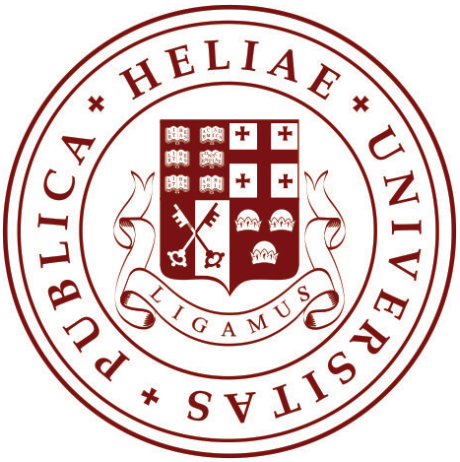

IoT with the Raspberry Pi under the LINUX OS, course script

Prof.-Dr. Ulrich Hauser-Ehninger

Ilia University Georgia, Tbilisi

2019

Abstract

The course *IoT with the Raspberry Pi under the LINUX OS* looks at the IoT in general and then focuses on the usage of the Raspberry Pi[1]. As the dominant OS (**O**perating **S**ystem) used to drive the Pi is LINUX[2], being productive with the Pi is almost impossible without fundamental knowledge of this OS. As communication is mostly done over the WWW (**W**orld **W**ide **W**eb), networking aspects and WWW-servers and -clients will be covered as far as necessary for the undestanding of the course content. Key points are:

- IoT (**I**nternet **o**f **T**hings)
- WoT (**W**eb **o**f **T**hings)
- WWW
- LINUX
- Raspberry Pi
- Sensor networks
- IoT communication
- M2M (**M**achine to (**2**) **M**achine) communication
- REST (**RE**presentational **S**tate **T**ransfer) interfaces
- network protocols
- JS (**J**ava **S**cript)
- nodeJS[3]

[1] https://www.raspberrypi.org/
[2] https://www.linuxfoundation.org/projects/linux/
[3] https://nodejs.org/en/

Contents

List of Figures

Acronyms

AJAX **A**synchronous **J**ava Script **A**nd **X**ML. 69

API **A**pplication **P**rogrammer **I**nterface. 14, 15, 26, 27

ARM **A**corn **R**ISC **M**achines. 60

bash **b**ourne-**b**gain-**sh**ell. 33, 42, 43

BT **B**lue**t**ooth. 11, 12

CLI **C**ommand **L**ine **I**nterface. 18

CoAP **Co**nstraint **A**pplication **P**rotocol. 14, 70

CPU **C**entral **P**rocessing **U**nit. 22, 23, 26

CSS **C**ascading **S**tyle **S**heets. 45, 84, 87, 88

DHCP **D**ynamic **H**ost **C**onfiguration **P**rotocol. 62

DOM **D**ocument **O**bject **M**odel. 46, 47, 56, 89

DoS **D**istributed **D**enial **o**f **S**ervice. 15

FHGR **F**achhochschule **H**ochschule **G**rau**B**ünden. 65

FHS **F**ilesystem **H**ierarchy **S**tandard. 30

FTP **F**ile **T**ransfer **P**rotocol. 30

Gimp **G**nu **i**mage **m**anipulation **p**rogram. 1

GNU **G**NU's **N**ot **U**nix. 42

GPIO **G**eneral **P**urpose **I**nput **O**utput. 60

GUI **G**raphical **U**ser **I**nterface. 50, 60, 63, 64

HAL **H**arware **A**bstraction **L**ayer. 22, 23

HCI **H**uman **C**omputer **I**nterface. 8

HTML **H**yper**t**ext **M**arkup **L**anguage. 7, 15, 45, 46, 69, 73, 84–86

HTTP **H**yper**t**ext **T**ransfer **P**rotocol. 7, 14, 15, 27, 46, 68, 69, 71, 74, 75

HW **H**ard**w**are. 13, 16, 20, 22, 23, 29, 32, 76, 77

I2C **I**nter-**I**ntegrated **C**ircuit (IIC $\rightarrow I^2C$). 60

IoT **I**nternet **o**f **T**hings. i, 3–8, 10, 12–15, 47, 60, 61, 63, 67–71, 73, 76

IP **I**nternet **P**rotocol. 62, 63, 67

IR **I**nfra**r**ed. 6

JS **J**ava **S**cript. i, 45–58, 66, 71, 75, 84, 86

JSON **J**ava**S**cript **O**bject **N**otation. 7, 15, 49, 58, 75

JSON-LD **J**ava**S**cript **O**bject **N**otation-**L**inked **D**ata. 15

LoRa **Lo**ng **Ra**nge. 12, 13

LOS **L**ine **O**f **S**ight. 12

LTE **L**ong **T**erm **E**volution. 13

M2M **M**achine to (**2**) **M**achine. i, 10, 15, 73

MIDI **M**usical **I**nstrument **D**igital **I**nterface. 9

MMU **M**emory **M**anagement **U**nit. 23

MQTT **M**essage **Q**ueue **T**elemetry **T**ransport. 13, 14, 69–71, 75, 94

NFS **N**etwork **F**ile **S**ystem. 30

npm **n**ode **p**acket **m**anager. 58

OS **O**perating **S**ystem. i, 17, 18, 22, 46, 59–61

PAN **P**ersonal **A**rea **N**etwork. 11

QoS **Q**uality **o**f **S**ervice. 69

RAM **R**andom **A**ccess **M**emory. 27, 28

REST **RE**presentational **S**tate **T**ransfer. i, 14, 15, 71

SCI **S**ystem **C**all **I**nterface. 26

SFTP **S**ecure **F**ile **T**ransfer **P**rotocol. 30, 63

sh bourne-**sh**ell. 33, 63

SMB **S**erver **M**essage **B**lock. 30

SPI **S**erial **P**eripheral **I**nterface. 65

ssh **s**ecure **sh**ell. 27, 62–64

ssid **s**ervice **s**et **id**entifier. 62

SW **S**oft**w**are. 1, 15, 27

TCP **T**ransmission **C**ontrol **P**rotocol. 67, 68, 70

TLS **T**ransport **L**ayer **S**ecurity. 70

UDP **U**ser **D**atagram **P**rotocol. 14, 70

UI **U**ser **I**nterface. 18, 21, 64

URI **U**nique **R**esource **I**dentifier. 14

URL **U**nique **R**esource **L**ocator. 14, 68, 84

VCS **V**ersion **C**ontrol **S**ystem. 59

VM **V**irtual **M**achine. 18

WAN **W**ide **A**rea **N**etwork. 11, 12

WoT **W**eb **o**f **T**hings. i, 3, 5, 14, 71, 76

WWW **W**orld **W**ide **W**eb. i, 1, 5, 6, 14, 29, 30, 36, 41, 45

XHTML **E**xtensible **H**yper**t**ext **M**arkup **L**anguage. 85

Preface

The script was developed using free SW (**S**oft**w**are) as far as possible. The reasons are:

- The Author is about the opinion that open SW is important in academic education, to avoid restricting students to commercial products during this period of time.
- Students should have the chance to use SW by themselves. This is most easy if
  - SW does not cost any money.
  - the lecturer/teacher/coach uses the same SW as the student.

The script was written using LaTeX[4]. The development environment is TeXStudio[5]. Images are taken from sources under the CC license model[6]. Image processing was done with Gimp (**G**nu **i**mage **m**anipulation **p**rogram)[7]. The author tried to reference sources wherever known. This applies as well for sources under CC. If copyright infringements are found, please do not hesitate to contact the author at ulrich.hauser@htwchur.ch to correct the mistake.

Most of the IoT knowledge was taken from the book *building the web of things*[8][1]. Information about LINUX[9] is drawn from a more elaborated lecture, given by the author at FHGR[10]. The Raspberry Pi itself[11] is widely documented in the WWW.

[4] https://www.latex-project.org/
[5] https://www.texstudio.org/
[6] http://www.creativecommons.ch/
[7] https://www.gimp.org/
[8] https://webofthings.org/book/
[9] https://www.linuxfoundation.org/projects/linux/
[10] https://www.fhgr.ch/
[11] https://www.raspberrypi.org/

The script is not exhaustive. Therefore, Students are required to take notes in the lectures themselves to complete the information.

Chapter 1

Introduction to the IOT

1.1 What is the IOT and where did it come from

There is no crisp definition of the IoT. One possible way to put it is[1]:

Definition 1 *The Internet of Things is a system of physical objects that can be discovered, monitored, controlled, or interacted with by electronic devices that communicate over various networking interfaces and eventually can be connected to the wider internet.*

The *physical object* can be everything between tiniest of sensors or actuators like RFID tags to big complex control systms like cars or even a standard PC. One central aspect of IoT is that the things themselves communicate with each other and not necessarily with a human. This communication is not seamless. There are plenty of protocols[1] communicating over a plethora of physical networks and network technologies, especially in the wireless area. Up to now, there is no end to this development and new protocols crop up everywhere (see fig. 1.1[2]). One approach to address the issue of heterogeinity is the WoT, trying to connect things using established and well-known technologies.

Mark Weiser, engineer at XEROX Parc, said[1]:

> The most profound technologies are those that disappear.

[1] https://en.wikipedia.org/wiki/List_of_automation_protocols
[2] https://xkcd.com/927/

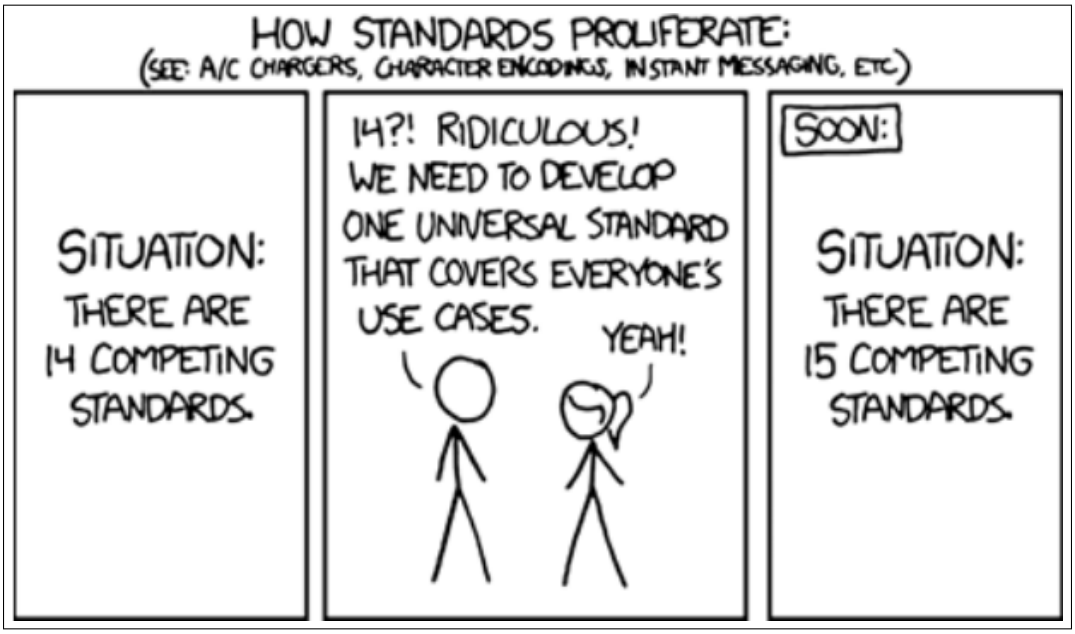

Figure 1.1: standardisation

This is, what the IoT is all about.

The term itself came up in 1999. Things developed away from the public until 2013 the term became very prominent in the WEB 2.0 movement. Forecasts back in 2014 estimated the number of IoT devices to be between 25 and 50 billion ba 2020. Today it looks like the number will be closer to the upper limit or even higher. Connected to this development is a growing demand in energy and other resources not to be underestimated.

1.2 The WOT and its relation to the IOT

As mentioned above, one of the challenges in the IoT-area is the number of technologies, protocols and 'standards' evolving around the subject. The companies developing IoT-devices developed new protocols or wireless technologies as the need arose. Huge players tried to establish their protocols as new standards. But keeping control over a proprietary protocol never worked well in the internet domain. The standards are more or less all open standards, developed largely not by industry but by academics. Why not use established standards

in the IoT-world as well? This is essentially what the WoT tries to accomplish. The main challenge is that IoTdevices have quite different requirements compared to standard WWW-devices. The main important ones are related to energy issues and issues of data rate and data amount. A sensor node at a remote location needs to consume as little energy as possible, transfer data at the lowest possible cost concerning energy and bandwidth aspects to be able to live on battery as long as possible or even draw all energy necessary from the sources available locally. On the other hand, sensors, but even more so actuators, need to be able to receive data, perhaps with quite low response times. This leads to receivers with as little energy consumption as possible as well as being receptive for as much a percentage of the time as posible.

The solution is to use well-known protocols where possible and combine them with technologies fullfilling the restricive requirements of the IoT. This is –in essence– abstracting the demands of the IoT behind the upper layers of the communication model.

1.3 Applications of the IOT

It is rather irrational to have a section about this topic as each day new applications for the IoT arise. But still, a small list of the most prominet areas shall be given.

- Monitoring of parameters in distributed contexts like
  - biology
  - geology
  - energy
  - manufacturing
  - meteorology
  - community (traffic, people commute etc.)
  - ...

- control in distributed contexts like
  - home automation
  - manufactoring
  - marketing
  - logistics and delivery
  - access technologies (entrance systems etc.)
  - ...

Plenty of information can be fount in the WWW[3,4,5,6,7,8,9,10][1].

1.4 Explore real IOT devices

Some IoT devices are directly visible in the WWW[11]. This is a website set up by the authors of the book this part of the lecture is based on. The devices accessible are (all sitting on a raspberry pi):

- an IR (Infrared) sensor
- a temperature sensor
- a camera
- a LCD display

[3] http://www.wired.com/2013/02/budweiser-red-light/
[4] http://www.altomagazine.com/newsdetails/travel/hotels/dom-prignon-at-the-press-of-a-button-431
[5] http://theinspirationroom.com/daily/2012/evian-smart-drop/
[6] http://postscapes.com/internet-of-things-investment
[7] http://en.wikipedia.org/wiki/List_of_wireless_sensor_nodes
[8] http://d3s.disi.unitn.it/projects/torreaquila
[9] http://www.cs.berkeley.edu/~culler/papers/wsna02.pdf
[10] http://www.mdpi.com/1424-8220/9/6/4728/htm
[11] http://devices.webofthings.io/

1.4.1 The human interface using a browser

The devices can be accessed via a web browser. The values can be explored, the LCD can be set. The display can be photographed with the camera. If the respective ports are open, the image can be downloaded and will be displayed.

1.4.2 The machine interface

A browser will normally send a HTTP (**H**yper**t**ext **T**ransfer **P**rotocol) header information to accept *text/html*. This indicates to the server that it is supposed to return something in HTML (**H**yper**t**ext **M**arkup **L**anguage)-code. If this information is replaced by *application /json*, the server (if programmed accordingly) should respond with information, coded by JSON (**J**ava**S**cript **O**bject **N**otation). To create the correct resquest, extra tooling is needed. There are browser plugins or applications like postman[12,13] which allow to generate a wide variety of requests and then show the responses.

1.4.3 Real time data-Event driven behaviour

Exploring a device is done by sending requests and evaluating the responses. In the IoT it would often be desirable to have an inversion of the control flow. This means that the device would start the communication by itself, because something was registered by some sensors or something reached a certain state. Normal HTTP did not support this, but the standard HTML5 introduced *websockets*[14] and uses HTTP upgrading[15] to allow communication to be initiated by the server itself.

Using this technology, real-time event driving is possible. It allows to very quickly after a certain condition is met, the associated server issue information which –in the client– causes an event to be triggered. The client can react to this event directly.

[12] https://www.getpostman.com/
[13] https://insomnia.rest/
[14] https://en.wikipedia.org/wiki/WebSocket
[15] https://en.wikipedia.org/wiki/HTTP/1.1_Upgrade_header

Without communication inversion, the client needs to use *polling*. Polling is a concept where a client requests information from a server in regular intervals to get to know about the state of the server. This, obviously, need much more data to be transferred and is less responsive compared to event-driven behaviour.

The thing explored above does not use this technology by the time of the writing of this script.

1.5 Networks of things

The IoT is a network of things, communicating with each other and communicating with humans over HCI (**H**uman **C**omputer **I**nterface)'s. If one wants to understand the IoT, a basic knowledge of networks and their properties is necessary.

1.5.1 Topologies

A *topology* of connections is the way in which the partners are connected to each other. An overview is given in Figure 1.2[16]. In fig. 1.3, a part of the internet is visualised as a map[17].

[16] https://simple.wikipedia.org/wiki/Network_topology

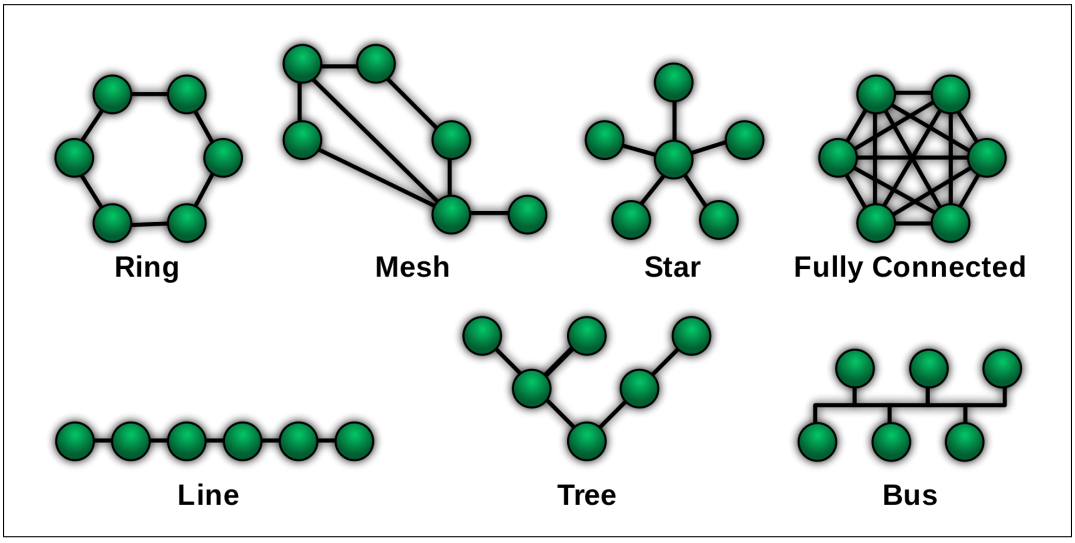

Figure 1.2: Network topologies

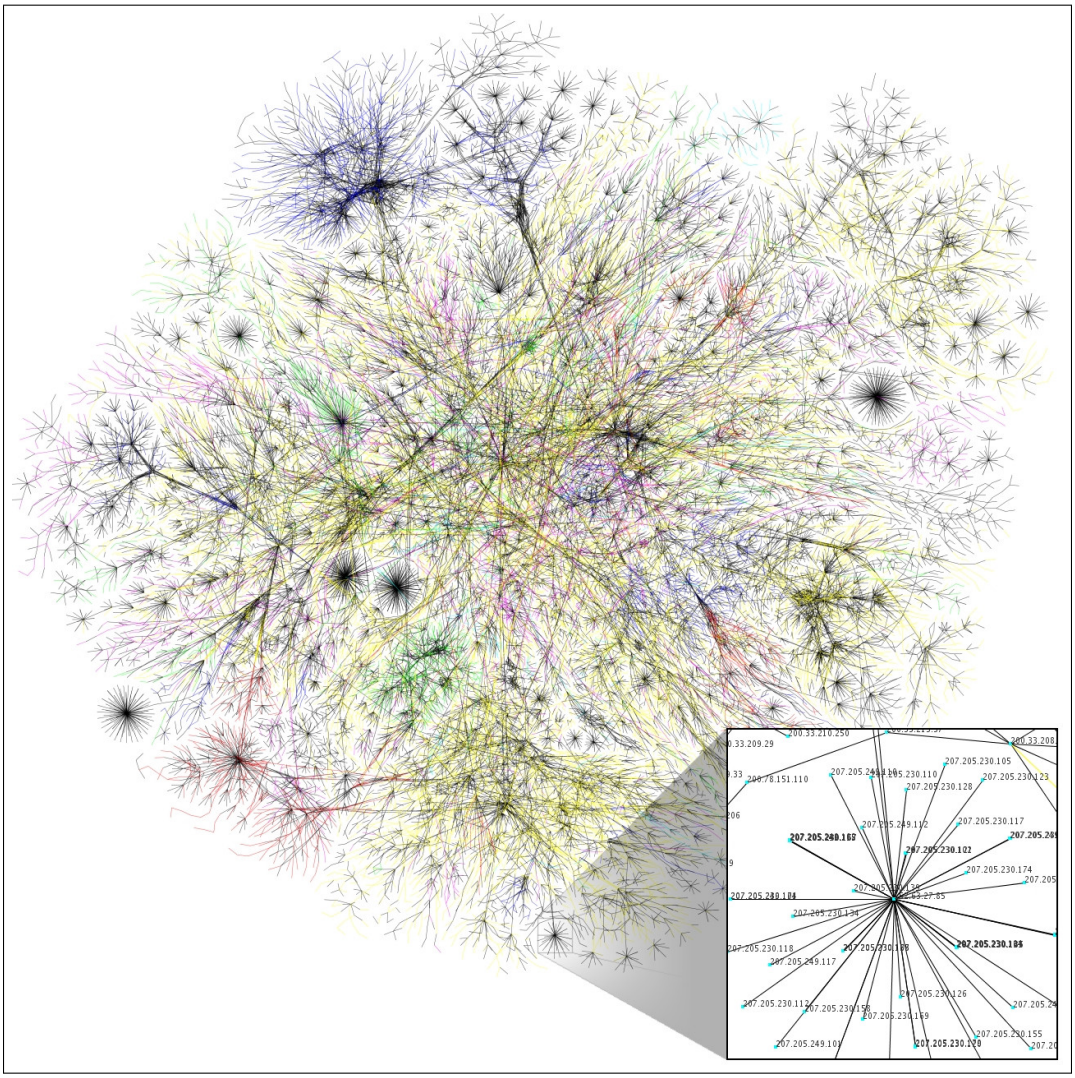

Figure 1.3: The internet, partial map

1.5.1.1 Ring

For electrical reasons, the ring used to be one of the most prominent network topologies for a long time[18]. Networking, including the internet, started out mostly as token ring networks.

1.5.1.2 Linear

Linear networks are quite basic. Often, they are unidirectional as well. Often, this is called *daisy chaining* as well. MIDI (**M**usical **I**nstrument **D**igital **I**nterface), used for music since

[17] https://en.wikipedia.org/wiki/Network_topology#/media/File:Internet_map_1024.jpg
[18] https://en.wikipedia.org/wiki/Ring_network

the 1980s is such a network.

1.5.1.3 Star

The most common network topology these days is the star of starts configuration. Ethernet clients are connected to a single gateway which itself connects to a switch or another gateway and so on, building *stars*[19] of devices, consisting of stars of devices, therefore named *star of stars*.

It is strictly hierarchical and allows hierarchical addressing over several levels of address spaces. If this network is strictly star or star of stars, only one way of communication between two partners is possible. As the failure of a single device could jeopardise a connection, redundency is normally built in the system. Gateway between two systems are implemented twice or more. Then –if one gateway or switch fails or is overloaded– another device can take over the necessary data transfers. But still the logic topology would be a star of stars.

1.5.1.4 Tree

Logically, a star of star network can be looked at as being a tree. The internet is one single gigantic tree.

1.5.1.5 Bus

Measurement systems are often arranged this way. Although today often connected to ethernet, *IEEE-488*[20] (often called HP-IB) used to be such a bus. As the measurement devices often talked to each other (M2M), it can be looked at as predecessors of IoT devices.

1.5.1.6 Mesh

IoT devices often use radio links of short range to save energy. This leads to the problem that, in larger areas, nodes may not be able to connect to the central data link device. Then,

[19] https://en.wikipedia.org/wiki/Star_network
[20] https://en.wikipedia.org/wiki/IEEE-488

a mesh network is a big advantage as nodes can use other nodes as links or relays towards the gateway. This way, a huge network can operate on quite small radio ranges. ZigBee[21] is such a network, but BT (**B**lue**t**ooth) in its last revisions is able to build meshes as well.

1.5.2 Layers

Networks are layered. The lowest layer normally is the physical medium. The higher the layer the mor abstract its working. The highest layer is the application using the network. A reference for networks is the OSI model[22].

1.5.3 Evaluation and decision

1.5.4 PANs

PAN (**P**ersonal **A**rea **N**etwork)s are local networks bridging distances from only mm or cm up to some m or dozens of meters. They are hugely use in home automation, logistics. Most of them are used in urban areas where distances are small and data needs to be kept local if possible.

1.5.5 WANs

WAN (**W**ide **A**rea **N**etwork)s span wide areas up some dozens of km. They are used for sensor networks, often in rural areas to be able to monitor conditions across large distances.

1.5.6 Examples

1.5.6.1 Bluetooth

BT is a typical network protocol for short distances. In its beginnings it was intended to reduce the cable chaos around computers. Printers, mice and other devices were supposed to

[21] https://zigbee.org/
[22] https://en.wikipedia.org/wiki/OSI_model

communicate through this interface. Data rates where quite low these days. Although data rates through the newest standards are much higher, still is not a connection type intended for the big data transfer.

Since version 4.0, BT supports low energy modes. As well starting with version 4.0, mesh networking is supported. Both aspects increased the for IoT applications to use BT significantly.

1.5.6.2 ZigBee

ZigBee is used in the same area as BT but was intended for IoT from the start. It was designed using low energy and mesh networking from the start. Ther are three main frequency bands, 915MHz and 868MHz, depending on regulatory limitation of the area of use and 2.5GHz worldwide. The intended range is 10-100m although outside with LOS (**L**ine **O**f **S**ight), up to 1500m can be obtained. There are proprietary ZigBee long range versions for much higher distances as well[23].

1.5.6.3 LoRa

LoRa (**Lo**ng **Ra**nge)[24] WAN is a technology to bridge large distances (typically 10 to 15km). It allows devices to live on single batteries for a number of years as the power consumption is very low. On the other hand, data rates are low, typically 10's of bytes per second up to some kBaud. This is ideal for sensor networks, transmitting just small bits of data with a large time inbetween transmissions. There are a lot of applications like mintoring in the IoT area where this technology is ideal.

[23] https://www.digi.com/products/embedded-systems/rf-modules/sub-1-ghz-modules/xbee-pro-900hp

[24] https://en.wikipedia.org/wiki/LoRa

1.5.6.4 SigFox

Technically, although being quite different to LoRa, *SigFox* has the same target application, long distance, low bandwidth, low energy. The business model is very different to LoRa. SigFox is proprietary, LoRa is open. Being direct competition SigFox seems to loose ground in recent times.

1.5.6.5 Cellular technologies

These technologies need support by the commercial mobile communications supplier as they partly use the technology provided there. and LTE (**L**ong **T**erm **E**volution)-M are examples. Together with 5G mobile communications big changes in IoT data transfer are to be expected during the coming years.

1.5.7 Protocols

The HW (**H**ard**w**are) technology is not everything needed to supply energy efficient data communication. The protocols defining the data exchange have a huge influence as well. It is important for energy efficiency to allow the link to be quiet for a long time. With WiFi e.g. this is not the case.

1.5.7.1 MQTT

MQTT (**M**essage **Q**ueue **T**elemetry **T**ransport) is a service based on the publish/subscribe pattern. A message delivery entity sends messages to a broker. 'Customers' can subscribe to listen to certain message types. Whenever a fitting message is issued to the broker it will inform the listening entities. MQTT knows different levels for the quality of service to give the guarantee (or not) that messages are delivered correctly.

1.5.7.2 CoAP

CoAP (**Co**nstraint **A**pplication **P**rotocol) –in contrast to MQTT– is placed on top of UDP (**U**ser **D**atagram **P**rotocol) to get rid of some amount of data traffic due to packet acknowledgement and keeping open connections. It is a request/response protocol and is based on REST [1]. It therefore has a lot in common with HTTP and REST.

The very small memory footprint in usage and the connectionless design help in its applicability in th IoT.

1.6 WOT architecture

We now leave the physical and logical structures of the networks and focus on the architecture of the WoT. There are four stages of using the connected things in the WoT:

- The *access* of devices through the technologies covered in section 1.5. The central aspect hier is how the API (**A**pplication **P**rogrammer **I**nterface)'s are designed to allow easy access of devices using well-known ways.
- Things need to provide means to lay open what they have to offer in a machine-readable way to make them *find*able.
- Things need to *share* data in a controlled, safe and secure way
- Tools can use the data found in the WoT to *compose* meaningful information from multiple sources of data.

1.6.1 Access

The WWW knows many techniques to access devices which are part of the internet. URL (**U**nique **R**esource **L**ocator)'s/URI (**U**nique **R**esource **I**dentifier)'s allow identification of devices and services on devices. *Websockets* are the base to use bidirectionally induced communication in web browsers. MQTT and CoAP help with lightweight protocols in

restricted environments. With HTTP, an application layer protocol to statelessly transfer data from one entity to another is at one's disposal. It helps with M2M communication as well through header attributes. REST API's support M2M communication representing states in a stateless surrounding. This is obtained by transmitting all information necessary for a process in each HTTP-message. HTML as the language of web-pages and JSON as a data format to transport structured data sets from small chunks of configuration data up to whole databases are languages for access by humand and by machines.

1.6.2 Find

To access devices the problems are mainly of syntactical nature. To find desired services, semantic information becomes more important. Marking parts of web pages (possibly using `schema.org`), linking to other data to get machine readable linked data, enhancing web pages with JSON-LD (**J**ava**S**cript **O**bject **N**otation-**L**inked **D**ata) to create the semantic web are all tools to help give data meaning. The same techniques can be used in the IoT to tell machines what given sensors or actors can deliver or do.

1.6.3 Share

The IoT would be meaningless without sharing data traveling through the internet. But for prober publushing of data, proper security means need to be considered. It is thought as one of the big threats, not only across the IoT devices, but across the internet and possibly even beyond it, that IoT devices might be (and have been) captured using malignant SW to do unintended things. Cameras have been captured in 2016[25],[26] to start DoS (**D**istributed **D**enial **o**f **S**ervice) attacks as they were so easily captured, available in large numbers and connected to the internet. The awareness about IoT security is still much too low, although it rises. More and more IoT devices come with security chips installed already or processors

[25] https://www.csoonline.com
[26] https://krebsonsecurity.com/2016/10

with security HW layers builtin[27].

1.6.4 Compose

Once meaningful data has been obtained, it can be used in conjunction to other data to crate new and meaningful knowledge. This is the huge market gaining momentum these years. It deals with mashups[28] (a word stemming from music[29]), industry 4.0 production, monitoring[30] and much more like smart cities etc., affecting all societies at all levels.

[27] e.g. https://www.arm.com/solutions/security
[28] https://en.wikipedia.org/wiki/Mashup_(web_application_hybrid)
[29] https://en.wikipedia.org/wiki/Mashup_(music)
[30] https://en.wikipedia.org/wiki/Industrial_internet_of_things

Chapter 2

Introduction to LINUX

2.1 Overview

LINUX –in general– is an OS with multiuser, multitasking and multiprocessing support. It even supports computer clusters like Beowulf[1] It started in 1991 as a project by Linus Thorvalds as a free alternativ to MINIX[2], which itself is an alternative to UNIX[3], all systems striving to be POSIX[4]-compliant OS's. The term LINUX is an acronym. Two following main exist:

- Linus Thorvald's UNIX
- Linus Thorvald's MINIX
- LINUX is not UNIX

Modern versions consist of more than 20 million lines of code and is published under the GNU General Public License[5]. Further details will be covered in section 2.3.2.

The first contact to LINUX in this lecture is supposed to be a practical one without having too much of a theoretical background obstructing one from having the first frustrations.

[1] https://en.wikipedia.org/wiki/Beowulf_cluster
[2] https://en.wikipedia.org/wiki/MINIX
[3] https://en.wikipedia.org/wiki/Unix
[4] https://en.wikipedia.org/wiki/POSIX
[5] https://en.wikipedia.org/wiki/GNU_General_Public_License

The author had his first contact with LINUX 1997 through a CD grabbed from a jumble sale stand in a book shop. Just two weeks later the system was up and running with a text console on a 80486 PC. Just to illustrate how far the system comfort progressed since then: A normal user with not more OS knowledge as a standard windows user now can set up a dual boot system with LINUX (.e.g. UBUNTU[6]) running in parallel to windows, with a comfortable graphical UI (**U**ser **I**nterface) in about an hours time.

To be able to use the same system with each student, VM (**V**irtual **M**achine)'s[7,8] are awsome tools to enable experiences without the danger of damaging a whole system and again and again spending hours of installing again and again. VirtualBox by Oracle will be the tool of choice in this script although every other VM able to host a LINUX system will be as sufficient as running on a native LINUX system will be.

2.2 First steps

LINUX-OS is started by running it from a VM or directly.

After the booting process, the user is asked to log in with a name and a password.

If the LINUX is non-graphical, the user now will be greeted by a console, the primary tool to interact with the system in this lecture. On systems with graphical UI, a terminal needs to bi started through the standard menu, which looks different in different distributions. The console cannot do more that accepting commands and printing answers. This limitation to entering text to make the system do certain things might be felt as a restriction at first, especially for users of windows or mac, used to just clicking around. After one gets used to it and one knows the most important commands, their interaction and interconnection, many of the everyday tasks can be accomplished much faster through a CLI (**C**ommand **L**ine **I**nterface).

The following command need to be issued exactly like given to get the same results. So

[6] https://ubuntu.com/
[7] https://www.virtualbox.org/
[8] https://wiki.qemu.org

playing around now is not a good idea.

2.2.1 First Commands

cd

cd / (keyed in and finished by hitting *Enter*) leads to the *root directory*. What is that and how do I know? LINUX does not know drive names. LINUX knows just a single gigantic directory tree starting with `/`. The *working directory* is the directory the actual commands are executed in and on. It normally is shown in the console *prompt*, the information preceding the blinking cursor. Before entering the command given above, the prompt showed `~` at the position of the working directory. This indicates the home directory, the place where each user will store all personal files.

ls

Now issuing cd home again changes into another directory.

ls shows the content of the working directory in a short and readable way. As we are in the directory `/home`, we should be given the names of the user accounts on this machine. Apart from these accounts there is a priviledged account, called `root`-account to administer the system.

pwd

cd takes us back to the home directory. But which one is that? pwd shows us, that it is a subdirectory of `/home`.

touch

touch somefile.txt will generate a file named `somefile.txt`. We can see this, issuing one more `ls`.

find

find /home > files.txt searches for all files in `/home` and pushes the result into a file named `files.txt`. cat files.txt will show the content of this file.

cat

top

top results in a well populated screen This command gives quite a lot of information about what is going on in the system. The number of tasks running, system utilisation, information about the most time consuming tasks are all shown and updated regularly. Even without performing high pressure tasks, quite a lot is going on here. q leaves this screen again.

2.2.2 A glimpse of the File system

As we have now seen the very first commands, it is time to look at the file system. Its role is quite different to the role in windows, where the file system is ther mostly to hold files. As mentioned above, LINUX knows a single directory tree. LINUX –from the user's point of view– doesn't know printers, network interfaces, graphics cards, USB nor any other HW. LINUX just knows files, even for the computer HW and the processes running at the moment. Generally speaking, a file in LINUX is *everything you can write to or read from.* proc

cd /proc, followed by `ls`, e.g. leads us to a very interesting directory. Many subdirectories with numbers as names are present. Each of this directory represents one of the running processes in the system. This is called a virtual directory (as there can be virtual files as well). These directories cannot be found on disk. They are there, created by the system in memory, when the associated information is asked for. Other files in the directory `/proc` are related to other system information. cat meminfo for example shows how the memory of the machine is used.

cd takes us 'home' again.

The following 'monster' command searches all files in the system starting with the letter `p` and looks for the content `notroot` inside the files. The result is stored in a file called `notroot.txt`.

```
find / -iname "p*" | xargs grep notroot >notroot.txt
```

This takes a while as the directory tree has hundreds of thousands of files. But now, the console is blocked by the running process. No prompt is visible any more.

Ctrl-C stops the command execution. this is called *break*.

The next trial:

```
find / -iname "p*" | xargs grep notroot >notroot.txt &
```

Now the promt is there again. But why does something write into the console, although we can write into the console ourselves? The ampersand (`&`) tells the command to run in the 'background'. It will run in parallel to everything what the console itself does. Therefore the

console is accessible to the user to issue new commands, but the command in the background will write to the console as soon as there is something to write. As the command issued above is supposed to write to a file anyway, it is not necessary to block the console. The problem is, that error message still are directed to the console not to clutter the normal output of the command.

ps shows the commands running in the console at the moment. This shows, that all the commands `find`, `xargs` and `grep` are active at the same time. `ps` as actually running command is listed as well..

We conclude that there is no problem with having multiple commands running i one console simultaneously.

A last example can show how powerful the concept behind the LINUX directory tree is: Having a graphical UI, Ctrl-AltF2 leads to a text console, without a graphical UI it is done by Alt-F2. After logging in, one can get back to the old console using Alt-F7 on most graphical UI's, Alt-F1 on text console systems.

ls -l >/dev/tty2 now seems to do nothing, but going back to the other console as before shows, that th output was channeled to the other console although `/dev/tty2` is supposed to be a file. There are two new concept here:

- There are several parallel consoles, even if the system is non-graphical. Not tested up to here but still possible: On the different consoles, different users can be logged in.
- In essence `/dev/tty2` is a file representing the console. Writing to the file is writing to the console, reading from the file is reading information that was typed on that console.

These first steps should have given a little teaser of LINUX. Some more basic understanding is stimulated in the next section.

2.3 Basics

2.3.1 Introduction

Users of computer systems would not like it, if they had to interact with a system differently when different HW would be mounted. Buying a new arddisk and because of that having to access files differently would be inacceptible. To solve this problem (among others) *operating systems* exist.

- An OS does HW abstraction to obstruct the details from the user. this is done by the HAL (**H**arware **A**bstraction **L**ayer).
- An OS manages resources like memory, hard disks, CPU (**C**entral **P**rocessing **U**nit) time etc. for the users. If a user program needs such a resource it asks for it in programmatical way. The OS the deals with the request. Generally, a means to deal with a request is called a *service*.

Therefore an OS can be described as a collection of services for the user/programmer to keep things HW independent. The OS 'sits' inbetween the HW and the user.

2.3.2 History

1. 1969: Start of the UNIX-development from conclusions from the MULTICs project by Ken Thompsen.
   (a) no closed Shop (punched card, given to an operator, no interaction between user and computer)
   (b) hierarchical file system
   (c) Multiuser
   (d) Multiprocessing
   (e) portable

Teaming up with Dennis Ritchie, Rudd Canady, Brian Kernighan

2. 1970: Name UNIX

3. 1972: New implemention in C (up to there too much assembly language)

4. later: Diversion into multiple branches (see fig. 2.1[9])

5. 1991: LINUX 0.99

6. 1994: LINUX 1.0

Today, several distributions are present, deveoped with different aims or due to disagreement over development issues. It is a very fractured landscape as can be seen in fig. 2.2[10]. The full image is of such detail that the names cannot be distinguished any more.

2.3.3 Structural overview

2.3.3.1 composition

In fig. 2.3, the basic laout of UNIX is shown. The device drivers –strongly dependent on the HW– are closely bound to the kernel and manage the data exchange between kernel and HW. The therefore form the HAL.

Most of the kernel is written in C. Some of the code very close to the HW needs to be written in assembly language and therefore will be dapted to fit the needs of the respective HW. This is mostly limited to the CPU and the peripherls close to it, but in modern computer systems there are a lot of components with processing power which needs to be programmed. Apart from CPU's these are for example MMU (**M**emory **M**anagement **U**nit)'s, graphic controllers, hard disk controllers, bus controllers for peripherals, bridges and sound controllers,

[9] https://de.wikipedia.org/wiki/Unix#/media/Datei:Unix_history-simple.svg
[10] http://futurist.se

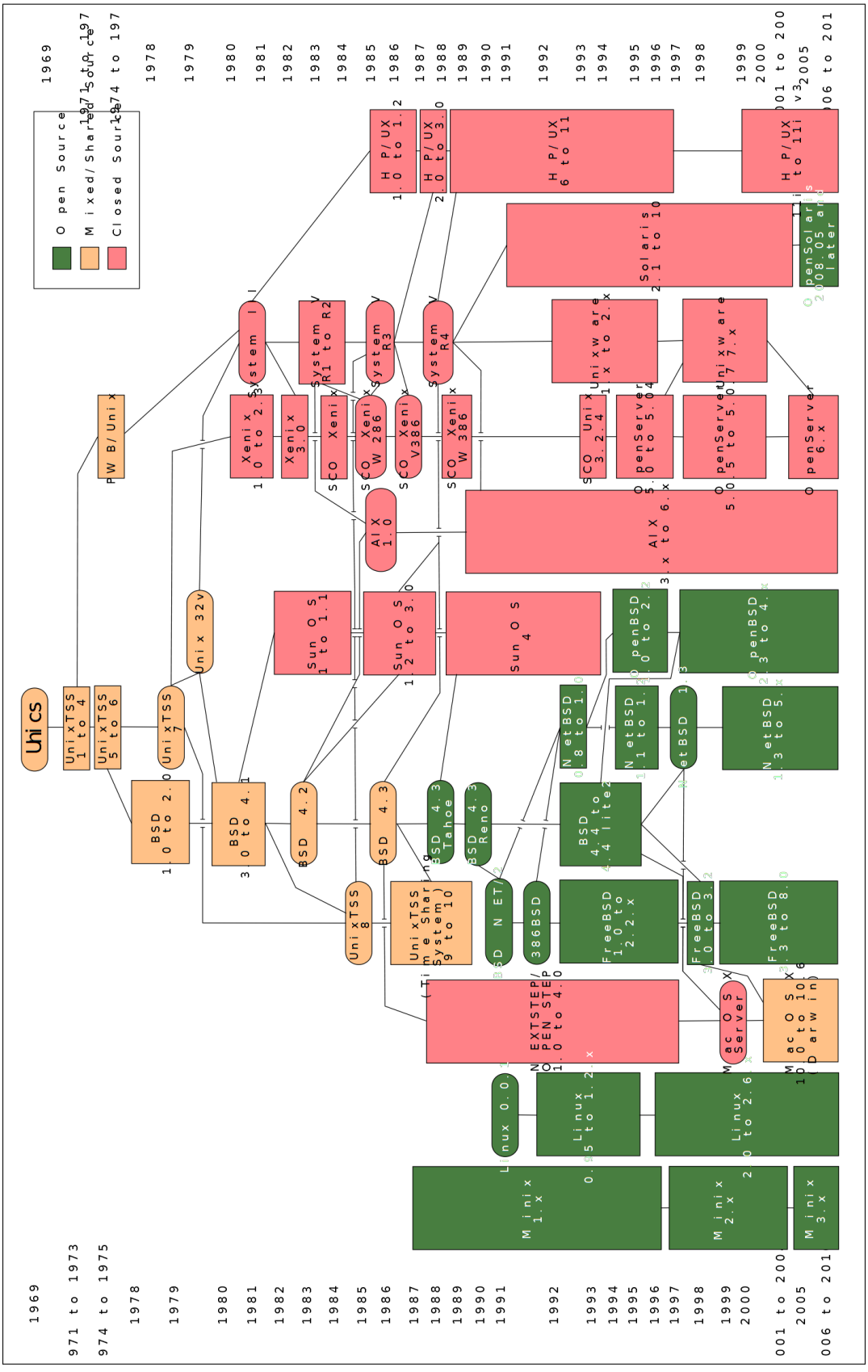

Figure 2.1: History of UNIX

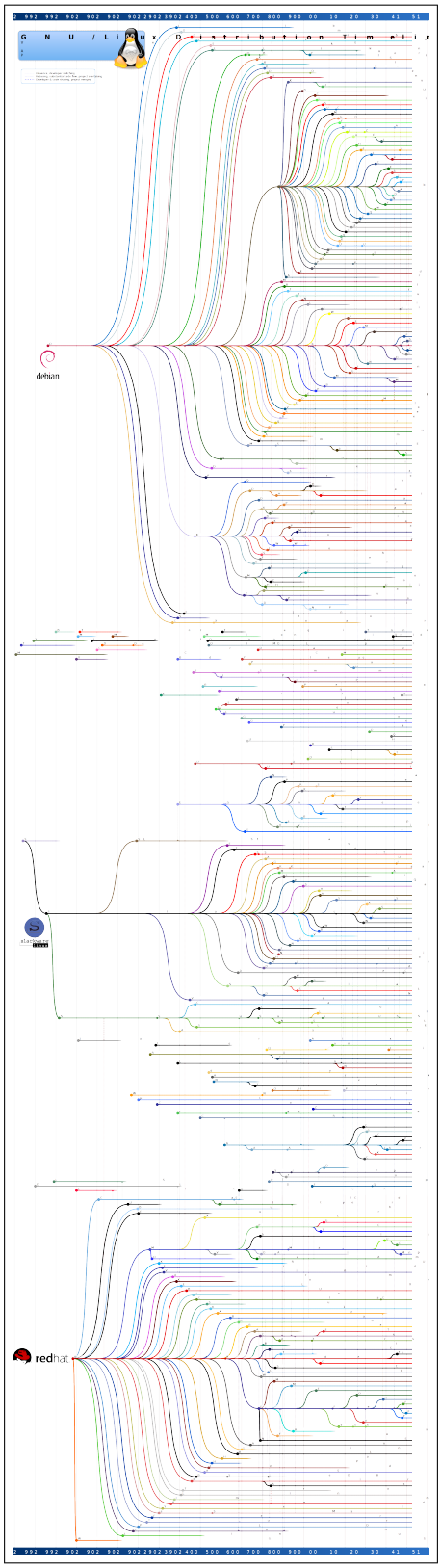

Figure 2.2: LINUX distributions

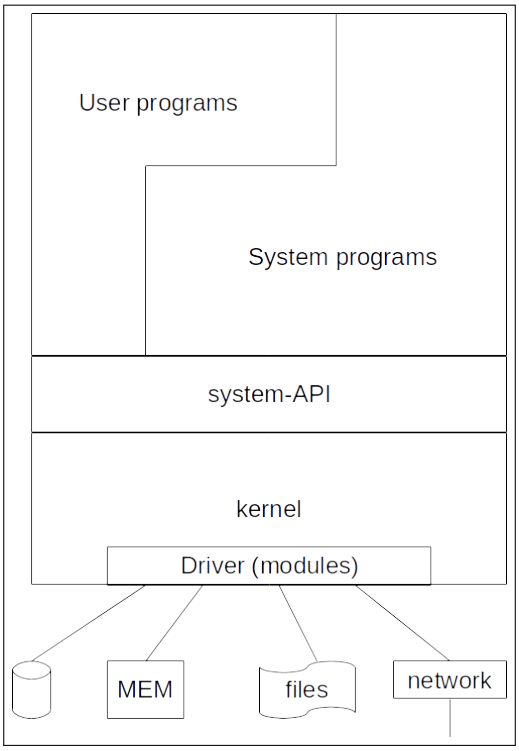

Figure 2.3: Composition of UNIX

The higher level services provided to application programmers are called API and in LINUX the SCI (**S**ystem **C**all **I**nterface).

2.3.3.2 Multitasking

UNIX can do several things at a time. This i called *multitasking*. But there is a big difference between *able to do at the same time* and *able to do at* nearly *the same time*. In systems with only a single CPU with a single core, only one command can be processed at a time. Nevertheless is UNIX able to give the user the impression that things happen at the same time by switching very cleverly and fast between jobs. Therefore, it is sufficient to say that UNIX is able to logically support multitasking. If multiple CPU's and/or multiple cores are active, things really habppen at the same time. But it does not make a difference to the user in most cases.

UNIX uses different priorities to deal with jobs according to their importance. There

needs to be abalnce between very fast switching to satisfy the expectancy of the users to have quick system responses and slow switching to have high performance in computing-intensive processes. The distribution of computing time is called *scheduling*[2, p.90ff]. Scheduling is an extremely complex subject and is solved in LINUX in a very modern way.

The kernel manage memory and other resources as well. A program simply cannot write everywher in memory. It is not just an issue about memory integrity but about safety as well. If applications could read everywhere it would be easy e.g. to write SW to spy on other users in the same system. Applications have to request memory, which e.g. could be done in C using `malloc`, which in UNIX systems is a call to the system API.

2.3.3.3 Multiuser

UNIX supports multiple users. Obviously it does not make much sense to have multiple users to sit in front of a single keyboard and a single screen. Therefore, most of the users will be logged in remotely using communication lines and protocols like telnet, ssh (**s**ecure **sh**ell) or HTTP. UNIX manages the integrity between users.

2.3.4 Kernel

fig. 2.4 shows the complexity of the system by showing the connection between the different parts of the kernel. Although it looks very dense and complicated, it can be seen, that the connections are mostly vertical and that the structure is 2-dimensional. Horizontally above each other there are layers of increasing abstraction, verticall next to each other different aspects of system control.

2.3.4.1 Memory management

Resources are limited and expensive. Therefore, memory management is a difficult task, striving to distribute the memory allocation in a way that everybode feels like having unlimited memory with no access time at all. This is not possible. Once, RAM (**R**andom

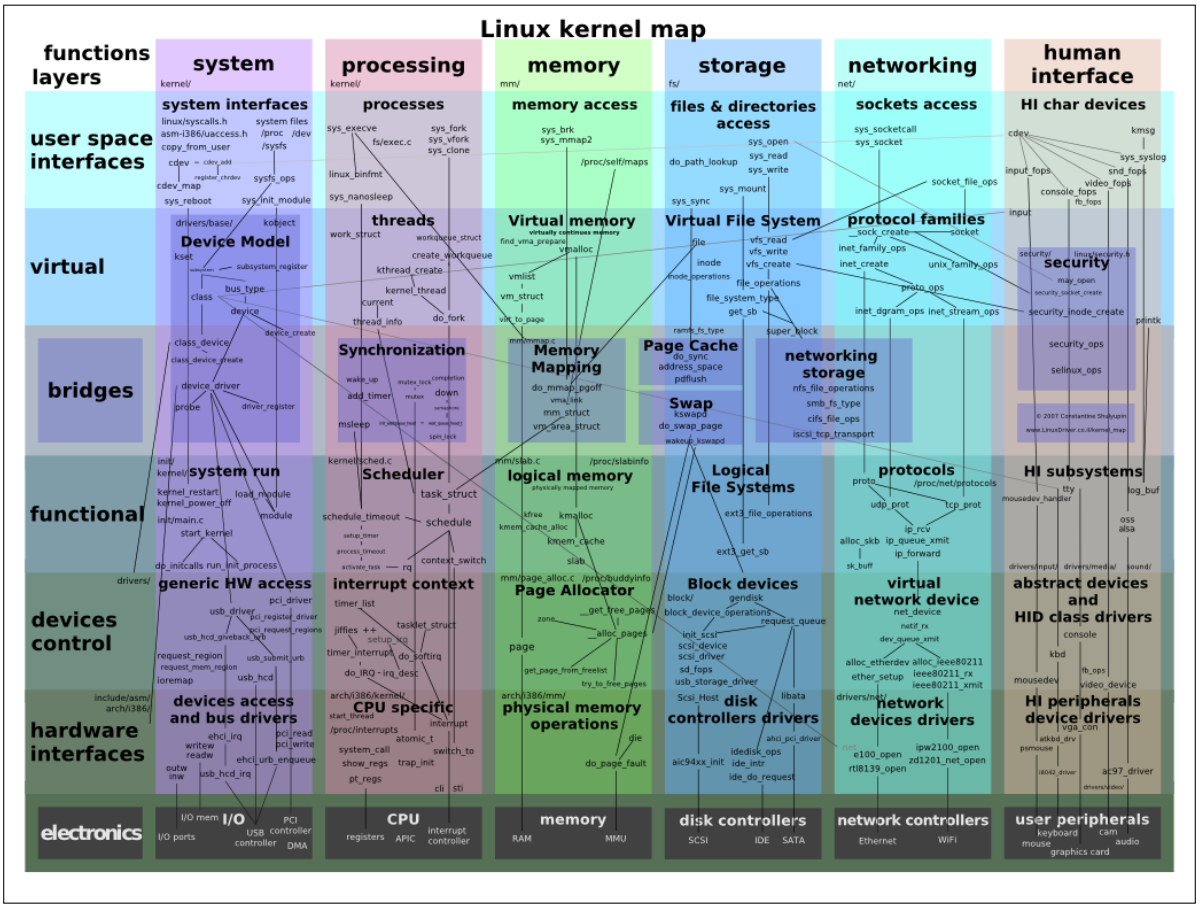

Figure 2.4: kernel structure (source: wikipedia)

Access Memory) comes to an end, fast memory will be outsourced by writing the content to files. When memory which is not directly accessible is requested, the system needs to again load the neccessary areas to RAM before. This can take up to seconds instead of ns or even sorter access times in direct access. A good strategy for this memory *swapping* helps to improve the user experience significantly.

Programs do not 'see' anything of swapping. They just see a nearly infinite amount of memory, called *virtual memory*.

2.4 The file system

2.4.1 Directories and the directory tree

The main difference, users 'see' between windows and UNIX file systems it the difference between `/` and `\`, the symbols to seperate directory levels from each other. Actually, the UNIX way is superior to the windows way as `\` is the *escape* symbol in many contexts.

2.4.2 Links

In LINUX, files can be indicated by referencing other files. this helps as often files should be visible or accessible from multiple points in the file tree. These links in UNIX can be treated in nearly exactly the same way as the files they refer to. Most commands make no distinction between accessing a file or accessing a link to a file.

For more information on links, there is much material on the WWW[11].

2.4.3 Device files

Hard drives and partitions in LINUX are erepresented as files, visible in the directory `/dev`, as well as other HW-components like printers, consoles, serial ports etc. and need to be specially treated when used. The dangerous aspects of these devices are not accessible to normal system users to protect the system. these devices are generally named *device*s.

Thee are some special and interesting devices. Writing to `/dev/null` lets the output just vanish into nothingness. Reading from `/dev/zero` or from `/dev/random` yields the expected output.

2.4.4 Remote file systems

File systems on rmote machines can be included into the file tree in many ways. They then appear as if they were local and can be accessed by all standard ways of accessing files. A

[11] https://www.linux.com/tutorials/understanding-linux-links/

small selection:

- NFS (**N**etwork **F**ile **S**ystem)
- FTP (**F**ile **T**ransfer **P**rotocol)
- SFTP (**S**ecure **F**ile **T**ransfer **P**rotocol)
- Samba (SMB (**S**erver **M**essage **B**lock))

2.4.5 FHS

The structure of the directory tree is standardised in the FHS (**F**ilesystem **H**ierarchy **S**tandard). It can be referred to at the WWW[12]. Not all LINUX distributions adhere to this standard, but most of them do.

The mot important directories will be covered later in the course as the need arises. Up to now, `/home` and `/proc` were touched briefly. Most interesting in the scope of this course are `/etc`, `/var` or `/srv`.

Table 2.1 istaken from version 2.3.

2.4.6 Permissions

In a multiuser environment it is compulsory to have permissions on different regions of the system. The system is much simpler, compared to windows systems, but still sufficient to implement secure multiuser environments.

There are three different aspects to each file or directory.

- *owner*: the file belongs to this user.
- *group*: the group the file is associated to. Users can belong to groups. Groups –and therefore users, belonging to such groups– have well-defined permissions

[12] http://refspecs.linuxfoundation.org/fhs.shtml

Table 2.1: Root-directory according to Filesystem Hierarchy Standard, Version 2.3

Verzeichnis	Bedeutung
bin	Essential command binaries
boot	Static files of the boot loader
dev	Device files
etc	Host-specific system configuration
lib	Essential shared libraries and kernel modules
media	Mount point for removeable media
mnt	Mount point for mounting a filesystem temporarily
opt	Add-on application software packages
sbin	Essential system binaries
srv	Data for services provided by this system
tmp	Temporary files
usr	Secondary hierarchy
var	Variable data

- everybode else or the *world*: everybody else (not owner or group member)

Permissions are '*read*', '*write*' und '*execute*', the shortcuts being 'r' (), 'w' () and 'x' (). ls-l lists a directory and the permissions of the entries., e.g.:

```
-rwxrwxr-x 1 hauseru users    870 10. Dez 11:32 abstract.tex~
drwxrwxr-x 2 hauseru users   4096 14. Feb 11:10 figures
```

The first entry belongs to the user `hauseru` and to the group `users`. The permissions rad from left to right

- owner: `rwx`, read, write and execute
- group: `rwx`, read, write and execute
- world: `rx`, read and execute

The second enry is a directory, indicated by the `d` at the begining of the line. With directories, permissions work slightly differently compared to files. Further information can be found at linux.com[13]. chmod

[13] https://www.linux.com/tutorials/understanding-linux-file-permissions/

chmod[14] is the command to change permissions.

2.5 The shell

A shell in UNIX has its windows counterpart in command.com. It is called a *command line* or *console* (see fig. 2.5). First steps in the console were done in section 2.2. It is important to note that a terminal for communication (e.g. a serial line) and a terminal for interaction are to be distinguished from each other.

2.5.1 Terminal

Historically, a terminal is a piece of HW with a keyboard and a display that can be connected to a computer system using a serial communication line. It was only used to display charcters sent by the conputer to the user and, accordingly, transfer characters keyed in by the user to the computer. The standards for communication were written down, amongst others, as

[14] https://en.wikipedia.org/wiki/Chmod

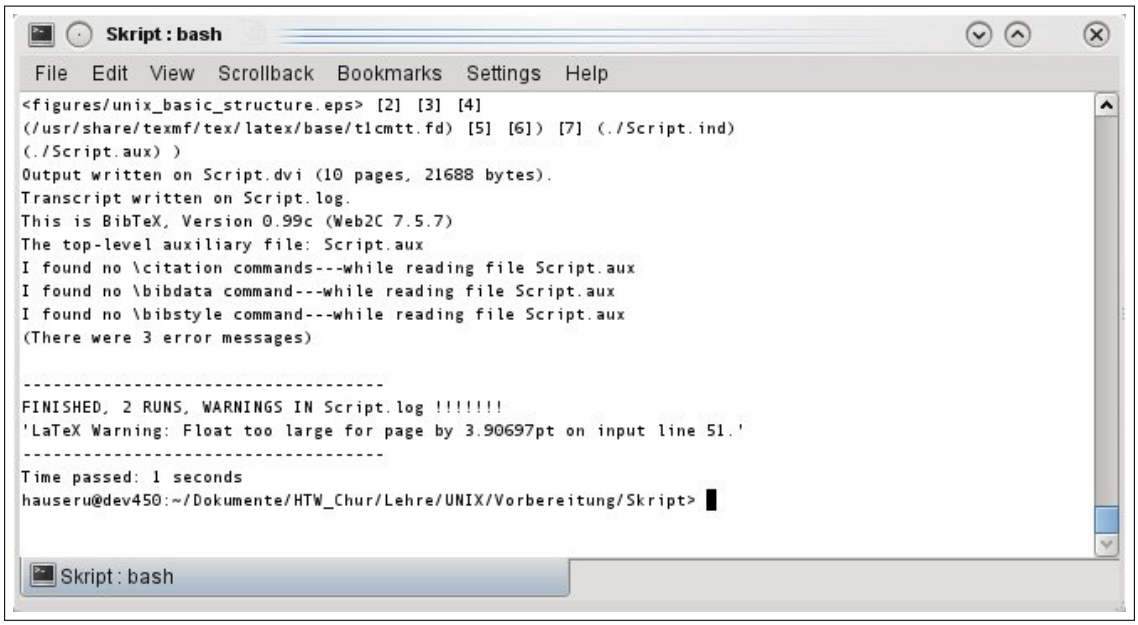

Figure 2.5: Eine Shell

ANSI escape codes, ANSI X3.64 und ISO/IEC 6429.

The shell today communicates to the computer in the same way as in the 70's of the last century. this has the advantage of always working inthe same way, no matter if communication is done locally or around the globe.

2.5.2 Command line

A *command line* is a program to receive and analyse instructions given by a user and to issue the appropriate commands to the addressed computer. Data resulting from the operations, normally will be presented to the user back on the same display.

UNIX/LINUX knows a whole bunch of command line programs. They share a common base, but have quite distinct differences when it comes to details. The most common used shells are sh (bourne-**sh**ell) and its succesor, bash (**b**ourne-**b**gain-**sh**ell).

The commands known by a shell are divided into

- *internal* commands: Commands directly known by the shell program itself
- *external* commands: Commands where the executable is a file that needs to be called
- *alias*-commands: Commands, created by a definition during shell configuration, mostly creating shortkeys for complicated command expressions. Alias commands are merely renaming operations for commands.

type

type command will reveal the type of a command.

2.5.2.1 Entering commands

Entering commands is supported by the intelligence of modern shells. Normally, a system is configured in a way that for external commands, no path information needs to be given. A *path* is a way through the directory tree which leeds to a file. The `tab` key supports the entry of commands tremendously. Hitting `tab` at nearly any time will give useful information about

possible entries or –if no alternativ is given– will complete the given expression automatically. Using `tab` becomes second nature quite quickly.

UNIX does not need special extensions for a file to be executable. If the permission (seesection 2.4.6) are set accordingly, a file simply is executed. If that is not possible, an error will be issued. Several types of files can be executed.

- executables themselves
- scripts for a shell
- files which need to be executed by an executable like PHP-Scripts etc.

LINUX –if configured properly– will know how to execute the necessary files.

Commands can be edited comfortably with →, ←, even in combination with `Ctrl`. Depending on the system configuation, the mouse can be used as well for pointing, marking and then appending by using the middle mouse button.

2.5.2.1.1 History The command line records and remembers typed command lines. this is called the *command history*. With ↑ ↓ the history can be walked back and forth. `Ctrl-r` allows the search for commands issued earlier by giving patterns. This is called *reverse intelligent search*. After proposing a command with Ctrl-r it can be edited normally before issuing it. `Ctrl-g` leaves this search mode.

2.5.3 The prompt

The *prompt* is the collection of characters in front of the blinking curser prior to entering a command. It can (and normally will) give essential information about the system. It can be configured widely[15].

In normal configurations it will at least give the name of the machine, the logged in user and the working directory, but much more information is possible.

[15] https://linuxconfig.org/bash-prompt-basics

2.5.4 Configuration

The shell configuration is divided into several aspects.

2.5.4.1 Variables

Shell variables control a lot of a shell's behaviour and can control a lot of application's behaviour as well. A shell variable consists of a name and a value. To define a shell variable its name is keyed in, followed by a `=` and its value.

```
MYVARIABLE='I am a variable'
```

Important variables will be covered seperately

2.5.4.1.1 `PATH` *PATH* controls the directories and their order in which the shell will search for executables if no explicit path is given. It should not include `.` for security reasons. Therefore, to call executables in the working directory, the executable needs to be prefixed with `./`.

2.5.4.1.2 `USER` *USER* holds the name of the logged-in user.

2.5.4.1.3 `HOSTNAME` *HOSTNAME* holds the name of the machine, the shell runs in.

2.5.4.1.4 `HISTSIZE` *HISTSIZE* controls how many commands are recorded for the history.

2.5.4.2 Configuration files

When a shell starts it calls on several files for configuration. The order of processing can be looked up in the `man`-page of the shell (see section 2.5.5.1.1). With these configuration files the behaviour of the shell can be widely influenced. There are global confguration files that are normally setup by the administrator and local ones to be adjusted by system users.

2.5.5 Important commands

The most important commands are given here. There is a wealth of further information available on the WWW[16].

Reading about the main commands is strongly recommended.

Commands consist of the command name, options, other parameters to be involved and a return type which reports on the command success. Successful operation normally will return 0.

The return value can be accessed by echo $?. $?

```
ls --xxxx
ls: unrecognized option '--xxxx'
Try 'ls --help' for more information.
echo $?
2
```

Above is a possible communication between shell and user. `2`, given in the last line is a return value indicating an error.

2.5.5.1 Basics, navigation and directories

2.5.5.1.1 `man` how to get help All commands should have a `man`-page associated, callable by man command. In fig. 2.6 the manpage to `ls` is given. man

Sometimes –when calling up the man page– the user is requested to isuue a number. This number refers to a *section*. The section divide the information into several aspects. Section 1 e.g. refers to shell commands, 3 are so called subroutines but effectively describe a huge number of C system calls. More information is given under `man man`.

2.5.5.1.2 `apropos` This command supports `man`. It searches the summaries of the man pages for keywords. `apropos directory` will pint all names of commands who deal with

[16] https://manpages.debian.org/

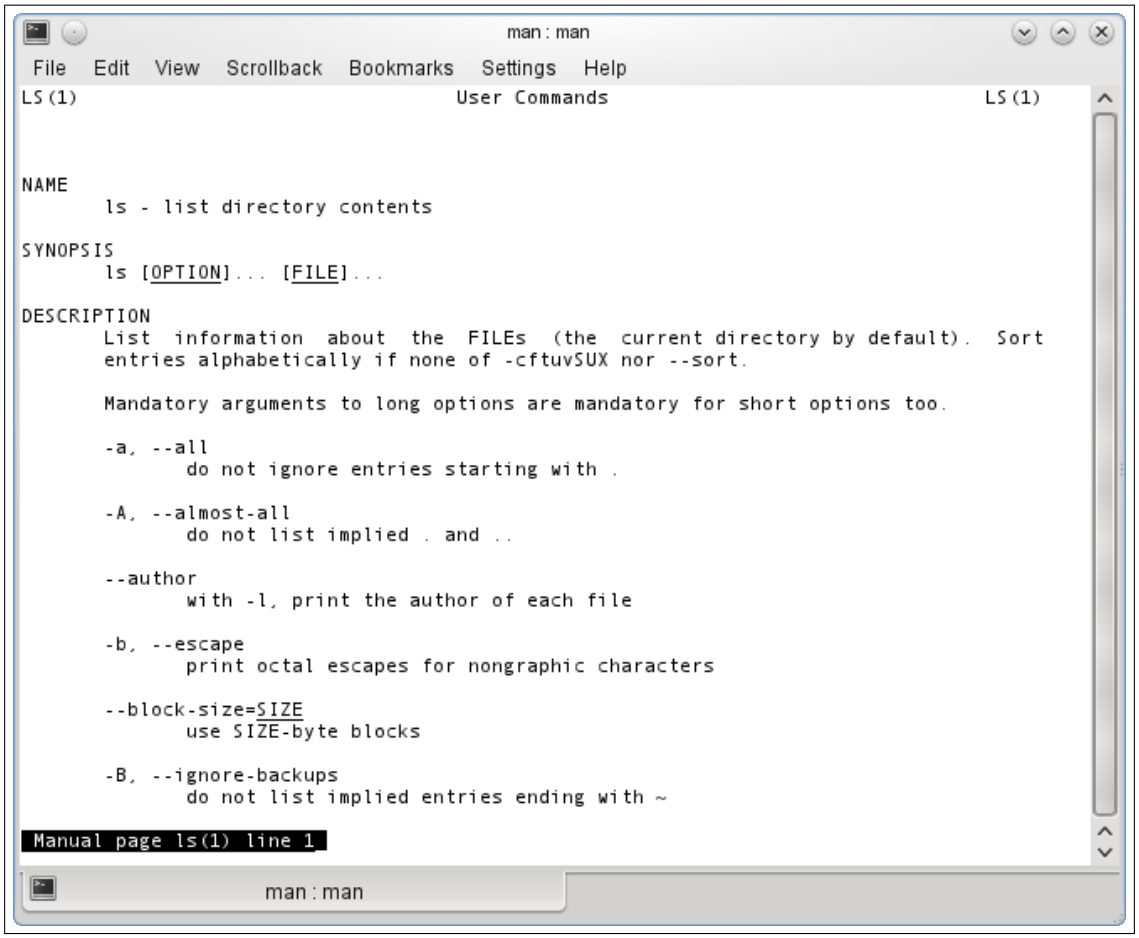

Figure 2.6: Man

directories. By using this command, comands asociated to some subject can be found.

2.5.5.1.3 `ls` `ls` shows directory content.

2.5.5.1.4 `pwd` This command shows the working directory.

2.5.5.1.5 `cd` This command changes the working directory.

2.5.5.1.6 `pushd`, `popd` **und** `dirs` These commands allow to change directories using a history stack.

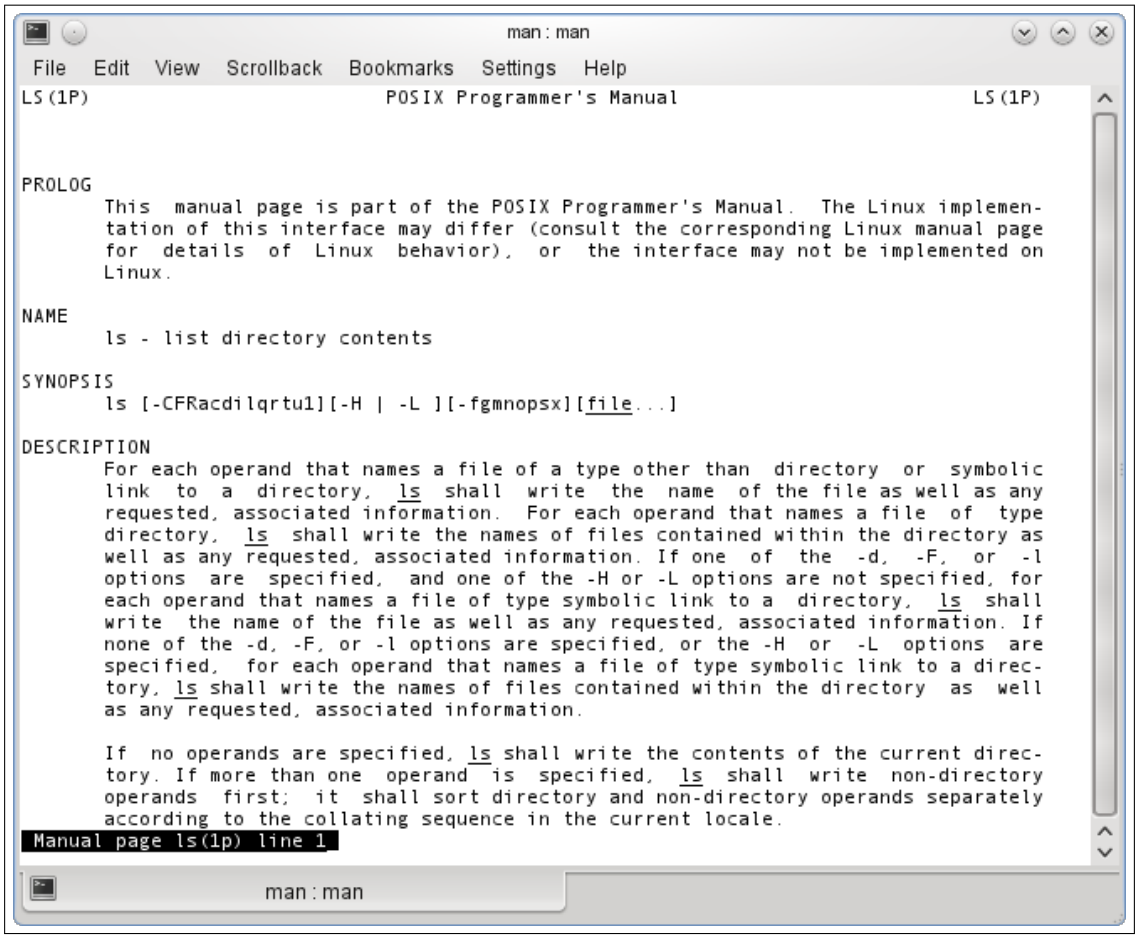

Figure 2.7: Man, verschiedene Versionen

2.5.5.2 Console

2.5.5.2.1 `clear` Clears the console.

2.5.5.2.2 `reset` Initialises the console.

2.5.5.2.3 `tput` Helps to control advanced activities on the console display. Uses *terminfo*

E.g. `tput cup 0 0` moves the cursor to position (0|0) of the terminal.

2.5.5.3 Information and search

2.5.5.3.1 `date` gives the date.

2.5.5.3.2 `echo` prints text to the console.

2.5.5.3.3 `du` Informs about disk usage. Interesting option: `-h`.

2.5.5.3.4 `find` Very versatile command for file searching.

2.5.5.3.5 `grep` Searches for character patterns, extremely flexible to use[17,18].

2.5.5.4 Account and system

2.5.5.4.1 `groups` Shows the groups a user belongs to.

2.5.5.4.2 `passwd` Change a password.

2.5.5.4.3 `shutdown` shutdown the machine. On many systems this is only allowed to be done by the administrator.

2.5.5.4.4 `mount` Connect to other file systems and include them into the directory tree.

2.5.5.5 Files

2.5.5.5.1 `cat` print some file content.

2.5.5.5.2 `less` File pager. This command can be used to show, search and navigate big files page by page.

2.5.5.5.3 `tail` Prints the end of a file.

2.5.5.5.4 `cmp`, `diff` Tools to compare files.

2.5.5.5.5 `sort` sort a file according to given aspects.

[17] https://en.wikipedia.org/wiki/Grep
[18] https://en.wikipedia.org/wiki/Regular_expression

2.5.5.5.6 `touch` Create a file or update its time stamp.

2.5.5.5.7 `chmod` Change file permissions.

2.5.5.5.8 `chown` Change file owner.

2.5.5.5.9 `chgrp` Change group of file.

2.5.5.5.10 `cp` Copy files.

2.5.5.5.11 `mv` Move files.

2.5.5.5.12 `rm` Remove files.

2.5.5.5.13 `rmdir` Remove directories.

2.5.5.5.14 `ar`, `compress`, `gzip`, `tar`, `unzip`, `zip` Deal with archives.

2.5.6 Pipes

Commands can be used in chains, chaining the output of one command directly to the input of another command. the notation is
`cmd1 | cmd2 | cmd3`. The symbol `|` therefore is often named the *pipe* symbol. Symbolically ther is a pipe created, chanelling the output from one command to the next. With *piping* very flexible command combinations can be created.

The concept of pipes change the complete system as the basic commands turn into little tools which can be combined to powerful processing chains, performing amazing things on a single line of command.

A special command in this context is xargs. This command takes the input and puts it at the end of the following command. This is often used to find a collection of files with one command and the do manipulations or searches on this collection of files.

xargs

```
find . -iname "*.txt" -type f | xargs grep -i boss
```

The command line above will search for all files with extension `txt` and look in all of them for the word `boss`, ignoring lower or upper case notaion.

Such an *anonymous pipe* ist unidirectional and is removed after the associated processes are finished.

There are named pipes. These must be explicitly created in the file system. They have no exiry time. To create a named pipe mkfifo thepipe is used to create a pipe named `thepipe`. mkfifo It will be indicated (using `ls -l`) as follows:

```
prw-r--r-- 1 hauseru users      0 28. Feb 11:55 thepipe
```

As UNIX is able to run processes at the same time, input and output of independent processes can now be coupled using such a pipe. Synchronisation is automatically managed.

2.5.7 Patterns

In section 2.5.5.3.5, the command `grep` is shown. This is command generally able to look for all kinds of patterns. The search for patterns is a very general subject in itself as it is necessary to search for things very often. The WWW produces a wealth of information if the search expressions 'regular expressiosns' or 'regex' are issued. Wikipedia[19]gives a good start.

2.5.8 Quoting und escape-Sequenzen

Quite often, the need to identify groups of words as a single text arises. The use od several types of quotation marks is very confusing at first. Evene experienced users sometimes run into difficulties when using quotes.

```
echo Hello, Don
echo "Hello, Don"
echo 'Hello, Don'
```

[19] https://en.wikipedia.org/wiki/Regular_expression

will all print the same string, `Hello, Don`. Many characters, however, have a special meaning in the bash (as in languages like C). This fact leads to new challenges. E.g., how can we print " itself? To accomplish tasks like this *escape* ()\) is there.

`echo Hello, \"Don\"` will print as `Hello, "Don"`.

The combination `\"` tells the system *to take the* " *literally* and not as an indication of the end of a string of words. In the same way, `echo Hello, \'Don\'` will yield `Hello, 'Don'`.

`name="Bruno";echo $name` yields `Bruno`. What happens here? First, a variable named `text` is defined, bearing the content `Bruno`. Then, the content of the variable is printed. Here `$` has a special meaning, starting the name of a variable.

Variables (section 2.5.4.1) play an important role in quoting.

`name="Bruno";echo "$name"` yields `Bruno` again. But

`name="Bruno";echo '$name'` yields `'$name'`.

The single quotes seem not to lead to a variable evaluation, but to a literal printing. It is often not really easy to predict the behaviour without a really acute knowledge of escaping:

`name="Bruno";echo -e "1:\\t$name"` yields `1: Bruno`,

`name="Bruno";echo -e '1:\\t$name'` yields `1:\t$nam`.

The option `-e` at `echo` is necessary to trigger the evaluation of escape sequences like `\t` themselves[20]. Even the GNU (**G**NU's **N**ot **U**nix)GNU bash manual-page (`http://www.gnu.org/software/bash/manual/bashref.html#Single-Quotes`) is quite involved to be understood correctly. The quickest option often is a good starting guess followed by strategically clever trials and tests.

2.5.9 Stream redirection

Commands do not necessarily need to print to the console. > filename will redirect the > output to a file. This first was used in section 2.2.1 without explanation. The target can be any kind of file, e.g. a device or a pipe.

[20] `https://en.wikipedia.org/wiki/Escape_sequence`

The output of `ls >/dev/null` discards the output.

Die error output is not redirected this way.

`cat xyz >/dev/null` will generate an error on the screen, if `xyz` does not exist. The reason is that output is divided into several channels. Giving no number uses channel 1 as the standard output channel. this channel is often named `stdout`, as this is the name for this channel in C. (`stderr`) has number 2. `cat xyz 1>/dev/null 2>error` discards the normal output and diverts `stderr` into the assigned file.

If `stdout` and `stderr` are to be redirected into the same target, this is indicated by `2>&1`. `cat xyz 1>/dev/null 2>&1` redirects `stdout` to `null` and *then* `stderr` to where `stdout` used to direct. The order need to be taken into account. `cat xyz 2>&1 1>/dev/null` redirects `stderr` to where `stdout` points to (still at the console) and then `stdout` to `null`.

Input can be redirected as well using `<`. `grep verb <Script.tex` is the same as `cat Script.tex | grep verb` (see section 2.5.6 for pipes).

For output to appear on the console as well as in a file e.g. ls | tee file can be used. It is derived from a piece of water pipe formed like a `T` to divert the flow into two directions at the same time.

tee

2.5.10 Proces control

UNIX is able to run processes concurrently. This can be used by tha bash as well as by programming languages like C. In a shell, a command can be topped by using CTRL-Z. Using bg (background) afterwords will restart the process, but in the background. This is visualised by the reappearing prompt. Now, new commands can be issued. Both commands now share the same output channel, the console. They will write on the console at the same time, normally resulting in a chaotic output. the command fg (foreground) pulls the process back to the foreground. A process can be started in background directly by appending `&` at the end of a command.

bg

fg

The following commands are helpful when working with processes.

ps

ps lists all processes running in shell the command is un on

pstree

pstree lists all processes in a tree structure

top

top shows the most important processas and some more system information. the command can be stopped with `q`.

Each process has a number which identifies the process. Using `pstree` shows, that all processes can be traced back to a general parent process called *init*. This process is the one created first when a LINUX system is started. All other processes are started from there as a cascade.

`ps -f` shows some additional information:

```
UID        PID  PPID  C STIME TTY          TIME CMD
hauseru  10043 10041  0 08:58 pts/1    00:00:00 /bin/bash
hauseru  20142 10043  0 12:55 pts/1    00:00:00 ps -f
```

Each process is allocated to a user (UID), knows its ID (PID), the ID (PPID) of its parents its (dynamic) priority (C) the starting time (STIME) and to which terminal it is associated to (TTY). Additionally, the processor time used (TIME) and the full command line itself (CMD) are known by the process.

Processes can be influenced by commands. nice influences the processor time, a process is allocated. kill sends a signal to a process. These signals can be caught (trapped) by a process and can stimulate whatever reaction the programmer wants to have. Often, this command is used to kill a process (Signal 9 is the standard there and is issued bc `Ctrl-C`), hence the command name `kill`.

nice

kill

Chapter 3

Introduction to JS

JS is a language stemming from predecessors intended to build dynamic web pages and get rid of java applets to obtain this. The first effort are from Netscape, a then famous browser provider. Brendan Eich started in 1995 with a language called mocha, then LifeScript, then JavaScript. By concept, no relation to Java was intended and in the language concept itself they are mo relations. By management the syntax was to look like Java, which explains some of the looks.

It is assumed that the reader is familiar with both, HTML and CSS (**C**ascading **S**tyle **S**heets). These are not programming languages but a markup language and a formalised style description language. As the target audience is computer scientists, it is expected from it to make itself familiar with document formats. Nevertheless, a very compact introduction is given in appendix A. A wealth of information is available on the WWW, e.g. the very elaborate w3schools[1].

3.1 JS in the Browser

JS can be used in web pages in several ways:

- code can be included in the web page directly using the tag `<script>`

[1] https://www.w3schools.com/

- script files can be included using the tag `<script>`
- code can be included directly at certain positions in the `HTML` code,
  e.g. as `<h1 onclick="this.innerHTML='Ooops!'">Click on this text!</h1>`[2]

The browser works as depicted in fig. 3.1. The structure in the HTML code is internally represented in the DOM (**D**ocument **O**bject **M**odel). JS normally starts to work as a reaction to changes in the state of the web page like clicked elements or answered HTTP-requests. JS can analyse and manipulate the DOM. These changes will then be brought to the display by a renderer engine when JS has finished its job and waits for events (see section 3.7) to be triggered.

3.2 NodeJS

With nodeJS, JS is used like a normal language on the console of an OS. Therefore, JS can do, whatever other languages can do as well, read files, use variables, print reults, setup servers etc. Libraries (modules) help with starting more complex things and .e.g. make

[2] https://www.w3schools.com/js/js_htmldom_events.asp

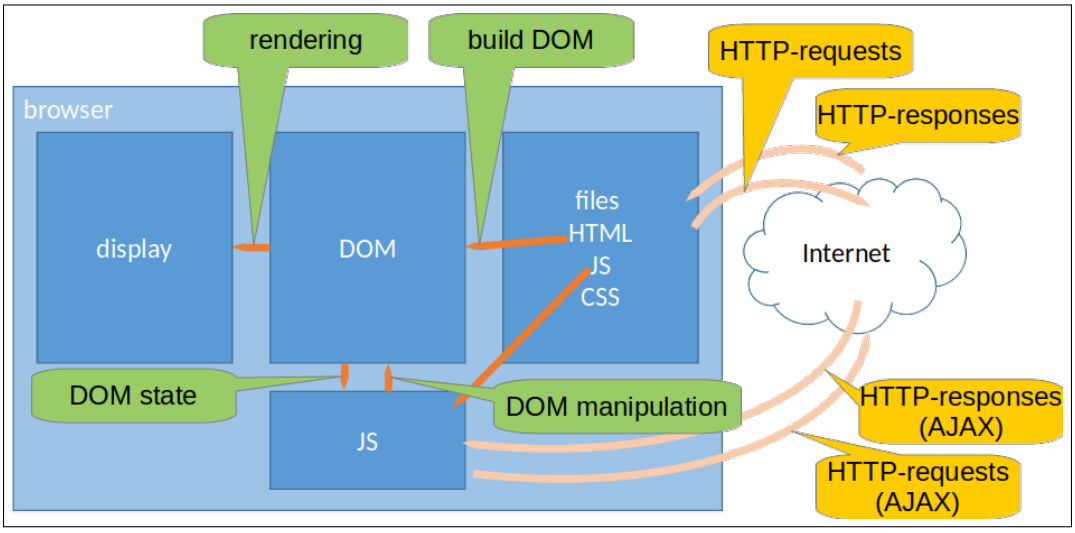

Figure 3.1: The browser

it extremely simple to build a complete web server with a handful of lines of code. The raspberry Pi uses nodeJS as well to implement server side functionality in IoT devices without leaving the language world of web developers.

As obviously there is no DOM in server side applications, JS will not manipulate any DOM there. But this is no change as the DOM is represented to JS like a single, large object or like a collection of large objects anyway.

3.3 Commonalities with C

A solid foundation in programming gernerally and in C especially is assumed.

Variables, conditions, loops and scope are some of the concepts common to C and JS. Functions are known as well in JS, although their calling can be quite different. A concept of divide-and-conquer like the modularisation with headers and source files in C is in place as well, called modules. Modern browsers support modules although this feature is quite new for browsers, but already used widely in server side projects.

3.4 Variables and objects

Variables are loosely and dynamically typed in JS. They are normally introduced by the keyword `var` or `let` (see section 3.6), but not necessarily.

3.4.1 Types

Types are not assigned to variables, but derived by the system according to assignments.

```
n = '838102050' // Set 'n' to a string
console.log('n = ' + n + ', and is a ' + typeof n);
n = 12345 * 67890; // Set 'n' to a number
console.log('n = ' + n + ', and is a ' + typeof n);
```

```
n += ' plus some text' // Change 'n' from a number to a string
console.log('n = ' + n + ', and is a ' + typeof n);
```

leads to[3] the console output

```
n = 838102050, and is a string
n = 838102050, and is a number
n = 838102050 plus some text, and is a string
```

As the example shows, a variable can even change its type during runtime.

The lack of typing has consequences for the use of variables. `5+5` will lead to `10`, but `'5'+'5'` will yield `"55"`, which will not be the desired result most of the time. This is a very common mistake in JS.

JS is interpreted, preventing the system from checking for errors before runtime. Mistakes in variable types will only occur in runtime.

3.4.2 Objects

Objects are compound types in JS. Apart from attributes they can hold operations (methods). But the way to type them is closer to function pointers in C than to methods in C++.

```
var person = {
firstName: "John",
lastName : "Doe",
id       : 5566,
fullName : function() {
return this.firstName + " " + this.lastName;
}
};
```

[3] https://www.quora.com/What-is-called-Variable-typing-in-Javascript

Above is an example[4] for an object, in this case built using JSON. Objects can change not only content like attributes, but structure as well. Attributes and operations can be added at runtime without restriction. Adding `person.birth=2002;` to the above example is no problem at all. But external functions can be used as operations as well. The following is perfectly legal:

```
var person = {
  firstName: "John",
  lastName : "Doe",
  id       : 5566,
  fullName : foo
};

function foo()
{
  console.log(this.firstName + " " + this.lastName);
}
```

Although giving a high degree of freedom, in programming it creates a whole jungle of traps and pitfalls. Concise and readable, let alone safe and secure code is much more difficult to write compared to a strictly typed and restrictive language.

3.5 Strict mode

Strict mode tries to compensate for some of the security issues JS as a language poses. Running in strict mode (`"use strict";`) forbids to use undeclared variables, which essentially means, that either `let` or `var` need to be used. Undeclared objects are forbidden as well.

[4] https://www.w3schools.com/js/js_object_methods.asp

`"use scrict"` can be used with functional scope as well. Only the associated function will then be strict.

There are more consequences to strict mode which does not need to be repeated here[5].

3.6 Scope

JS has function scope. Each function generates a new scope. Variables declared with `var` are local to this function. Newer versions of JS support block scope as well, which is identified by `let`. Therefore `let` is the most restrictive declaration and should be used wherever possible. Variables without the keywords `var` or `let` are global automatically, except the script is run in Sctrict Mode.

3.7 Events and listeners

Web pages are *event driven*. The program reacts to user activities and other events brought to its attention and –after finishing these activities– goes back into a state of waiting. Most programs in the pc market, in mobile phones and other devices work this way. It is the opposite way of information flow, traditional programs work and students normally learn how to program. There, the programs compute along a given process and –at the apropriate locations– asks users for input. The reason is found in the way comping worked, when high level programming languages like C were created. These days, a programmer went to a computer operator with his/her code. The code was executed and the result was given back to the programmer. The communication inversion took place when the first interactive GUI (**G**raphical **U**ser **I**nterface)'s came to the market. The Apple Lisa[6] was possibly the first computer with a GUI worth mentioning. That was 1983, years after the creation of C or PASCAL.

[5] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Strict_mode
[6] https://en.wikipedia.org/wiki/Apple_Lisa

JS is well suited for the inverted comminucation of event driven programming. Functions can be set up as target function to be executed when certain events happen. This is very general and JS uses it to much further extend

3.7.1 Asynchronism and single tasking

3.7.1.1 Overview

To cope with multiple requirements at the same time, some languages turn to built-in concurrent programming or to libraries supporting concurrent programming. There, multiple program threads are executed logically at the same time. Depending on the computer architecture and language there are several technologies not being covered here. They have in common, that switching between the different strands of execution takes time and interrupts code execution at sometimes unpredictable or unsuitable locations.

JS is single tasked. This could be seen as a disadvanzage first. But as it is event driven, it can compensate for many disadvantages of multithreading like deadlocks, mutual exclusions etc., as code will not be interrupted at unknown locations. JS sits and sleeps until an event is triggered and goes to sleep again after the event has been handled, unless new events have arrived in the meantime. This principle is shown in fig. 3.2

JS has proven to be able to handle huge amounts of requests, serving many customers at the same time in large servers.

An example of reading a file asynchronously is given in Listing 3.1

Listing 3.1: readFileAsync.js

```
var fs = require('fs');
fs.readFile("readFileAsync.js", fileIsRead);
function fileIsRead(err, data)
{
    if (err)
    {
        console.log(err);
    }
    else
    {
```

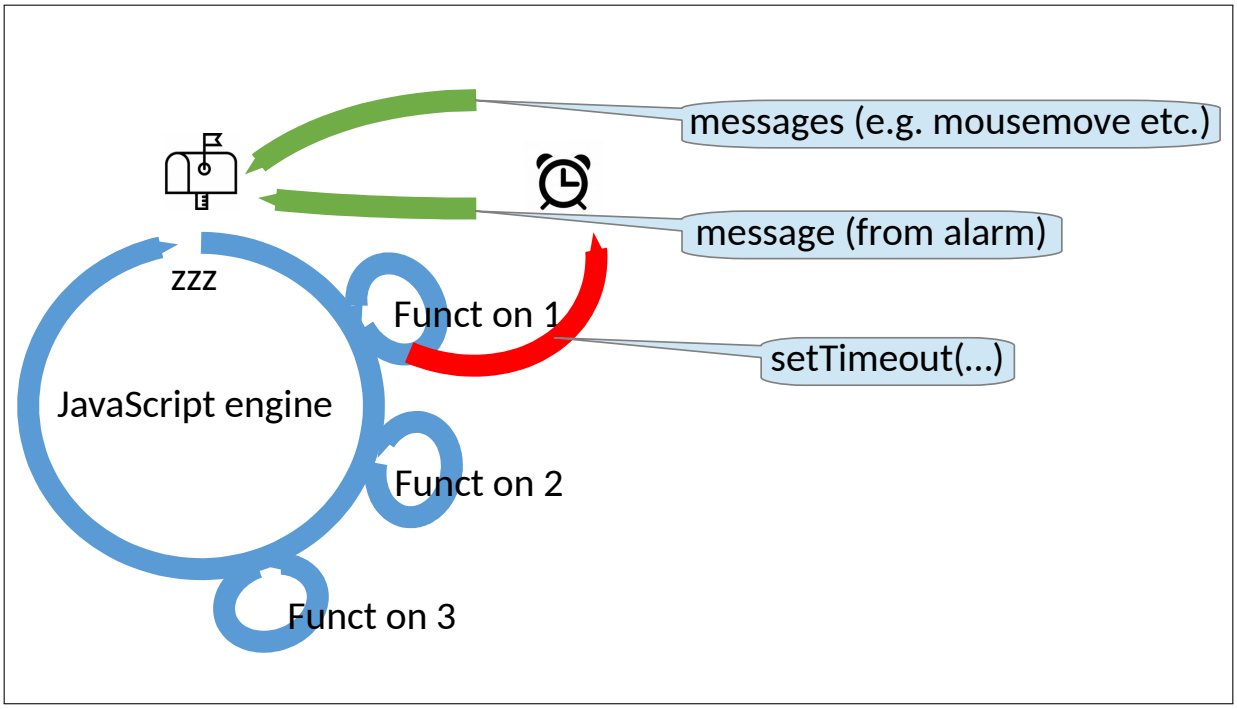

Figure 3.2: JS engine

```
        console.log(data.toString());
    }
}
```

The background to of the first line (containing `require`) is given in section 3.8.2.1. Essentially it includes a library. The function `fs.readFile` is passed two arguments, the filename itself and a function to call, when the file is read. The function itself (`fileIsRead`) is passed two parameters, an error, which should be `null` in normal cases, and the content of the file itself.

3.7.1.2 Registering events

Registration involves to elements:

- the event type
- the function to handle the event

3.7.1.2.1 Client-side With JS in browsers there are many possibilities to register events.

```
<button onclick="document.getElementById('demo').innerHTML = Date()">
The time is?
</button>
```

is the most direct method. After the name of the event (here `onclick`), the code to be executed is given directly. From a designers point of view it is not recommendable as it only results in readable code for extremely small chunks of code. It undermines the principle of seperation between code and document. In addition to this, it uses an anonymous function (see section 3.7.1.2.1.1), although it is not directly visible.

```
<button onclick="displayDate()">
The time is?
</button>
```

is much better already. The code itself will be inside the function `displayDate()`, possibly in a seperate JS file.

```
document.getElementById("myBtn").onclick = function() {...};
```

is even better, keeping the whole dynamic part in the JS file. Here, an anonymous function is used (`function() {...}`).

3.7.1.2.1.1 anonymous functions Anonymous functions are nameles functions put directly at the place where the code is to be executed. They are quite poular in JS, as they allow very quick implementations. Their drawback is, that they make it very difficult to write readable and maintainable code, especially when nested inside each other. The author advices not to use anonymous functions if possible. However, aspecially when transporting short term variables as parameters to anonymous functions, they are advantageous, because they allow special memory tricks due to scope reasons. Therefore, a good balance needs to be stricken.

```
document.getElementById("myBtn").onclick = displayDate;
```

is really good, using an explicit reference to a function and not an anonymous function.

3.7.1.2.2 Server-side Here only nodeJS will be covered. A large number of functions looking like a single process in C can be divided into more steps in JS giving the programmer the choice between synchronous and asynchronous execution. For example, reading a file can be seen as opening the file in the first step and then waiting for the file to be read. In C it might take some time to read a file, especially when it is located in a remote file system. This will block C until the file is read. If dealing with a web page or a server this behaviour would leave the web page or the server unresponsive for some time, which is not acceptable. Therefore, JS can signal the system to open and read the file and register an event listener to trigger a function once the file content is ready for further processing. Then, as a second step, reacting to the event that the data is ready, JS can further process the content of the file.

Reading large files can be divided into reading blocks of file content as multiple steps, indicated by an event for each chunk of file content being ready for processing.

3.8 Modules

Clean code needs modularisation. In C the usage of header and source files serves the purpose. JS in web pages has not used modularisation for long time except from including several JS-scripts into one web page. As JS is used on the server side or as a non-web language, modularisation became an issue. So modern versions of JS include methods for modularisation.

3.8.1 Modules in the browser

As mentioned above, modules in the browser are loaded by including the associated JS source files als scripts with the `<script>`-Tag. The scripts are included in the order in which they appear and are executed immediately.

Unfortunately it can happen, that the scripts are loaded out of order, as some scripts may take longer to load than others. Therefore, it is a good idea to have an initialisation function which is called by the `window.onload` event. This event is issued after all files belonging to a web page have finished loading.

There is a plethora of libraries available for free. A selection of popular ones is:

- jQuery *is a fast, small, and feature-rich JavaScript library. It makes things like HTML document traversal and manipulation, event handling, animation, and Ajax much simpler with an easy-to-use API that works across a multitude of browsers. With a combination of versatility and extensibility, jQuery has changed the way that millions of people write JavaScript*[7], see section 3.8.1.1
- bootstrap (although it is more a css library than a JS library) *is the most popular HTML, CSS, and JS framework for developing responsive, mobile first projects on the web*[8]
- d3 *is a JavaScript library for manipulating documents based on data*[9]
- chart.js, *Simple yet flexible JavaScript charting for designers* & developers"[10]
- popper, *a kickass library used to manage poppers in web applications*[11]
- ..., according to personbal need and habit

For this course, jQuery is the most important library and therefore has its own section associated to it.

[7] https://jquery.com
[8] https://getbootstrap.com
[9] https://d3js.org/
[10] https://www.chartjs.org
[11] https://popper.js.org/

3.8.1.1 jQuery

jQuery does not extend JS. It itself is written in JS and just uses some of the more intricate features of JS to allow for more compact writing of code, much easier access of DOM elements and manipuation and therefore much quicker and more reliable coding. It abstracts a number of standard procedures –normally requiring the setting up of loops through selected DOM elements– behind simple interfacing methods. Chaining[12] is used extensively for drilling down in selections of large DOM trees and manipulating all selected elemenets in one single command. Setting up listeners is simplified considerably.

Additional features not necessary for this course –and therefore not covered– are animations, themes (lQuery UI) and other add-on's to make JS programming less work.

3.8.1.2 Bootstrap

Bootstrap 's main intend is to make it easier to develop responsive web sites. The term *responsive* means, that a web page reacts to properties of the presenting device like screen resolution, input devices etc. to allow for optimal presentation. In combination with this term, often *mobile first* is mentioned to point out the fact that a web site is more likely to be viewed on a mobile device than on e.g. a pc. Therefore, the main design decitions are taken for mobile devices as well.

To chieve layout control, bootstrap uses mainly the concept of attributing classes to tags to influence their appearance. This is not limited to single tags, as tags can contain other tags. E.g., a table is a tag and bootsrap can control the appearance of the whole table, rearrange columns to fit the mobile first approach, have buttons look consistently throughout a page. A button, created with the code

```
<button type="button" class="btn btn-primary">Primary</button>
```

will have a look decided on by bootstrap due to simply adding the given class attribute.

[12] https://www.w3schools.com/jquery/jquery_chaining.asp

3.8.2 Modules in node JS

3.8.2.1 Including modules

```
var httpServer = require('http');
```

includes a module called `http` and assigns it to an object called `fooHttp`. From there, the module can be accessed through the object like in

```
httpServer.createServer(
 function (req, res)
 {
  res.writeHead(200, {'Content-Type': 'text/html'});
  res.end('Hello World!');
 })
 .listen(8080);
```

creating a server which just gives an echo of *Hello World*.

3.8.2.2 Exports

A JS module is a normal JS file with a single specialty: It carries a object called `exports`. This object is the interface of the module. If this file is drawn as a module using `require`, the return value is the `export` object of the called file. As it is an object, attributes and operations can be part it. A simple example:

Listing 3.2: module.js

```
module.exports =
{
    firstName: 'James',
    lastName: 'Bond'
}
```

Listing 3.3: require.js

```
var person = require('./data.js');
console.log(person.firstName + ' ' + person.lastName);
```

Listing 3.2 defines the `export` object, Listing 3.3 draws on the module and uses the defined values[13].

3.8.2.3 npm

Modules are one of the main reasons for the success of nodejs. It is a good reason to use JS if –as seen in section 3.8.2.1– it is possible to create a server with so little an effort. And, as will be seen later in the course (see section 6.1), extending this minimal echo server to something really useful is not much more difficult.

The management of modules is important to keep control in larger projects. npm (**n**ode **p**acket **m**anager)[14,15] is a packet manager for nodeJS. It can automatically manage the availability of modules in a nodeJS project. Other things npm is capable of are beyond this course.

3.8.2.3.1 using npm A nodeJS project can define a file `package.json`, normally located in the project root directory. The easiest way to achieve this is by typing `npm init` and answer the following questions or even by `npm init --y` and amending the necessary things afterwards. In this file (obviously writtenin JSON), the packages used by the project, are configured. If this is done correctly, the call `npm install` in the project root will download all the necessary modules and store them in a directory called `node_modules`.

Dependencies between modules (modules needing other modules) are cared for automatically.

To add a downloadable module to the project, it is necessary to tell `package.json` to include the module by `npm install <modulename>`.

It needs to be noted that global modules can be used and installed as well supposed having the necessary user permissions. To add a module globally, the command is `npm install <modulename> --global`. The location of these modules depend on the

[13] https://www.tutorialsteacher.com/nodejs/nodejs-module-exports
[14] https://en.wikipedia.org/wiki/Npm_(software)
[15] https://www.npmjs.com/

underlying OS.

The directory `node_modules` tends to grow very large even for small pojects. If using git or other VCS (**V**ersion **C**ontrol **S**ystem)'s, it is a good idea to add `node_module` to the ignore list.

Chapter 4

Introduction to the Raspberry Pi

4.1 Overview

The Raspberry Pi is a minicomputer based on the ARM architecture. If compared with e.g. the Arduino line of computers, there is a massive increase in processing power. There are several models and revisions which grew to 'a little family' over time, with model 4 being the most recent addition.

A screen and a keyboard can be connected to th Pi. The resolution and performance are different between the Pi models. For IoT applications, a *headless* configuration is standard. No screen or keyboard are attached to the Pi. Access is only possible using network connections. The OS is then set up without GUI in most cases. using LINUX, a compromise can be striked by installing X11, but no X-server.

The Pi provides a header to add experimental electronics, which lead to the development of many shields called *hat*'s. The header has –amongst others– GPIO (**G**eneral **P**urpose **I**nput **O**utput)'s and I2C (**I**nter-**I**ntegrated **C**ircuit (IIC $\rightarrow I^2C$)) for simple connection of popular electronic circuits –often already known from the Arduino 'electronics zoo'.

Pi's are most often run with a LINUX OS, sometimes with Windows. With LINUX, the Pi presents a versatile machine, manageable like a standard LINUX PC, but much smaller, using less power and still having enough performance to serve many IoT needs.

4.2 Setting up

The OS is stored on a memory card. This memory card can be written to by standard card readers. Therefore, the chosen OS can be written by a standard PC, Windows or LINUX, inserted in the Pi and will then run from there. The most popular images are provided by raspberry themselves on their web sitehttps://www.raspberrypi.org/downloads/. How to write a disk image to a memory card is shown on their site as well [1].

As IoT devices usually are not equipped with a screen and/or keyboard, a headless installation is a realistic scenario and favoured for this course. Then, an image without desktop, as found on the raspberry web site[2] is the sensible way to choose.

4.2.1 Preparation

A headless install requires some extra preparations for a successful start. Starting with model 3 the Pi has WiFi on board. To connect to a WiFi directly, the memory card with the OS needs to be configured.

The memory card with the written image consists of two partitions, called *boot* and *rootfs*. In the root directory, an empty file called `ssh` needs to be created, e.g. using the command touch. Additionally, a file `wpa_supplicant.conf` with the content

```
country=GE # Your 2-digit country code
ctrl_interface=DIR=/var/run/wpa_supplicant GROUP=netdev
update_config=1
network={
ssid="THE_SSID"
psk="WIFI_PW"
key_mgmt=WPA-PSK
}
```

[1] https://www.raspberrypi.org/documentation/installation/installing-images/
[2] https://www.raspberrypi.org/downloads/raspbian/

is neccessary, with `THE_SSID` replaced by the real ssid (**s**ervice **s**et **id**entifier) and `WIFI_PW` with the wifi password.

After the first start, these files will vanish from the boot drive, ssh will be enabled and the network configuration should be at its correct place in the configuration.

4.2.2 Starting

The correctly configured memory card is inserted in the Pi and the Pi is connected to ethernet (if wired network is used) and to power. It should start up and should automatically connect to the network.

4.2.3 Finding

To find the Pi, the DHCP (**D**ynamic **H**ost **C**onfiguration **P**rotocol) client list of the associated router can be looked up. If there is no such control over the network, a search for the Pi needs to be started. *nmap* is the tool of choice in the LINUX/UNIX world.

```
nmap -sP 192.168.0.0/24 | awk '/^Nmap/{ip=\$NF}/B8:27:EB/{print ip}'
```

searches in a pouplar situation (a local private network 192.168.0.0) for a device with a MAC address matching the factory code of Raspberry Pi Foundation, isolating all Pi's in the network.

If installed, *avahi* is a good choice too. `avahi-browse -avc` will list all devices in the local network. This could be filtered further to isolate Pi's.

4.2.3.1 Contacting via ssh

With a known IP (**I**nternet **P**rotocol) address, the Pi can be connected using `ssh -l pi IP` with `IP` being the IP-address of the Pi. If successful, a terminal should b available to the user to do work on the Pi.

4.2.4 Improvements

To have just a terminal is somehow unsatisfactory. Sending files to the Pi, editing files on the Pi, all this is cumbersome. Enhanced connectivity would be helpful and can be achieved easily, especially when working with a LINUX environment.

4.2.4.1 SFTP

SFTP is a protocol on top of ssh, allowing to transfer files over ssh. most SFTP clients like filezilla[3] or WinSCP[4] support this protocol. As soon as sh is enabled on the pi, files can be transferred this way.

4.2.4.2 Mounting

In LINUX, the filesystem of a remote Pi can even be integrated into the file system of a local machine transperently. The package *sshfs* supports mounting a remote machine, connected to using SFTP.

```
sshfs pi@IP:/ ~/mnt/MP
```

The login name `pi` is followed by the IP-address of the Pi. the password will be asked for.

Windows knows a tool named *Win-SSHFS* to serve the same purpose.

4.2.4.3 X-Server

A complete desktop is not typical for Pi environments in a IoT setting. Therefore, working with the support of a GUI is not self-evident. The graphical environment in LINUX is seperated into two parts, *X-server* and the *X-client*. The X-server provides the communication for graphical primitives like points or lines for the output and the communication with input devices like a mouse or keyboard. The X-client uses this communication protocol to produce the display of an application and the communication interface of the user to this application.

[3] https://filezilla-project.org/
[4] https://winscp.net/eng/index.php

The footprint of just the X-server is by far less than the footprint of a graphical desktop as the whole window management –depending on the type and the 'luxury' of the configuration– eats up far more file and memory space, but is not needed on the machine itself.

An application running a GUI does not deal with its display on the machine, but will just send the graphical primitives needed for display to the displaying machine and in return will receive the interactions of the user with the program's UI. In a typical context, a Pi will run an X-client with the machine which connects to it running the X-server, window manager etc. to manage display and i/o communication.

On LINUX the support for such remotely running graphical programs is native as long as the machine runs with a GUI. In Windows, a server for the X11-protocol is not included. There are such servers available, e.g. Cygwin/X.

On a LINUX system, the tunneling of the X11-protocol through an ssh connection is allowed using the -X option. Therefore, logging into a PI with `ssh -XL pi IP` allows for GUI-applications to be run and displayed on the connecting machine. The Pi needs to have X11 installed, but not with a server or a window manager.

On Windows, the approach is the same.

To be able to forward X11, the Pi needs to be configured accordingly. With most installations it works out of the box. A good explanation is given in the documentation of Cygwin/X[5].

[5] https://x.cygwin.com/docs/ug/using-remote-apps.html

Chapter 5

Building a IOT device

In this course, focused on computer science and not on electronics, a prefabricated hat is used, called PiFace. It provides 2 relays to switch domestic appliances (240V~), 8 LED's (two of them in connection with the relays) and 4 pushbuttons. These peripherals are used to convey the concepts of sensors and actors without the hassle of building the circuits in the course. An example what can be done with this equipment was demomstrated by a project at the FHGR (**F**achhochschule **H**ochschule **G**rau**B**ünden)[1].

5.1 Pi, accessing peripherals

The Pi as it is used in this course is a classical LINUX machine. Therefore, HW access needs to be made through the HAL by using a driver dedicated to the Pi. This driver is build through the standard toolchain. But first of all, the SPI needs to be enabled through the command

```
sudo raspi-config
```

in the interfacing options. The following commands will build and install the driver

```
sudo apt-get install automake libtool git
```

[1] https://vimeo.com/219830495

```
git clone https://github.com/thomasmacpherson/piface.git
cd piface/c
./autogen.sh && ./configure && make && sudo make install
sudo ldconfig
cd ../scripts
sudo ./spidev-setup
```

Some of the commands might not be necessary on some systms as e.g. `automake`, `libtool` or `git` may already be installed because they are standard tools.

Building the driver is only the first step to use the PiFace. It needs to be made accessible with JS, as nodejs is supposed to be used in this course.

```
npm install piface
```

achieves this goal. It is a module abstracting the access to the driver to make it easy to command the PiFace using nodejs.

```
var pfio = require("piface");
pfio.init();
```

provides the functionality of the PiFace in the object `pfio`, with `init()` initialising the board.

```
pfio.digital_write(pin, state);
```

writes to a pin, e.g. `pfio.digital_write(0, 1);` will switch on LED 0. As relay 0 and relay 1 are connected to LED 0 and LED 1 respectively, both will switch together.

```
var state = pfio.digital_read(pin);
```

will read the state of the pushbuttons 0..3. `read_input` and `write_output` will read and write as well, but will use the whole register in on access cycle. The codes will then be bit fields, which are more tedious to be constructed or dissected. In the course they will not be used.

5.2 Access over a network

TCP (**T**ransmission **C**ontrol **P**rotocol) is one of the major protocols of the internet. It is located in the transport layer. It is connection oriented, recognises data loss and repairs it. Most of the time TCP works togeter with IP which is a packet oriented data transfer from one computer to another. A TCP connection is identified by both the trasnmitter and the receiver and at both ends by the pair of IP-address and *port number*. Data content is not defined, so any stream of data can be sent. A server providing TCP data transfer can be sufficient for IoT communication. This is often the case when e.g. mobile apps talk to IoT devices over the internet.

A rudimentary example using a *pi with a digital face 2* as a server and any other device running JS as a client is given in appendix B.1. This sample program can be used as a starting point for more elaborate function implementations considering basic TCP as a connection.

Chapter 6

Building a WOT device

6.1 Introduction

A TCP connection is nice and easy to handle, but it is low level and quite basic. As this course presses to use the web standard technologies, it would be appropriate to use higher level web protocols to obtain the same results as with a TCP connection.

6.1.1 Websockets

Websockets fullfill the requirement given above, being at the same communication layer as HTTP. As HTTP has `http:` (`https:`) as URL schema, websockets use `ws:` (`wss:`). The standard ports for websockets are identical to http ports. These are open in nearly all systems involved in web activity. Websockets are initiated by a HTTP request with upgrade.

The main difference between the two is that websockets can keep connections open ater a communication has finished. HTTP will just open a connection, communicate via a *request/response* pair and close the connection again. Both types of communication are initiated exclusively by the client. This –obviously– blocks the server from pushing data on its own initiative in HTTP. Having a websocket open, the server can use this channel to push data. Obviously, this way of communication is very attractive for IoT-Applications as data

needs to be pushed from the device to a client when it becomes available. Otherwise the client would need to poll data in intervals, which is much more effort and less data efficient.

An application can therefore access an IoT device through standard web technologies on at least the same layer level as HTTP.

Starting with HTML5, the major markup language in the web supports web sockets. All recent major browsers support this technology. This allows for the inclusion of websocket connections into web pages, which opens up completely new possibilites. Like AJAX (**A**synchronous **J**ava Script **A**nd **X**ML) opened up the way for dynamic web pages without reloading the whole page, websockets allow for web pages which are updated automatically by remote data being generated and pushed to the client by the server. Incoming data will issue an event which can be caught by a function. The data then can be used for manipulating the web page, which in turn will show the new data.

6.1.2 MQTT

Even higher in the protocol stack there is MQTT. Actually, MQTT is often implemented using websockets. It is a lightweight protocol following the publish/subscribe pattern. A central broker manages the transfer of messages. Devices can register as publishers for a given subject. The subject has the form of a hierarchical path like a directory. With this path, a device can issue messages, which are received by the broker. Subscribers can subscribe to a given subject and listen to messages sent to the broker with this associated path. The broker then will send a message to each subscriber to a subject when a publisher sends a message associated with this subject. The subscriber and the publisher do not need to know about each other.

6.1.2.1 Properties

MQTT knows about QoS (**Q**uality **o**f **S**ervice) in three levels.

- *fire-and-forget* There is no guarantee that a message reaches the subscriber at all. This can be a good choice e.g. for measurement data, which is only interesting in real-time
- *deliver-at-least-once* a message is guaranteed to be delivered, but may be delivered multiple times. This is easier than the next level, because subscribers, which are temporarily disconnected may get a message again after reconnection.
- *deliver-exactly-once* A message will be delivered exactly once. This is the highest level of quality.

Security can be implemented using TLS (**T**ransport **L**ayer **S**ecurity). Additionally, authorisation may be required for clients.

MQTT originally was intended to be situated on top of TCP. But TCP keeps connections open until they are closed. This is not optimal from a energy-efficiency point of view. There are basically two solutions, open and close the connection for each transaction and getting away from TCP. A version of MQTT on top of UDP is called MQTT-SN with *SN* standing for *sensor networks*. It is very well suited for IoT environments.

6.1.2.2 Implementations and tools

Possibly the most common broker is *mosquitto*[1]. It is very leightweight and (especially on linux) very easy to use.

A tester is *MQTT.fx*[2]. With this tool, brokers can be connected. When connected, subscriptions can be issued and messages published. All reactions of the broker can be displayed.

6.1.3 COaP

CoAP has the same target domain as MQTT, but is designed with even more focus on efficiency. Using UDP on the network level by design it doesn't need connections. On the

[1] https://mosquitto.org/
[2] https://mqttfx.jensd.de/

network layer it is mostly used with 6LoWPAN. It is a request/response characteristic like REST. It can use a pub/sub mechanism, but without a broker inbetween. Therefore, the things must themselves know about the clients and push all messages to the subscribing clients.

6.1.4 Security

Security is one of the big issues in IoT. Considering that there are about 20 billion devices around at the time of writing this course script, with a strong tendency of growth, there exists a new and very big playground for blackhats with a lot of power to be tapped into. However, this is not in the focus of this course, as –especially when using WoT-technologies– security is no different to security in normal web applications. Application layer protocols like MQTT have their own measures for security e.g. authorisation on top of the security technologies of the layers below.

It may still be noted that the author wants to press the subject and motivate all users of IoT in a commercial context to put their highest priorities on safety and security issues.

6.2 Building a WOT WWW server using NodeJS

A simple HTTP server in JS can be set up very quickly using the library `http`. The following echo-server shows the principle

Listing 6.1:

```
// content of index.js
const http = require('http')
const port = 3000
const server = http.createServer(requestHandler)

server.listen(port,listenerAdded)

function listenerAdded(err)
{
  if (err) {
    return console.log('something bad happened', err)
  }
```

```
  console.log(`server is listening on ${port}`);
}

function requestHandler(request, response)
{
  console.log(request.url)
  response.end('Hello Node.js Server!')
}
```

It could be even simpler as the example given, but a minimum of error handling is included.

6.2.1 A WWW server for the browser

The next example shows, how –with very little effort– a server reacting to directory or file requests can be implemented using `express`.

Listing 6.2:

```
var express = require('express');
var app = express();
var responseTemplate="<html><head><title>Small Server</
  ↪   title></head></html><body>You requested
  ↪   xxxbodytextxxx</body>"
var replaceTemplate=/xxxbodytextxxx/g;

port = 8080;

app.listen(port);
console.log("Server listens at " + port);

app.get('/', reqHome);
app.get('/directory', reqDir);
app.get('/directory/*', reqPoint);
app.get('*', reqDefault);

// respond to root request
function reqHome(request, response)
{
  finishResponse(response,"the home of me.");
}

// respond to directory request
function reqDir(request, response)
{
  finishResponse(response,"the directory 'directory'.");
}

// respond to directory request
function reqPoint(request, response)
{
```

```
  finishResponse(response,"something in the directory '    ↩
    ↪    directory'.");
}

// respond to all uncaught requests
function reqDefault(request, response)
{
  finishResponse(response,"just something.");
}

function finishResponse(response, fillText)
{
  response.writeHead(200,
  {
    'Content-Type' : 'text/html'
  });
  responseText = responseTemplate.replace(replaceTemplate    ↩
    ↪    ,fillText);
  response.write(responseText);
  response.end();
}
```

It needs to be noted that this server does not return HTML files from disk but HTML text synthesised directly in the server itself. It demonststrates how arbitrary answers can be created as a response to given requests.

6.2.2 A WWW server for M2M communication

IoT devices are predominantly intended for M2M communication. Therefore, HTML is not the most prevalent way to communicate. But there is only little change necessary in Listing 6.2 to make it communicate with machines.

Listing 6.3:

```
var express = require('express');
var app = express();

port = 8080;

app.listen(port);
console.log("Server listens at " + port);

app.get('/', reqHome);
app.get('/directory', reqDir);
app.get('/directory/*', reqPoint);
app.get('*', reqDefault);

// respond to root request
function reqHome(request, response)
```

```
{
    finishResponse(response, "/");
}

// respond to directory request
function reqDir(request, response)
{
    finishResponse(response, "/directory");
}

// respond to directory request
function reqPoint(request, response)
{
    finishResponse(response, "/directory/???");
}

// respond to all uncaught requests
function reqDefault(request, response)
{
    finishResponse(response, "/???");
}

function finishResponse(response, fillText)
{
    var replaceTemplate = "xxxresponsexxx";
    var replaceRegEx = new RegExp(replaceTemplate, "g");
    var responseTemplate = "[ 'targetURL':'" +
      ↪   replaceTemplate + "']";
    response.writeHead(200, {
        'Content-Type':'application/json'
    });
    responseText = responseTemplate.replace(replaceRegEx,
      ↪    fillText);
    response.write(responseText);
    response.end();
}
```

As can be seen in Listing 6.3, only the synthesis of the answer itself needs to be adapted to make it respond in JSON.

6.2.3 A universal WWW server

The HTTP header can tell the server the expected format of the answer. This allows for the design of servers which, depending on a request originating from a browser or from a communicator being a machine, can answer appropriately. Associated sample program code can be seen at Listing B.3. There, it can be seen that the strcuture is very similar to the examples in subsection 6.2.1 and subsection 6.2.2. The response needs to discriminate

by checking the *accept* field in the HTTP-header and react accordingly. The example is kept simple and just discriminates between JSON and everything else. It could be easlily compromised. But the principle becomes clear.

6.3 Sending and receiving over MQTT

Browsers do not directly support MQTT. Therefore, MQTT needs to be transported by other means. Websockets are ideal although they keep open a connection. This is normally no problem at this end of the communication as the energy-efficiency is not of high priority.

A JS library to provide such MQTT over websockets is paho[3].

Listing B.5, Listing B.6 and Listing B.7 set up a primitive MQTT publisher and a web site working with MQTT over websockets with the use of the *paho* library. It can be seen that the web site does not pull messages. The messages are pushed by the publisher to the broker. The broker then pushes each message to the subscribing web page. Thus, a very quick updating of the web page occurs, whenever a message is generated by the publisher.

[3] https://www.eclipse.org/paho/

Chapter 7

Final statements

This script gives an introduction into all important subjects of IoT and WoT devices. It is not intended to hold all information necessary to become an expert in the area. Personal research into the depths of the individual areas is demanded to become a productive engineer. However, the contents of the course should be sufficient to develop first insightful applications on the server side as well as on the client side.

Students of this course are encouraged to focus on the concepts given in the material and play with the knowledge acquired. To extend on the knowledge, it is even not necessary to have typical IoT HW to one's disposal. On the other hand, usable IoT HW became quite affordable in the last years. Recommendable platforms are:

- The arduino family, the door-opener into the IoT-world[1,2]
- Node MCU, extremely cheap, WiFi (and BT) on board, more processing power than the Arduinos, most models with a small footprint[3]
- Raspberry Pi[4]

[1] https://en.wikipedia.org/wiki/Arduino
[2] https://en.wikipedia.org/wiki/List_of_Arduino_boards_and_compatible_systems
[3] https://en.wikipedia.org/wiki/NodeMCU
[4] https://en.wikipedia.org/wiki/Raspberry_Pi

With the example HW-platforms above, a wide range of applications in the *things world* can be implemented.

Bibliography

[1] Dominique Guinard and Vlad Trifa. *Building the Web of Things: With Examples in Node.Js and Raspberry Pi*. Manning Publications Co., Greenwich, CT, USA, 1st edition, 2016.

[2] Phillip A. Laplate. *Real-Time Systems Design and Analysis*. IEEE Press, 2004.

Index

Appendix A

HTML and CSS

The two components, HTML and CSS are complementary components, the first dealing with document structure and content, the second with appearance, supporting the pradigm of seperation of content and presentation[1].

HTML is a markup language, indicating the document structure by marking the elements by tags. A good source, quite exhaustive, is w3schools[2], providing tutorials and references to HTML, CSS and JS.

As an URL for a browser, a HTML document is given. This docuent will normally first specify a document type, e.g.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
"http://www.w3.org/TR/html4/strict.dtd">
```

for a HTML4 document. It defines, to which sec:tandard a document adheres. Most browsers tolerate to omit this information.

Most browsers deal with the syntax of HTML documents with high tolerance. They will try to present a document as sensibly as possible. The positive cconsequence is that even badly-formed documents will be displayed as far as possible. On the other hand, precise and correct expression is not forced and therefore not followed as closely as e.g. in

[1] https://en.wikipedia.org/wiki/Separation_of_content_and_presentation
[2] https://www.w3schools.com/

strict programming languages like C or C++ where malformed documents will lead to error messages in most cases immediately.

A.1 Tags

Tags are given by `<...>`. Each tag normally has a counterpart leading to `<tag>....</tag>` with `tag` being a single word. There are exceptions though. `<br>` for example indicates a line break. In fact, this example is not as simple as it seems as it is only valid for HTML. XHTML (**E**xtensible **H**yper**t**ext **M**arkup **L**anguage) being a close relationship to HTML, needs this tag to be `<br />` to indicate that this tag is the opening tag and the closing tag combined in one tag. HTML5 accepts both variants.

Optional attributes are given in the form

```
<tag attr1="content 1" attr2="content 2" ... attrN="content N">
```

with no limit to the number of attributes.

Tags can be nested into each other, building a tree-like organisation. Tags are not to be crossed.

```
<a>
  <b>
  </b>
</a>
```

is legal, while

```
<a>
<b>
</a>
</b>
```

is not. This rule is obvious, taking into consideration that a HTML document must be a hierarchical tree.

A HTML document consists of two major parts, head and body, indicated by `<head>` and `<body>` respectively. Both parts are contained in a `<html>` tag. A basic document can look like:

```
<!DOCTYPE html>
<html>
  <head>
    <title>my title</title>
  </head>
  <body>
    <h1>a heading</h1>
    <p>a paragraph</p>
  </body>
</html>
```

JS can be placed everywhere in the script. Popular locations are between header and body or at the end of the body.

A.1.1 Header

The header contains the title, and meta-information with the allowed tags being

- <title> (this element is required in an HTML document)
- `<style>`
- `<base>`
- `<link>`
- `<meta>`

- `<script>`
- `<noscript>`

A.1.2 Body

The body holds everything else, which is the main part of the document.

The document is structured by several levels of sectioning. This is achieved with the tags `<h1>` to `<h6>`. The structuring can be extended by the use of `<div>` and `<span>` tags apart from many other possible tags to seperate elements from each other. For further reading the audience is adviced to refer to the sources given above.

A.2 CSS

CSS is style information and copes with appearance. As a teaser what can be achieved using CSS, a visit to the CSS zen garden[3] is recommended.

CSS works with a two-step approach:

1. Select, what you want to show in a given way
2. Define, how you want to show, what was selected

It is a key-value pair with the selector being the key and the definition being the value.

Additionally, if selections collide –which often is intentional– there are rules of precedence. This leads to definitions overloading each other. The more specific definitions will overwrite more general ones, why the system is called *cascading*. More information about specificity is found in w3schools[4]. It allows for specifying appearance rules for general element selections without prohibiting further change and particularising appearance definitions for more 'local'

[3] http://www.csszengarden.com/
[4] https://www.w3schools.com/css/css_specificity.asp

elements or groups of elements. These cascades can become quite intricate. To follow up specific cascades, the debug mode of browsers like chrome[5] can be extremely helpful.

Comments in CSS are given in C style.

A.2.1 Selectors

Selection can be done by specifying a tag name, an id or another attribute etc., and combinations of such tag identifications. The very entertaining CSS diner[6] is a very good sparring environment to get used to the selectors and the possibilities associated to them.

A.2.2 Definitions

The definitions start with `{` and end with `}`. Inbetween these brackets, a single definition is a key-value pair of a attribute and its value.

A.2.3 Examples

```
<style>
p {
color: red;
text-align: center;
}
</style>
```

will select all `<p>`-tags and present them in red and centered.

```
<style>
#theID {
text-align: center;
```

[5] https://developers.google.com/web/tools/chrome-devtools/css
[6] https://flukeout.github.io/

```
color: red;
}
</style>
...
<p id="theID">I'm selected</p>
<p>I'm not selected</p>
```

selects all elements with the id attribute set to `"theID"`, so `#` indicates an id. An id shall be used only once in the DOM.

```
<style>
.theClass {
color: red;
text-align: center;
}
</style>
...
<h1 class="theClass">I'm selected</h1>
<p class="theClass">I'm selected</p>
```

Both elements will be selected, although they are of different tag names.

Appendix B

Listings

B.1 TCP on the Pi

B.1.1 Server (Pi with PiFace Digital 2)

Listing B.1:

```
var net = require('net');
var ip="0.0.0.0";

var server = net.createServer(connect);
var pfio = require("piface");
pfio.init();

function connect(socket)
{
  console.log('connected');
  socket.on('data', receive.bind(socket))
}

function receive(data)
{
  let dataOK=false;
  if (data=="[0,1]")
  {
    pfio.digital_write(0,1);
    dataOK=true;
  }
  if (data=="[0,0]")
  {
    pfio.digital_write(0,0);
    dataOK=true;
  }
  if (dataOK)
```

```
  {
    this.write('data OK:' + data);
  }
  else
  {
    this.write('data not OK:' + data);
  }
  this.pipe(this);
}

if(2<process.argv.length)
{
  //get connection parameters from program options
  port=process.argv[2];
  console.log('listen at: ' + port);
}
else
{
  console.log("Usage: nodejs server.js ip port");
  process.exit(-1);
}

server.listen(port);
```

B.1.2 Client

Listing B.2:

```
var net = require('net');

var client = new net.Socket();

if(3<process.argv.length)
{
  //get connection parameters from program options
  ip=process.argv[2];
  port=process.argv[3];
  console.log('listen at: ' + ip + ":" + port);
}
else
{
  console.log("Usage: nodejs client.js ip port");
  process.exit(-1);
}

//listener for replies
client.on('data', function(data) {
  console.log('Received: ' + data);
});

client.on('close', function() {
  console.log('Connection closed');
});

relay1On();
setTimeout(relay1Off, 2000);
```

```
setTimeout(killClient, 5000); //kill client at the end

function relay1On()
{
  client.connect(port, ip, function() {
    console.log('Connected');
    client.write('[0,1]');
  });
}

function relay1Off()
{
  client.connect(port, ip, function() {
    console.log('Connected');
    client.write('[0,0]');
  });
}

function killClient()
{
  client.destroy();
}
```

B.2 HTTP

B.2.1 HTTP server respecting accept fields

Listing B.3:

```
var express = require('express');
var app = express();

port = 8080;

app.listen(port);
console.log("Server listens at " + port);

app.get('/', reqHome);
app.get('/directory', reqDir);
app.get('/directory/*', reqPoint);
app.get('*', reqDefault);

// respond to root request
function reqHome(request, response)
{
    if (isM2M(request))
    {
        finishResponseJSON(response, "/");
    }
    else
    {
        finishResponseHTML(response, "the home of me.");
    }
```

```
}

// respond to directory request
function reqDir(request, response)
{
    if (isM2M(request))
    {
        finishResponseJSON(response, "/directory");
    }
    else
    {
        finishResponseHTML(response, "the directory '    ↩
          ↪    directory'.");
    }
}

// respond to directory request
function reqPoint(request, response)
{
    if (isM2M(request))
    {
        finishResponseJSON(response, "/directory/???");
    }
    else
    {
        finishResponseHTML(response, "something in the    ↩
          ↪    directory 'directory'.");
    }
}

// respond to all uncaught requests
function reqDefault(request, response)
{
    if (isM2M(request))
    {
        finishResponseJSON(response, "/???");
    }
    else
    {
        finishResponseHTML(response, "just something.");
    }
}

function finishResponseJSON(response, fillText)
{
    var replaceTemplate = "xxxresponsexxx";
    var replaceRegEx = new RegExp(replaceTemplate, "g");
    var responseTemplate = "[ 'targetURL':'" +    ↩
      ↪    replaceTemplate + "']";
    responseText = responseTemplate.replace(replaceRegEx,    ↩
      ↪    fillText);
    finishResponse(response, 'application/json',    ↩
      ↪    responseText)
}

function finishResponseHTML(response, fillText)
```

```
{
    var replaceTemplate = "xxxbodytextxxx";
    var replaceRegEx = new RegExp(replaceTemplate, "g");
    var responseTemplate = "<html><head><title>Small
      ↪   Server</title></head></html><body>You
      ↪   requested " + replaceTemplate + "</body>";
    responseText = responseTemplate.replace(replaceRegEx,
      ↪    fillText);
    finishResponse(response, 'text/html', responseText)
}

function finishResponse(response,type,responseText)
{
    response.writeHead(200, {
        'Content-Type':type
    });
    response.write(responseText);
    response.end();
}

function isM2M(request)
{
    /*
    -get accept field of request's header
    -split up into an array
    -test, if one array element indicates json format
    */
    if(-1==request.headers.accept.split(',').indexOf("
      ↪   application/json"))
    {
        //no json format founs
        return(false);
    }
    else
    {
        //json format found
        return(true);
    }
}
```

B.3 MQTT

For the following listings it is assumed, that an MQTT broker runs on localhost with the standard ports for MQTT configured. Additionally on port 9001 there need to be configured a `ws:` protocol. A smple configuration for *mosquitto* is given here:

Listing B.4:

```
listener 1883 0.0.0.0
protocol mqtt

listener 9001 0.0.0.0
```

```
protocol websockets
```

B.3.1 publisher

A very basic publisher:

Listing B.5:

```
'use strict';

//application consts and vars
const topic = 'iliauni';
var urlBroker = 'mqtt:127.0.0.1';
var mCount = 0;

const mqtt = require('mqtt')
const client = mqtt.connect(urlBroker);
client.on('connect', onConnect);

function onConnect() {
    console.log('connected to ' + urlBroker);
    client.subscribe(topic);
    MessageLoop();
}

function send() {
    var message='Message ' + mCount++
    client.publish(topic, message);
    console.log(message);
}

function MessageLoop() {
    send();
    setTimeout(MessageLoop, 10000);
}
```

B.3.2 subscriber

A very basic subscriber web page:

Listing B.6:

```
<html xmlns="https://www.w3.org/1999/xhtml">

<head>
    <title>MQTT over websockets</title>
</head>

<body>
    <div>URL:
    </div>
    <div id="url">
```

```
    </div>
    <div>Subject:
    </div>
    <div id="subject">
    </div>
    <div>Status:
    </div>
    <div id="status">
    </div>
    <div>Last Message:
    </div>
    <div id="message">
    </div>
</body>
<script src="https://cdnjs.cloudflare.com/ajax/libs/paho-mqtt/1.0.1/mqttws31.js" type="text/javascript"></script>
<script src="./main.js" type="text/javascript"></script>

</html>
```

Listing B.7:

```
const mttqClientHOST = "ws://127.0.0.1";
const mttqClientPort = 9001;
const mttqClientPath = "/";
const mttqClientURL = mttqClientHOST + ":" + mttqClientPort + mttqClientPath;
const mttqClientId = '';

const topic = 'iliauni';

var mttqClient;

function mqttOnConnect() {
    alertUser('Info', "connected");
    mttqClient.subscribe(topic);
    document.getElementById("subject").innerHTML = topic;
    mttqClient.onMessageArrived = mqttOnMessage;
}

function mqttOnMessage(message) {
    document.getElementById("message").innerHTML = message.payloadString;
}

function setupMQTT() {
    alertUser('Info', "trying to connect: " + mttqClientURL + ", id: " + mttqClientId);
    mttqClient = new Paho.MQTT.Client(mttqClientURL, mttqClientId);
    mttqClient.connect({ onSuccess: mqttOnConnect });
}

function shutdownMQTT() {
    mttqClient.disconnect();
}
```

```
function alertUser(type, message) {
    document.getElementById("status").innerHTML = type +
      ↪    ": " + message;
    console.log(type + ':' + message);
}

function init() {
    document.getElementById("url").innerHTML =
      ↪    mttqClientURL;
    setupMQTT();
}

window.onload = init;
```