Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Networked Systems
ETH Zürich — seit 2015

Make your own Internet - Setting up a brand new P&S Project

Group Thesis

Author: Nicolas Adam

Tutors: Alexander Dietmüller, Rui Yang,
Roland Meier, Edgar Costa Molero

Supervisor: Prof. Dr. Laurent Vanbever

Autumn Semester 2020

1 Introduction

The final goal of this project is to create a completely new course for bachelor students, who want to learn about the Internet for the first time. In this course, we want to give these students an intuition about how a local area network works, how switches work and how they can think about the internet in a simplified manner.

This is also the keyword for this P&S project: **simplification**. Students taking this course, will learn about the Internet by creating their own Internet. We want them to program a simplified network, which, in principle, works similar to a real one. This would make concepts like the link forwarding, DHCP or ARP much easier to grasp for the first time learner and could help to really visualise, what is going on inside a network.

To do so, we prepared a board with 16 micro-controllers on it. Each one of them has a screen and can be programmed separately. They can all be connected to one another physically with cables. It will then be the students job to implement a working network by first connecting all the devices into a spanning tree and then programming each of the devices. They will be provided with a library written by us, that will help them, but generally we still want students to understand everything, that is happening in these routers by letting them program themselves as much as possible and letting them see inside all the devices at the same time.

Each of these little devices/nodes can be thought of as a layer-2 switch, a host or a NAT-router. The devices have different colours to symbolise what they are: pink is a NAT-router, yellow is a switch and violet is a host. The students will then need to program each device accordingly. They are all screwed to a board which helps keeping everything clean and orderly.

Because creating an entire course is beyond the scope of this kind of project, we limit our selves only to some aspects of it, which we will see in the next section.

2 The Project

We will focus on following aspects of the P&S course here:

- Ideas: Drafting a course outline
- Hardware: Making the devices and a demo board
- Software: Making a library and code template for the students
- Documentation: Making a setup guide for the software

As it was already mentioned, we're not going to have the P&S course fully setup and planned in this project. But we will have some ways and ideas to realise it

and we're going to know what devices we could use, what problems there are and how we could fix some of them.

3 Hardware

Since we wanted to make a simplified internet in this course, a real router or switch wasn't an option as hardware. First of all switches and routers are expensive but more importantly they are not very vivid. A switch is just a box without a screen and you never really know what is exactly happening inside. This is why we chose micro-controllers to do the job. We wanted each device to have a screen, so we can always see, what's happening inside.

3.1 Devices

The name of the chosen micro-controller is "Wemos Lolin ESP32 OLED". It's based on the ESP32 micro-controller with an integrated OLED screen, Bluetooth and WIFI unit. There are multiple reasons we chose this board:

- It's quite new and very powerful compared to other micro-controllers.
- It's cheap. ($\approx$ 12CHF / board)
- The screen is included. (onboard)
- The USB converter is included. (onboard)
- We can easily solder plugs onto it. ($\approx$ 2CHF / board)
- Except for plugs and power no additional elements are needed.
- It works with the Arduino IDE.

The reason we decided to use this controller is, because first we wanted to use the "ATmega328p", which was very cheap ($\approx$ 1CHF / board) but did not include a lot features. When we found out, that the USB-converters we needed, were actually more expensive ($\approx$ 16CHF / board!) than any other modern micro-controller, we decided to change the board. Because we already implemented a lot of code, we wanted a board, that is compatible with the Arduino IDE. After extensive search, it turns out, that "Wemos Lolin ESP32 OLED" is a better fit, than the "ATmega328p" because:

- The "ATmega328p" has very limited capabilities.
- We already used 50% of its RAM just for the template.
- We would have needed to design and print PCBs,
- Together with USB adapter and PCB it's more expensive than any modern micro-controller. (like ESP32)
- The "Wemos Lolin ESP32 OLED" is cheaper, more powerful and less work is needed to build a device from it.

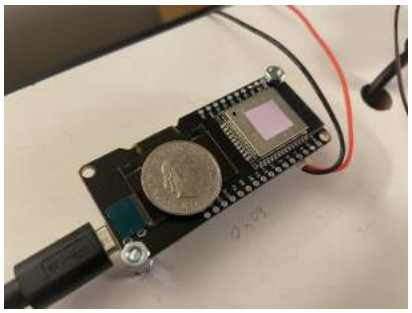

Figure 1: ESP 32

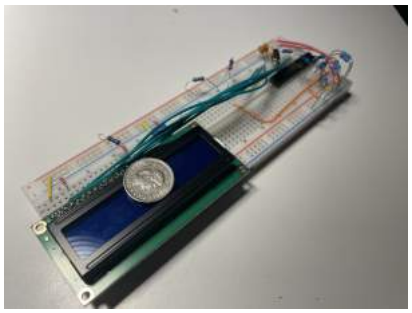

Figure 2: Atmega32p-prototype

3.2 Plugs and Cables

For the plugs we chose normal 1.25mm JST-plugs with cables already soldered onto them. We ordered them in pairs, as seen in the figure below. We then proceeded by soldering the cables and gluing them onto the device.

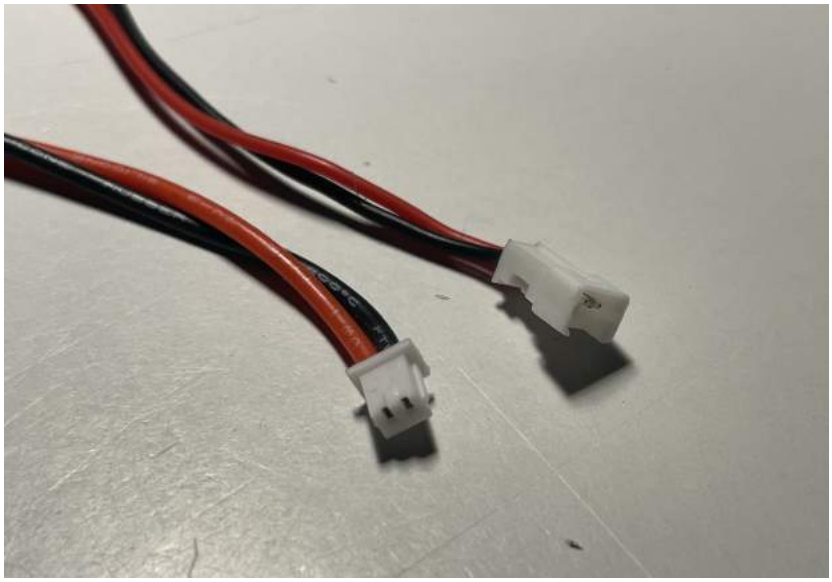

Figure 3: JST-cable pair

3.3 Board

Because every device has a power cable, things can get quite tedious. That's why we decided to screw everything onto a wooden board. This allows us to

hide all the power cables under the board. That way it's much clearer how the boards are connected.

3.4 USB-Hub

There are a lot of devices, so a lot of USB-Ports are needed. We found a nice solution: The "USB 3.0 16-port Charging HUB" from i-tec. The device is of good quality and has a button for every USB-Port, which allows us to turn every device on and off directly at the hub.

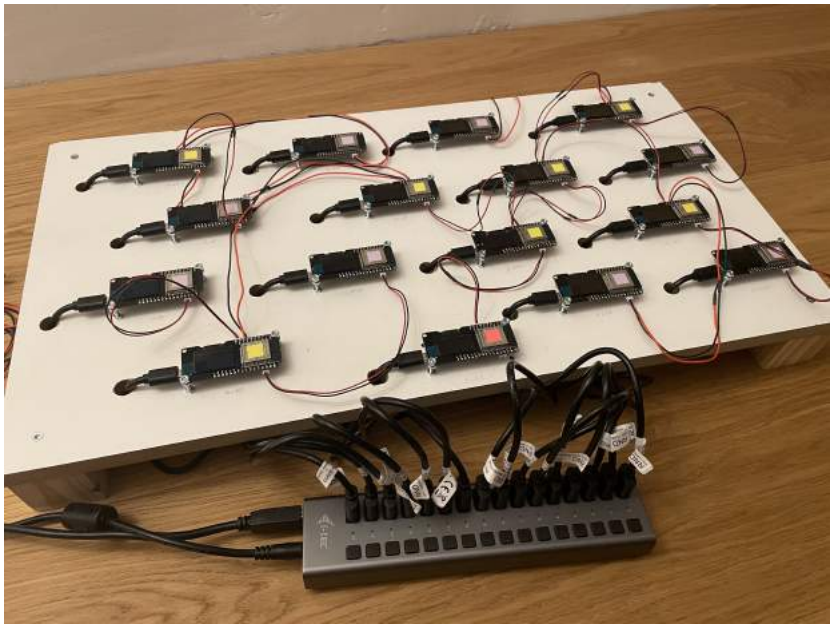

Figure 4: The Wooden Board with USB-Hub

3.5 Hardware Setup Guide

3.5.1 Requirements

- 16x "Wemos Lolin ESP32 OLED"
- 16x USB-mini cable
- 1x "USB 3.0 16-port Charging HUB" from i-tec
- 64x 1.25mm JST-plugs with cables
- 1x Wooden Board 48cm x 36cm
- 32x Screws M3, length: 30mm
- 96x Nuts M3

3.5.2 Soldering the Cables

Soldering the cables is very simple. You always have to solder the black cables to the pink ones and vice versa. This becomes very clear, when you take a look at the cables used in the demo-board.

3.5.3 Soldering the Devices

Soldering of the Devices is a bit harder. Each cable has to be soldered to the correct port as seen in the following picture:

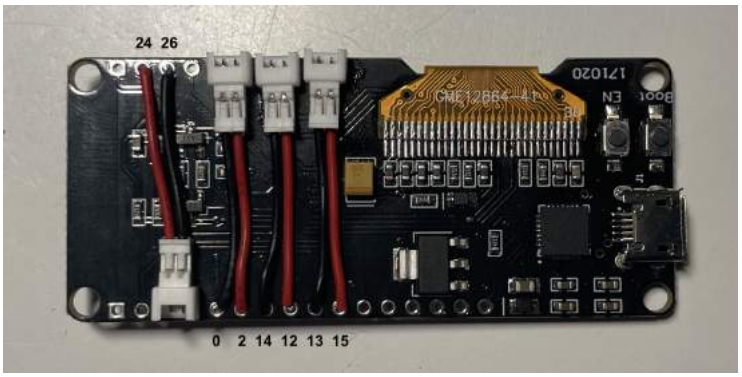

Figure 5: Correct plug placement

3.5.4 Building the Board

To build the board you have to drill three holes per device: Two, which are big enough to fit an M3 screw and one, 13mm wide, big enough to fit the power cable. Using nuts you can then fix the devices onto the board. Take a look at the demo-board to understand how it was done.

4 Software

4.1 Arduino IDE

The Arduino IDE is very easy to use and install. It's available on all platforms and is generally very well known and understood. It's always clear, what is happening and it works in C++, the first programming language you learn, when studying at ETH. Therefore, we chose to use this IDE from the very beginning of the project.

4.2 Software Setup Guide

4.2.1 Arduino IDE

The Arduino software can simply be downloaded here:

https://www.arduino.cc/en/software

Just follow the instructions on the Arduino website.

4.2.2 Arduino - ESP32 Tools

We also need to add all the ESP32 boards to the Arduino Boardmanager.

1) **Find the "Arduino" folder:** Arduino always sets up a folder called "Arduino". Per default you can find it in your Documents folder. This is the same folder, where all your code is being stored, if you can't find it, you could try to find out, where your code is stored by creating a new sketch and searching for this sketch on your computer.

2) **Download the board tools:** For now all the tools needed are stored on my own private ETH webpage:

https://n.ethz.ch/~adamn/arduino/

3) **Add the board tools:** Unzip the "ESP32-Tools" and copy the folder "hardware" to the "Arduino" folder. If the folder "Arduino/hardware" already exists, then only add the contents of "ESP32-Tools/hardware" folder to "Arduino/hardware".

4) **Select the board:** In the menu bar select:
Tools > Board: > ESP32Arduino > WEMOS LOLIN32

5) **Check if the library is available:** In the menu bar select:
Sketch > Include Library: > ESP8266 and ESP32 OLED driver for SSD1306 dispays

4.2.3 SLAB to UART driver

To talk with our ESP32 we'll need a driver that converts SLAB to UART. This is the most difficult part of the setup because depending on your system, there are different kinds of drivers. Try to download and install this driver, if this one does not work, you'll need to look for another one.

https://www.silabs.com/developers/usb-to-uart-bridge-vcp-drivers

4.2.4 Programming Guide

There are two buttons underneath the micro-controller. **The outside button needs to be held down while programming a device.** This needs to be done carefully because it could break the device if pushed to hard. This also ensures that the programmer doesn't accidentally program the wrong device.

5 Library

This is a library of classes which are used to program the devices. The goal is to provide the students a general skeleton so they can just focus on the concepts rather than programming every detail. In addition to the library a lot is already implemented in the template.

5.1 Byte

In the Arduino IDE there is a data type called byte, which stores an unsigned 8-bit integer from 0-255. We will use this byte for a lot of our classes, because the length of the packet we transmit is very limited and byte was therefore the most reasonable type to use.

5.2 Template

The entire code structure is already implemented in the template. All objects, that are needed, like the display object or the PhysicalLayer object are already instantiated. This will provide the students with a good starting point, so they could focus more on implementing network functions. The template itself has three different parts:

5.2.1 Device.ino

This is where the main part of the code runs. Here you can see the part of the file where the students would then start to implement their devices.

```
////////////////////////////
// GLOBAL VARIABLES/////////
/////////////////////////////

// PUT YOUR GLOBAL VARIABLES HERE  ....
      // ...
      // ...


////////////////////////////////////////
// LOOP ///////////////////////////////
////////////////////////////////////////

while(true){

  //CHECKS IF PORTS ARE ACTIVE
  pLayer.update(receive_flag);

  //PHYSICAL - LAYER - HANDLE ALL RECEIVING - NO INTERRUPTS DURING THIS TIME
  if(pLayer.receiveAvailable()){
```

```
    pLayer.receiveHandle(pLayer.receiving_port);
    attachAllInterrupts();
    continue;
  }

  //PHYSICAL - LAYER - HANDLE ALL SENDING - NO INTERRUPTS DURING THIS TIME
  if(pLayer.sendAvailable())
  {
    pLayer.sendHandle();
    attachAllInterrupts();
    continue;
  }

  //LINK - LAYER
  if(pLayer.available())
  {
      ////////////////////////////////
      // PROCESS RECEIVED PACKET////////
      ////////////////////////////////
      Packet thispacket;
      pLayer.receive(thispacket);

      // START PROGRAMMING YOUR DEVICES FROM HERE  ....
      // ...
      // ...

      //pLayer.send(thispacket);
   }
}
```

5.2.2 Definitions.h

This is where all the necessary parameters are defined, which can be used by any class or function in the entire project. These paramaters include values like:

```
//PLACE OF THE DEST_MAC
#define DEST_MAC 0
//PLACE OF THE SRC MAC
#define SRC_MAC 1
//PLACE OF THE TYPE HEADER  (0=ip, 1=dhcp, 2=arp)
#define TYPE 2
//PLACE OF CHECKSUM
#define CHECKSUM 3
//PLACE OF DEST IP ADDRESS
#define DEST_IP 4
```

```
//PLACE OF SRC IP ADDRESS
#define SRC_IP 5
//PLACE OF TTL HEADER
#define TTL 6
//PLACE OF NEXT PROTOCOL HEADER
#define NEXT_PROTOCOL 7
```

The definitions above can be used to simplify the code, as stated below:

```
byte src_mac = thispacket.returnByte(SRC_MAC);
thispacket.changeByte(DEST_MAC, src_mac);
```

This would for example swap the destination to the source MAC address.

5.2.3 Functions.h - Functions.cpp

This is where functions or classes, that can be called in the main loop, could be defined. Students are free to use this document to implement their own things.

5.3 Packet Class

The packet class is used to store packets that have been received or that will be sent. Each packet consists of 16 bytes, which are the bytes that were received. One byte stores the port, on which the packet has been received and a bool array stores the ports, on which the packet will be sent. In addition, there is a public variable called id, which can be used to give a packet an unique id, which can be used for anything. The class has the following member-functions:

5.3.1 Member Functions

- Constructor: creates a new packet object.

```
Packet thispacket;
```

- Destructor: deletes packet.
- Return: returns a header/payload byte at byte_nr location.

```
byte thispacket.returnByte(unsigned int byteNnr);
```

- Change: changes a header/payload at byteNr location to value (byte)

```
void thispacket.changeByte(unsigned int byteNr, byte value);
```

- Receive Port: returns port on which packet has been received

```
byte thispacket.returnReceivePort();
```

- Change Receive Port: changes port on which packet has been received (used by PhysicalLayer class)

```
byte thispacket.changeReceivePort(byte newReceivePort);
```

- Send Port: packet will be sent on all these added ports

```
void thispacket.addSendPort(byte portNr);
```

- Send Port Active: returns true if that port is activated as send port

```
bool thispacket.returnSendPortActive(byte portNr);
```

- All active send ports: returns all activated send ports as a string

```
String thispacket.returnAllSendPorts();
```

- Don't Send: packet will not be sent unless a new send port is added

```
void thispacket.removeAllSendPorts();
```

- To String: returns a string containing all bytes in hexadecimal

```
String thispacket.toString();
```

- To String: passing two integers to this function will return only a part of the packet as a string. start_byte is the first printed byte, end_byte is the first **not** printed byte.

```
String thispacket.toString(u int start_byte, u int end_byte);
```

5.3.2 Examples

```
//Here we're going to create a packet and change it's headers
Packet thispacket;
byte dest_mac = 0xff;
byte src_mac = 0xa2;
void thispacket.changeByte(0, dest_mac); // change first byte
void thispacket.changeByte(1, src_mac); // change second byte

// get strings from the headers
String linkHeader = thispacket.toString(0,4);
String ipHeader = thispacket.toString(4,8);
String fullPacket = thispacket.toString();
display->display(fullPacket);

// send packet on port 2,3
void thispacket.addSendPort(2);
void thispacket.addSendPort(3);

//Here we're going to read data from a received packet
byte dest_mac = thispacket.returnByte(0);
byte src_mac = thispacket.returnByte(1);
byte receivePort = thispacket.returnReceivePort();
```

5.4 PhysicalLayer Class

The PhysicalLayer class handles all sending and receiving on the ports. (See Appendix for more info) It's a global object that is created once and should never be deleted. This object is already defined in the template as "pLayer". If a packet has been received the pLayer object will buffer it until pLayer.receive() is called, which returns the packet and removes it from the buffer. All these functions are already added at the right place in the template. The class has following member-functions:

5.4.1 Member Functions

- Constructor: creates a new PhysicalLayer object that handles all sending and receiving.

```
PhysicalLayer pLayer;
```

- New Packet available: returns true if a new packet has been received and added to the buffer.

```
bool pLayer.available();
```

- Return new packet: loads the received packet into the packet passed by reference &packet.

```
void pLayer.receive(Packet &packet);
```

- Send packet: sends packet on specified ports (send ports of packet), return false if failed.

```
bool pLayer.send(Packet packet);
```

- Port Active: returns true if the port in the input is active.

```
bool pLayer.portActive(byte port);
```

5.4.2 Examples

```
//pLayer object is already defined in template.

//this forwards any received packet to port two if the port is connected
Packet thispacket;
pLayer.receive(thispacket);
if(pLayer.portActive(2)) {
    thispacket.addSendPort(2);
}
pLayer.send(thispacket);
```

5.5 Display Class

The Display class is a very simple wrapper used to display text on the screen and serial console at the same time. It's a global object that is created once and should never be deleted, it's already defined in the template as "display" but as a pointer. The class has following member-functions:

5.5.1 Member Functions

- Constructor: creates a new Display object that handles all displays.

```
Display* display = new Display();
```

- Display: prints string s to the display and serial console. \n is used as an escape character for a line-break.

```
void display->display(String s);
```

5.5.2 Examples

```
//display object is already defined in the template AS A POINTER

display->display("Hello World");
//Trick to display a number in hex, bin or dec:
display->display(String(27,DEC)); //14
display->display(String(27,HEX)); //1B
display->display(String(27,BIN)); //11011
```

5.6 IdleDisplay Class

The IdleDisplay class is used to print strings to the screen, that should stay there even if the device is idle. It's a global object that is created once and should never be deleted. It's already defined in the template as "idleDisplay" but as a pointer. In the template the MAC-address is always printed if the device is idle.
For example if a ping message is received and you want to display it for 10 seconds you can use **void IdleDisplay→changeString("Ping", 10);** The class has following member-functions:

5.6.1 Member Functions

- Constructor: creates a new IdleDisplay object that handles all displays while the device is idle (while waiting for a new packet).

```
IdleDisplay* idleDisplay = new IdleDisplay(Display* _display,
String defaultString);
```

- Display: needs to be called inside the main loop to re-display the idle default string

```
void idleDisplay->display();
```

- Change default: changes the string which is printed by default.

```
void idleDisplay->changeDefault(String defaultString);
```

- Change String: changes the string which is printed to a non-default string until enableDefault() is called.

```
void idleDisplay->changeString(String nondefaultString);
```

- Change string: changes the string which is printed to a non-default string for displayTime seconds.

```
void idleDisplay->changeString(String nondefaultString,
unsigned int displayTime);
```

- Change back to default: changes the string, which is printed, back to the default string.

```
void idleDisplay->enableDefault();
```

- Disable timeout: disables any timeout set by the changeString() - function.

```
void idleDisplay->disableTimeout();
```

5.6.2 Examples

```
//display object is already defined in the template AS A POINTER

idleDisplay->changeDefault("Hello World");
Packet thispacket;
pLayer.receive(thispacket);
byte src_mac = thispacket.returnByte(1);

//If a packet by 0x04 is received, print it for 10 seconds.
//If a packet is received by another mac, then go back to the default string.
if(src_mac == 0x04){
    idleDisplay->changeString("Recieved Packet by 4", 10);
}
else {
    idleDisplay->enableDefault();
}
```

6 The P&S Course

First, let's have a look what such a course could look like. These are only ideas and concepts, that we think are nice to visualise using our board.

6.1 Outline

The course will last for around 12 weeks at around 4 hours a week. (4KP) Students are encouraged to work everything out by themselves and find their own creative ways to solve the problems.
There will be three different kinds of devices, namely: routers, switches and hosts. They are all actually implemented on the same micro-controller but they have to be programmed in a different way. Here is some outline of what such a course may look like:

- Lesson 1: Introduction to Arduino, C++ and router class
- Lesson 2: Introduction to LINK-Layer/DHCP/ARP, Begin LINK project
- Lesson 3: Work on LINK project
- Lesson 4: Work on LINK project
- Lesson 5: Finish LINK project
- Lesson 6: Introduction to Transport-Layer, Begin TCP project
- Lesson 7: Work on TCP project
- Lesson 8: Work on TCP project
- Lesson 9: Work on TCP project
- Lesson 10: Finish TCP project
- Lesson 11: Introduction to NAT, Begin NAT project
- Lesson 12: Work on NAT project
- Lesson 13: Finish NAT project - hackathon

There will be three projects as seen in the list. Each of the projects will have their own short introduction. Let's go more into detail what these projects could be exactly:

6.2 LINK Project

6.2.1 Link-Layer Forwarding

This first project is about how the link-layer and Ethernet switches work. So students will learn how switches store MAC addresses and how they build their forwarding table. Each switch has four ports: port numbers = {1,2,3,4}. They then need to build forwarding tables upon receiving frames. They do that by storing which source MAC-address has been received on which port. Whenever a new frame with that same destination MAC-address arrives, the switch knows which port to forward it to. [1] As an example:

- 1. Frame with DEST_ADD = 3 arrives
  $\Longrightarrow$ Frame is forwarded to all ports
- 2. Frame with SRC_ADD = 3 arrives on PORT = 2
  $\Longrightarrow$ SRC_ADD = 3 at PORT = 2 is stored in table
- 3. Another frame with DEST_ADD = 3 arrives
  $\Longrightarrow$ Frame is forwarded to PORT = 2

To make it a bit more challenging, the students will then start to program the hosts and the router. They will learn how the IP to MAC binding works by implementing ARP[2] and DHCP[3] and are allowed to use their own simplified versions of these protocols or invent their own. The suggested way to use the headers would be:

Dest Mac | Src Mac | Type | Check Sum || ... | ... | ... | ... | ...
Where Dest Mac = byte0, Src Mac = byte1 and so on.

- Type = 0: Normal Frame
- Type = 1: DHCP Message
- Type = 2: ARP Message
- Type = 3: SP Protocol (if implemented)

6.2.2 DHCP Message

A DHCP message could have the following form: (Type = 1) [3]

Dest Mac | Src Mac | Type | Check Sum || DHCP Type | DHCP ip | ...

- DHCP Type = 0: DISCOVER (DHCP ip = none, Dest Mac = 0xff)
- DHCP Type = 1: OFFER (DHCP ip = offered ip)
- DHCP Type = 2: REQUEST (DHCP ip = requested ip)
- DHCP Type = 3: ACK (DHCP ip = none)
- DHCP Type = 4: NACK (DHCP ip = none)

6.2.3 ARP Message

An ARP message could have the following form: (Type = 2) [2]

Dest Mac | Src Mac | Type | Check Sum || ARP Type | ARP ip | ...

- ARP Type = 0: REQUEST (ARP ip = requested ip, Dest Mac = 0xff)
- ARP Type = 1: REPLY (ARP ip = request ip, Src Mac = searched Mac)

After that we will take a look at the importance of a spanning tree for the link layer by connecting them in a loop and look how the network behaves strange. We're not going to implement the SP protocol but students are always encouraged to do so.

6.2.4 IP-Packets

An IP-packet[4] could then have the following structure:

LINK HEAD || Dest IP | Src IP | TTL | Next Protocol || ... | ... | ...

6.3 IP Project (Alternative for LINK Project)

As an alternative to the link project: This project is about IP-forwarding[4] in a network. Here we need to think of our devices not as switches and host but as routers=yellow, red and servers=violet. Again since this would be a simplification we would ignore the link-layer and just forward IP-packets to one of the 4 ports. This is what the headers could look like:

Dest IP | Src IP | TTL | Next Protocol || ... | ... | ..

6.3.1 Static Routes

At first students will hard-code a forwarding table for the network and learn how they have to make forwarding tables.

6.3.2 Distance Vector

In the second part of the project students will then implement the distance vector protocol.[5] They will figure out their very own way to transmit the forwarding information of one router to the next. To prevent flooding and huge loss inside the network it's better, if the router only sends its distance vector if it has changed, eg. if a port has been disconnected or connected.

6.4 TCP Project

Since this network has a rather high probability of a packet loss it would be a very nice opportunity to see TCP[6] in action. In this project students are

encouraged to invent a reliable transmission strategy. They will implement their own TCP completely form scratch. There is only a suggestion for the choice of headers:

LINK HEAD || IP HEAD || Dest Port | Src Port | Seq Nr | Type || ...

This project is meant to be similar to the reliable transmission project form the communication networks lecture but a bit more creative and free. By implementing TCP[6] completely by themselves, students are really forced to understand what they are doing.

6.5 NAT Project

This project is thought to be done in the future, when many more boards are available. The idea is to connect all the boards to each other. Just like home-networks, we're going to use NAT to do so. Each group will get their own public IP. To get this to work they would all need to agree on the headers first, which could be:

LINK HEAD || IP HEAD || Dest Port | Src Port | Seq Nr | Type || ...

The students will then need to program the pink router such that it receives outside traffic on port 4 and inside traffic on ports 1-3. The router will then have to translate between the two. Since the transport layer needs to be already implemented for this project, it is recommended to do the project in the following order: 1) link-layer, 2) TCP, 3) NAT.

The goal of this project is to have all the boards connected to each other and then have a little hackathon where students will try to hack into the other networks and exploit their loopholes while fixing their own.

7 Demo

In this section we'll see how our network actually behaves, when the link-layer is fully implemented by looking at the outputs of the console.

7.1 DHCP and ARP - Demo

This demo is basically the same as the LINK project in "Project Ideas" chapter but in action. The host (MAC=10=0x0a) wants to send a Ping-IP-packet to the destination IP: DEST_IP = 3.

First, the host (MAC=10) will start by connecting to the network, therefore a DHCP discover is sent:

```
19:38:53.046 -> My MAC: a
19:38:53.046 -> My IP: 0
19:38:53.120 -> DHCP: Sending DHCP-Discover
19:38:53.154 -> LINK: Sending to:ff
19:38:53.154 -> PHY: Sending: ff0a01000000010000000000000000000
19:38:53.154 -> to ports: 4,
```

The DHCP-server receives that message and responds by sending an offer:

```
20:30:43.021 -> My MAC: 1
20:30:43.021 -> My IP: 1
20:30:45.983 -> PHY: Receiving
20:30:45.983 -> PHY: Received: ff0a01000000010000000000000000000
20:30:46.021 -> on port: 2
20:30:46.021 -> DHCP: Received DHCP-Discover
20:30:46.021 -> DHCP: Sending DHCP-Offer: IP = 5
20:30:46.021 -> LINK: Sending to:a
20:30:46.807 -> PHY: Sending: 0a01010000050200000000000000000
20:30:46.844 -> to ports: 2,
```

Subsequently the host (10) receives the offer and sends a request: (You can see the ARP trying to send the buffered elements. That's because the actual Ping-IP-packet could not be sent, because neither DHCP nor the ARP table were ready. Anytime either of them update it tries to send all buffered elements.)

```
19:38:53.191 -> My MAC: a
19:38:53.191 -> My IP: 0
19:38:59.164 -> PHY: Receiving
19:38:59.164 -> PHY: Received: 0a01010000050200000000000000000
19:38:59.201 -> DHCP: Received DHCP-Offer: IP = 5
19:38:59.201 -> DHCP: Sending DHCP-Request: IP = 5
```

```
19:38:59.201 -> LINK: Sending to:1
19:38:59.201 -> ARP: Resend Buffer Elements (len = 1)
19:38:59.201 -> ARP: DHCP NOT READY
19:38:59.997 -> PHY: Sending: 010a01000005030000000000000000
19:38:59.997 -> to ports: 4,
```

The DHCP server responds by sending an acknowledgement:

```
20:30:46.844 -> My MAC: 1
20:30:46.844 -> My IP: 1
20:30:52.722 -> PHY: Receiving
20:30:52.760 -> PHY: Received: 010a01000005030000000000000000
20:30:52.760 -> on port: 2
20:30:52.793 -> DHCP: Received DHCP-Request: IP = 5
20:30:52.793 -> DHCP: Sending DHCP-ACK: IP = 5
20:30:52.793 -> LINK: Sending to:a
20:30:53.594 -> PHY: Sending: 0a01010000050400000000000000
20:30:53.594 -> to ports: 2,
```

The host (10) now has an IP address and tries yet again to send the ping packet, but first he has to know what MAC-address to send it to, so he sends an ARP-request to the broadcast MAC-address first.

```
19:39:00.034 -> My MAC: a
19:39:00.034 -> My IP: 0
19:39:05.936 -> PHY: Receiving
19:39:05.936 -> PHY: Received: 0a01010000050400000000000000
19:39:05.936 -> on port: 4
19:39:05.974 -> DHCP: Received DHCP-ACK: IP = 5
19:39:05.974 -> ARP: Resend Buffer Elements (len = 1)
19:39:05.974 -> ARP: Sending ARP-Request: Who has IP: 3 ?
19:39:05.974 -> LINK: Sending to:ff
19:39:06.765 -> PHY: Sending: ff0a02000305010000000000000000
19:39:06.765 -> to ports: 4,
```

The other host with IP-address 3 responds with an ARP-resonse:

```
20:19:28.498 -> My MAC: 3
20:19:28.498 -> My IP: 3
20:19:41.273 -> PHY: Received: ff0a02000305010000000000000000
20:19:41.273 -> on port: 4
20:19:41.273 -> ARP: Received ARP-Request: Who has IP: 3 ?
20:19:41.273 -> ARP: Sending ARP-Response: I have IP: 3 at MAC: 3
20:19:41.273 -> LINK: Sending to:a
```

```
20:19:42.108 -> PHY: Sending: 0a030200030302000000000000000000
20:19:42.108 -> to ports: 4,
```

Finally the host (10) can send the ping packet: (ARP: Forwarding is actually looking up the destination MAC-address for the given destination IP-address in the ARP-table.)

```
19:39:06.802 -> My MAC: a
19:39:06.802 -> My IP: 5
19:39:16.153 -> PHY: Receiving
19:39:16.191 -> PHY: Received: 0a030200030302000000000000000000
19:39:16.191 -> on port: 4
19:39:16.191 -> ARP: Received ARP-Response: I have IP: 3 at MAC: 3
19:39:16.191 -> ARP: Forwarding IP: 3 to MAC: 3
19:39:16.191 -> LINK: Sending to:3
19:39:17.011 -> PHY: Sending: 030a0000030500ff0000000000000000
19:39:17.011 -> to ports: 4,
```

The other host (3) receives the ping and wants to send a pong answer, but there is no entry in the ARP-table yet. Therefore he has to send an ARP-request first:

```
20:19:42.108 -> My MAC: 3
20:19:42.108 -> My IP: 3
20:19:51.380 -> PHY: Receiving
20:19:51.417 -> PHY: Received: 030a0000030500ff0000000000000000
20:19:51.417 -> on port: 4
20:19:51.417 -> LINK: Received:
20:19:51.417 -> 030a0000030500ff0000000000000000
20:19:51.451 -> APP:PING by 5
20:19:51.451 -> ARP: Sending ARP-Request: Who has IP: 5 ?
20:19:51.451 -> LINK: Sending to:ff
20:19:52.235 -> PHY: Sending: ff030200050301000000000000000000
20:19:52.235 -> to ports: 4,
```

The Host (10) now responds with an ARP-response:

```
19:39:17.047 -> My MAC: a
19:39:17.047 -> My IP: 5
19:39:26.371 -> PHY: Receiving
19:39:26.408 -> PHY: Received: ff030200050301000000000000000000
19:39:26.408 -> on port: 4
19:39:26.408 -> ARP: Received ARP-Request: Who has IP: 5 ?
19:39:26.408 -> ARP: Sending ARP-Response: I have IP: 5 at MAC: A
19:39:26.408 -> LINK: Sending to:3
```

```
19:39:27.234 -> PHY: Sending: 030a02000a05020000000000000000
19:39:27.234 -> to ports: 4,
```

In the end the other host (3) can send the pong answer:

```
20:20:01.465 -> My MAC: 3
20:20:01.465 -> My IP: 3
20:20:01.575 -> PHY: Receiving
20:20:01.610 -> PHY: Received: 030a02000a05020000000000000000
20:20:01.610 -> on port: 4
20:20:01.644 -> ARP: Received ARP-Response: I have IP: 5 at MAC: A
20:20:01.644 -> ARP: Forwarding IP: 5 to MAC: A
20:20:01.644 -> LINK: Sending to:a
20:20:02.435 -> PHY: Sending: 0a03000005030ffe0000000000000000
20:20:02.473 -> to ports: 4,
```

The Host (10) receives the pong response on the application layer (APP):

```
19:39:27.270 -> My MAC: a
19:39:27.270 -> My IP: 5
19:39:36.536 -> PHY: Receiving
19:39:36.536 -> PHY: Received: 0a03000005030ffe0000000000000000
19:39:36.572 -> on port: 4
19:39:36.572 -> LINK: Received: 0a03000005030ffe0000000000000000
19:39:36.572 -> APP:PONG by 3
```

7.2 Errors, Problems and Fixes

7.2.1 Device failed to connect

A fatal error occurred: Failed to connect to ESP32: Timed out waiting for packet header.

Whenever you program the device you need to hold the button at the bottom of the device. Else, the device cannot be reached. **See Section 4.2.4 - Programming Guide**

7.2.2 Device does not light up

Error
Sometimes there is an issue where the screen of one device doesn't light up and doesn't do anything although the power is connected.

Fix
There is a problem during the startup of the device. Disconnect all the ports and turn the power off and on again. You can also try to only unplug port 1 first. Also monitor the serial output of the device when doing so.

7.2.3 Device does nothing

Error
A device just sits there, idle.

Fix
This could be a problem during the startup of the device. Disconnect all the ports and turn the power off and on again. Also monitor the serial output of the device.

7.2.4 Large Packet Loss

Problem
When multiple frames are propagating through the network, there is a huge chance for that frames are lost. This happens when for example switch 2 sends to switch 3 while switch 3 wants to send to switch 2. This will trigger the timeout and the packet will be dropped.

8 Conclusion

In this project, we found out, that making a simple network from micro-controllers is indeed possible. We were able to prepare an entire demo network consisting of 16 devices, all connected to one another. In addition we implemented a set of classes, that drastically simplify the programming workflow for anyone wanting to program these devices.

We've also seen that making such a setup is a lot effort, as there are more than 128 connections to solder per setup and the entire board needs to be built. Considering the many hours invested in building just one setup, it will probably take a lot of effort and/or money to create many of them.

Let's consider how useful this setup actually is for teaching. On the one hand, this network can be used nicely to visualize, what is happening inside any local-area-network. Watching the packets propagating through the network is very satisfying. On the other hand, our network was overloaded very quickly and even with just two hosts sending at the same time, almost all data was lost, which could be considered not a bug, but a feature! This would make for very nice setup to implement a reliable transport protocol, like TCP.

There's also room for improvement: The just mentioned bug could be fixed by completely restructuring the way the devices are programmed. The ESP32 has a dual core. Figuring out how one of these cores could be used only for receiving and the other one for processing and sending could drastically improve the performance of the device and the network.

An other aspect, which could be improved, is the programming process. Currently every device has to be programmed separately, which, when programming 16 devices, is a big hassle. Also having to type the correct MAC-address for every device allows for a lot of mistakes to be made. This could maybe be improved by writing a python script, which automatically detects the devices and programs all devices automatically while also setting a different MAC-address for each one.

In the future there are many more things, which could be implemented in addition to the current setup. First of all, to make a P&S course possible, many more of these setups are necessary.

Also taking advantage of the build in Wi-Fi module of these device could enable a direct connection from out model-internet to the real internet. This could also allow all of these setups to be connected to each other.

Using Wi-Fi, another idea would be to have all devices connected to a raspberry pi (or any Linux) server from which all the console logs could be read or from which it would be possible to send inputs to any device. The students

could then connect to this server via their web-browser (for example).

Instead of having every student installing the Arduino software, libraries and drivers a raspberry pi could be directly connected to each setup via USB. This raspberry pi would then run a simple web-tool, such that students can very easily program their devices on a web-browser.

The possibilities for the future of this project are endless, which is also why we had to limit this project to where we are now. A lot could still be improved and even more could be added. At least, after finishing this project, we know where to start.

References

[1] Laurent Vanbever. *Communication Networks - Lecture*, Feb 2020. (Accessed on 03/06/2020).

[2] David C. Plummer. *RFC 826 - An Ethernet Address Resolution Protocol*. https://tools.ietf.org/html/rfc826, Nov 1982. (Accessed on 10/03/2021).

[3] R. Droms. *RFC 2131 - Dynamic Host Configuration Protocol*. https://tools.ietf.org/html/rfc2131, Mar 1997. (Accessed on 10/03/2021).

[4] University of Southern California Information Sciences Institute. *RFC 791 - Internet Protocol*. https://tools.ietf.org/html/rfc791, Sep 1981. (Accessed on 10/03/2021).

[5] S. Deering D. Waitzman, C. Partridge. *RFC 1075 - Distance Vector Multicast Routing Protocol*. https://tools.ietf.org/html/rfc1075, Nov 1988. (Accessed on 10/03/2021).

[6] University of Southern California Information Sciences Institute. *RFC 793 - Transmission Control Protocol*. https://tools.ietf.org/html/rfc793, Sep 1981. (Accessed on 10/03/2021).

Appendix

8.1 Implementation of the Physical Layer

8.1.1 Transmission of Bits

Each port of a device consists of two connections: Tx (write only, voltage=low per default) and Rx (read only). Let's consider a sender (A) and a receiver (B). The senders Tx is always connected to the receivers Rx and vice versa. The transmission now happens in three steps:

1) Sender Initialization
The sender, A starts a transmission by changing Tx(A) from low to high.

2) Receiver Detection
This also changes the receivers Rx(B) to high, which triggers a hardware interrupt, which setts a flag, that a message is incoming. In the next iteration of the main loop the receiver will start to receive the message by sending a clock signal on Tx(B).

Exception: The receiver does not respond - After a short amount of time ($\approx 1000ms$) this will trigger a timeout and the packet will then be discarded by the sender.

3) Transmission
The sender now detects the clock signal on Rx(A) = Tx(B) and he sets Tx(A) on every positive flank to the next transmitted bit. The receiver on the other hand reads Rx(B) = Tx(B) on every negative flank. The actual packet is sent after the preamble = {1,0,1,0}.

Exception: The preamble is not {1,0,1,0} - This will trigger an invalid message exception an the packet will be discarded by the receiver.

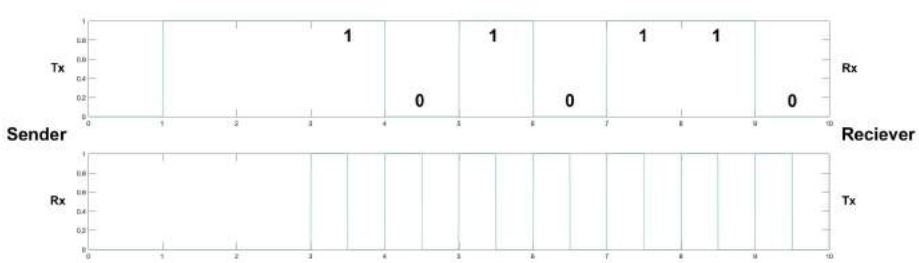

Figure 6: Transmission of: {1,0,1,0,1,10}

4) Ending the Transmission
The length of packet is fixed in **Definitions.h** as MESSAGE_SIZE in bytes. So after all the bytes are transmitted the sender simply stops sending an the receiver simply stops receiving.

Exception: The sender and receiver have different message sizes - If the sender has a bigger message size, then the receiver will stop the clock to soon and will only receive part of the message while the sender runs into a timeout.
If the sender has a smaller message size, then the last bits will most likely be received as zeros but it is not guaranteed, since the sender could interpret the clock of the receiver as incoming message and start sending a clock signal.

8.1.2 Detection of Port Connections

Every Rx has a built-in pull-up resistor. Therefore when nothing is connected to it, Rx reads high. Tx is initially set to low.

If two devices are connected the Rx of one device is always connected to the Tx of the other one. So as soon as two devices are connected their Rx will be pulled to low. Therefore as soon as any Rx is set to low, it is assumed that the port is connected and will be activated inside the PhysicalLayer object.

Now a change on Rx from low to high could mean two cases:
1) A message is incoming.
2) The port has been disconnected.

To detect which case we have, the devices treats any change from low to high as an incoming message, but if the message is invalid, it assumes the port to be disconnected and deactivates it. If the message was actually invalid, but the port is still connected, then the device will detect, that Rx is set to low again and will automatically activate the port again.