Sem vložte zadání Vaší práce.

České vysoké učení technické v Praze
Fakulta informačních technologií
Katedra počítačových sytémů

Diplomová práce

Sběr dat z meteorologických stanic

Bc. Slavomír Dittrich

Vedoucí práce: Ing. Pavel Vít

11. ledna 2016

Poděkování

Rád bych poděkoval všem, kteří mi byli nápomocni při tvorbě této diplomové práce. Děkuji především vedoucímu této práce Ing. Pavlu Vítovi za věcné připomínky a cenné rady. Dále touto cestou děkuji rodičům za jejich trpělivost a neustálou podporu nejen při studiu, ale i v osobním životě, které si moc vážím a za kterou jsem a vždy budu moc vděčný.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů, zejména skutečnost, že České vysoké učení technické v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona.

V Praze dne 11. ledna 2016 .....

Odkaz na tuto práci

Dittrich, Slavomír. *Sběr dat z meteorologických stanic*. Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2016.

Abstract

Thesis deals with collecting, storage and evaluation of data obtained from Arduino based meteorological stations. The purpose of the thesis is to design a complex system for data collecting. The primary aim of the thesis is to design and implement the way of communication between meteorological stations and server environment. Additionally the thesis contains an overview and discussion about possible solution of each part of the whole system and a model example of its usage.

Keywords Arduino, meteorological stations, sensors, MQTT, big data, NoSQL, distributed systems

Abstrakt

Práce pojednává o problematice sběru, ukládání a vyhodnocování dat, získávaných z meteorologických stanic, vytvořených na platformě Arduino. Výsledkem práce je vytvoření univerzálního systému pro sběr dat. Hlavním bodem je návrh způsobu komunikace mezi meteorologickými stanicemi a serverovým prostředím. Součástí práce je také přehled a diskuze o možnostech řešení jednotlivých částí výsledného systému a příklad a ověření modelového řešení.

Klíčová slova Arduino, meteorologické stanice, senzory, MQTT, big data, NoSQL, distribuované systémy

Obsah

Seznam obrázků

Seznam tabulek

Úvod

Internet věcí, senzory, big data, cloud. Jedná se o pojmy, které se v dnešní době stále více používají a rozšiřují. Lidé a jejich aktivity produkují obrovská množství dat, jejichž celkový objem exponenciálně roste. Na základě výzkumu a predikcí společnosti IDC [1] se každé dva roky počet všech dat v Internetu zdvojnásobí. Jedním z příčin a nemalý podíl na této skutečností má také rozvoj a neustálý vývoj a inovace v oblasti mikroporecesorů. Především z hlediska HW dochází v posledních letech k rapidnímu rozvoji technologií. Technologické postupy výroby jsou více sofistikovanější a umožňují vytvářet zejména HW součásti malých rozměrů. Jsou to převážně nejrůznější vestavěná zařízení, typicky různé druhy senzorů, připojené k jednoduchému mikroprocesoru a síťovému adaptéru, jejichž úkolem je produkovat data a další užitečné informace. Taková data je nutné následně někde uchovávat, zpracovávat a vyhodnocovat. Na rozdíl od doby před 20 lety, není dnes problém najít HW, mnohem větší problém je najít pro něj vhodné využití.

Oblastí, ze kterých je možné získávat užitečná data, je nepřeberné množství. Mezi nejvýznamější odvětví patří například zdravotnictví, doprava, průmyslová výroba, vědecká činnost, ale i domácnost a aktivity, které jsou spojeny s běžným životem člověka. Ve zdravotnictví se jedná především o senzory pro měření zdravotního stavu pacientů, jako je teplota, tlak, srdeční činnost, a další. Nositelná elektronika je dalším odvětvím, které bylo značně inspirováno použitím různých senzorů ve zdravotnictví, a které se dnes také velice rychle vyvíjí. V oblasti průmyslu to mohou být prakticky jakékoliv informace typu provozní teploty zařízení, stavy měřičů, polohy servomotorů, apod. Je zřejmé, že jsou to právě i meteorologická data, která mají v dnešním světě svoji důležitost a na základě kterých jsou následně prováděny analýzy a předpovědi počasí.

Tato práce se zaměřuje na sběr dat z meteorologických stanic, které si dnes může kdokoliv, zejména díky cenové dostupnosti potřebných HW prostředků, sám sestavit, naprogramovat a provozovat. Hlavním přínosem práce je však výsledná architektura systému, způsob komunikace, uložení

a práce se sbíranými daty. Výsledný systém bude umožňovat sběr a vyhodnocení nejrůznějších druhů dat z různých aplikací. Systém bude také zajišťovat snadný přístup k těmto datům z dalších aplikací.

První kapitola se zabývá analýzou problematiky sběru dat, dnes v rozlehlém a rychle se měnícím prostředí. Jsou zde popsány především požadavky a charakteristika zadaného problému. Dále jsou zde uvedeny možnosti a přístupy, které je možné použít pro následnou realizaci řešení.

Ve druhé kapitole je popsán návrh a odůvodnění modelového řešení. Jsou zde uvedeny a porovnány různé možnosti, síťové protokoly a obecné postupy.

Třetí kapitola obsahuje ověření modelového řešení. Klade důraz na popis systému pro sběr dat, vytvořeného na základě návrhů z předchozí kapitoly a obsahuje jeho základní testování.

Výsledkem a hlavním přínosem práce je návrh systému pro sběr dat, který bude možné použít za účelem sběru různých druhů dat a podpora pro jejich následné zpracování dalšími aplikacemi. Výsledný systém bude možné nasadit a provozovat v různých prostředích, ať v cloudovém nebo jako řešení v lokální síti.

KAPITOLA 1

Analýza zadané problematiky

Sběr dat se už z historického pohledu řadí mezi důležité části systémů v nejrůznějších odvětvích. Umožňuje zejména vyhodnotit stav daného systému a stanovit stav jeho funkčnosti v průběhu času. U většiny systémů se jedná o proprietární řešení daného výrobce nebo dodavatele, které je vyrobeno pouze pro danou aplikaci. Výpočetní prostředky jsou stále výkonnější a systémy jsou větší a komplexnější. S tím souvisí i generované množství dat, které se stějně tak zvětšuje. Sběr a zpracování těchto dat je dnes již ve většině případů řízen počítačem.

Obecně platí, že čím větší je objem sbíraných dat, tím přesnější jsou výsledky při jejich zpracování a nedochází tolik ke zkreslení pozorovaných údajů. Avšak před nasazením nejrůznějsích nových systémů do provozu ještě někdy není předem známo, zda bude v budoucnu potřeba nějaká data sbírat, kde se budou případně ukládat a jak se budou zpracovávat. Právě tyto drobné detaily mohou být v budoucnu zdrojem například vysokých finančních výdajů a mohou být i časově náročné na samotnou implementaci. Obecným trendem pro návrh různých systému je dnes zajištění modulárnosti. Jedná se o použití takových komponent systému, které spolu mohou jednoduše komunikovat a předávat si informace a data. Pro většinu těchto komponent je dnes k dispozici několik možných řešení. Pro výběr toho vhodného z nich je třeba možná řešení porovnat a zhodnotit jejich vlastnosti.

Dnes existují řešení pro sběr dat, která jsou ve většině případů provozována v prostředí cloudu jako služby. Jedná se například o projekt

ThingSpeak[1] nebo Beebotte[2]. Uložení velkého objemu dat v cloudovém prostředí vyžaduje odpovídající technologii k zabezpečení uložených dat. Cloudové prostředí je vhodné zejména pro práci s menšími vzorky dat, které jsou využívány větším počtem uživatelů. Na druhou stranu pro velké objemy dat, se kterými obecně pracuje menší počet uživatelů a kdy cloudové řešení nenabízí dostatečný výpočetní výkon, nebo je zajištění požadovaného výkonu příliš finančně náročné, je výhodnější ukládat a zpracovávat data na prostředcích v lokálních sítích.

1.1 Požadavky a charakteristika zadání

Zadání práce je zaměřeno na vytvoření systému pro sběr dat, který bude sbírat data z jednoduchých meteorologických stanic a bude umožňovat jejich následné uložení, zpracování a vyhodnocování. Za účelem univerzálnosti použití navrhovaného řešení budou popsány postupy, na základě kterých bude možné sbírat data nejen z meteorologických stanic, ale i dalších různých zařízení.

1.1.1 Vytvoření systému pro sběr dat

Hlavním cílem práce je vytvoření systému pro sběr dat. Ve výsledku se bude jednat o univerzální systém, umožňující sběr libovolných dat z libovolných zařízení. Obecně existuje několik možných řešení tohoto problému, které budou diskutované v dalším textu.

1.1.2 Jednoduché meteorologické stanice

Za účelem ověření modelového řešení systému pro sběr dat budou vytvořeny jednoduché meteorologické stanice, použitím jedné z dnes běžně dostupných HW prototypových platforem a k nim dostupných senzorů. Tyto meteorologické stanice budou připojeny k síti Internet a budou umožňovat pravidelné, periodické zasílání dat, která budou následně zpracována na serverové části systému.

[1] ThingSpeak je open source cloudová platforma, poskytující API pro sběr a analýzu dat.

[2] Beebotte je cloudová platforma pro práci s živými daty z různých zařízení (fyzická zařízení nebo aplikace).

1.1.3 Serverové prostředí

Serverové prostředí bude zajišťovat zejména spolehlivé ukládání sbíraných dat. Za tímto účelem bude především nutné zvolit vhodný databázový nástroj. Mainframe servery a monolitické databáze jsou dnes již 50 let starou technologií, která stále uchovává více než 60% veškerých dat v Internetu. Z důvodu exponenciálního růstu objemu dat je nutné volit taková řešení, která umožňují a jsou připravena pro uchovávání velkých objemů a velkého množství operací nad těmito daty.

1.1.4 Komunikace mezi stanicemi a serverem

Komunikace mezi meteorologickými stanicemi a serverovým prostředím bude hlavním tématem práce. Přístupů a metod pro samotnou komunikaci dnes existuje značné množství. Za účelem obecného a univerzálního použití systému bude nutné zvolit vhodnou architekturu systému.

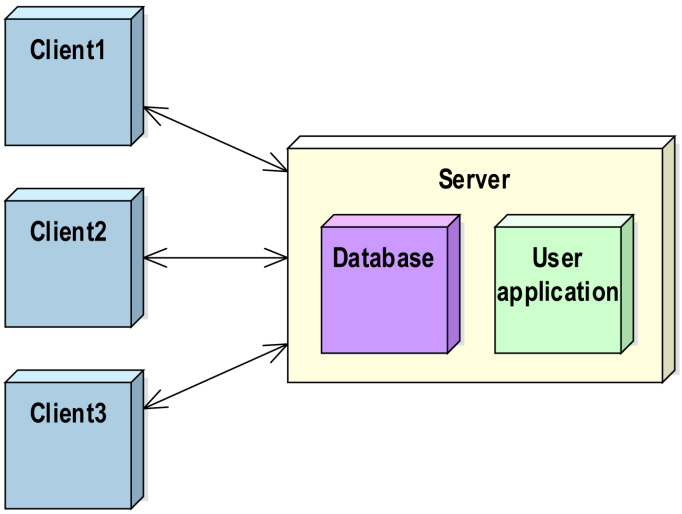
1.1.5 Prostředek pro práci s daty

Součástí řešení bude zajistit možnost přístupu ke sbíraným datům, která budou uložena na datovém úložišti serverového prostředí. Hlavním kritériem bude univerzálnost použití, aby byl zajištěn přístup k datům z různých dalších aplikací a systémů.

1.2 Architektura systému pro sběr dat

Možností a přístupů jak sbírat data je obecně velké množství a každé z nich má své výhody a nevýhody. V konkrétním případě se jedná o všechny možné architektury systémů z pohledu typu propojení jednotlivých uzlů a jejich rolí. Základní a zároveň nejjednodušší schéma, které se nabízí za tímto účelem použít je, dnes již běžná, síťová architektura typu **klient - server**. V tomto případě hrají roli klientů všechna zařízení, která data produkují i vyhodnocují data a tyto pak odesílají přímo na server, kde jsou dále ukládána. Zařízení, která produkují data jsou v tomto případě meteorologické stanice. Vedle toho mohou existovat klientské aplikace, jejichž hlavním účelem je umožnit uživateli vizualizaci, vyhodnocení a další požadované operace s daty, uchovávanými na serveru. Příklad této architektury je uveden na obrázku 1.1.

Hlavní výhody pro zvolení architektury systému typu klient - server jsou následující:



Obrázek 1.1: Typická architektura typu klient - server.

- **Centralizovanost**: server je hlavním prvkem architektury. Díky tomu je možné snadno spravovat zdroje a přístupová práva na jednom místě.
- **Bezpečnost**: pravidla pro bezpečnostní a přístupová práva mohou být jednoduše definována a spravována.
- **Zálohování**: data, která jsou uchovávána na serveru, mohou být jednoduše zálohována a v případě potřeby i obnovena.

Jako druhé řešení se nabízí použití vícevrstvé architektury, která přináší komplexnější možnosti použití výsledného systému. **Mezivrstva (middleware)** v takovém případě přináší především univerzálnost a širší možnosti použití celého systému. Většina mezivrstev je založena na výměně zpráv prostřednictvím různých frameworků (SOAP, REST), které používají široce používané formáty přenosu dat (XML, JSON). Důvody pro její začlenění mohou být následující:

- **Rozložení zátěže**: při použití dedikovaných výpočetních prostředků pro aplikaci mezivrstvy dojde ke snížení nároků na výpočetní prostředky ostatních částí systému.
- **Filtrace dat**: jeden z hlavních úkolů mezivrstvy je zajištění toho, aby komunikující strany (meteorologické stanice, výpočetní prostředí a další klientské aplikace) systému mohly pracovat pouze se správnými daty a to s takovými, která očekávají. Tímto se značně snižuje riziko, kdy se výpočetní prostředí zabývají zpracováním nežádoucích dat.

- **Nezávislost**: middleware vrstva je často implementována jako samostatná entita v síťové infrastruktuře, čímž logicky odděluje jednotlivé části systému. Toto logické členění s sebou přináší právě nezávislost při komunikaci s ostatními částmi systému, které spolu nekomunikují přímo, ale právě prostřednictvím mezivrstvy. Konkrétní výčet výhod použití mezivrstvy jsou uvedeny v dalším textu.
- **Interakce s dalšími službami**: hlavním účelem middleware vrstvy je zajištění jednodušší interakce s dalšími službami a aplikacemi. Tato skutečnost zajišťuje zejména spolehlivější a univerzálnější předávání dat a dalších informací mezi jednotlivými prvky v systému.

Typický příklad nasazení mezivrstvy je uveden na obrázku 1.2. Použití mezivrstvy pro podobné aplikace spolu přináší i některé nevýhody.

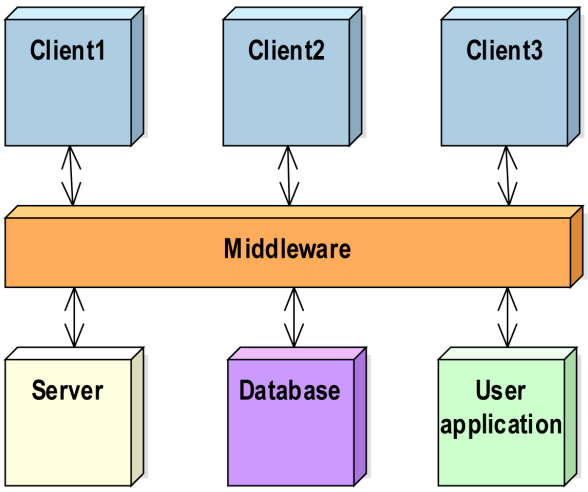
- **Snížení rychlosti**: rychlost přenosu informací a dat může být při používání mezivrstvy mírně snížena. Tuto skutečnost je však možné dostatečně kompenzovat při použití výkonnějších zařízení.
- **Bezpečnost**: použití mezivrstvy přináší do výsledného systému další prvek, který je třeba dostatečně zabezpečit především před případnými síťovými útoky. Vhodně zvolený bezpečnostní model je v tomto případě nezbytnou součástí.
- **Spolehlivost**: z hlediska spolehlivosti se jedná o další prvek architektury, jehož stav je třeba monitorovat, zejména z důvodu dostupnosti. Při správném a dostatečném nastavení administračních a dohledových nástrojů se však nejedná o zásadní problém.

1.3 Dílčí součásti řešení

V následujícím textu budou popsány jednotlivé dílčí součásti řešení, ze kterých se bude výsledný systém sestávat. Jednotlivé části řešení budou podrobně prozkoumány a budou uvedeny také příklady možností realizace.

1.3.1 Meteorologická stanice

Meteorologická stanice je HW zařízení, které je určené pro měření různých fyzikálních veličin, potřebných pro analýzu a předpověď počasí v daném místě. Nejčastější měřené veličiny jsou teplota, tlak, vlhkost, směr a rychlost větru, případně množství vodních srážek, apod.



Obrázek 1.2: Systémová architektura s použitím mezivrstvy.

Existuje několik druhů meteorologických stanic. Nejdůležitější jsou profesionální meteorologické stanice, které produkují data pro státní organizace, jako je Český hydrometeorologický ústav (ČHMÚ), na základě kterých jsou nad sbíranými daty následně prováděny předpovědi počasí a další analytické úkony. Vedle toho se v poslední době těší oblibě i amatérské meteorologické stanice, které si pořizují nebo sami sestavují zejména lidé se zájmem o meteorologii.

1.3.2 Přenos dat

Existují různé druhy senzorů, ze kterých je možné získávat data. Tyto data je pak možné z meteorologických stanic přenášet mnoha různými způsoby. Zpravidla záleží pouze na tom, jaké knihovní funkce síťových protokolů jsou pro danou HW platformu dostupné. Existuje několik základních síťových protokolů, které je možné použít, a které je možné rozdělit do dvou skupin na základě čtvrté, transportní vrstvy **ISO/OSI modelu**[3], a sice protokoly založené na protokolu **UDP** a **TCP**.

1.3.2.1 UDP (UDP-based)

UDP je jednoduchý, bezstavový síťový protokol čtrvté transportní vrstvy, založený na jednoduchém předávání zpráv (datagramů) mezi odesílatelem

[3]ISO/OSI model je teoretický model, skládající se, obvykle, ze sedmi nezávislých vrstev, který se především v počítačových sítích používá jako referenční model pro popis prvků síťové komunikace.

a příjemcem. Výhoda použití protokolu UDP pro přenos dat z malých zařízení spočívá zejména v nízké náročnosti na výpočetní prostředky, paměť a šíři síťové konektivity. Na druhou stranu je nutné ošetřit správné doručení paketů, ať na straně klienta, tak na straně serveru, což s sebou přináší mírné komplikace, zejména v aplikační logice. Obecně neexistují síťové protokoly pro přenos dat založené na UDP, na rozdíl od TCP, na kterém staví například protokol HTTP.

Pro přenos dat, založeném na UDP, je tedy nutné vytvořit vlastní síťový protokol, resp. definovat struktury posílaných datagramů, podle kterých bude probíhat komunikace mezi koncovými zařízeními.

1.3.2.2 TCP (TCP-based)

TCP je stavový síťový protokol, umožňující předchozí sestavení spojení mezi odesílatelem a příjemcem po dobu přenosu dat. Garantuje zejména samotné doručení datových paketů, včetně potvrzovacích mechanismů. Při použití v malých zařízeních jsou u protkolů tohoto typu obecně vyšší nároky na výpočetní prostředky a paměť, což je důležitý aspekt, který je nutné brát v potaz při návrhu obslužného programu. Vedle toho záleží na konkrétní implementaci použité knihovny, které se mohou také výrazně lišit.

1.3.3 Protokoly pro Internet věcí

Ve světě Internetu věcí[4] je několik komunikačních konceptů, které lze rozdělit do několika skupin. Pro každou z těchto skupin pak existuje minimálně jeden komunikační síťový protokol, který je možné použít. Samotná zařízení musí umět komunikovat mezi sebou (Device to device, D2D). Data, získávaná ze zařízení musí být někde sbírána a uchovávána, typicky na serveru (Device to server, D2S). Data, která jsou uchovávaná na serverech se musí dále prezentovat uživatelům nebo být dostupná dalším aplikacím, které nad nimy provádí další úkony (Server to server, S2S) [2].

- **MQTT (Message Queuing Telemetry Transport)**: protokol, vyvinutý převážně za účelem sběru dat ze zařízení a jejich odesílání na server (D2S). Jeho primární použití je určeno do rozlehlých sítích s velkým množstvím malých zařízení, ze kterých je třeba sbírat data a tyto zařízení monitorovat a řídit, ve většině případů nasazení z cloudového prostředí.

[4]Internet věcí (Internet of Things, IOT) je pojem pro propojení jednocuhých, většinou vestavěných, zařízení s Internetem.

- **XMPP (Extensible Messaging and Presence Protocol)**: protokol, zajišťující připojení zařízení s uživateli. Jedná se o speciální případ D2S, kdy uživatelé jsou připojení k serverům, kde běží různé aplikace pro práci s daty. Protokol je známý spíše pod původním označením Jabber, který byl původně vyvinut jako protokol pro výměnu textových zpráv mezi uživateli (instant messaging, IM). XMPP využívá XML formát pro výměnu zpráv. XMPP není navržen jako rychlý protokol. Většina implementací používá k získávání dat techniku polling nebo kontroluje aktuální stavy zpráv a dat pouze na vyžádání.
- **DDS (Data Distribution Service)**: rychlá sběrnice pro integraci a zajištění komunikace mezi zařízeními (D2D). Jedná se o protokol, který zajišťuje rychlou distribuci dat mezi více zařízení. Jedná se o middleware vrstvu, která je orientovaná na data. DDS je navržen tak, aby zajistil spolehlivé doručení milionů zpráv za vteřinu několika příjemcům.
- **AMQP (Advanced Message Queuing Protocol)**: systém, založený na konceptu fronty, zajišťující komunikaci a výměnu dat především mezi servery (S2S). Koncept fronty zajišťuje schopnost systému neztrácet přenášené zprávy. Protokol byl původně vyvinut pro bankovní prostředí. Odtud pochází zejména mechanismy pro spolehlivé doručování zpráv prostřednictvím vícefázových potvrzovacích zpráv.

Všechny zmíněné protokoly jsou široce používané a od každého z nich existuje několik různých implementací. Na první pohled se může zdát, že se jedná o protokoly, které všechny zajišťují jednu a tu samou funkčnost. Je pravdou, že všechno jsou to síťové protokoly pro komunikaci v reálném čase, založené na mechanismu publish/subscribe a určené primárně pro aplikace v prostředí Internetu věcí. Všechny jsou také schopné propojovat tisíce zařízení prostřednictvím zasílání zpráv. Záleží pouze na tom, co přesně jak který protokol definuje významy slov komunikace v reálném čase, Internet věcí a zařízení. Od toho se pak odvíjí možnosti použití daného protokolu v dané aplikaci.

1.3.4 Formát dat

Data je možné přenášet v různých formátech a strukturách. Právě formát a struktura přenášených dat je hlavním kritériem pro univerzálnost použití výsledného systému pro další aplikace. Dnes nejpoužívanější formáty pro přenos dat, zejména v prostředí Internetu a webových aplikací jsou následující textove formáty:

- **XML** bylo navrženo jako univerzální textový formát pro různé aplikace. Výhodou je podpora schémat, na základě kterých je možné provádět validace a transformace XML dokumentů. Na rozdíl od JSON a YAML jsou XML dokumenty obecně náročnější na zpracování a ve většině případů je vyžadován XML parser.

- **JSON** vzniknul jako JavaScriptový objektový zápis. Samotná strukturovaná data mohou být organizována v polích nebo objektech. Hlavní výhodou oproti XML je rychlost zpracování, které není obecně tak náročné. Jednou z hlavních charakteristik JSON je skutečnost, že v tomto textovém formátu není možné použít komentáře.

- **YAML** je méně známý textový formát dat, jehož hlavní výhodou je dobrá čitelnost uchovávaných dat i pro člověka. Pro zápis dat používá odsazování, které je charakteristickým rysem skriptovacího jazyka Python.

Všechny tři výše zmíněné formáty jsou dnes hojně využívané a hodí se zejména pro výměnu strukturovaných dat mezi různými aplikacemi. Záleží pouze na konkrétní implementaci, požadavcích a možnostech daného systému, který z těchto formátů použít. Pro účely použití ve vestavěných systémech se nejlépe hodí formát JSON, protože je možné jej zapsat jako textový řetězec s danou strukturou. YAML vyžaduje odsazení a XML používá XML tagy, které zbytečně zabírají ve výsledném řetězci místo.

1.3.5 Serverové prostředí

Výsledný systém je nutné provozovat na nějakých výpočetních zařízeních. V dnešní době je k dispozici nepřeberné množství HW prostředků, které je možné za tímto účelem využít. Z pohledu jejich umístění a používání jsou k dispozici dvě hlavní řešení:

- **cloud computing** a

- **lokální (on-premise) řešení**.

Hlavní rozdíl mezi těmito dvěma řešeními je ve fyzickém umístění výpočetních prostředků a způsobu jejich financování. Obecně lze říci, že prvním z řešení, která se začala využívat v cloudovém prostředí byl provoz webových stránek. Tehdy se sice ještě nejednalo o pojem cloudového prostředí, které se začalo používat kolem roku 1999, ale celkový koncept byl velmi podobný. Byla to právě cena, která byla pro většinu subjektů hlavním faktorem

pro provoz webových stránek u poskyovatelů webhostingu. Stejným směrem dnes směřuje (a v budoucnu bude jistě směřovat) stále více uživatelů a již se nejedná pouze o webové stránky, ale o různé druhy aplikací a systémů. První společností, která začala nabízet cloudové služby byl v roce 2002 Amazon. Zpočátku se jednalo především o datová úložiště a výpočetní prostředky. V roce 2006 začal Amazon nabízet dnes již dobře známé a hojně využívané webové služby[5], jejichž portfolio se stále rozrůstá. Vedle Amazonu postupně přibývají i další poskytovatelé, nejvýznamější z nich jsou například Google nebo Microsoft [3].

1.3.5.1 Cloud computing

Technologie cloud computingu nabízí několik hlavních výhod, které je vhodné dnes zohlednit v případě nasazení nových systémů a aplikací do provozu.

- **Ekonomika**: koncový subjekt platí pouze za ty výpočetní prostředky, které skutečně využívá. Výsledné finanční náklady jsou ve u většiny poskytovatelů dopředu známé a predikovatelné.

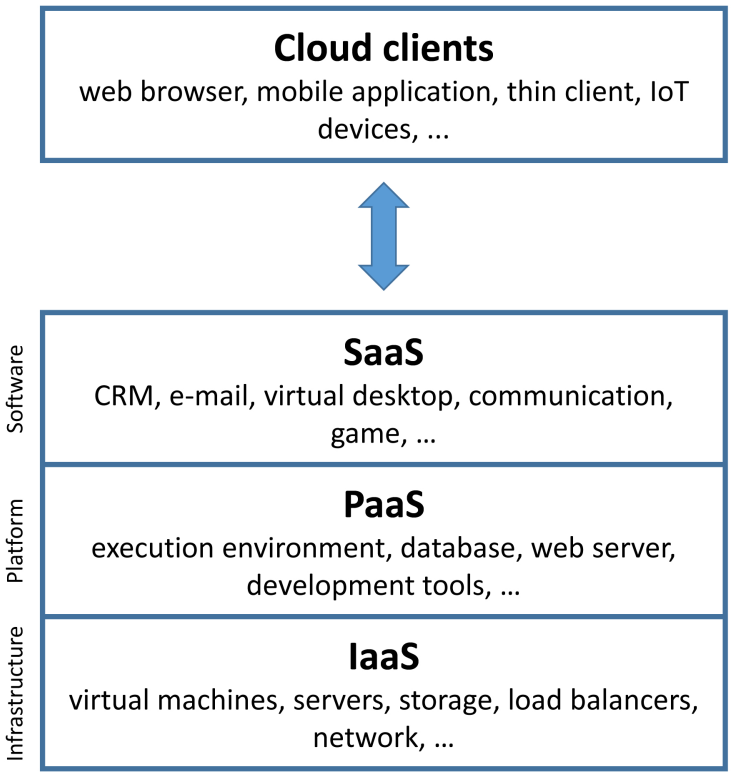
- **Správa**: využívané systémy nevyžadují zpravidla základní správu ze strany koncového uživatele, ale je zajišťena poskytovatelem dané služby. Většina poskytovatelů cloudových služeb také zajišťuje spolehlivou bezpečnostní ochranu provozovaných systémů.

- **Produktivita**: systémy a aplikace, provozované v cloudovém prostředí jsou dostupné z Internetu. Společně s vhodným nástrojem je možné spolehlivě spravovat přístupy k daným aplikacím a systémům.

Cloud computing nabízí architekturu, která je zaměřená na poskytování služeb. Existují tři základní druhy služeb, poskytovaných v cloudovém prostředí. Tyto základní služby jsou uvedeny na obrázku 1.3 a jejich stručný popis následuje. Kromě nich dnes existují prakticky jakékoliv další druhy služeb dle konceptu XaaS (Anything as a Service).

- **IaaS (Infrastructure as a Service)**: infrastruktura je nejnižší úroveň v hierarchii výpočetních prostředků. Poskytuje HW prostředky jak jsou CPU, paměť, úložiště, síťové prvky nebo load balancery. Nejznámějším poskytovatel IaaS je společnost Amazon.

[5] Amazon Web Services (AWS) je dnes jedna z nejvíce využívaných webových služeb.



Obrázek 1.3: Základní druhy služeb v cloudovém prostředí.

- **PaaS (Platform as a Service)**: PaaS je o jednu úroveň výše než IaaS. Hlavní výhodou je prostředí pro vývoj a provoz samotných aplikací. Koncoví uživatelé si tak nemusí dělat starosti s instalací, správou a údržbou samotného serveru. Typickým příkladem poskytovatelů PaaS je například Microsoft Azure, Google App Engine nebo Heroku [4].
- **SaaS (Software as a Service)**: software patří dnes k nejvíce používaným cloudovým službám. Pro většinu dodavatelů SW je provoz v cloudovém prostředí finančně výhodnější a stejně tak je výhodnější i model pro získávání poplatků od uživatelů za používání dané služby. Typickými poskytovateli služeb SaaS jsou dnes známé Google Apps nebo Microsoft Office 365.

1.3.5.2 Lokální řešení

Lokální on-premise řešení je tradiční přístup pro provozování aplikací a systémů, které jsou přístupné pro koncové uživatele v lokálním umístění. Pro-

On-premise		Cloud	
Výhody	**Nevýhody**	**Výhody**	**Nevýhody**
Data jsou uložena lokálně	Vysoké pořizovací náklady HW a SW	Nízké pořizovací náklady HW a SW	Uložení dat obecně není pod kontrolou vlastníka
Žádné měsíční poplatky při využívání HW nebo SW	Závislost na lokálním správci IT	Vždy aktuální verze SW	Bezpečnost a dostupnost
HW může být sdílen více aplikacemi a systémy	Delší odezva při nasazování systémů a aplikací	Platba pouze za skutečně využité prostředky	Vysoké požadavky na Internetové připojení

Tabulka 1.1: Porovnání vlastností cloudových a lokálních on-premise řešení.

vozovatel je sám zodpovědný za dostupnost, zabezpečení a celkovou správu těchto systémů a aplikací. Vedle samotného SW je třeba zajistit a spravovat také HW prostředky. Celkově lze říci, že lokální řešení nabízí obecně vyšší úroveň zapezpečení a především kontrolu nad daty, se kterými se pracuje. Na druhou stranu je zřejmé, že v porovnání s cloudovými službami může být provoz těchto systémů finančně náročnější. Srovnání obou přístupů je uvedeno v tabulce 1.1.

Hlavní vlastností přístupu XaaS je provoz daného prostředku (software, infrastruktura, platforma) jako služby ve formě pronájmu. Koncový uživatel pak neplatí samotný prostředek, ale platí pouze za jeho využívání ve formě pravidelných, obvykle měsíčních, splátek.

1.3.5.3 Počítačový cluster

Počítačový cluster je skupina dvou a více počítačů, které spolu úzce spolupracují a typicky se navenek tváří jako jeden počítač. Jednotlivé uzly spolu sdílejí výpočetní výkon, počítačovou síť a dále pak výpočetní prostředky, například filesystém, databázi, apod. Existují různé druhy clusterů, které zajišťují požadovanou funkcionalitu celého systému. Základní funkcionality počítačových clusterů jsou následující.

- **Výpočetní výkon (high-performance)**: výpočetní cluster slouží k zajištění zvýšení výpočetního výkonu. Obvykle se jedná o skupinu

počítačů, které jsou mezi sebou propojeny rychlou počítačovou sítí a sdílí své výpočetní prostředky.

- **Vysoká dostupnost (high-availability)**: požadavkem je zajištění vysoké dostupnosti obvykle nějaké služby. V případě poruchy jednoho z uzlů clusteru je služba dostupná na jiném.
- **Rozložení zátěže (load-balancing)**: v případě velké zátěže, typicky velké množství klientů, využívajících danou službu, je daná služba poskytována paralelně na více uzlech clusteru. Konzistence obsahu a zejména dat služby je zajištěna replikací mezi požadované uzly.
- **Úložiště (storage)**: clusterové úložiště je používáno především ke zvýšení úložné kapacity systému, ale také z důvodu získání vyššího toku dat a spolehlivosti pro tok dat mezi poskytovanou službou a klienty, využívající tuto službu.

Pro potřeby sběru dat jsou hlavní požadavky právě vysoká dostupnost systému, rozložení zátěže a dostatečné datové úložiště.

KAPITOLA 2

Návrh modelového řešení

V předchozí kapitole byly popsány obecné přístupy a možnosti řešení zadané problematiky. V následující kapitole bude prezentován návrh konkretního modelového řešení. Systém pro sběr dat se bude sestávát ze tří hlavních částí.

- **Meteorologické stanice**: hlavním úkolem bude produkce dat z přímo připojených senzorů.
- **Mezivrstva pro výměnu zpráv**: místo, kde budou předávány zprávy mezi stanicemi, serverovým prostředím a klientskými aplikacemi.
- **Serverové prostředí**: zpracování, archivace a práce se sbíranými daty.

Další skupinu tvoří uživatelské aplikace, které budou zajišťovat samotnou práci s daty, jejich prezentaci koncovým uživatelům a budou moci nabídnout i konfiguraci zařízení (nastavení provozních parametrů).

2.1 Meteorologické stanice

Nejdůležitější součásti při návrhu meteorologických stanic je správný výběr HW platformy, použité senzory pro měření pozorovaných údajů a správný návrh a struktura obslužného programu pro danou HW platformu.

2.1.1 Výběr vhodné HW platformy

Dnes je na trhu k dispozici mnoho výrobců, kteří poskytují HW součástky pro vývoj vlastních zařízení. Tato skutečnost je do jisté míry zapříčiněna

vyšší dostupností a nižší pořizovací cenou vlastních HW komponent (mikroprocesory) a elektronických součástek (rezistory, diody, kondenzátory). S potřebnou znalostí není problém vytvořit vlastní návrh desky plošných spojů, tu osadit potřebnými HW komponenty, připojit požadované snímače a senzory a výsledné zařízení naprogramovat.

Pro vytvoření meteorologické stanice je však možné použít již existující, prototypovací HW platformy s využitím běžně dostupných senzorů pro měření libovolných údajů. Na trhu je několik výrobců, kteří poskytují již sestavené integrované obvody, které je třeba pouze naprogramovat. Tři nejrozšířenější prototypovací platformy jsou následující:

- **Arduino**: open source prototypová platforma, založená na mikroprocesorech ATmega společnosti Atmel. Kromě samotných programovatelných desek je k dispozici i SW programovací prostředí. Platforma původně vznikla jako podpora pro výuku a měla studentům poskytnout podporu pro snadné a rychlé vytváření prototypů. Dnes se, zejména díky nízké ceně a velkému množství příslušenství, řadí mezi nejpoužívanější prototypovací HW platformy na celém světě. Také je ukázkovým příkladem toho, jak mohou být HW a SW technologie, původně vytvořené pro potřeby armády, obchodu a vědeckého výzkumu, použity pro nejrůznější projekty [5].

- **Raspberry Pi**: narozdíl od Arduina je považováno spíše za plnohodnotný, miniaturní počítač. Samotná deska disponuje HDMI konektorem pro přenos video/audio signálu, síťovým konektorem RJ-45 a USB porty. Díky svým rozměrům a programovatelným vstupně/výstupním GPIO pinům může být též použito pro prototypování různých HW projektů. Samotná deska byla také vyvinuta pro podporu studentů při výuce informatiky ve školách.

- **Intel Edison**: z hlediska poskytovaných funkcí a možností řešení je Intel Edison něco mezi Arduinem a Raspberry Pi. Jedná se o jednočipový počítač, vyvíjený zejména pro aplikace Internetu věcí a nositelné elektroniky. Samotný čip je asi o velikosti běžné SD karty a jsou k němu k dispozici různé typy desek plošných spojů se vstupy a výstupy a paticí, do které je možné jej umístit.

Tabulka 2.1 uvádí jejich základní parametry a srovnání. Raspberry Pi a Intel Edison nabízí možnost běhu samostatného operačního programu. Ve výchozím nastavení se jedná o speciálně upravenou verzi některé z Linuxových distribucí. Zajímavou skutečností je také možnost instalace operačního systému Windows 10 ve verzi Embedded pro Internet věcí.

	Arduino UNO	**Raspberry Pi**	**Intel Edison**
Procesor	ATmega328	ARM Cortex-A7	ARM Cortex-A8
Frekvence	16Mhz	900MHz	700MHz
Paměť RAM	2kB	1GB	256MB
Paměť FLASH	32kB	SD karta	4GB
Počet výstupů	14	8	66
Cena*	$20	$35	$130

*Ceny jsou pouze orientační a byly převzaty z prodejní sítě internetového prodejce http://www.amazon.com.

Tabulka 2.1: Srovnání nejběžnějších prototypovacích HW platforem.

Všechny zmíněné HW platformy jsou určeny pro seznámení se s těmito technologiemi jak z HW hlediska, tak i z hlediska programování cílových zařízení. Hlavním přínosem prototypovacích platforem je rychlost zhotovení výsledného, plně funkčního prototypu [6].

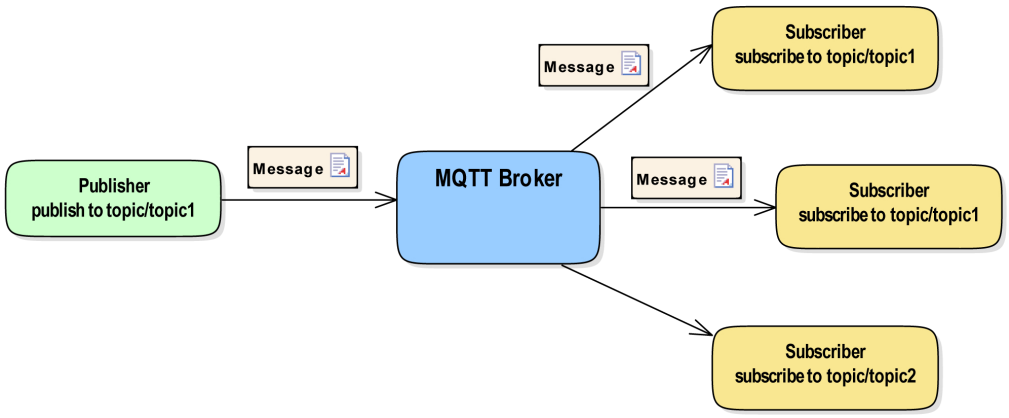
2.2 Mezivrstva pro výměnu zpráv

Při návrhu řešení pro sběr dat se nabízí použití mezivrstvy jako prostředku pro výměnu zpráv, odtud přímo pochází pojem Mezivrstva pro výměnu zpráv[6]. Jedná se o prostředek, zajišťující výměnu zpráv v distribuovaném prostředí mezi různorodými platformami a zařízeními. MOM je jedním ze základních kamenů, na kterých jsou dnes postaveny podnikové systémy. Jednotliví účastníci komunikace posílají a odebírají prostřednictvím MOM zprávy od dalších účastníků. Platforma MOM nabízí vytvoření flexibilních systémů, které umožňují její univerzální a nezávislé použití [7].

2.2.1 MQTT

MQTT je síťový protokol založený na mechanismu publish-subscribe, pracující na čtvrté TCP transportní vrstvě. Jedná se o spolehlivý, snadno implementovatelný a na prostředky nenáročný síťový protokol. Primární účel MQTT je použití v jednoduchých zařízeních, s malým výpočetním výkonem a malou pamětí, u kterých je požadován častý přenos malých dat v rozhlehlém a nestabilním prostředí. MQTT pochází z laboratoří firmy IBM,

[6]Message Oriented Middleware (MOM) je mezivrstva, kde komunikace mezi účastníky je založená na výměně zpráv. Účastníci komunikace mohou být nejrůznější aplikace a systémy na různých platformách.



Obrázek 2.1: Mechanismus výměny zpráv, založený na konceptu předávání publish/subscribe.

kde vznikl v roce 1999 za účelem vytvoření snadno implementovatelného průmyslového protokolu pro bezdrátové senzory a řídící sítě [8]. Do posledních let nebyl protokol široce využíván. Tato skutečnost se však změnila s příchodem Internetu věcí a přinesla výrobcům a inovátorům v této oblasti spolehlivou platformu přenosu dat pro jejich projekty. Obrázek 2.1 znázorňuje koncept publish/subscribe mechanismu pro přenos zprávy mezi odesílatelem a více příjemci. Odesílatel odešle zprávu do příslušného kontextu na MQTT broker, který je dále zodpovědný za jejich doručení příjemcům. MQTT broker pak zkontroluje každou takto přijatou zprávu a na základě připojených klientů, kteří jsou přihlášeni k odběru zpráv z daného kontextu, jim tuto zprávu doručí.

2.2.2 MQTT broker

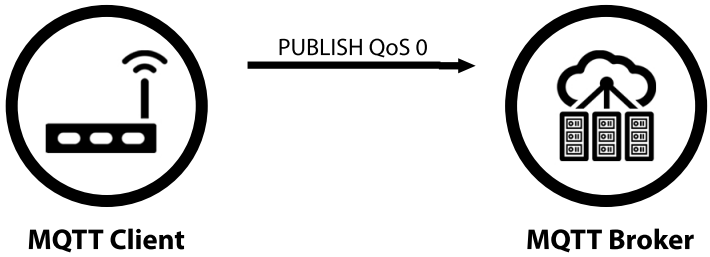
MQTT broker je základní prvek mezivrstvy. Existuje několik řešení a některé z nich implementují více funkčností než jen základní vlastnosti MQTT protokolu dle specifikace. Ve většině případů se jedná o proprietární funkčnosti pro konkrétní aplikace. K dispozici je celá škála řešení, od komerčních (HiveMQ, CloudMQTT) po open source řešení (Mosquitto, RabbitMQ, VerneMQ).

2.2.3 Výhody využití MQTT

Použití modelu MQTT broker má několik výhod oproti jiným způsobům přenosu dat.

- **Nezávislost umístění (space decoupling)**: aplikace, využívající publish/subscribe mechanismus pro přenos zpráv, nepotřebují znát přesné umístění komunikujících uzlů. Jediná adresa, která je potřebná pro výměnu zpráv je právě adresa MQTT brokeru. MQTT broker je pak sám zodpovědný o samotný přenos zpráv cílovým účastníkům komunikace. Za tímto účelem se používá koncept **témat (topics)**. Téma je textový řetězec, na základě kterého MQTT broker filtruje a doručuje zprávy pro každého z připojených klientů.
- **Nezávislost času (time decoupling)**: komunikující uzly mohou být při výměně zpráv časově nezávislé. Za tímto účelem implementuje MQTT broker možnost využití persistentního úložiště zpráv. Typicky se jedná o jednoduchou databázi, do které jsou příchozí zprávy ukládány pro následné vyzvednutí příjemcem. Odesílatel pak odešle do příslušného kontextu na MQTT broker zprávu a následně může být (z různých důvodů) odpojen. Daná zpráva je pak právě díky konceptu MQTT brokeru a použití úložiště spolehlivě doručena příjemci.
- **Spolehlivost (reliability)**: spolehlivost je hlavní výhodou při použití MQTT brokeru. Zejména v heterogenních a obecně nespolehlivých sítích, kdy může docházet k výpadkům a ztrátám síťových paketů. MQTT je nenáčočný síťový protokol, jehož použití je výhodné pro časté přenosy zpráv v rozlehlém prostředí.
- **Bezpečnost (security)**: bezpečnost přenosu dat může být v MQTT protokolu řešena na různých vrstvách. Hlavním cílem MQTT protokolu je nabídnout jednoduchý a nenáročný síťový protokol. Obecně platí, že je relativně obtížné zapezpečit samotnou komunikaci z vestavěných zařízení, ale místo toho je lepší používat již ověřené bezpečnostní standardy. Za tímto účelem podporuje protokol MQTT šifrování dat na čtvrté, transportní vrstvě pomocí SSL/TLS protokolu.

Koncept MQTT brokeru je obecně výhodný zejména v jednom konkrétním případě, a sice, když je třeba předat zprávu příjemci, který se nachází v lokální síti, od odesílatele, který je z pohledu příjemce v Internetu. Jelikož odesílatel zprávy v Internetu nezná přesnou adresu příjemce v lokální síti, nemůže tedy zprávu odeslat přímo (pokud není adresa příjemce veřejně přístupná přes směrovač lokální sítě například pomocí techniky NAT apod.).



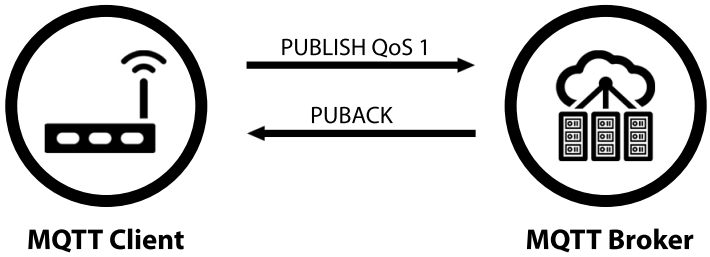
Obrázek 2.2: Úroveň QoS 0 MQTT protokolu. Zprávy nejsou potvrzovány, ani ukládány na MQTT brokeru.

Jedná se například o nastavení provozních a konfiguračních parametrů vestavěných zařízení (perioda odesílání dat, konfigurace sítě, apod.). Právě použití publish/subscribe mechanismu je v tomto případě nejspolehlivějším řešením především z hlediska bezpečnosti.

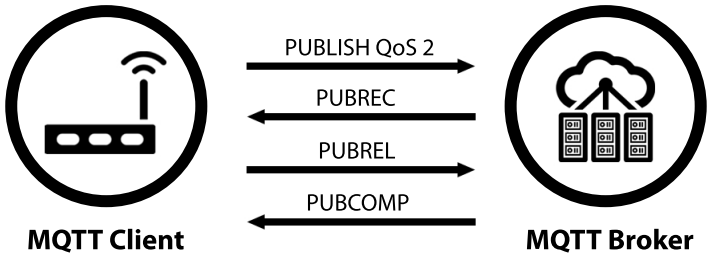
2.2.4 MQTT QoS

MQTT Podporuje také QoS. Úroveň QoS rozhoduje o tom, jak bude každá z posílaných MQTT zpráv doručena a je jednou z hlavních vlastností, která zajišťuje spolehlivé doručování zpráv v nespolehlivých sítích, jako je síť Internet. Pro správné doručení zpráv musí být nastavená příslušná úroveň QoS pro obě komunikující strany [9]. Podpora úrovní QoS se může lišit v daných implementacích MQTT protokolu. Vyšší úroveň QoS značí vyšší spolehlivost přenosu dat.

- **QoS 0 (nejvýše jednou)**: Zpráva je doručena nejvýše jednou nebo není doručena vůbec. Potvrzovací pakety, ani jiné doručovací mechanismy vyšších vrstev, nejsou při komunikaci posílány a samotné zprávy nejsou na MQTT brokeru ukládány, pokud nejsou ihned odebrány (minimálně) jedním z odebíratelů. QoS 0 je očividně nejrychlejší, ale zároveň nejméně spolehlivý typ MQTT komunikace [10]. Spolehlivost komunikace je zajištěna pouze na transportní vrstvě protokolem TCP. MQTT QoS 0 je vhodné k použití pro zařízení ve stabilním síťovém prostředí s využitím kabelového připojení (například Ethernet), kde není kladen důraz na fronty zpráv (Message Queueing) a také v systémech, ve kterých je možné ztratit čas od času zprávu (typicky při periodickém, pravidelném odesílání dat).

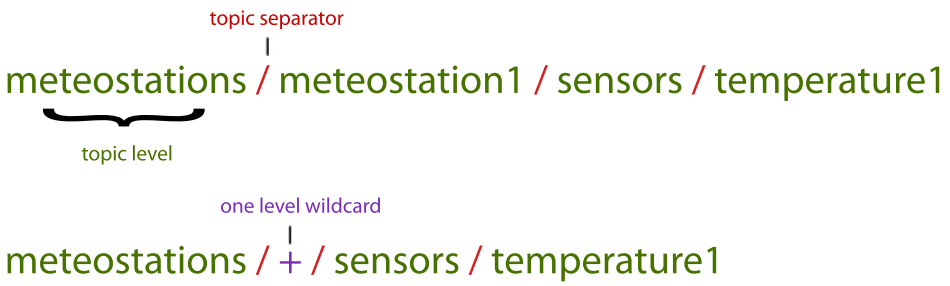


Obrázek 2.3: Úroveň QoS 1 protokolu MQTT. Zprávy jsou potvrzovány pouze příznakem PUBACK. Nedoručení potvrzení znamená opětovné odeslání duplikátu původní zprávy.



Obrázek 2.4: Úroveň QoS 2 MQTT protokolu. Pro spolehlivé doručení zpráv, na úkor rychlosti přenosu, je použita sada potvrzovacích příkazů mezi MQTT klientem a brokerem.

- **QoS 1 (alespoň jednou)**: Doručení zprávy je zajištěno a samotná zpráva může být doručena vícekrát (alespoň jednou). Pokud odesílatel zprávy nedostane potvrzení o doručení zprávy příjemci, duplikát zprávy je odesílán tak dlouho, dokud odesílatel nepřijme potvrzení o doručení. Důsledkem tohoto je možnost opakovaného přijetí a zpracování stejné zprávy příjemcem. Tato skutečnost musí být implementována v logice aplikace. Příklad komunikace je uveden na obrázku 2.3.
- **QoS 2 (právě jednou)**: Nejspolehlivější a zároveň nejpomalejší typ QoS. Při komunikaci mezi klientem a brokerem je garantováno právě jedno doručení přenášené zprávy. Tato skutečnost je zajištěna zasíláním potvrzovacích paketů, jak je zřejmé z obrázku 2.4.



Obrázek 2.5: Hierarchická struktura MQTT kontextu. Implementovaný koncept je téměř shodný se strukturou unixových filesystémů.

Využití MQTT protokolu má proto své výhody. Zejména pro aplikace, kde je třeba zajistit spolehlivé doručování zpráv v poměrně nestabilním prostředí. Typickým příkladem je dnes značně se rozšiřující pojem Internetu věcí. Při použití protokolu MQTT je možné klást při vývoji hlavní důraz na logiku samotné aplikace a společně se správnou volbou úrovně QoS je zajištěna spolehlivá komunikace a předávání zpráv. Hlavní výhodou je možnost použití obousměrné komunikace.

2.2.5 MQTT kontext

Protokol MQTT je založen na výměně zpráv prostřednictvím příslušného **kontextu** na straně MQTT brokeru, které mají pevně danou hierarchickou strukturu [11]. Jedná se o textový řetězec ve formátu UTF-8, kde jsou jednotlivé hierarchické úrovně odděleny znakem lomítko, jak je uvedeno na obrázku 2.5.

V porovnání s konceptem **fronty zpráv**[7], je použití kontextu mnohem méně náročnější na výpočetní prostředky a obecně i jednodušší. Klient může předat zprávu do libovolného kontextu libovolné struktury, odkud si jej příjemce zprávy následně vyzvedne. MQTT broker tedy akceptuje zprávy v libovolném kontextu, a to bez nutnosti jeho předchozího vytvoření či alokování.

[7]Fronta zpráv (Message Queueing, MQ) je tecnhologie pro předávání zpráv, založená na modelu fronty. Zprávy jsou producenty ukládány do fronty, odkud jsou danými konzumenty odebídány a zpracovávány.

2.2.6 Zabezpečení přenosu dat

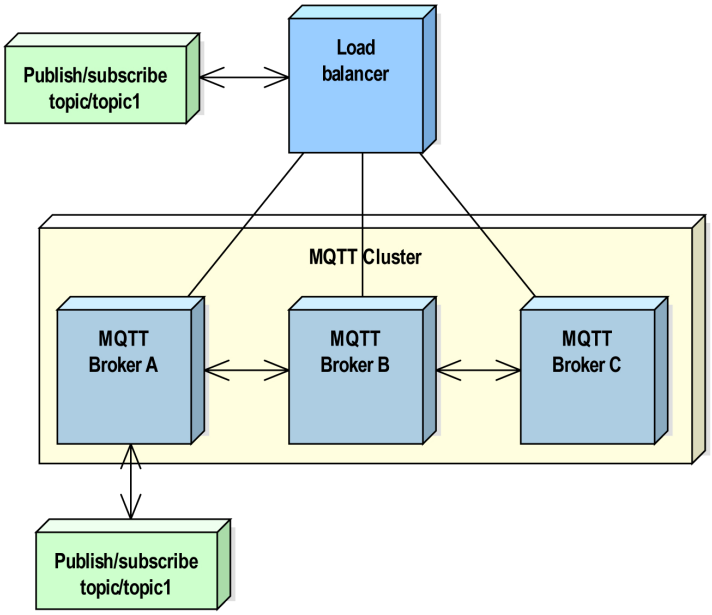
MQTT protokol obecně podporuje zabezpečení ve formě šifrování přenosu mezi klienty a MQTT brokerem prostřednictvím SSL/TLS protokolu. Nevýhodou tohoto řešení pro použití v navrženém systému pro sběr dat je skutečnost, kdy zařízení, založené na platformě Arduino a jim podobné, nemají dostatečné výpočetní prostředky pro použití této techniky. Tento scénář nabízí dvě možné řešení. První možností je použití výkonnější HW platformy, například Raspberry Pi, které má dostatečný výkon pro provádění šifrování datových paketů protokolem SSL/TLS. Na druhou stranu je ale jeho pořizovací cena výrazně vyšší. Druhou možností je použití MQTT brokeru v módu bridge v lokální síti, kde je meteorologická stanice provozována. MQTT bridge by měl disponovat dostatečným výpočetním výkonem, aby byl schopen zajistit šifrovanou komunikaci s centrálním MQTT brokerem. Tento MQTT broker v módu bridge pak může běžet na libovolném stroji, kterým může být například pouze virtuální server s minimální instalací OS Linux. Tento systém pak bude nakonfigurován jako gateway (MQTT broker v módu bridge) pro komunikaci mezi meteorologickými stanicemi a centrálním MQTT brokerem.

2.2.7 Použití MQTT v clusterovém prostředí

Existují implementace MQTT brokerů, které umožňují provozovat samotný MQTT broker v clusterovém prostředí. Jejich hlavním účelem je zajistit vysokou dostupnost a rozložení zátěže pro připojené klienty a klientské aplikace. Obecný přístup pro použití MQTT brokeru v clusterovém prostředí je zvýšení počtu zároveň připojených klientů. Tato skutečnost je zejména ovlivněna prostředky OS daného stroje, na kterém MQTT broker běží. Jedná se především o následující prostředky:

- maximální počet otevřených portů (number of open ports),
- maximální počet popisovačů socketů (socket descriptors) a
- maximální počet otevřených souborů (number of open files).

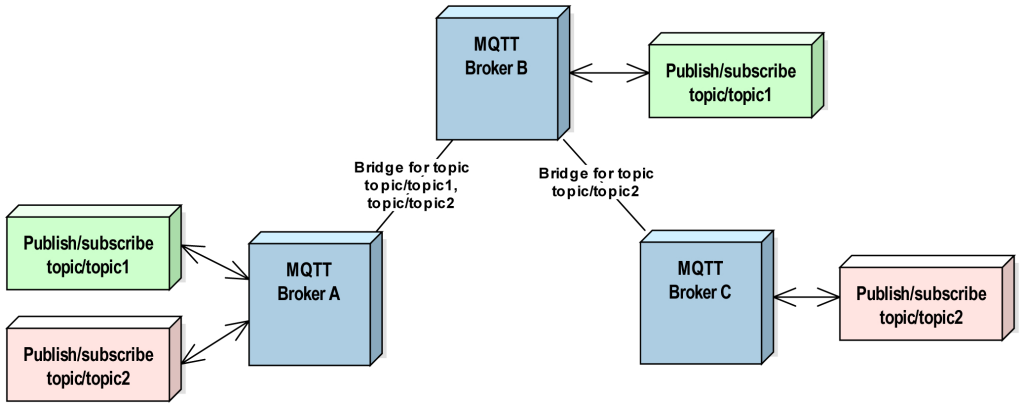
Vě většině případů jsou MQTT brokery instalovány na stroj s OS unixového typu. V prostředí těchto systémů platí skutečnost, kdy všechno je soubor. V zásadě se pak výčet výše zmíněných prostředků OS redukuje na maximální počet vytvořených souborů, pod které pak spadají všechny zbývající prostředky. Pro samotné použití MQTT brokeru v clusterovém prostředí jsou pak možné tři základní scénáře, případně jejich hybridní kombinace.



Obrázek 2.6: Příklad architektury MQTT brokeru v clusterovém módu.

Clusterový mód V clusterovém módu mohou fungovat pouze taková řešení, která mají tuto funkci přímo implementovanou. V současnosti existuje několik takových řešení (VerneMQ, HiveMQ, JoramMQ, apod.). Většina implementací umožňuje přidání a odebírání uzlů za běhu clusteru, čímž je zajištěna vysoká dostupnost a horizontální škálovatelnost. MQTT broker se pak chová jako jeden, nezávisle na množství uzlů, které celý cluster tvoří. Narozdíl od izolovaného módu jsou potom mezi všemi jednotlivými uzly clusteru sdíleny potřebné informace pro zajištění celkové funkčnosti. Jedná se zejména o informace o připojených MQTT klientech. Klient je ve výsledku připojen pouze k jednomu konkrétnímu uzlu, na kterém je otevřený příslušný socket. V případě, že je třeba tomuto klientovi předat nějakou zprávu, clusterový MQTT broker si pak tyto informace předává pomocí pravidelného zasílání zpráv mezi všemi uzly a tím je zajištěno správné předání dané zprávy správnému klinetovi. Architektura tohoto módu je zřejmá z obrázku 2.6.

Bridge mód Jedná se o mód, umožňující vytvoření mostů (bridges) mezi jednotlivými, izolovanými MQTT brokery na základě předávání zpráv v daném kontextu. Prakticky se jedná o sdílení všech zpráv, které spadají do příslušného kontextu, mezi dvěma a více MQTT brokery. Příklad tohoto módu je uveden na obrázku 2.7. Použití tohoto módu je výhodné zejména

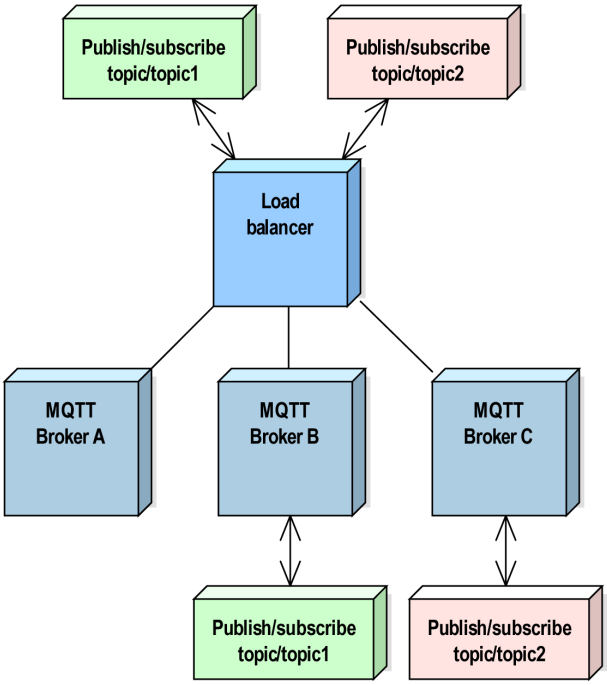


Obrázek 2.7: Příklad architektury MQTT brokeru v módu bridge.

v případě, kdy MQTT klient nemá dostatečný výpočetní výkon pro zajištění šifrování přenosu dat prostřednictvím SSL/TLS protokolu. Řešením je použití dostatečně výkonného zařízení (samostatný HW nebo virtuální instalace OS Linux), které bude hrát roli prostředníka mezi MQTT klientem a centrálním MQTT brokerem. Toto zařízení bude spojeno s centrálním MQTT brokerem právě prostřednictvím bridge módu a bude umožňovat šifrovaný přenos dat ve veřejné síti Internet.

Izolovaný mód V izolovaném módu figurují pouze nezávislé MQTT brokery, které spolu nijak nekomunikují. Příkladem pro nasazení takového řešení může být pouze potřeba zajištění většího množství připojených klientů a správné předávání a zpracování zpráv je pak zajištěno pomocí dalších mechanismů. Příklad této architektury je uveden na obrázku 2.8. Příkladem pro nasazení MQTT brokerů v izolovaném módu je například požadavek sběru dat a tím i jednosměrná komunikace od zařízení k MQTT brokeru. Klienti jsou pak připojeni k jednotlivým MQTT brokerům clusteru, na kterých běží nezávislé instance obslužného skriptu, který zpracovává příchozí požadavky.

Ve všech výše uvedených případech je nezbytné zahrnout do architektury také load balancer, který je pak zodpovědný za rozložení zátěže mezi jednotlivé MQTT brokery. Zejména pro případ izolovaného použití MQTT brokerů, kdy pak není jiný způsob, jak jednotlivé zprávy mezi jednotlivými uzly předávat.



Obrázek 2.8: Příklad architektury MQTT brokeru v izolovaném módu.

2.3 Serverové prostředí

Serverové prostředí je místem, kde budou probíhat všechny operace s daty. Od sběru, ukládání až po prezentaci dat uživateli. Jedná se o síťové prostředí, které se sestává z několika částí. Popis a možnosti řešení jednotlivých částí následuje v nalším textu.

2.3.1 Load balancer

Load balancing je technika pro rozložení zátěže mezi dva a více strojů za účelem zvýšení propustnosti a celkové optimalizaci využití prostředků.

K dispozici je obecně několik metod pro vyvažování zátěže mezi množinu uzlů v clusteru. Jednotlivé metody se liší ve způsobu směrování příchozích požadavků na cílové uzly.

- **Round Robin**: základní metoda pro vyvažování zátěže. Load balancer směruje příchozí požadavky na cílové uzly ve smyčce. Použití RR je vhodné zejména v prostředí, kde mají cílové uzly podobnou HW konfiguraci (procesor, paměť).

- **Least connections**: load balancer si udržuje seznam vytíženosti cílových uzlů, zejména aktuální hodnotu současně připojených klientů. Na základě této metriky pak směruje příchozí požadavky na ty uzly, které mají tuto hodnotu nejnižší. Tato metoda je vhodná opět pro použití v prostředí, kde jsou cílové uzly podobné konfigurace.

- **Weighted least connections**: metoda WLC je velmi podobná metodě předchozí s tím rozdílem, že load balancer má k dispozici i váhu (weight), preferenci, jednotlivých strojů z hlediska výkonu. Pokud pak mají dva cílové uzly stejný počet aktuálně připojených klientů, load balancer bere v úvahu váhu daného uzlu a příchozí požadavek směřuje na ten, který má tuto hodnotu vyšší. Tuto metodu je vhodné použít především v prostředí, ve kterém mají cílové uzly různé HW konfigurace.

Load balancer může být umístěn před různé druhy cílových uzlů (databázové a výpočetní uzly, MQTT brokery). Pro systém sběru dat se nabízí použití zejména před databázovým clusterem a MQTT brokery.

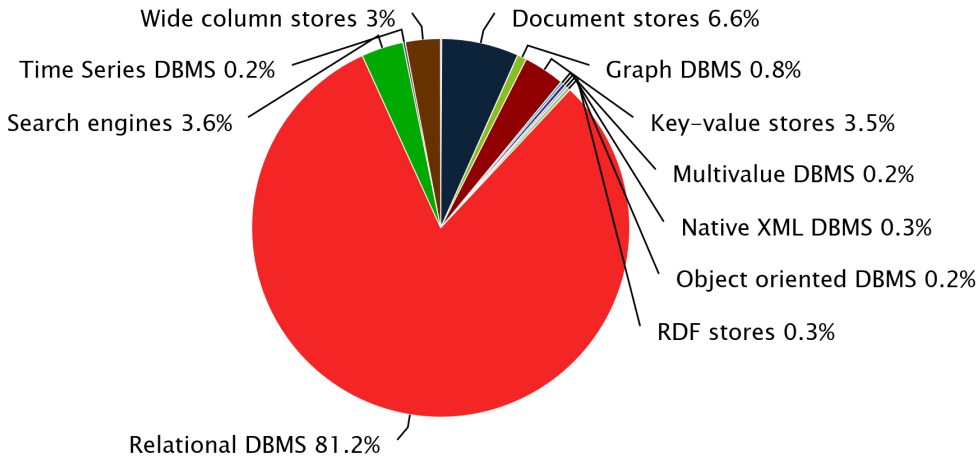
2.3.2 Databázová vrstva

Pro požadavky úkládání dat z meteorologických stanic je nutné zvolit vhodný databázový nástroj. Meteorologické stanice jsou navrženy tak, aby v určitých periodách posílaly aktuální data pro archivaci. Výsledné množství uchovávaných dat je odhadem možné stanovit předem a tím zajistit spolehlivé uložení. Hlavním kritériem při návrhu datového úložiště pro taková data je také možnost budoucího škálování úložiště. S tímto souvisí zejména výběr konceptu samotné databáze, zda zvolit tradiční **RDBMS**[8] model nebo dnes stále více se rozšiřující koncept **NoSQL**[9]. RDBMS databáze jsou stále používány celkem asi v 80% případů, jak je uvedeno na obrázku2.9, ale jsou to právě NoSQL databáze, které dnes získávají na popularitě a nabízí značné množství výhod. Většina RDBMS databází je, zejména z historického důvodu, primárně určena pro monolitické aplikace a poskytuje omezené možnosti škálování a také omezené použití v clusterovém prostředí.

Existují dva přístupy pro škálování systémů. První, ten jednodušší, je **vertikální škálování**. Prakticky se jedná o rozšíření výpočetních kapacit

[8]Relational Database Management System (RDBMS) je systém pro řízení báze dat, založený na relačním modelu.

[9]Not Only SQL (NoSQL) je databázový koncept, ve kterém jsou data zpracovávána a ukládána jinými prostředky, než tradičními tabulkovými schématy s relacemi mezi jednotlivými tabulkami, jako v relačním modelu.

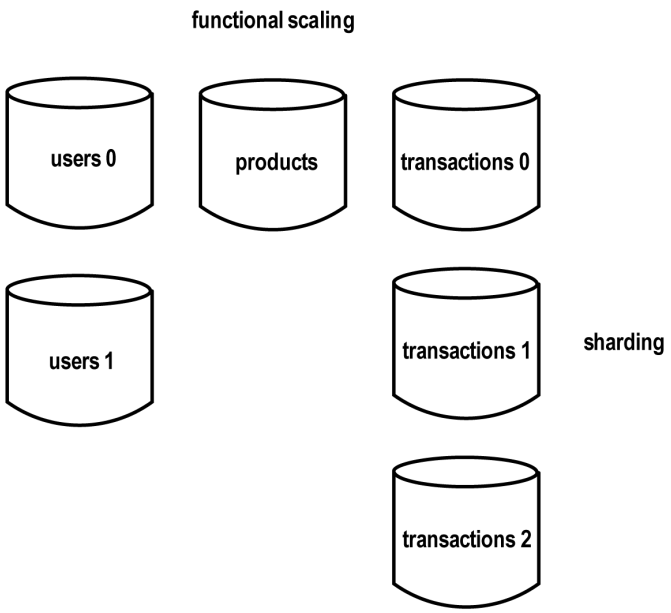


Obrázek 2.9: Diagram, znázorňující procentuální použití jednotlivých typů databází. Založeno na celkovém hodnocení dané kategorie (převzato z http://www.db-engines.com).

daného stroje. V případě ukládání dat se může jednat například o navýšení kapacity datového úložiště. Vertikální škálování funguje obecně dobře, ale zároveň má i svá omezení. Hlavním nedostatkem je skutečnost, že takto nelze škálovat do nekonečna. Každý fyzický stroj má své HW limity. U databázových systému je to především kapacita datového úložiště. Na rozdíl od toho **horizontální škálování** je mnohem komplexnější a nabízí značnou flexibilitu. Horizontální škálování databázových systémů může obsahovat dva vektory:

- **Splitting**: první složka značí rozdělení dat z hlediska funkčnosti mezi více databázových strojů.
- **Sharding**: přidává horizontálnímu škálování druhou složku a sice v podobě rozložení zátěže exponovaných částí databáze přes více databázových strojů. Prakticky se jedná o operace nad samotnými řádky, kdy jsou jednotlivé řádky jedné fyzické databáze rozloženy mezi více strojů a tím je dosaženo vyššího výkonu při operacích nad těmito daty.

Obě složky horizontálního škálování moho být aplikovány najednou. Příklad je uveden na obrázku 2.10, kde tabulky users, products a transactions mohou být uloženy na různých databázových strojích. Dle požadovaného výkonu se může jednat o logické nebo o fyzické stroje. Jednotlivé databázové tabulky mohou být škálovány nezávisle na sobě.



Obrázek 2.10: Příklad horizontálního škálováni databázových tabulek.

2.3.2.1 Typy NoSQL databází

V posledních letech jsou NoSQL databáze stále používanější zejména díky jejich výkonu, škálovatelnosti, flexibilitě a analytickým možnostem. Zejména díky vysokým nárokům na množství ukládaných dat (big data) jsou to právě NoSQL databáze, které umožňují nové metody práce s těmito daty. Obecné důvody pro používání NoSQL databází souvisí zejména s přístupem tvorby nových systémů. Dříve byly vytvářeny zejména monolitické aplikace, které běžely na jednom stroji, zajišťovaly jednu danouo funkčnost a potřebovaly typicky pouze jednu databázi. Dnešní požadavky na systémy jsou mnohem komplexnější a vyžadují sofistikovanější přístupy. Jedná se především o distribuované systémy a aplikace, které běží na několika tisicích strojích, zajišťují několik funkcionalit a uchovávají obrovská množství dat. Zejména požadavky na ukládání a práci s velkými objemy dat jsou hlavním důvodem pro vývoj a častější nasazování NoSQL databází. Graf v příloze na obrázku C.1 zobrazuje obecný trend popularity jednotlivých databázových nástrojů v posledních 3 letech.

K dispozici je několik druhů NoSQL databázových systémů. Každý z nich je vhodný pro jinou aplikaci, zejména díky možnostem formátu a struktur pro ukládání dat. Od každého databázového systému existuje několik různých implementací.

- **Dokumentově orientované (document store)** databáze jsou největším kontrastem k RDBMS databázím. Obecně jsou podtřídou datových úložišť typu klíč-hodnota, hlavním rozdílem je způsob uložení a zpracování dat. Samotná data jsou fyzicky uložena v dokumentech, které mohou mít různou strukturu. Nejčastěji se jedná o XML, YAML, JSON a případně BSON, cože je binární verze JSON. Mezi hlavní představitele patří MongoDB, Couchbase a CouchDB.
- **Grafové databáze (graph database)** fungují na principu grafových struktur, které používají pro ukládání a práci s daty (uzly, hrany a jejich vlastnosti). Hlavní přínos je v objektově orientovaných aplikacích, kdy dovolují přesnější mapování na zpracováváná data a poskytují operace s daty, které by v RDBMS byly značně obtížné (nejkratší cesty v grafu, průměry v grafu, apod.). Mezi hlavní a zároveň nejpoužívanější řešení patří Neo4j, Titan a OrientDB.
- **Úložiště typu klíč-hodnota (key-value store)** využívá jako datový model asociativní pole (mapa, slovník). Data jsou reprezentována jako páry **klíč** a **hodnota**. Některá řešení na základě toho umožňují lexikografické uspořádání ukládaných klíčů, což následně zajišťuje efektivní práci s různými rozsahy dat. Mezi typické představitele patří Redis, Memcached a Amazon DynamoDB.
- **Úložiště s širokými sloupci (wide columnar store)** je založeno na typu klíč-hodnota. Pro uložení dat využívá tabulek, řádků a sloupců, ale na rozdíl od relačních databází nemá pevné schéma tabulek, používá speciální typy sloupců (column families), čímž je možné ukládat různé formáty a typy hodnot na jednotlivých řádcích. Nejpoužívanější řešení jsou Apache Cassandra, Apache HBase a Apache Accumulo.

NoSQL databáze mají obecně několik pozitivních vlastností, kterými se odlišují od RDBMS systémů.

2.3.2.2 Přednosti NoSQL databází

- **Vysoká škálovatelnost (highly scalable)** zajišťuje zejména možnost **horizontálního** škálování (přidání nových databázových serverů a jejich začlenění do již existujícícho clusteru) distribuovaného databázového systému. Na rozdíl od většiny RDBMS systémů, které jsou škálovatelné pouze **vertikálně** (navýšení výpočetních prostředků daného databázového serveru přidáním procesorů, operační paměti nebo kapacity diskového úložiště).

- **Flexibilní datový model (flexible data model)** umožňuje ukládání nejen strukturovaných, ale i nestrukturovaných dat. Na rozdíl od RDBMS, kde je nutné předem definovat fixní datové schéma, umožňují NoSQL databáze využití rodin sloupců (column families), ve kterých je možné ukládat různá data v různých strukturách. Příklad rodiny sloupců je uveden na obrázku 2.12.

- **Paralelní výpočet (parallel computing)** je obecně jednou z výhod distribuovaného databázového prostředí. Již z podstaty věci se nabízí možnost paralelního zpracování velkého množství uložených dat za účelem zvýšení celkového výkonu a doby zpracování. Typickou metodou je model **MapReduce** [12].

Z hlediska požadavků na škálovatelnost databázových systémů je kladen důraz především na dvě hlavní složky, a to datové úložiště a výpočetní výkon. Při škálování je od systémových administrátorů požadováno škálovat daný systém v rozumném časovém horizontu. Ve většině případů je tohoto docíleno přidáním dalších výpočetních prostředků do již existujícího prostředí a tím navýšit výpočetní kapacitu systému. V databázovém prostředí je právě tento přístup však penalizován a přináší s sebou drobné nedostatky. Tyto nedostatky popisuje **CAP teorém**. CAP teorém říká, že v distrbuovaném systému[10] může být zároveň docíleno pouze dvou ze tří níže uvedených vlastností [13].

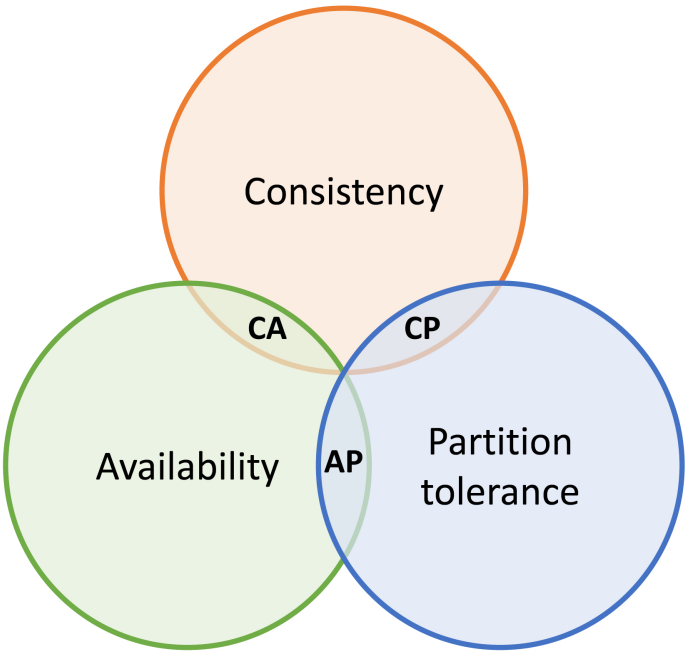
- **C - Consistency (konzistence)**: na daný požadavek čtení je systém schopen vrátit nejaktuálnější stav dat pro daného klienta.

- **A - Availability (dostupnost)**: správně pracující uzel je schopen poslat klientu danou odpověď v rozumném čase.

- **P - Partition tolerance (tolerance rozdělení)**: systém je schopen normálně fungovat i v případě, kdy dojde k výpadku jednoho z uzlů a tím zároveň k rozdělení daného systému na více částí (partitions).

Distribuované systémy přináší řadu výhod, na druhé straně společně s nimi se obecně jedná o poměrně složité systémy. Při jejich návrhu je nutné brát ohled na všechny tři výše zmíněné vlastnosti a tím čelit případným chybám. Tato skutečnost platí zejména pro distribuované prostředí. Počítačové sítě nejsou vždy spolehlivé a proto je nutné brát v potaz právě

[10]Distribovaný systém je obecně skupina několika výpočetních uzlů, která sdílí data a další výpočetní prostředky.



Obrázek 2.11: Znázornění CAP teorému a jeho tří základních principů a vztahů mezi nimi.

možnost rozdělení na více segmentů při výpadku některé části sítě (partition tolerance).

Běžně nastává situace, kdy se distribuovaný systém, který obsahuje na tisíce výpočetních uzlů, dostane do nekonzistetního módu na základě nedostupnosti jednoho z uzlů. Může se jednat o selhání daného stroje nebo jeho nedostupnosti v distribuovaném prostředí. V této chvíli dochází právě k rozdělení systému a z pohledu CAP teorému nastávají dva stavy, které jsou kombinací při toleranci rozdělení (parition tolerance) a dvěma zbývajícími, a sice dostupností (availability) a konzistencí dat (consistency) [14].

- **CP (consistency/partition tolerance)**: volba konzistence dat před dostupností je obecně doporučována v takovém systému, kdy je kladen požadavek na atomické operace čtení a zápisu.

- **AP (availability/partition tolerance)**: dostupnost systému může být jeden z hlavních požadavků především v případě, kdy je třeba zajistit funkčnost celého systému vůči dalším, na něm závislým aplikacím.

	id	timestamp	senzor 1	senzor 2	senzor 3	senzor 4
1	1	1448523476	20.54	5.41	88	135
2	2	1448523486	20.56	5.94	89	133
3	3	1448523496	20.58	5.84	88	139
4	4	1448523506	20.54	5.44	87	140
5	5	1448523516	20.51	5.28	86	137

Tabulka 2.2: Příklad SQL databázového schématu tabulky pro uložení dat ze senzorů.

Na obrázku 2.11 jsou zobrazeny jednotlivé kategorie a kombinace tří základních principů CAP teorému. Na základě těchto principů jsou navrženy a implementovány samotné databázové systémy. Do kategorie **CA** patří obecně tradiční RDBMS systémy a do dalších dvou kategorií patří většinou NoSQL databáze. V kategorii **CP** se jedná třeba o řešení Apache HBase, MongoDB nebo Redis. Do poslední kategorie **AP** patří například CouchDB, DynamoDB nebo Apache Cassandra.

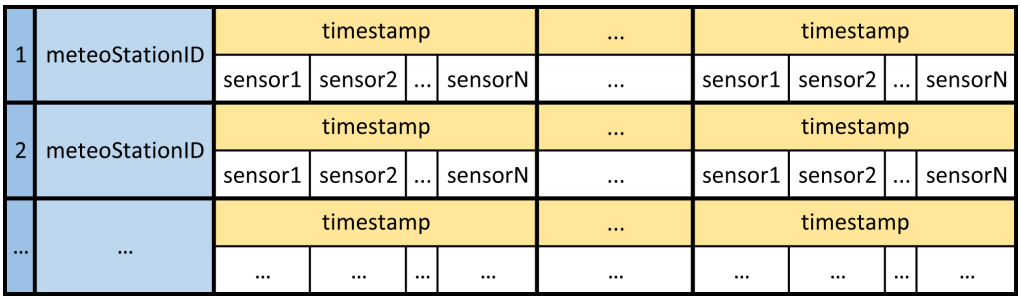
2.4 Problematika časových dat

Požadavky pro ukládání časových dat jsou stále náročnější. Zejména s příchodem Internetu věcí a dalších typů vestavěných zařízení jsou tyto požadavky v poslední době stále vyšší a vyšší. Vhodně zvolená databáze a struktura uložení dat jsou pro ukládání dat ze senzorů (časová data[11]) velice důležitá část celého systému. Nejsnadnějším řešením je použití tradiční relační SQL databáze, kdy se pro jednotlivé senzory ukládají jejich příslušné hodnoty v daném čase. Příklad takového schématu databázové tabulky je uveden v tabulce 2.2.

2.4.1 Flexibilita NoSQL databází

Relační databáze nabízí především výhodu v dotazovacím jazyce SQL, který je hojně a často používán. Hlavní nevýhodou relačních databází pro ukládání časových dat je flexibilita ve smyslu pevně daného schématu příslušné tabulky. Prvním krokem před nasazením SQL databáze je návrh a ověření používaného databázového schématu (tabulky). Jelikož však mohou

[11] Časová data (time series data) jsou data, která jsou obvykle reprezentována datovými body a jejich hodnotami v určitém časovém intervalu. Typickým příkladem jsou právě meteorologická data, která je nutná sbírat v časových intervalech, například průběh teploty.



Obrázek 2.12: Příklad struktury schématu NoSQL datbází při použití konceptu rodiny sloupců pro data z meteorologických stanic.

při ukládání nastat změny ve struktuře ukládaných dat (typicky přidání nebo odebrání senzoru pro případ meteorologických stanic), je to právě flexibilita relačních databází, která je značně limitujícím faktorem při jejich použití. NoSQL databáze za tímto účelem využívají koncept **širokých sloupců (wide column stores)**, který umožňuje ukládat velké množství dynamicky měnících se dat (struktury sloupců). Obecně lze říci, že se ve většině řešení NoSQL databází jedná o dokumentově orientované úložiště bez použití schémat, ačkoliv konkrétní implementace jsou různé [15].

Ze způsobu ukládání dat do relačních databází, jak je uvedeno v tabulce 2.2, je patrné, že data přibývají do nových řádků směrem dolů. Tento přístup je typický pro SQL. Na časová data je ale třeba se dívat jinak [16]. Stejně tak, jak plyne čas, přibývají i data. Z tohoto pohledu data pro jeden konkrétní klíč přibývají horizontálně, do strany. V tomto případě je výhodné uplatnit právě koncept širokých sloupců. Většina NoSQL databází je navržena tak, že pro jeden primární klíč umožňuje ukládat až miliardy libovolných sloupců. Tato skutečnost je možná především díky konceptu použití **rodin sloupců (column families)**. Příklad rodiny sloupců je uveden na obrázku 2.12, kde je znázorněna struktura ukládaných dat právě pro meteorologické stanice.

Pokud se ale nejprve zamyslíme nad formátem ukládaných dat, je zřejmé, že pro časová data jsou nutné pouze 2 údaje, a sice **hodnota** daného senzoru v určitém **čase**. Tomuto požadavku se nejvíce přibližuje právě databázový koncept z kategorie NoSQL databází, a sice formát uložení dat typu klíč-hodnota. Pro ukládání časových dat je k dispozici obecně několik možných řešení. Ve většině případech se jedná o nadstavbu, framework nad nějakou NoSQL databází. Tři nejznámější a nejrozšířenější řešení jsou následující.

- **KairosDB** je databázový framework, operující nad Apache Cassandra. Jedná se tedy o nadstavbu Apache Cassandra, využívající všechny její vlastnosti, zaměřený právě na práci s časovými daty.

- **OpenTSDB** využívá Apache HBase jako datové úložiště. Apache HBase je databázové řešení na frameworku Apache Hadoop. OpenTSDB tedy pro svoji funkčnost vyžaduje fungující cluster Hadoop, což mírně komplikuje její použití.

- **InfluxDB** je databáze pro ukládání časových dat a metrických údajů, která není závislá na dalších prvcích (HBase, Cassandra, Redis, apod.) a tudíž funguje plně samostatně. Hlavní nevýhodou je, že v současné verzi není implementována podpora pro clusterové prostředí.

Všechny z výše uvedených řešení jsou, stejně jako většina NoSQL databázových řešení, open source, takže jsou pod neustálým vývojem.

2.4.2 Práce a přístup k datům

Pokud jsou data někde uložena, je třeba k nim zajistit přístup a možnosti s nimi pracovat.

REST je architektura, která umožňuje přistupovat k datům pomocí standardních metod protokolu HTTP (GET, POST, PUT, DELETE). Výhodou této architektury je možnost přístupu k datům z různých aplikací pomocí jednotného rozhraní a jedná se o bezstavový protokol. REST dnes společné s formátem JSON tvoří základ většiny webových aplikací, ale jeho použití můžeme najít i v dalších aplikacích.

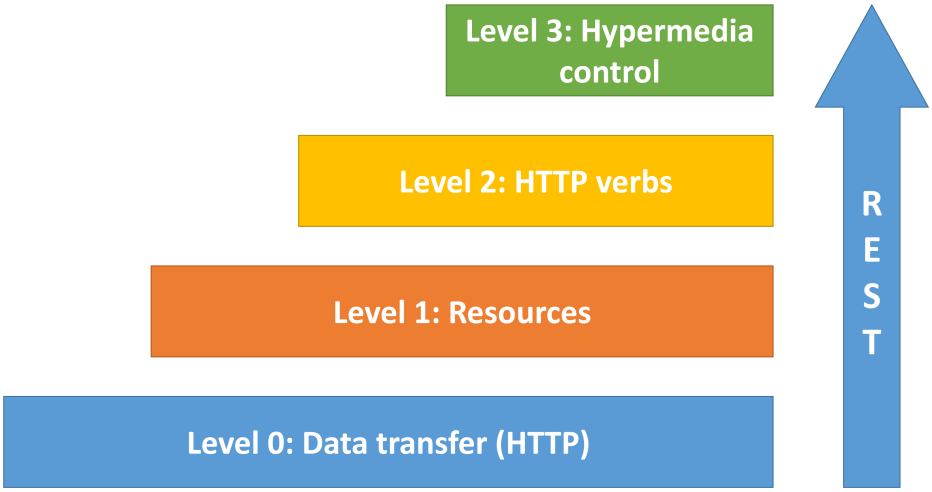
REST je architektura rozhraní, určená pro distribuované prostředí. Zajišťuje přístup a manipulaci s daty z různých umístění. Na rozdíl od procedurálních metod XML-RPC nebo SOAP je REST orientován více na data a obsah [17].

Z pohledu práce s daty existuje dnes dobře známý koncept **CRUD**, jehož jednotlivé části tvoří základní operace nad daty, které jsou hodně spojované s databázovými operacemi. Můžeme říci, že REST tvoří mapování vlastních metod právě na metody modelu CRUD. Samotné mapování je zřejmé z tabulky 2.3.

REST architektura se sestává z několika úrovní, které je možné jednolivě popsat a kde každá z nich má konkrétní účel. Rozdělení do úrovní provedl a popsal Leonard Richardson ve své publikaci Richardson Maturity Model. Jednotlivé úrovně jsou zobrazeny na obrázku 2.13.

REST	CRUD
POST	Create
GET	Read
PUT	Update or Create
DELETE	Delete

Tabulka 2.3: Mapování HTTP metod REST architektury na standardí CRUD operace.



Obrázek 2.13: REST úrovně a jejich pořadí.

- **Nultá úroveň**: na nejnižší úrovni se definuje pouze přenos dat. Dnes je REST architektura používaná ve většině případů nad protokolem HTTP.

- **První úroveň - zdroje (resources)**: zdroje jsou data nebo informace, se kterými se pomocí REST pracuje. Jejich definice a jednoznačné označení jejich umístění je žádoucí. Např. GET /articles vrátí seznam všech článků, ale GET /articles/1 vrátí pak pouze jeden konkrétní článek. Správné pojmenování zdrojů je nutná podmínka pro jejich správnou manipulaci.

- **Druhá úroveň - HTTP slovesa (verbs)**: nejznámější metodou je metoda GET. Vedle ní existují i další metody. Jsou to POST, PUT, DELETE, OPTIONS a další. HTTP slovesa definují akci, kterou bude požadavek představovat. S požadavky souvisí také **stavové kódy**. Jedná se o předem definované odpovědi na požadavky. Typickým pří-

kladem je odpověď 200 OK - požadavek byl zpracován v pořádku nebo 404 Not Found - zdroj nebyl nalezen a další.

- **Třetí úroveň - Hypermedia controls**: třetí úroveň je známá především pod zkratkou **HATEOAS** (Hypermedia as the Engine Of Application State). Z pohledu uživatele je znám pouze jeden zdroj (koncový bod), který vrací na požadavky odkazy na další zdroje, ten zase na další atd. Hlavní výhodou je nezávislost klienta na URL adresách.

Velice důležitým požadavkem je udržet REST API bezstavové. Každý zaslaný požadavek by měl být přesně specifikován a měl by obsahovat všechny potřebné informace k jeho vykonání. Použití REST API s sebou přináší řadu možností, které jsou pro systém sběru dat výhodné. Zejména se jedná o přístup k datům, ke kterým pak mohou přistupovat různé aplikace a další systémy z různého umístění.

KAPITOLA 3

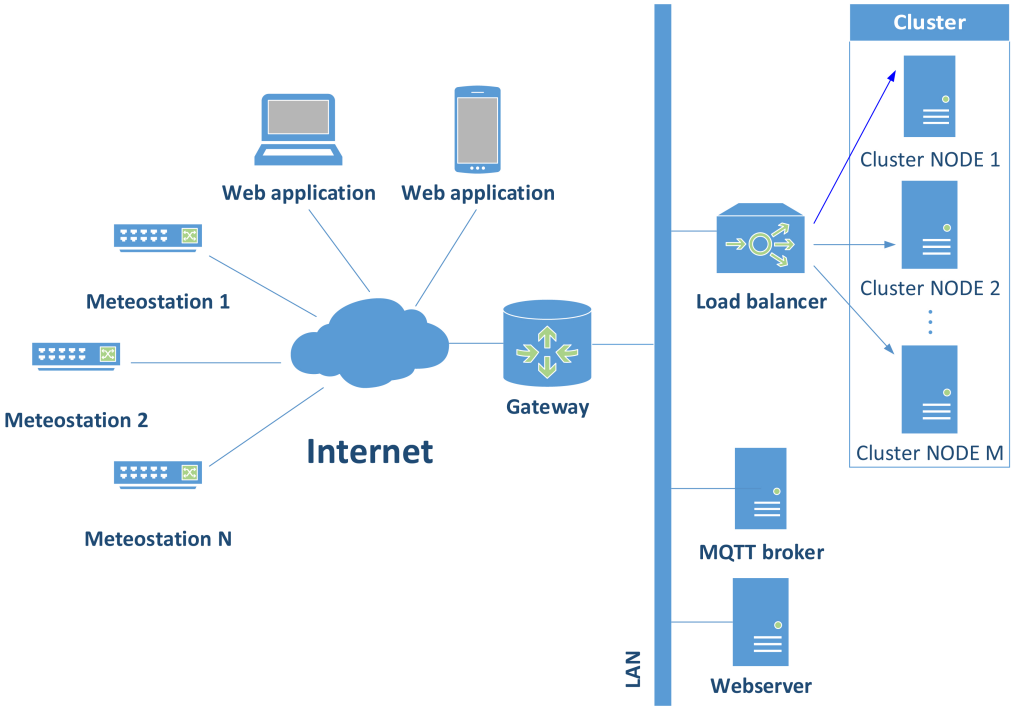
Realizace modelového řešení

V předchozí kapitole byly popsány možnosti řešení, pomocí kterých je možné sestavit výsledný systém pro sběr dat. Tato kapitola obsahuje shrnutí, popis a základní testování výsledného systému. Architektura výsledného systému pro sběr dat je nejlépe zřejmá ze schématu na obrázku 3.1. Schéma definuje všechny komponenty systému a vztahy mezi nimi.

3.1 Meteorologické stanice

Pro realizaci samotné meteorologické stanice byla zvolena otevřená HW platforma Arduino, která nabízí oproti ostatním HW platformám několik hlavních výhod.

- **Cenová dostupnost**: cena byla jedním z hlavních faktorů pro volbu platformy. Jelikož je Arduino otevřená platforma, jsou k dispozici schémata samotných desek. Díky tomu existuje několik výrobců, kteří tyto desky vyrábí, a proto je zde prostor pro velkou konkurenci a tím relativně nízké koncové ceny.
- **Velké množství periferií**: pro Arduino existuje velké množství různých periferií, ať se jedná o síťové rozšiřující desky (WiFi, Ethernet, XBee) nebo senzory (teplota, tlak, vlhkost, detektory pohybu).
- **Uživatelsky přívětivé IDE**: programovací prostředí Arduina je velice přehledné a intuitivní. Postačující podmínkou je dobrá znalost programovacího jazyka C, ve kterém se Arduino desky pomocí IDE programují.



Obrázek 3.1: Architektura a vazby mezi jednotlivými komponentami výsledného systému pro sběr dat.

- **Podpora a aktivní komunita**: díky otevřenosti platformy je zde velice aktivní komunita uživatelů a vývojářů. K dispozici je proto velké množství knihoven a zdrojových kódů, které je možné použít.

Konkrétní použitý model je Arduino UNO. Arduino UNO je deska, která je založena na mikroprocesoru Atmega 328. Jedná se o nejpoužívanější a nejvíce robustní model, který je poměrně odolný i například vůči nesprávnému zapojení a tím se dobře hodí pro prototypování a vývoj podobných aplikací.

3.1.1 Síťová konektivita

Za účelem připojení desky Arduino k Internetu je použito rozhraní Ethernet. K dispozici je rozšiřující Ethernetový modul se standardním RJ-45 konektorem a SW knihovnou, podporující síťovou komunikaci. Použití modulu je velice jednoduché a přímočaré. Fyzicky se jedná o desku stejných rozměrů, jako má Arduino UNO, která navíc obsahuje síťový konektor a je osazena mikroprocesorem Wiznet W5100, který poskytuje možnosti práce s IP stackem jak pro UDP, tak i pro TCP komunikaci. Součástí obslu-

Senzor	Typ	Popis
Teplota	DALLAS DS18B20	Digitální teplotní čidlo určené pro měření teploty v rozsahu (-10)-(+85)°C.
Světlo	BH1750	Digitální světelný senzor používaný pro měření světelné intenzity v rozsahu 1-65535lux.
Vlhkost a teplota	DHT11	Digitální čidlo určené pro měření teploty v rozsahu 0-50°C a vlhkosti v rozsahu 20-90%. Pro zapojení do digitálního vstupu desky Arduino je třeba využít pomocný odpor o hodnotě $4.7k\Omega$.

Tabulka 3.1: Seznam senzorů, použitých pro získávání dat z meteorologické stanice.

ného programu je pak volání funkce pro získání IP adresy pomocí protokolu DHCP. Tímto je zajištěna především možnost použití meteorologické stanice v libovolné lokální síti bez nutnosti její předchozí konfigurace.

3.1.2 Senzory

Pro Arduino existuje velké množství senzorů. Pro potřeby meteorologické stanice byly zvoleny nejběžnější a dobře dostupné senzory, jejichž výčet a základní parametry je uveden v tabulce 3.1. Fyzické zapojení senzorů k desce Arduino je uvedeno na obrázku 3.2. Pro jednotlivé senzory existují různé implementace knihoven, které umožňují jejich použití. Z tohoto důvodu je nutné zvolit správné verze, které jsou ve většině případů závislé na konkrétním modelu a typu daného senzoru.

3.1.3 Obslužný program

Jak již bylo zmíněno, pro Arduino je k dispozici multiplatformní, uživatelsky přívětivé IDE, pomocí kterého je možné v programovacím jazyce C programovat zařízení. IDE je napsané v jazyce Java a provádí kompilaci z jazyka C do binárního kódu příslušného procesoru, který je následně přes USB rozhraní nahrán do samotné desky Arduino.

Pro naprogramování meteorologické stanice jsou nutné knihovny, umožňující práci se všemi použitými komponentami. Jedná se zejména o knihovny

pro práci s ethernetovým adaptérem, která implementuje základní síťové protokoly, jako je například ARP, DHCP, apod., a je standardně součástí vývojového prostředí. Další důležitou součástí je knihovna pro podporu samotného protokolu MQTT. Za tímto účelem je použita knihovna **PubSubClient**[12].

Obslužný program desky Arduino má dvě hlavní části:

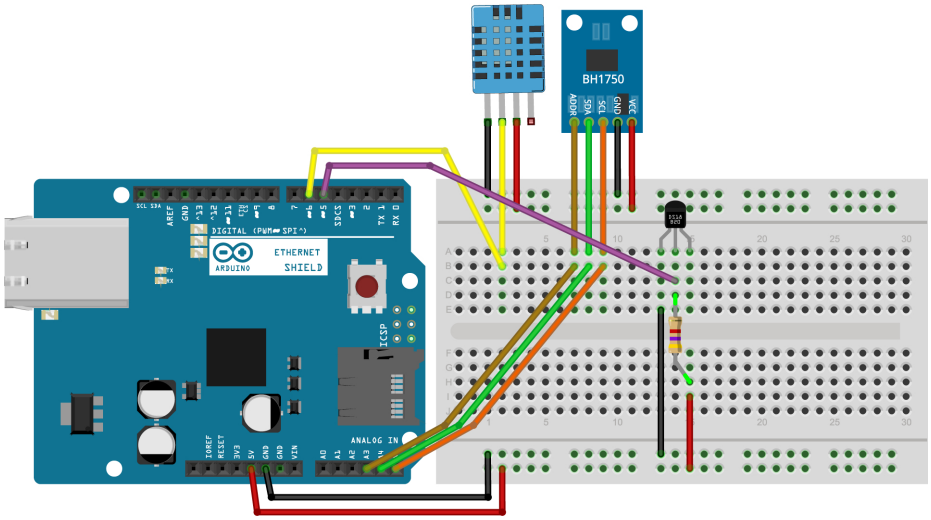
- **Funkce setup**: funkce setup je volána pouze jedenkrát, a sice při fyzickém zapnutí zařízení. V této funkci probíhá inicializace knihoven a nastavení jejich parametrů. V případě meteorologické stanice se nejprve jedná o inicializaci časových funkcí pro čtení dat ze senzorů a odesílání dat. Dále pak probíhá inicializace knihoven pro práci s připojenými senzory a získání IP adresy pomocí protokolu DHCP. Následně je provedeno připojení k MQTT brokeru na příslušné adrese a portu. V případě, že jedna z těchto fází selže, je smyčka zavolána znovu.
- **Funkce loop**: funkce loop je hlavní smyčka obslužného programu. Jak její název napovídá, je vykonávána stále dokola a jsou v ní vykonávány všechny funkce a akce obslužného programu. Hlavní dvě fuknce, které jsou volány, jsou knihovní funkce časových čítačů, které zajišťují odpočítávání času před voláním samotných funkcí pro čtení dat ze senzorů a odesílání dat. Tato skutečnost je důležitá zejména pro čtení dat ze senzorů, kdy použité knihovny přímo vyžadují delší časový interval před čtením samotných hodnot. Pokud není tento minimální časový interval dodržován, může docházet k nepřesnostem čtení.

Schéma provádění příslušných částí v obou smyčkách obslužného programu je uvedeno na obrázku v příloze A.

3.2 Mezivrstva pro výměnu zpráv

Middleware vrstva v tomto případě slouží zejména jako prostředník pro předávání dat mezi meteorologickými stanicemi, serverovým prostředím, na kterém jsou data dále zpracována, a případnými klientskými aplikacemi, která s daty pracují. Pro systém sběru dat bylo zvoleno řešení Mosquitto MQTT broker.

[12]PubSubClient je knihovna pro Ethernetový modul Arduino, zajišťující API pro práci s MQTT protokolem. Knihovna je dostupná z https://github.com/knolleary/pubsubclient.



Obrázek 3.2: Fyzické zapojení senzorů k desce Arduino pomocí nepájivého pole (breadboard).

3.2.1 Mosquitto

Mosquitto je open source broker, který implementuje MQTT protokol. Hlavní jeho výhodou je jednoduchost instalace, široké možnosti konfigurace a od verze 1.4 také zabudovaná podpora protokolu websockets. Konfigurace brokeru je možná pomocí konfiguračního souboru **mosquitto.conf** a dle online dokumentace je možné snadno nakonfigurovat potřebné vlastnosti brokeru. Pro účely sběru dat z meteorologických stanic bylo nutné nakonfigurovat především tyto základní komponenty:

- **MQTT listener**: komponenta, která naslouchá na TCP portu 1883 a udržuje spojení s klienty. V systému jsou dvě základní role klientů: meteorologické stanice a MQTT subscribe agent, který odebírá zprávy z příslušných kontextů, do kterých je zasílají stanice.

- **Websockets listener**: komponenta, která naslouchá na TCP portu 9001 a umožňuje trvalé TCP spojení s klientem, kterým je v tomto případě webová aplikace, umožňující zobrazování aktuálních dat z meteorologických stanic, které jsou v pravidelných intervalech odesílány.

- **Persistence**: jedná se o základní komponentu, která umožňuje uchovávat informace o připojených klientech a příchozí zprávy nejen v operační paměti, ale také v interní databázi brokeru (obvykle reprezen-

tována jako soubor na disku). Při restartu služby jsou pak tyto informace z databáze obnoveny.

Na rozdíl od jiných řešení MQTT brokeru je konfigurace u Mosquitto velice jednoduchá a přímočará. Ukázka konfigurace tří základních výše zmíněných parametrů z konfiguračního souboru mosquitto.conf je uvedena níže.

```
persistence true
persistence_location /var/lib/mosquitto/

listener 1883 127.0.0.1
protocol mqtt

listener 9001 127.0.0.1
protocol websockets
```

3.2.2 Použití Mosquitto v clusteru

Mosquitto MQTT broker nemá implementovány mechanismy pro použití v clusterovém prostředí. Existují však jiná řešení (například VerneMQ), které tuto funkcionalitu nabízí, ale pro použití v navrhovaném systému pro sběr dat pak narážíme na dva možné scénaře problémů, dle operačního módu, ve kterém by daný MQTT broker fungoval. Problémy se týkají skriptu, který zajišťuje odběr a zpracování zpráv od klientů (MQTT subscribe agent). Z hlediska distribuovanosti systému se nabízí řešení, kdy by skript běžel paralelně na více uzlech, společně s MQTT brokery. Příchozí zprávy by byly dle nějakého algoritmu na předsazeném load balanceru předávány na tyto uzly, kde by byly jednotlivými skripty zpracovány. Výsledkem by byla větší propustnost systému právě díky většímu počtu současně připojených klientů.

- **MQTT broker v clusterovém módu**: pro otestování MQTT brokeru pro navržený systém sběru dat bylo využito řešení VerneMQ, které přímo implementuje podporu pro clusterové prostředí. Clusterový mód dovoluje všem MQTT brokerům clusteru předávání informací o připojených klientech prostřednictvím zasílání vnitřních zpráv. U VerneMQ se jedná o proprietární komunikační mechanismus, založený na protokolu Erlang[13]. Problémem tohoto přístupu je vícenásobné zpracování příchozích zpráv MQTT subsribe agenty. Jelikož na

[13]Erlang je funkcionální programovací jazyk, ve kterém je napsán clusterový MQTT broker VerneMQ.

každém MQTT brokeru běží jeden, všichni jsou připojeni pro odebírání zpráv z jednoho konkrétního kontextu, který je určen maskou a pokrývá všechny připojené klienty.

- **MQTT broker v izolovaném módu**: pro koncept sběru dat, kdy je vyžadována pouze jednosměrná komunikace ze strany připojených klientů, je možné toto zajistit použitím váce MQTT brokerů v izolovaného módu. Jednotliví klienti jsou pak připojeny ke konkrétním MQTT brokerům, na kterých běží skript pro zpracování příchozích zpráv MQTT subscribe agent. Tento skript naslouchá pouze na adrese daného MQTT brokeru, odkud odebírá a zpracovává zprávy z daných kontextů (všechny kontexty mohou být stejné). Problém tohoto řešení je však v obousměrné komunikaci směrem ke klientům, například v případě, kdy je nutné nastavit provozní parametry zařízení. Jelikož se cluster nechová jako jeden celek, nemohou být zprávy předány konkrétnímu klientu, protože si jednotlivé uzly nepředávají informace o místě, kde na kterém brokeru je konkrétní klient v tu danou chvíli připojen.

Z tohoto důvodu je řešení MQTT brokeru omezeno a běží pouze na jednom uzlu. Za účelem ověření modelového řešení netvoří tato skutečnost žádná zásadní omezení, pouze v počtu současně připojených klientů a tím v omezené možnosti škálování.

3.3 Serverové prostředí

Pro potřeby ověření funkčnosti a testování modelového řešení výsledného systému je nutné vytvořit testovací serverové prostředí. Kvůli omezené dostupnosti HW prostředků byla zvolena metoda virtualizace a bylo vytvořeno virtuální serverové prostředí pomocí hypervizoru **Xen**. Xen je open source virtualizační technologie, která je dnes poměrně hojně využívána v produkčním prostředí a na které běží například Amazon Web Services, Verizon Cloud a další komerční cloudová řešení.

Parametry fyzického stroje, na kterém testovací virtuální serverové prostředí běží, jsou pro ilustraci uvedeny v tabulce 3.2. Za účelem simulace databázového clusteru byly vytvořeny tři dedikované uzly. Všechny komponenty systému jsou ve virtuálním prostředí propojeny virtuálním přepínačem a nacházejí se ve společné virtuální LAN síti. Celá architektura prostředí je zřejmá z obrázku 3.1.

CPU	Intel Core i5-2500, @3,3GHz
RAM	8GB, DDR3, 1067MHz
HDD	240GB, 7200RPM
LAN	1Gbit/s

Tabulka 3.2: Parametry fyzického stroje pro ověření modelového řešení.

3.3.1 Load balancer

Hlavním účelem použití load balanceru je zajištění vysokého výkonu a vysoké dostupnost pro clusterové prostředí, které je pak snadno škálovatelné, spolehlivé a odolné vůči výpadkům. Pro potřeby systému pro sběr dat bylo vyvažování zátěže použito pouze pro databázový systém. Cílem nasazení load balanceru je skutečnost, aby celý databázový cluster vypadal z pohledu klienta (MQTT subscribe agent, který zapisuje data, nebo klientské aplikace, které data získávají) jako jeden výkonný databázový server.

Pro zajištění této funkčnosti byl použit Linuxový Virtuální Server (LVS). LVS pracuje pouze na čtvrté, transportní vrstvě **ISO/OSI modelu** a tím zajišťuje rychlejší zpracování příchozích požadavků na rozdíl od jiných řešení, pracujících na vyšších vrstvách, kdy je nutné jednotlivé příchozí datové pakety zkoumat do hloubky. Z důvodu, že jsou všechny uzly databázového clusteru stejné konfigurace, byla pro samotné vyvažování zátěže byla zvolena metoda Round Robin.

Load balancer požaduje také znalost stavu jednotlivých serverů, na které pak může předávat příchozí požadavky. Tyto servery je nutné spravovat a minotorovat. Za tímto účelem byla použita unixová utilita **mon**. Jedná se o unixový program, který v pravidelných časových intervalech monitoruje stavy jednotlivých serverů na základě dostupnosti monitorované služby na daném portu. V případě nedostupnost/dostupnosti umožňuje volání uživatelských skriptů a tím zajišťuje možnosti úprav dané konfigurace uzlů.

Následující skript umožňuje v případě výpadku jednoho ze serverů jeho odebrání z množiny aktivních serverů, využívaných programem LVS. V opačném případě, při opětovné dostupnosti serveru, umožňuje jeho přidání zpět do množiny aktivních serverů. Příklad takového skriptu je uveden níže.

```
#!/bin/bash

if [ "$9" = "-u" ]; then
        echo "`date`: KairosDB $6 is UP."
        ipvsadm -a -t lb:8080 -r $6:8080 -m
else
        echo "`date`: KairosDB $6 is DOWN."
        ipvsadm -d -t lb:8080 -r $6:8080
fi
```

3.3.2 Databázový cluster

Jako databázový engine pro ukládání a archivaci dat byla zvolena NoSQL databáze Apache Cassandra. Apache Cassandra je široce škálovatelná open source NoSQL databáze. Byla navržena primárně za účelem ukládání velkého množství dat napříč několika servery a datovými centry, zajišťující vysokou dostupnost bez jediného bodu chyby (SPOF). Topologie databáze je založena na kruhové architektuře (ring) [18].

Hlavní výhody použití databáze Cassandra:

- **Škálovatelnost**: díky vnitřní architektuře systému umožňuje Cassandra lineární škálování. Nové uzly je možné přidávat za běhu. Pokud 2 uzly umožňují 100000 transakcí za vteřinu, 4 uzly umožňují 200000 transakcí za vteřinu [19].

- **Automatická replikace**: Cassandra je navržena pro nasazení na několika stovkách uzlů, mezi různými clustery v různých datových centrech. V takovém prostředí jsou běžné výpadky některých uzlů. Pro každé tabulkové schéma existuje parametr replikačního faktoru, díky čemuž je zajištěna dostupnost všech ukládaných dat.

- **Decentralizovanost**: architektura Cassandry je založena na konceptu, kdy jsou si všechny uzly v systému rovny. Není zde žádný hlavní uzel a data jsou rovnoměrně rozložena mezi všemi uzly clusteru (v závislosti na nastaveném parametru stupně replikace). Tímto je zajištěno, že v systému není žádný SPOF.

- **Dotazovací jazyk**: Cassandra používá proprietární dotazovací jazyk CQL (Cassandra Query Language), který svou konstrukcí vychází z tradičního SQL jazyka. Umožňuje tak jednoduché dotazování nad uchovávanými daty.

Databázový cluster byl zvolen zejména pro požadavky ukládání velkého množství dat a pro snadnou možnost vertikálního škálování.

3.3.3 Databázový framework

KairosDB je databázový framework, vyvinutý primárně za účelem práce s časovými daty. Jedná se o framework, pracující nad samotnou databází, kterou může být H2 (SQL databáze), Apache HBase nebo Apache Cassandra (obě NoSQL). Apache Cassandra umožňuje ukládání dat podle konceptu širokých sloupců. Každý řádek pak může obsahovat až 2 miliardy samotných sloupců. Schéma KairosDB pro tuto databázi je nastaveno tak, aby každý řádek uchovával 3 týdny záznamů při periodě ukládání dat 1ms (1 * 1000 * 3600 * 24 * 21 = 1 814 400 000). Tato skutečnost umožňuje vyšší výkon databáze při dotazování nad uchovávanými daty.

Databázový framework pracuje primárně s **datovými body**, což je základní entita uchovávaných dat. Každý tento datový bod obsahuje

- jméno metriky (metric name),
- hodnotu (value),
- časové razítko (timestamp) a
- seznam značek (list of tags).

První tři vlastnosti datových bodů jsou zřejmé. Čtvrtý z nich (seznam značek) obsahuje další informace, které následně slouží k upřesnění druhu a vlastností daných datových bodů. Pomocí těchto vlastností je dále možné data seskupovat a logicky členit. Seznam značek se využívá především při dotazování, kdy je třeba výsledná data seskupovat pomocí klasické SQL operace GROUP BY. Datové body mohou být seskupovány na základě několika značek. Typickým příkladem pro meteorologické stanice může být dotaz pro zjištění všech teplot z daného senzoru, napříč všemi meteorologickými stanicemi v daném časovém období pro následné zjištění například nejvyšší teploty.

KairosDB nabízí dvě základní API pro práci s daty, a sice pomocí protokolu **Telnet** a **HTTP**.

3.3.3.1 Telnet

Pomocí protokolu Telnet je možné se dotazovat i vkládat datové záznamy do databáze. Konfigurace KairosDB umožňuje definovat číslo portu pro samotný Telnet server, na kterém bude aplikace naslouchat. Příklad jednoduchého shellového skriptu pro vložení nových záznamů (datových bodů) je uveden níže. Hlavní využití Telnet protokolu pro práci s daty je zejména při testování systému.

```
#!/bin/bash

# Current time in milliseconds
now=$(($(date +%s%N)/1000000))
metric=ms.t1
value=21.56
host=10.1.1.201

echo "put␣$metric␣$now␣$value␣host=A" | nc -w 30 $host
```

3.3.3.2 HTTP

KairosDB nabízí HTTP REST API pro práci s daty. Oproti rozhraní Telnet se jedná se o rozhraní, které nabízí širší možnosti využití a bohatší formy dotazování. Základním formátem pro přenos dat je formát JSON. Záleží pouze na infrastruktuře síťového prostředí, odkud je možné se na uchovávaná data dotazovat. Ve výsledném systému je HTTP API přístupné pouze z lokální sítě clusteru, což s sebou přináší dvě hlavní výhody:

- **Rychlost** je kritickým požadavkem pro správné ukládání dat z meteorologických stanic. Jelikož je HTTP API přístupné z lokální sítě, kde se nachází i ostatní prvky systému (především MQTT subscribe agent), nemusí datové pakety cestovat přes dlouhé vzdálenosti například přes Internet.

- **Bezpečnost**: KairosDB nabízí mechanismy pro zajištění kontroly přístupů pomocí uživatelských jmen a hesel společně s možností použití SSL připojení. Fyzická data jsou také k dispozici pouze na strojích clusteru.

Příklady struktury těla HTTP POST požadavku pro dotazování a vkládání dat nad KairosDB jsou uvedeny v příloze B.

3.3.3.3 Analýza a vyhodnocení dat

REST API u KairosDB poskytuje základní analytické metody pro práci s uchovávanými daty. Součástí je webová stránka, která je hostována na každém uzlu databázového clusteru a poskytuje rozhraní pro manipulaci s daty. Pomocí ní je možné zobrazovat grafické průběhy dat společně s dalších parametrů.

- **Časový rozsah**: data je možné vybrat v absolutním časovém intervalu nebo zadáním relativního časového úseku od aktuálního data.
- **Volba metriky**: metrika je základním parametrem pro práci s daty. Jedná se typicky o jeden požadovaný druh dat, například teplota z určitého teplotního senzoru. Obecně je možné vybrat více těchto metrik a pro každou z nich lze definovat upřesňující údaje, kterými mohou být seskupování dat (obdoba agregační gunkce GROUP BY v SQL) nebo vzorkování výsledků (sampling rate). Pro každou metriku je možné dále definovat sadu značek dle požadovaných parametrů, které jsou v databázi uložené společně se samotnými datovými body.
- **Agregační funkce**: nad výsledky jednotlivých metrik je možné provádět různé agregační funkce typu součet, průměr, maximum, minimum, apod.

Samotná webová aplikace umožňuje pohodlnou práci s daty. Nabízí především možnosti grafické reprezentace dat ve formě grafů. Zároveň slouží jako účinný nástroj pro vývojáře klientských aplikací, protože umožňuje ladit formulace jednotlivých dotazů, pomocí kterých je pak možné se dotazovat prostřednictvím HTTP REST API. Prostředí webové aplikace je zobrazeno na obrázku 3.3.

Webová aplikace umožňuje prezentaci dat také ve formě grafů. Příklad grafu je zobrazen na obrázku 3.4, který zobrazuje průběh tří sbíraných dat z meteorologických stanic (modře teplota 1, červeně teplota 2, zeleně vlhkost).

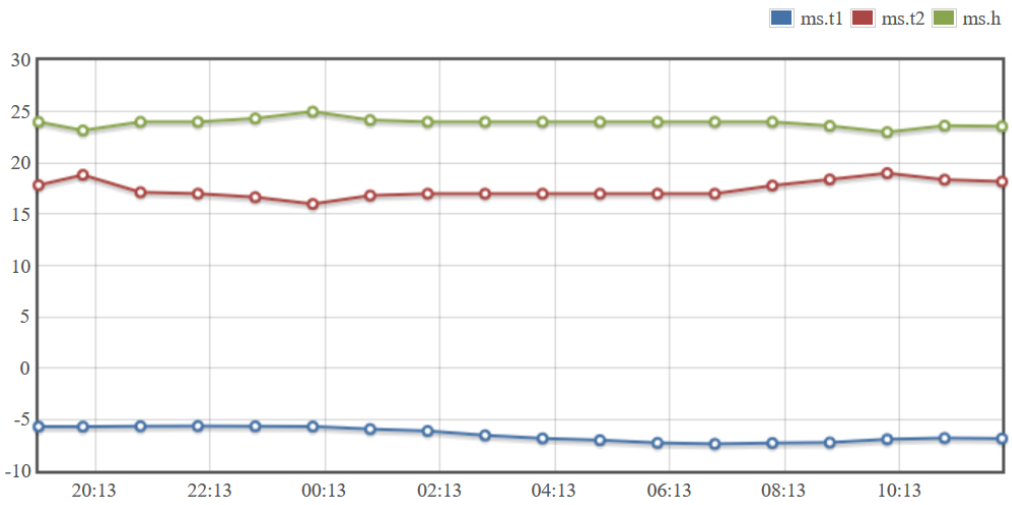
Když už je možné data získávat, nabízí se další úkony, které je možné nad daty provádět. Jedná se o analytické a statistické metody, které přináší podrobný pohled nad sbíraná data a umožňují jejich vyhodnocení. Data je možné reprezentovat **graficky**, a to pomocí základních statistických typů grafů:

- **Spojnicový graf**: zobrazuje trend vývoje jednotlivých měřených veličin po dané časové období.

Obrázek 3.3: Prostředí uživatelské webové aplikace KairosDB pro práci s daty.

- **Krabicový graf**: krabicový graf je vhodný ukazatel pro grafickou vizualizaci dat na základě jejich kvartilů. Kvartily rozdělují statistický soubor na čtvrtiny. 25% prvků pak má hodnoty nižší než dolní kvartil $Q_{0,25}$ a 75% prvků má hodnoty větší než horní kvartil $Q_{0,75}$.
- **Histogram**: slouží k zobrazení četnosti výskytu jednotlivých hodnot měřené veličiny ve sloupcovém grafu se sloupci stejné šířky.

Pro podrobnou statistickou analýzu dat je dnes k dispozici několik SW nástrojů, které se soustředí především na práci s daty a jejich grafické zpracování. Může se jednat o jednoduché tabulkové procesory nebo specializované statistické nástroje. Stručný výčet dnes nejpoužívanějších SW nástrojů je uveden níže.



Obrázek 3.4: Příklad grafického výstupu z webové aplikace KairosDB.

- **Matlab**: Matlab nabízí prostředí pro práci s daty, zejména vícerozměrného maticového charakteru. Pro provedení analýzy a zpracování dat záleží pouze na schopnostech programátora a použití dostupných toolboxů, například Statistics and Machine Learning Toolbox.

- **R**: R patří mezi nejznámější statistické nástroje, které se používají pro provádění statistických úkonů nad daty. Nabízí široké množství statistických metod a funkcí jak pro statistické, tak pro grafické zpracování dat.

- **Microsoft Excel**: mezi tabulkovými procesory je právě Microsoft Excel nejrozšířenější aplikací. Nabízí základní statistické funkce a umožňuje snadnou grafickou reprezentaci dat.

Velkou výhodou pro zpracování dat z meteorologických stanic je použití aplikace Matlab, který přímo podporuje získávání dat pomocí RESTful webových služeb. Pomocí funkce webread[14] je možné přímo získat data z HTTP REST API. Při použití s KairosDB, které vrací data ve formátu JSON, umožňuje funkce webread automatickou konverzi získaných dat do datového typu pro Matlab a jejich následné zpracování.

[14]Webread je název knihovní funkce Matlabu pro získávání dat z webových zdrojů prostřednictvím RESTful rozhraní.

3.3.4 Webový server

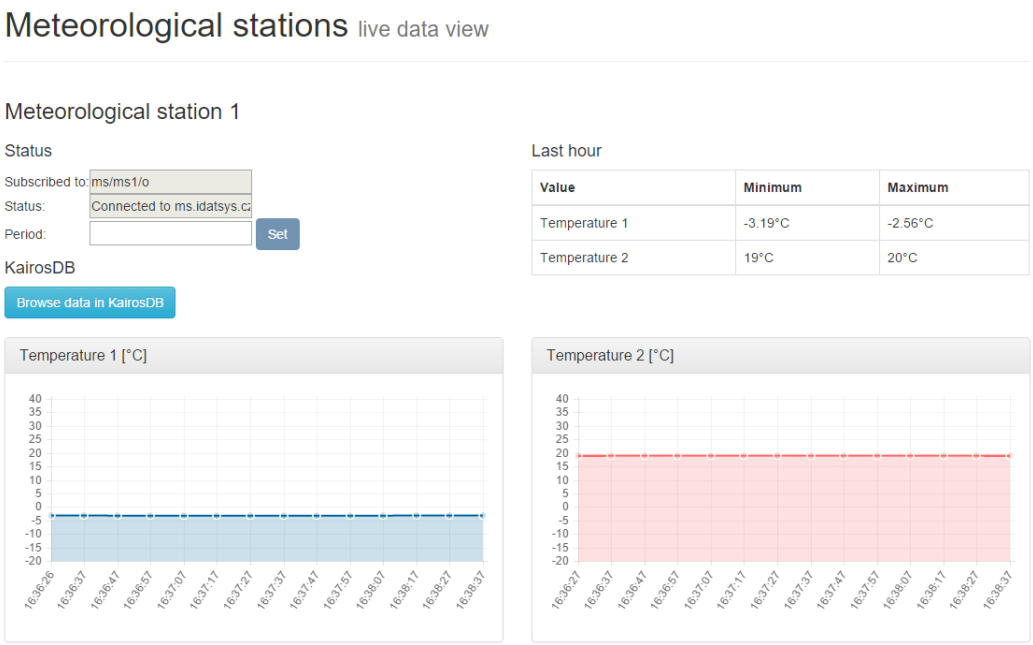
Webový server je místem pro hostování klientských webových aplikací, které umožňují především prezentaci dat. Součástí modelového řešení je jednoduchý webový server, který beží ve virtuálním systému a je součástí serverového prostředí. Pro hostování webových stránek je využit dnes běžně používaný webový server **Apache** s jeho minimální konfigurací, která dostatečně zajišťuje potřeby pro hostování klientské aplikace.

3.3.5 Klientská aplikace

Součástí modelového řešení je jednoduchá webová aplikace pro zobrazování aktuálních dat z meteorologických stanic a zobrazení několika údajů z KairosDB, získaných pomocí POST HTTP požadavků. Jedná se o aplikaci, která je založená na technologii websockets a která se sestává ze tří základních částí:

- Kód v programovací jazyce JavaScriptpro získávání aktuálních dat, který využívá JS knihovnu **PAHO-MQTT**, která umožňuje připojení k MQTT brokeru právě s využitím protokolu websockets a tím získává aktuální data, která jsou na broker odesílána z meteorologických stanic. Prakticky se jedná o připojení k danému kontextu na MQTT brokeru pomocí protokolu websockets a pravidelnou aktualizaci aktuálních dat na stránce.
- Grafická JavaScript knihovna **Chart.js**, která umožňuje aktuální data zobrazovat ve formě grafů. Díky websockets jsou data pravidelně doplňována a tato grafická knihovna je umožňuje aktuálně zobrazovat bez potřeby aktualizace samotné HTML stránky.
- Pro příklad získávání dat z KairosDB pomocí HTTP REST API je součástí aplikace také práce s XML HTTP POST požadavky, pomocí kterých je možné získávat data z KairosDB. Pro ilustraci jsou zobrazeny minimální a maximální hodnoty ze dvou teplotních čidel, které jsou v pravidelných časových intervalech obnovovány.

Pro ilustraci komunikace z klientské aplikace směrem k meteorologickým stanicím byla do klientské aplikace zakomponována možnost nastavení periody zasílání dat. Uživatel zadá požadovanou periodu, která je následně prostřednictvím MQTT protokolu předána na MQTT broker, odkud si ji vyzvedne příslušná meteorologická stanice. Právě v tomto případě lze příkladně demonstrovat vhodnost použití protokolu MQTT pro podobné aplikace. Ukázka výsledné aplikace je na obrázku 3.5.



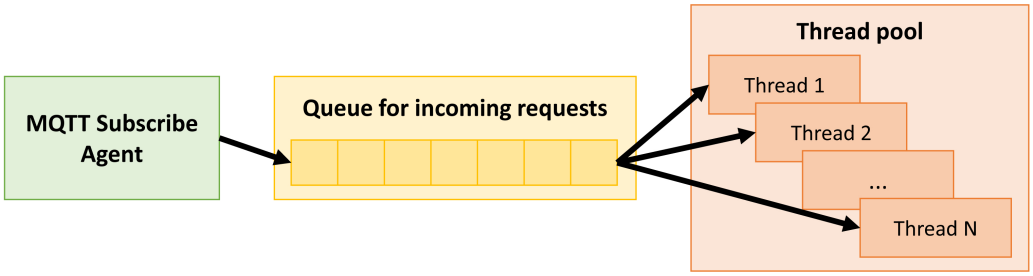
Obrázek 3.5: Ukázka prostředí webové klientské aplikace.

3.3.6 MQTT subscribe agent

Za účelem ukládání dat z meteorologických stanic do databáze bylo třeba zajistit jejich zpracování z middleware vrstvy na straně backendu. Za tímto účelem bylo vytvořeno řešení ve formě MQTT odběratele zpráv, který odebírá všechny zprávy v daném MQTT kontextu. Pro odebírání zpráv byla využita funkcionalita víceúrovňového maskování kontextu a při zpracování zprávy je využito neblokujícího I/O modelu.

Víceúrovňové maskování kontextu umožňuje MQTT klientu odebírat zprávy, které jsou určené do různých kontextů, ale mají společnou nějakou část. Za tímto účelem byla definována struktura kontextů pro sběr dat z meteorologických stanic, která umožňuje jejich zpracování právě jedním MQTT klientem.

Neblokující I/O model je používán primárně za účelem rychlejšího zpracování příchozích požadavků. Nezbytným prvkem tohoto modelu je fronta, která uchovává příchozí požadavky pro obslužná vlákna, která jsou následně zodpovědná za jejich zpracování. Koncept je zřejmý z obrázku 3.6.



Obrázek 3.6: Hlavní komponenty služby MQTT subscribe agent pro odebírání a zpracování dat z MQTT brokeru.

Samotná služba je napsaná ve skriptovacím jazyce Python a má tyto tři základní komponenty:

- **MQTT odběratel** je zodpovědný za odebírání zpráv z příslušného MQTT kontextu, který je určen víceúrovňovou maskou. Tato maska je ve tvaru **ms**/#, kde znak # zastupuje všechny podúrovně daného kontextu. Za tímto účelem byla použita knihovna PAHO-MQTT[15].

- **Fronta** je nejdůležitějším prvkem této služby. MQTT knihovna pro Arduino totiž podporuje pouze zasílání zpráv QoS 0, u kterých není zaručeno žádné doručení zprávy příjemci. Fronta v tomto případě tedy zajišťuje dočasné uložení zpráv před jejich zpracováním obslužnými vlákny a tím zaručuje jejich zpracování.

- **Obslužná vlákna** jsou vytvořena při startu služby a hrají roli konzumentů zpráv z fronty. Jakmile fronta není prázdná, zpráva ve frontě je okamžitě zpracována jedním z obslužných vláken. Při větším počtu připojených klientů do systému je díky obslužným vláknům zajištěno paralelní zpracování více příchozích zpráv v jeden časový okamžik. Obslužná vlákna jsou zodpovědná za odebírání a zpracování požadavků z fronty a především uložení samotných příchozích dat do databáze.

[15]PAHO-MQTT je knihovna pro Python, implementující funkce protokolu MQTT. Dostupná je na adrese https://pypi.python.org/pypi/paho-mqtt.

3.4 Testování systému

Za účelem testování navrženého řešení byly vytvořeny dva pomocné skripty, které simulují chování dvou základních komponent celého systému. Jedná se o aplikaci **MQTT Subscribe Agent** a **samotné meteorologické stanice**. Oba skripty byly napsány ve skriptovacím jazyce Python a jsou součástí této práce. Jedná se o skripty, které simulují chování reálných klientů a skriptu pro zpracování příchozích požadavků s daty pro uložení do databáze. Za účelem reálné simulace klientů, tj. vytvoření jejich dostatečně velkého počtu, bylo třeba využít víceprocesových a vícevláknových technik. Konkrétním případem je skriptovací jazyk Python, který pro jeden proces umožňuje vytvořit maximálně 877 vláken (tento údaj se může lišit v závislosti na testovacím stroji). Tento počet může být navýšen právě použitím víceprocesových technik. Při zvětšování tohoto limitu pak dochází k omezení samotného OS, na kterém tento skript běží. Jedná se především o nastavení parametrů jádra. U unixových OS jsou k dispozici techniky, které umožňují tyto parametry měnit a jsou spojovány s pojmem **kernel tuning**. Jedná se o tyto dvě nejzákladnější techniky:

- **Limit otevřených souborů (file handle limits)**: jádro OS má typicky omezený počet souborů, se kterými může každý z běžících procesů pracovat. Navýšením tohoto limitu roste nárok na paměť.
- **Ladění socketů (socket tuning)**: servery obvykle potřebují současně komunikovat s několika klienty a s tímto souvisí právě maximální počet současně otevřených TCP spojení.

mqttTestServer.py Jedná se o odlehčenou verzi plnohodnotné verze MQTT subscribe agent s drobnými modifikacemi pro vyhodnocení parametrů pro testování výkonosti systému. Skript hraje roli odběratele zpráv z daného kontextu na MQTT brokeru a ukládá je do fronty, odkud je dále obslužná vlákna konzumují a zpracovávají. Výstupem skriptu je log soubor, který obsahuje informace o stavu systému (časová razítka odeslání a uložení zprávy, perioda odesílání zpráv, stav fronty), na základě kterých pak byla provedena analýza a vyhodnocení výkonnosti systému.

mqttTestClients.py Skript umožňuje díky víceprocesovému a vícevláknovému návrhu, simulovat velké množství klientů, kteří pravidelně zasílají zprávy do daného kontextu na MQTT broker. Samotný výsledný počet simulovaných klientů je zde omezen právě výše zmíněnými limity OS. Záleží tedy čistě na možnostech stroje, na kterých je skript spuštěn.

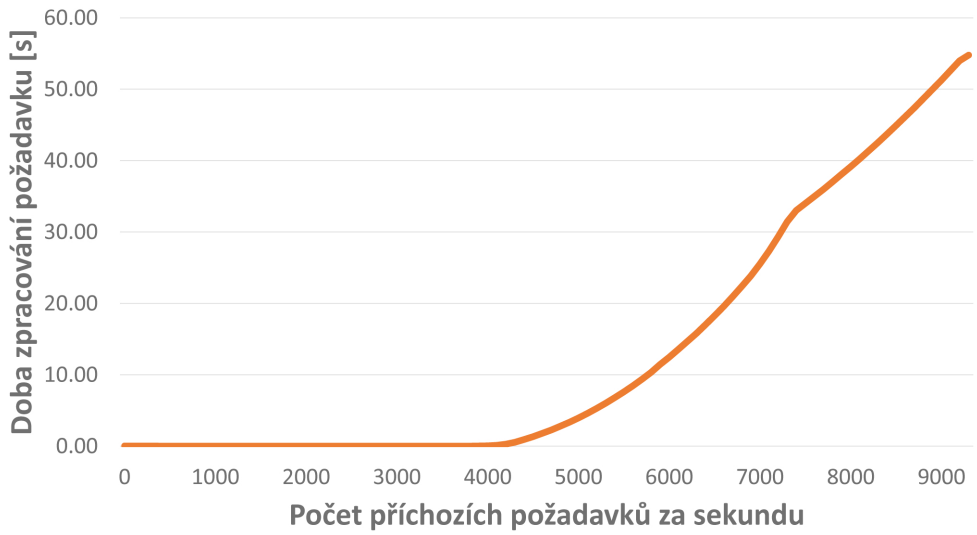
3.4.1 Vyhodnocení výkonnosti systému

Při testování výkonnosti systému byl kladen důraz zejména na **propustnost** systému. Propustnost zde značí schopnost systému zpracovat množství příchozích zpráv za jednotku času. Testovací skript byl nastaven tak, aby simuloval klienty, odesílající data s rychlostí 10 paketů za sekundu. Počet simulovaných klientů se postupně zvyšoval na 1000, čím bylo ve výsledku dosaženo maximálního výkonu odesílání dat 10000 zpráv za sekundu. Odesílané zprávy odebíral z MQTT brokeru skript, simulující MQTT subscribe agent a přijaté zprávy zpracovával. Tímto byla stanovena hranice, při které je systém schopen zpracovávat příchozí požadavky s minimálním zpožděním.

Pro testování jsem na klientské straně simuloval 1000 klientů a na straně odběratele zpráv 1 obslužné vlákno. Při dosažení odesílání 10000 požadavků za sekundu jsem testování ukončil. Gaf na obrázku 3.7 (oranžová křivka) zobrazuje závislost doby zpracování požadavků na počtu příchozích požadavků za sekundu.

Maximální hodnoty 10000 příchozích požadavků za sekundu jsem ve skutečnosti nedosáhl, protože jednotlivá vlákna simulující klienty nebyla vzájemně synchronizována. Toto však není na závadu, neboť významným bodem měření je hodnota okolo 4000 příchozích požadavků za sekundu. Hodnoty počtů odesílaných požadavků za sekundu jsem převzal podle skutečného počtu odeslaných paketů v jednotlivých sekundách měření tak, jak jsem postupně spouštěl jednotlivá vlákna simulující klienty. Celkem bylo vyhodnoceno asi 500 tisíc požadavků (růst požadavků od 10 do 10000 za sekundu po dobu 100 sekund odpovídá celkovému počtu 500 tisícům požadavků).

Graf na obrázku 3.8 (modrá křivka) pak zobrazuje závislost délky fronty na počtu příchozích požadavků za sekundu do systému. Výsledná modrá křivka má dvě hlavní části.



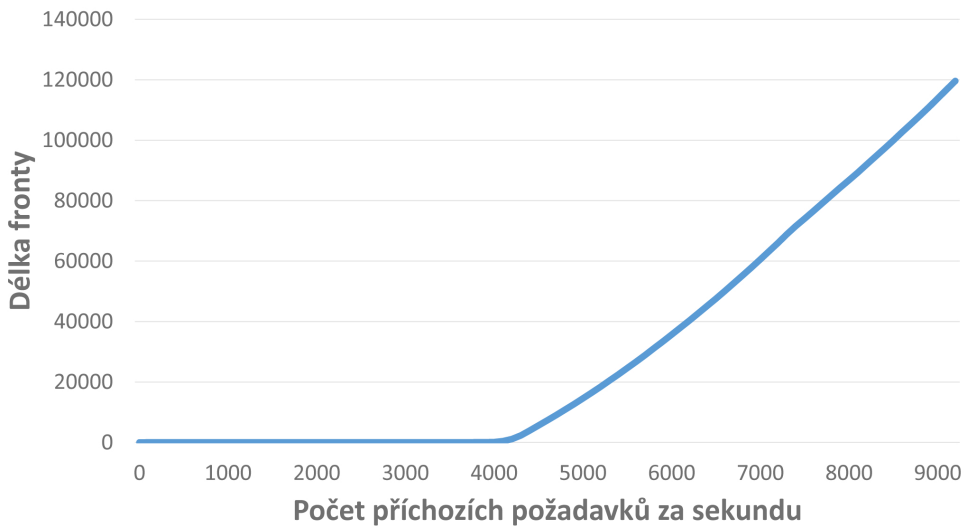
Obrázek 3.7: Graf závislosti doby zpracování na počtu příchozích požadavků do systému.

- **První část**: na intervalu od 0 do 4000 požadavků za sekundu, vyznačuje schopnost systému zpracovat příchozí požadavky za trvalého provozu bez viditelného zpoždení. Obslužné vlákno je tedy schopno včas odebírat a zpracovávat všechny příchozí požadavky z fronty.

- **Druhá část**: bod zlomu, v hodnotě asi 4000 požadavků za sekundu pak vyjadřuje stav, kdy se příchozí požadavky začínají více hromadit ve frontě a systém nestíhá tyto požadavky včas obsluhovat. Interval od 4000 do 9000 požadavků za sekundu pak značí nárůst velikosti fronty při současném odbavování požadavků z vrcholu fronty. V tomto intervalu je i patrný rostoucí trend požadavků ve frontě tak, jak se navyšuje objem odesílaných zpráv. V této fázi je systém schopen zpracovávat krátkodobé nárůsty objemu dat. Pro dlouhodobější zatížení systému je možné za účelem snížení zátěže systému zvýšit počet obslužných vláken.

Na základě výsledků testování lze pak definovat dva možné stavy, ve kterých se může systém pro sběr dat nacházet.

- **Vyvážený stav**: vyvážený stav značí schopnost systému zpracovávat všechny příchozí požadavky s minimálním zpožděním. Z obou uvede-



Obrázek 3.8: Graf závislosti délky fronty na počtu příchozích požadavků do systému.

ných grafů je zřejmé, že se jedná o stav, kdy do systému přichází méně než 4000 požadavků za sekundu.

- **Vytížený stav**: vytížený stav je definován stavem, kdy do systému přichází více než 4000 požadavků za sekundu. V tento moment dochází ke značnému nárustu fronty, kdy obslužné vlákno není schopné dostatečně rychle konzumovat a zpracovávat příchozí požadavky a tím se zároveň zvyšuje doba zpracování jednotlivých požadavků. Všechny požadavky jsou přesto zpracovány, pouze doba jejich zpracování roste.

Zvýšení propustnosti systému (zvýšení počtu požadavků, které je systém schopen za sekundu zpracovat) je možné především zvýšením počtu obslužných vláken na straně odebíratele zpráv, kterým je skript MQTT subscribe agent. Tato skutečnost prakticky zajistí zvýšení rychlosti odebírání a zpracování zpráv z fronty.

Výsledek měření dokazuje, že i takto realizovaný systém, sestavený z běžných HW prostředků s použitím skriptovacích jazyků bez optimalizace pro výkon dokáže v praxi obsluhovat až jednotky tisíc stanic.

Závěr

Cílem práce bylo navrhnout a ověřit řešení, které bude umožňovat sběr dat z meteorologických stanic. Výsledný systém se skládá ze tří hlavních částí. Jedná se o meteorologické stanice s obslužným programem, mezivsrtvu pro výměnu zpráv a serverové prostředí. Pro každou z těchto částí jsem v příslušné kapitole diskutoval různé možností řešení, jejich obecné výhody a nevýhody použití a především také možnosti škálování. Na základě těchto možností jsem realizoval jedno konkrétní modelové řešení, jehož základní funkčnost jsem následně ověřil a otestoval. Vedle toho jsem vytvořil i jednoduchou klientskou webovou aplikaci, která umožňuje zobrazení aktuálních hodnot z několika sestavených meteorologických stanic a nastavení jejich provozních parametrů.

Vytvořené meteorologické stanice je možné díky podpoře DHCP protokolu provozovat v libovolné počítačové síti. Jedinými dvěma nutnými požadavky jsou možnost připojení pomocí standardního síťového konektoru do počítačové sítě a zajištění komunikace na předem definovaný MQTT broker z lokální sítě, ve které je stanice provozována. Veškerá komunikace je pak realizována prostřednictvím MQTT protokolu.

Mezivrstva pro výměnu zpráv tvoří hlavní prvek celého systému, na které jsou závislé jeho ostatní komponenty (meteorologické stanice, serverové prostředí a klientské aplikace). Výhodou zvolené technologie je nejen podpora samotného protokolu MQTT, který zajišťuje spolehlivé a hlavně také obousměrné předávání zpráv, ale také podpora protokolu websockets, díky kterému je možné vytvářet uživatelsky přívětivé webové aplikace.

Součástí práce není řešení zabezpečení. Jedná se zejména o zabezpečení komunikace mezi meteorologickými stanicemi a MQTT brokerem, která z většiny případů probíhá v nezabezpečené síti Internet. Součástí práce jsou i dva možné scénáře řešení zabezpečení, které by bylo vhodné zohlednit před případným nasazením systému v produkčním prostředí.

Výsledkem práce je systém pro sběr dat z různých zdrojů, jehož použití je vhodné především v lokálních sítích. Zejména díky vhodně zvolenému databázovému frameworku je možné sbírat data z různých zařízení (systém

tedy není limitován pouze na sběr dat z meteorologických stanic) a díky clusterové databázové technologii je zároveň možné sbírat velké objemy těchto dat. Datové úložiště je pak díky použití clusterového databázového systému snadno škálovatelné, především horizontálně. KairosDB pak nabízí bohaté RESTful rozhraní, nad kterým je možné snadno vyvíjet nejen webové aplikace, které umožňují práci se sbíranými daty, ale umožňuje také snadné získávání dat pro libovolné sofistikovanější analytické a statistické nástroje pro zpracování dat.

Navržený systém jsem otestoval a jeho funkčnost ověřil pomocí zkušebních skriptů. Na základě výsledků testování jsem stanovil jeho propustnost, při které je systém schopen zpracovávat příchozí požadavky bez viditelného zpoždění.

Literatura

[1] Gantz, J.; Reinsel, D.: The Digital Universe in 2020: Big Data, Bigger Digital Shadows, and Biggest Growth in the Far East. 2012.

[2] Electronic Design: *Understanding The Protocols Behind The Internet Of Things*. [online]. [cit. 2015-12-30]. Dostupné z: `http://electronicdesign.com/iot/understanding-protocols-behind-internet-things`

[3] ComputerWeekly.com: *A history of cloud computing*. [online]. [cit. 2015-12-21]. Dostupné z: `http://www.computerweekly.com/feature/A-history-of-cloud-computing`

[4] MSV, J.: *Demystifying the cloud*. Janakiram & Associates, 2012.

[5] Ken Leyung: *Arduino: A brief history*. [online]. [cit. 2015-11-25]. Dostupné z: `http://www.kenleung.ca/portfolio/arduino-a-brief-history-3`

[6] Readwrite: *How Intel's Edison Stacks Up Against Arduino And Raspberry Pi*. [online]. [cit. 2015-11-25]. Dostupné z: `http://readwrite.com/2014/09/10/intel-edison-raspberry-pi-arduino-comparison`

[7] *Middleware for Communications*. John Wiley & Sons, první vydání, 2004, ISBN 0-470-86206-8.

[8] Aziz, B.: *A formal model and analysis of an IoT protocol*. School of Computing, University of Portsmouth, 2015.

[9] HiveMQ: *HiveMQ Essentials - Quality of Services*. [online]. [cit. 2015-11-16]. Dostupné z: `http://www.hivemq.com/blog/mqtt-essentials-part-6-mqtt-quality-of-service-levels`

[10] *Building Smarter Planet Solutions with MQTT and IBM WebSphere MQ Telemetry*. IBM Redbooks publication, první vydání, 2012, ISBN 978-0738437088.

[11] HiveMQ: *HiveMQ Essentials - MQTT topics*. [online]. [cit. 2015-11-23]. Dostupné z: `http://www.hivemq.com/blog/mqtt-essentials-part-5-mqtt-topics-best-practices`

[12] Sadalage, P. J.; Fowler, M.: *NoSQL distilled: a brief guide to the emerging world of polyglot persistence*. Upper Saddle River, čtvrté vydání, 2014, ISBN 978-0321826626.

[13] Robert Greiner: *CAP Theorem: Revisited*. [online]. [cit. 2015-12-21]. Dostupné z: `http://robertgreiner.com/2014/08/cap-theorem-revisited`

[14] *Large Scale and Big Data*. CRC Press, první vydání, 2014, ISBN 978-1466581517.

[15] DB-Engines: *Wide column stores*. [online]. [cit. 2015-11-30]. Dostupné z: `http://db-engines.com/en/article/Wide+Column+Stores`

[16] Baron Schwartz's blog: *Time Series Databases and InfluxDB*. [online]. [cit. 2015-11-30]. Dostupné z: `http://www.xaprb.com/blog/2014/03/02/time-series-databases-influxdb`

[17] *REST in Practice: Hypermedia and Systems Architecture*. O'reilly, první vydání, 2010, ISBN 978-0596805821.

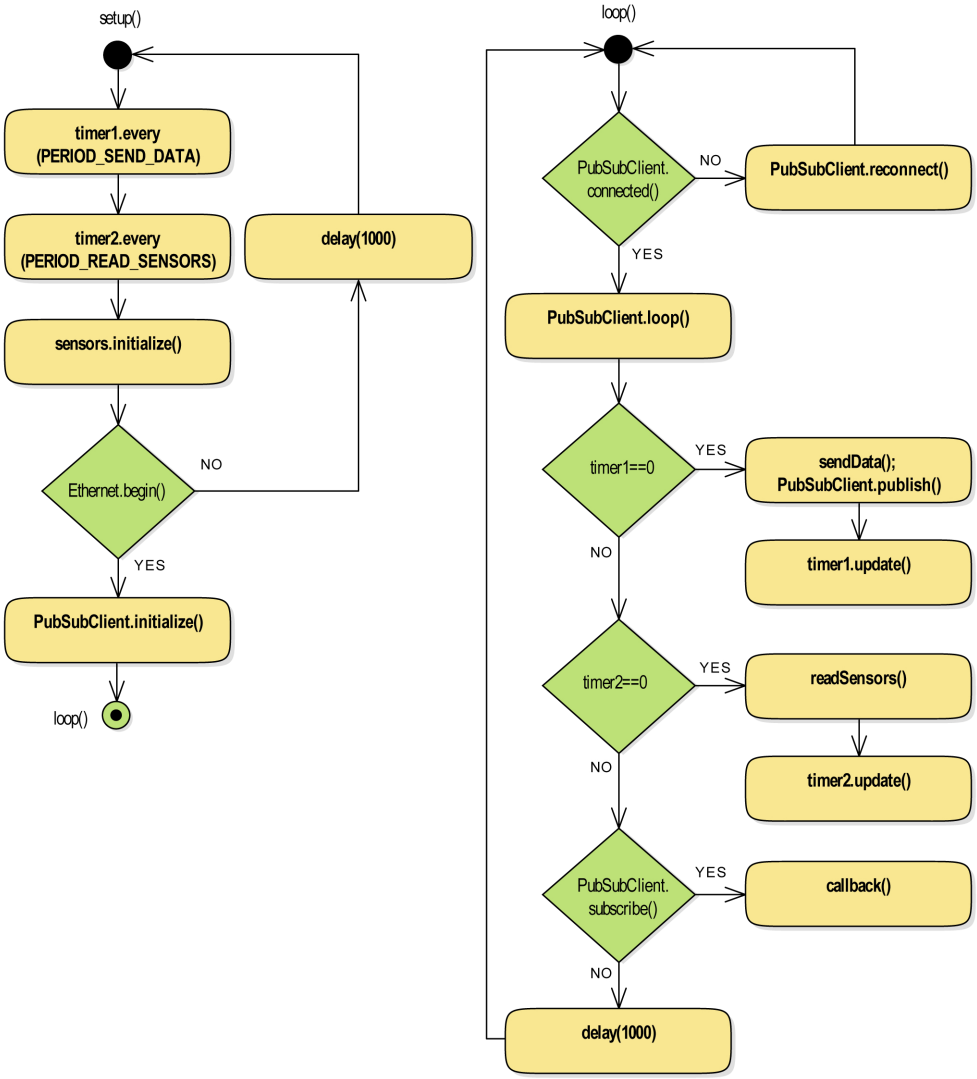
[18] Datastax: *About Client Requests in Cassandra*. [online]. [cit. 2015-11-27]. Dostupné z: `http://docs.datastax.com/en/archived/cassandra/1.0/docs/cluster_architecture/about_client_requests.html`

[19] Datastax: *About Apache Cassandra*. [online]. [cit. 2015-12-01]. Dostupné z: `http://docs.datastax.com/en/cassandra/2.1/cassandra/gettingStartedCassandraIntro.html`

PŘÍLOHA A

Schéma obslužného programu

Hlavní funkcionalitou obslužného programu meteorologické stanice je pravidelné odesílání dat. Knihovny pro práci se senzory nabízejí metody a funkce, pomocí kterých je možné zjistit aktuální hodnoty senzorů. Dle dokumentace knihoven je nutné nastavit určitou časovou prodlevu mezi jednotlivým čtením hodnot. Proto bylo nutné funkce pro čtení hodnot ze senzorů a funkce pro odesílání dat navrhnout tak, aby se prováděly pouze v určitých intervalech. Tato skutečnost byla zajištěna použitím knihovny časového čítače, který dané funkce volá pouze po uplynutí zadaného času.



Obrázek A.1: Vývojový diagram, znázorňující logiku obslužného programu desky Arduino.

PŘÍLOHA B

Příklady HTTP POST požadavků

Příklad struktury těla POST požadavku pro HTTP API KairosDB pro dotaz nad množinou datových bodů pro metriku se jménem **ms.t1**, značkou **msID:1** a časovým razítkem v relativním rozsahu **10 minut** zpětně.

```
{
 "metrics": [
  {
   "tags": {
   "msID": [
    "1"
   ]
  },
  "name": "ms.t1"
  }
 ],
  "cache_time": 0,
  "start_relative": {
  "value": "10",
  "unit": "minutes"
 }
}
```

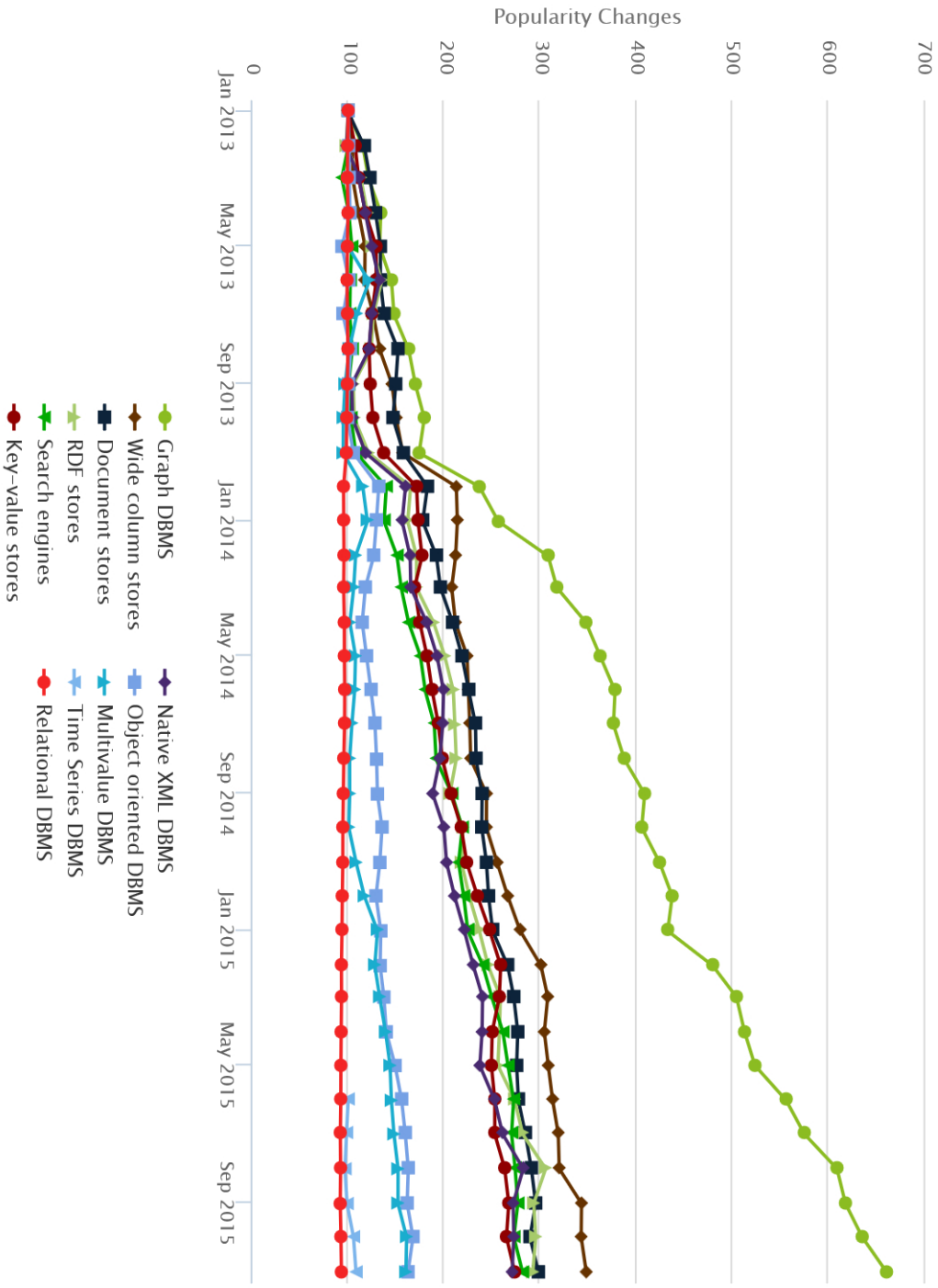
Příklad struktury těla HTTP POST požadavku pro vložení nových datových bodů.

```
[
 {
  "name": "ms.t1",
  "timestamp": 1359786400000,
  "value": 20.54,
  "tags": {
   "msID": 1
  }
 },
 {
  "name": "ms.t2",
  "timestamp": 1359786400000,
  "value": 5.98,
  "tags": {
   "msID": 1
  }
 }
]
```

PŘÍLOHA C

Graf popularity databázových systémů

Graf na obrázku C.1 zobrazuje roustoucí trend jednotlivých databázových systémů. Je zajímavé, že většina těchto systémů je založena právě na koneptu NoSQL.



Obrázek C.1: Graf rostoucího trendu používání databázových systémů (převzato z http://www.db-engines.com).

PŘÍLOHA D

Seznam použitých zkratek

AMQP Advanced Message Queuing Protocol

API Application Programming Interface

AWS Amazon Web Services

DDS Data Distribution Service

GPIO General Purpose Input Output

HTTP Hyper Text Transfer Protocol

HW Hardware

IAAS Infrastructure As A Service

IDE Integrated Development Environment

JSON JavaScript Object Notation

MOM Message Oriented Middleware

MQTT Message Queueing Telemetry Transport

PAAS Platform As A Service

QoS Quality Of Services

REST Representational State Transfer

RDBMS Relational Database Management System

SAAS Softwre As A Service

SPOF Single Point Of Failure

SW Software

TCP Transmission Control Protocol

UDP User Datagram Protocol

XML Extensible Markup Language

XMPP Extensible Messaging and Presence Protocol

YAML YAML Ain't Markup Language

PŘÍLOHA E

Obsah přiloženého CD

- `arduino` ..... složka s podpůrnými soubory pro Arduino
  - `libraries` ..... složka s knihovními soubory
  - `meteostation.ino` ..... zdrojový kód obslužného programu
- `client-application` ..... složka se soubory webové klientské aplikace
- `mqtt-subscribe-agent` .. složka se soubory skriptu MQTT Sub. Agent
- `server-configuration-files` ..... adresář s konfiguračními soubory
  - `mosquitto.conf` ..... konfigurační soubor MQTT brokeru
  - `cassandra.yaml` ..... konfigurační soubor Apache Cassandra
  - `kairosdb.properties` ..... konfigurační soubor KairosDB
- `test-scripts` ..... adresář s testovacími skripty
  - `mqttTestClients.py` ..... skript pro simulaci MQTT klientů
  - `mqttTestServer.py` ..... skript pro simulaci odběratele zpráv
- `text` ..... text práce
  - `DP_Dittrich_Slavomir_2016.pdf` ..... text práce ve formátu PDF
  - `DP_Dittrich_Slavomir_2016` .. zdrojové soubory ve formátu LaTeX
- `readme.txt` ..... stručný popis obsahu CD