, to set the `DF` flag which is the first bit in the offset field of the IP header, you will have to set the *first bit from the left*. This has the byte representation `0x40`, to make it end up at the first position, it has to be the higher byte of the short that you write into the offset field. Hence to set the DF flag you can use `ip->offset = htons(0x4000);`

The maximum size of an IP packet is 64KiB. When the underlying link layer does not support packets that big, *IP fragmentation* allows to split IP packets in smaller packets. You do not have to implement fragmentation, but you should detect and drop incoming fragmented packets.

The IP header contains a checksum, the checksum algorithm is defined in RFC 1071[3]. We provide an implementation of the checksuming function in `netutil/checksum.h`.

[2] https://en.wikipedia.org/wiki/Ping_(networking_utility)#ICMP_packet
[3] https://tools.ietf.org/html/rfc1071

Assign the static IP *10.0.2.1* to your PandaBoard. You have to set this as source address in your IP packets. You should not hard-code the host address when replying to echo request messages.

You should now be able to successfully ping your PandaBoard from the host.

### Hints

- https://www.wireshark.org.
- To write structs that match the UDP/IP headers, remember gcc's support for `__attribute__((__packed__))`. You can also write a mackerel definition. See the files in `devices/` as an example.

## 4 UDP echo server

The next step is to create a UDP echo server that listens on a port and replies to the sender by sending back the same packet. You can hardcode the port on which the server is listening. Use the IP/UDP header information source information to determine the destination address and port.

Use the following netcat command to test your echo server. It will open a UDP connection and send a packet once you press return. It also displays any answer that your server sends back.

```
nc -u 10.0.2.1 Port
```

You should be able to demonstrate with the command above that your implementation is functioning properly.

## 5 Multiple clients

For this section, you will export a service to other client applications. The client should be able to send UDP packets and listen for UDP packets on a specific port. Hence, your network service has to track open ports and dispatch incoming packets according to this information to the correct client. Also, ensure that the clients are not able to set an arbitrary source IP and port. As a demo application move your echo server into a separate process. You should be able to demonstrate that you are indeed running the server in a separate domain by first pinging your machine without the echo server running. Then, after starting the echo server, it should also act as an echo server.

The next step, is to allow multiple echo servers running on different ports at the same time.

## 6 Further challenges

- If you have a shell, instead of relying the packets execute the command and send back the output of your command.
- Connect to a NTP server and implement a real-time clock. You may have to change `tunslip` and your settings on the host, so that you are able to reach the internet.

# 7 Lab demonstration

You are expected to demonstrate the following functionality during the lab-session:

- 1) Display a message on the new serial line. If you have completed further steps, it is fine to just show a SLIP encoded packet. (2pt)
- 2) Dump a small ping packet to standard output. (1pt)
- 3) Show that you can ping your system. (2pt)
- 4) Send and receive back a packet from the UDP echo server by using netcat. (1pt)
- 5) That your echo server is running in a separate domain. (By starting the process at runtime). (2pt)
- 6) By starting two UDP echo server on different ports. (2pt)