DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

800 ANNI 1222·2022

UNIVERSITÀ DEGLI STUDI DI PADOVA

SIGNET

signet.dei.unipd.it

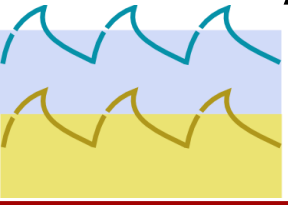
Device drivers

UNWiS - Padova (Italy)

30th of January – 3rd of February 2023

Filippo Campagnaro, Roberto Francescon, Angela Soldà, Antonio Montanari, Michele Zorzi

filippo.campagnaro@unipd.it



What are drivers

What we mean by drivers in DESERT

Actual modem

To connect to a real device and test new protocols developed through DESERT:
it is not enough to develop the packers

Once you have the correct translated bitstream, you have the content...
but:
to effectively run a protocol, you need to ensure timings and decisions, as defined by that protocol and complying with modem operations

Modem (*)ware

- A COTS device, usually, has its own firmware and, often, even its software
- It also has some way to communicate with it, such as a socket or serial interface: a connector
- Finally, also some sort of notifications that allow to know the status of the device and commands operating, receiving, transmitting

What we call driver

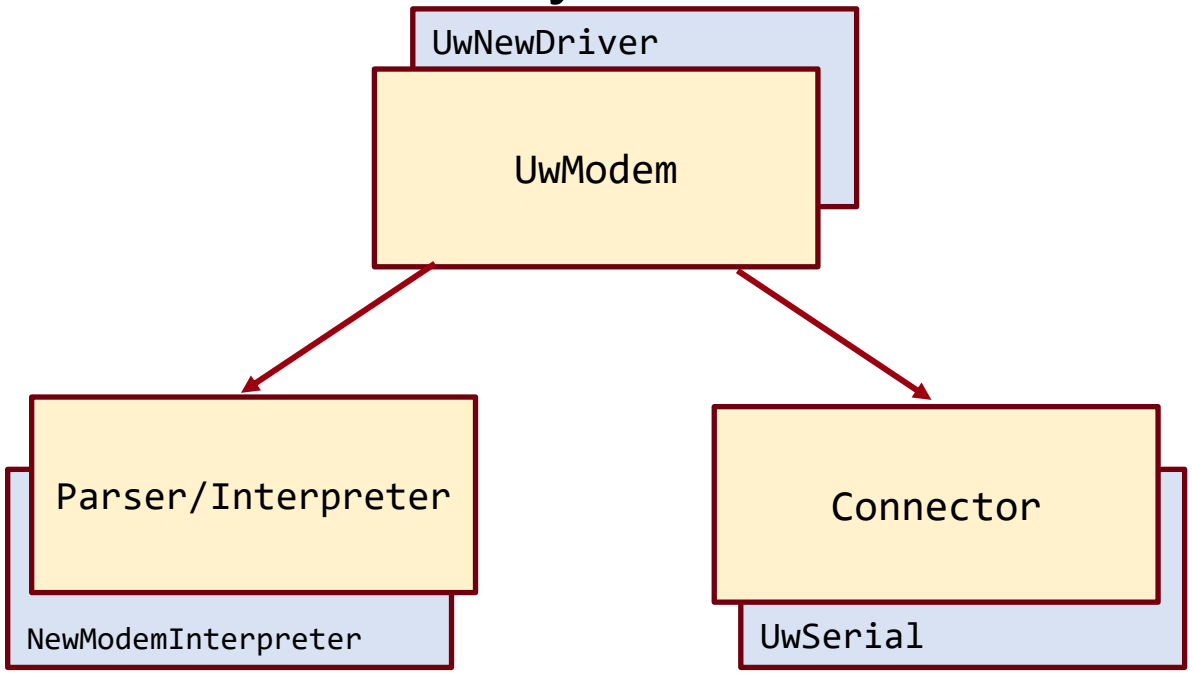
- A device driver in DESERT is a software module that understands the language of a modem, be it AT commands, serial bit sequences or APIs
- A device driver is able to operate the modem: when to transmit and how, when to receive and how, etc.
- It parses responses from the modem and can correctly format commands
- It knows the status of a device and what this status entails

Defining your own driver

You have your brand-new underwater modem, and you want to connect it with DESERT

Structure (1/2)

The basic structure of any driver is shown



The driver makes use of the connector, to retrieve and send bits, and of the interpreter to understand those bits

Structure (2/2)

The previous structure, based on 3 classes, is not mandatory, but is good practice: it helps in defining your new driver

Interpreter: usually manages strings or chars (`uint`). It is able to understand received bits, classifying received tokens

Connector: Connector interface provide APIs to send and retrieve bits. It is already implemented in sockets (TCP and UDP) and serial. *Evaluate if your device uses another technology*

The basics

- Device drivers are found in the *physical* layer folder
- To declare your driver for, say, `UwNewDriver`, start by declaring a class that inherits from `UwModem`

```
class UwNewDriver : public UwModem
{
 public:
    UwNewDriver();
    virtual ~UwNewDriver();

}
```

UwModem

The class `UwModem` provides a basis to build upon

- transmission and reception queues

```
std::queue<Packet *> tx_queue;
std::queue<Packet *> rx_queue;
```

- a handler for executing modem events
- the declaration of abstract methods such as
  - `recv(Packet* p)`
  - `startTx(Packet* p)`
  - `startRx(packet* p)`
  - `endRx(Packet* p)`

Packet arriving at the layer

We need to define what to do with packets arriving at the layer

Because PHY is the *last* layer, the valid option is packet is exiting (to be transmitted)

Packet from upper layers is received through the `recv(Packet* p)` method

- first, we strip the headers (macros) then we can assure the packet is going down

```
hdr_cmn *ch = HDR_CMN(p);
hdr_MPhy *ph = HDR_MPHY(p);
if (ch->direction() == hdr_cmn::DOWN);
```

Transmission of a packet (1/2)

- If the packet is going down the stack, we need to fill in the fields of the PHY header

```
ph->dstSpectralMask = 0;
ph->dstPosition = 0;
ph->dstAntenna = 0;
ph->modulationType = getModulationType(p);
ph->duration = getTxDuration(p);
```

- then we can proceed with the transmission: this means pushing the packet in the transmission queue

```
tx_queue.push(p);
```

Transmission of a packet (2/2)

The transmission of the packet, then, proceeds by selecting the packet from the tx queue, extracting the payload, formatting the bits and sending them down the connector

```
hdr_mac *mach = HDR_MAC(p);
hdr_uwal *uwalh = HDR_UWAL(p);
std::string msg;
msg.assign(uwalh->binPkt(), uwalh->binPktLength());

std::string send_cmd = "SENDING"; // sending command
uint length = msg.size(); // size of the payload
std::string destination = std::to_string(dest); //destination node

std::string cmd = send_cmd +":"+ length+":"+ msg+":"+ destination;

connector->writeToDevice(cmd);
```

In this case, the command to be sent was a string

Reception of a packet (1/2)

- To receive a packet, we need to retrieve bits from the interface that is connecting the modem to the host machine (PC): we need a **connector**
- The first thing to do is understand what is reserved word (for the modem) and what is payload (data for the upper layers)
- That is, we need an **interpreter**

Reception of a packet (2/2)

Retrieving bits from the connection interface makes use of the connector APIs

```
uint READ_BYTES;
char* buffer;
int n_bytes = connector->readFromDevice(buffer, READ_BYTES);
```

Then, we must proceed to parse the received bits to understand what they carry

```
status_t state = interpreter->parse(buffer);
switch (state) {
  case RECV:
  // proceed to receive stuff
  case ERROR:
  // analyze the error
}
```

UwConnector

Connector interface declares methods such as

```
class UwConnector
{
public:
  virtual bool openConnection(const std::string &path) = 0;
  virtual int writeToDevice(const std::string& msg) = 0;
  virtual int readFromDevice(void *wpos, int maxlen) = 0;
}
```

Currently implemented in DESERT:

- TCP (server/client)
- UDP
- serial interface

```
auto connector =
      std::make_shared<UwSocket>();

connector->openConnection("1.1.1.1:555");
connector->writeToDevice(command);
```

Interpreter (1/2)

It is often cleaner and better to define a separate class for the interpreter

```
class NewInterpreter
{
 public:
    NewInterpreter();
    virtual ~NewInterpreter();

    status_t parse(char* message);
}
```

The definition of the `parse()` method varies greatly, but the goal is to understand what the received message contains

Interpreter (2/2)

- The output of the method should be information that helps the driver decide what to do:
  - error messages: take action to understand and maybe fix
  - if configuration/monitoring is detected, cast the needed values and schedule application of the new config
  - if data is detected, locate and extract this data, and schedule it to be sent to the upper layers
  - AoB...

Data

Data-carrying packets contain bits that are to be sent up

```
retrievePayload(std::string message) {
  char init_cmd; //bits or char identifying payload start
  char end_cmd;  //bits or char identifying payload end

  auto it = std::search(message.begin(), message.end(),
                        init_cmd, end_cmd);

  std::string data = std::string(it, message.end());
}
```

```
Packet* p;
hdr_uwal* uwalh = HDR_UWAL(p);
uwalh->binPktLength() = rx_payload.size();
std::copy(data.begin(), data.end(), uwalh->binPkt());
HDR_CMN(p)->direction() = hdr_cmn::UP;
```

Internal messages

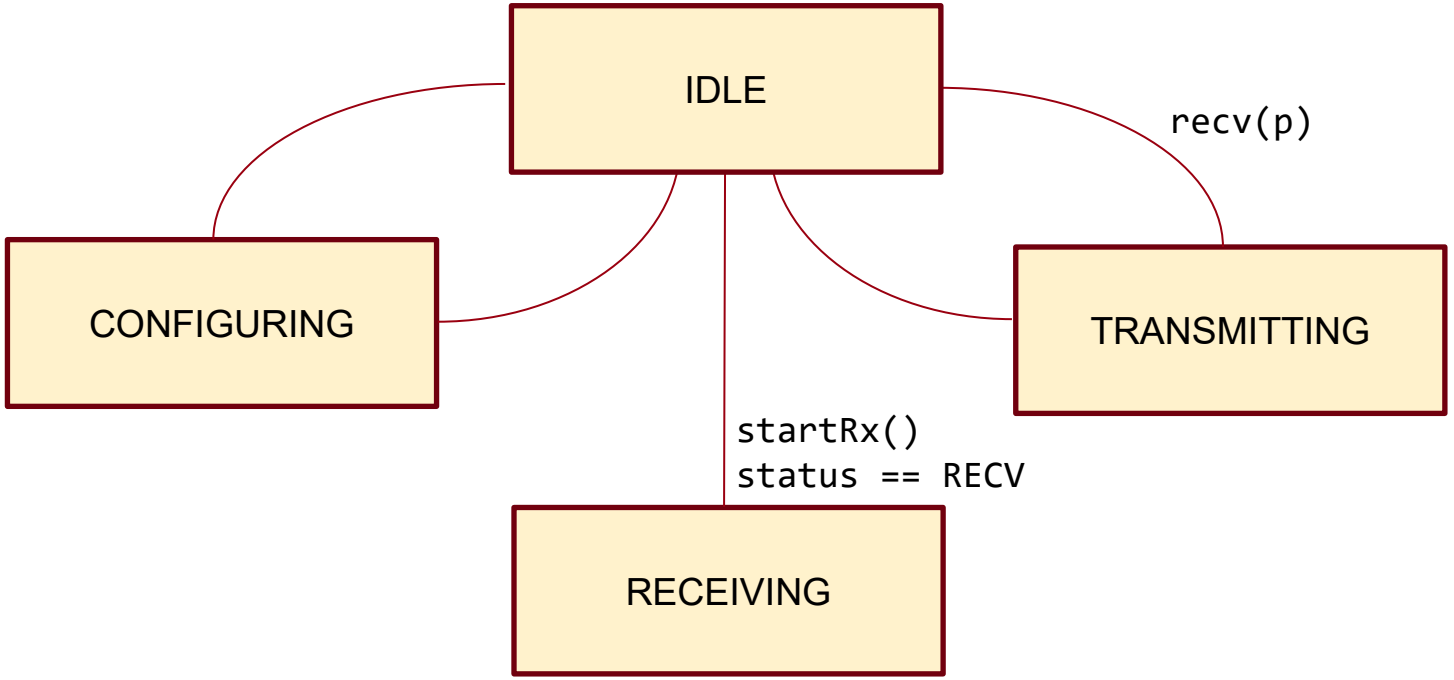
Information like error messages and warnings or configurations, act on the drivers itself

A good knowledge of the device processes is necessary

```
switch (response) {
    case ERROR:
        std::cout << "Modem ERROR" << std::endl;

    case SENT_MSG:
        std::cout << "Modem message sent" << std::endl;
        nextPacket();

    case INACTIVE:
        std::cout << "Modem INACTIVE" << std::endl;
        unblockModem();
}
```

State machines

It turns out it is effective to use state machines to manage modems: by carefully designing a state machine it is possible to ensure great deal of stability



and simplify the whole driver design

Running an emulation/sea trial

How to actually use our brand-new driver in an emulation, compared to a simulation

Overview

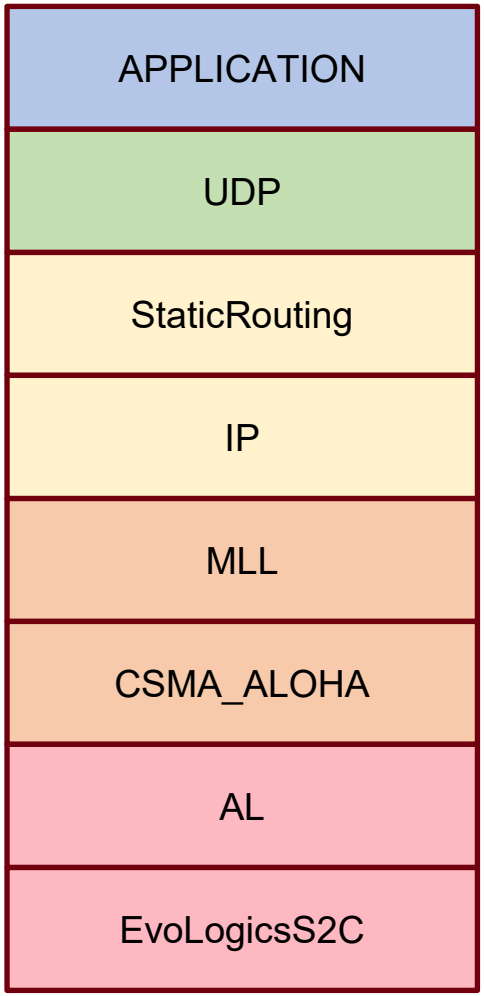
When we run an emulation or a sea test, we need to substitute the simulated physical layer with the device driver of the modems we are going to deploy

We need to make sure to have all the addresses (IP and ports, or parameters of the serial conn.) and check that everything is correctly set

Let's see how the protocol stack is set up, compared to a simulation

Instantiating objects

```
# APPLICATION LAYER
set app [new Module/UW/APPLICATION]
# TRANSPORT LAYER
set transport [new Module/UW/UDP]
# NETWORK LAYER: Static Routing
set routing [new Module/UW/StaticRouting]
# IP interface
set ipif [new Module/UW/IP]
# DATA LINK LAYER - MEDIA LINK LAYER
set mll [new Module/UW/MLL]
# DATA LINK LAYER - MAC LAYER
set mac [new Module/UW/CSMA_ALOHA]
# ADAPTATION LAYER
set uwal [new Module/UW/AL]
# PHY LAYER
set modem [new Module/UW/UwModem/EvoLogicsS2C]
```



Instantiating objects

Things to notice

The IP interface (`[new Module/UW/IP]`) is different from the routing protocol layer: it allows to be IP-compliant, avoiding the predefined routing protocols, which are instead implemented in the `routing` object

The Adaptation Layer is fundamental in the passage from the software packets to the physical packets: it manages translation of virtual packets (defined but non-existent) to actual stream of bits, to be modulated and sent over the channel

Adaptation Layer

The Adaptation Layer (AL) is responsible for converting virtual packets into actual binary streams

It makes use of the packer to forge the header and the payload from a virtual packet

The AL performs fragmentation and reassembling of packets

```
class Uwal : public MPhy // AL inherits from MPhy basic physical module
```

```
std::queue<Packet *> sendDownPkts; /**< queue of the packet to send down to the
modem */
std::queue<Packet *> sendDownFrames; /**< queue of the frames to send down */
std::queue<Packet *> sendUpFrames; /**< queue of the frames to send up to the
upper protocols */
std::queue<Packet *> sendUpPkts; /**< queue of the packets to send up to the upper
protocols */
```

Adaptation Layer

- It is not needed to modify the Adaptation Layer
- It is necessary to include it in the protocol stack and connect it to the right layers

```
$node_  addModule  3  $mac_   1  "ALOHA"
$node_  addModule  2  $uwal_  1  "UWAL"
$node_  addModule  1  $modem_ 1  "S2C"
```

```
$node_  setConnection  $mll_  $mac_   trace
$node_  setConnection  $mac_  $uwal_  trace
$node_  setConnection  $uwal_ $modem_ trace
```

```
$uwal_  linkPacker  $packer_
```

Packers setup

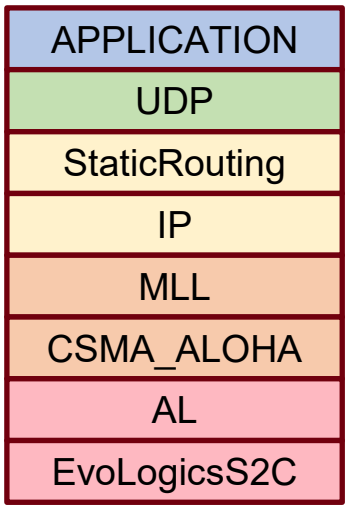
To have a correct translation from virtual packets to actual binary streams, we need to correctly configure and add the packers for the various layers

And then link the packer object to the AL

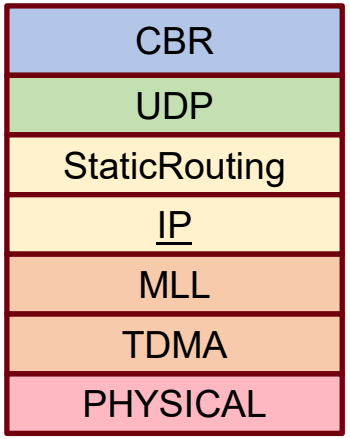
```
set packer_ [new UW/AL/Packer]
set packer_payload0 [new NS2/COMMON/Packer]
set packer_payload1 [new UW/IP/Packer]
set packer_payload2 [new NS2/MAC/Packer]

$packer_ addPacker $packer_payload0
$packer_ addPacker $packer_payload1
$packer_ addPacker $packer_payload2

$uwal_ linkPacker $packer_
```

Emulation/Sea test vs Simulation

```
set app  [new Module/UW/APPLICATION]
set udp  [new Module/UW/UDP]
set ipr  [new Module/UW/StaticRouting]
set ipif [new Module/UW/IP]
set mll  [new Module/UW/MLL]
set mac  [new Module/UW/CSMA_ALOHA]
set uwal [new Module/UW/AL]
set modem [new Module/UW/UwModem/EvoLogicsS2C]
```




```
set app($id)  [new Module/UW/CBR]
set udp($id)  [new Module/UW/UDP]
set ipr($id)  [new Module/UW/StaticRouting]
set ipif($id) [new Module/UW/IP]
set mll($id)  [new Module/UW/MLL]
set mac($id)  [new Module/UW/TDMA]
set phy($id)  [new Module/UW/PHYSICAL]
```



Emulation/Sea test vs Simulation

It is worth noting that, apart from the different physical layer, there is, usually, other differences

A simulation can host all the nodes in the network

An emulation and a sea test usually run a single node, as nodes are far away in a real-field trial and thus, if no other way exist to reach the far away nodes (e.g., Wi-Fi), for each node, we will have a computer (or an SBC such as RasPi) on which the DESERT instance is running