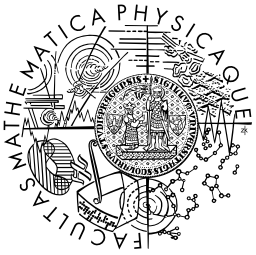


MATEMATICKO-FYZIKÁLNÍ FAKULTA
Univerzita Karlova

BAKALÁŘSKÁ PRÁCE

Milan Lamplot

Návrh frameworku pro edge-cloud continuum

Katedra distribuovaných a spolehlivých systémů

Vedoucí bakalářské práce: doc. RNDr. Petr Hnětynka, Ph.D.

Studijní program: Informatika (B0613A140006)

Praha 2024

Prohlašuji, že jsem tuto bakalářskou práci vypracoval(a) samostatně a výhradně s použitím citovaných pramenů, literatury a dalších odborných zdrojů. Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona v platném znění, zejména skutečnost, že Univerzita Karlova má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle §60 odst. 1 autorského zákona.

V ..... dne ....

Podpis autora

Děkuji svému vedoucímu doc. RNDr. Petru Hnětynkovi, Ph.D. za rady, ochotu a čas, který věnoval této práci.

Název práce: Návrh frameworku pro edge-cloud continuum

Autor: Milan Lamplot

Department: Katedra distribuovaných a spolehlivých systémů

Vedoucí bakalářské práce: doc. RNDr. Petr Hnětynka, Ph.D., Katedra distribuovaných a spolehlivých systémů

Abstrakt: Cílem této práce je navrhnout a implementovat framework pro Edge-Cloud Continuum (ECC) v jazyce Java. V dnešní době je stále více zařízení IoT (Internet of Things) a dat, která tato zařízení generují. Edge-Cloud Continuum řeší tento problém tak, že výpočetní sílu přiblíží ke koncovým zařízením. Zároveň umožňuje přesouvat umístění zpracování dat podle potřeby aplikace. Pokud úloha potřebuje vyšší výkon nebo vyžaduje nějaké specializované funkce, lze ji plynule přesunout na uzel, který tyto požadavky splňuje. Práce se zaměřuje na klíčové aspekty ECC, jako je komunikace s nízkou latencí nebo migrace služeb mezi vrstvami. Výsledkem je funkční framework, který je demonstrovaný na zjednodušeném příkladu aplikace pro řízení dopravy ve městě.

Klíčová slova: edge-cloud continuum, edge computing, cloud computing

Title: Designing a framework for edge-cloud continuum

Author: Milan Lamplot

Department: Department of Distributed and Dependable Systems

Supervisor: doc. RNDr. Petr Hnětynka, Ph.D., Department of Distributed and Dependable Systems

Abstract: The aim of this work is to design and implement a framework for Edge-Cloud Continuum (ECC) in Java. Nowadays, there is an increasing number of Internet of Things (IoT) devices and the data that these devices generate. Edge-Cloud Continuum solves this problem by bringing the computing power closer to the end devices. It also allows to move the location of data processing according to the application's needs. If a job needs higher performance or requires some specialized functionality, it can be seamlessly moved to a node that meets those requirements. This work focuses on key aspects of ECC, such as low-latency communication or migration of services between layers. The result is a functional framework that is demonstrated with a simplified example of an urban traffic management application.

Keywords: edge-cloud continuum, edge computing, cloud computing

Obsah

1 Úvod

V dnešní době, kdy narůstá počet zařízení IoT (Internet-of-Things)[1] a IoE (Internet-of-Everything)[2], je potřeba jejich data nějak zpracovávat. Tradiční postupy, jako je *cloud computing*, kde se data odesílají na centrální servery, se potýkají s řadou výzev, jako je škálovatelnost, latence a omezené kapacity při rostoucím objemu dat[2].

Aby se tyto problémy vyřešily, vznikl koncept *edge computing* [2]. Tento přístup přibližuje výpočetní sílu ke koncovým zařízením, tedy *na okraj* sítě. To přináší řadu výhod, kterými jsou:

- Snížení latence: Zpracování dat blíže u jejich zdroje snižuje zpoždění a umožňuje rychleji reagovat na změny.
- Lepší škálovatelnost: Distribuováním zátěže mezi několik uzlů umožňuje lépe se přizpůsobit rostoucímu počtu zařízení a objemu dat.
- Snížení zátěže datových center: Zpracováním dat na okraji sítě se snižuje množství dat, které je nutné přenést a zpracovat v datových centrech.
- Odolnost při výpadku: Díky distribuci uzlů síť lépe reaguje na výpadek jednoho z nich.

Edge-Cloud Continuum (ECC)[3] je koncept, který dále rozvíjí myšlenku *edge computing* a spojuje ji s *cloud computing*. ECC vytváří architekturu, která kombinuje výhody obou přístupů a umožňuje efektivní zpracování dat v různých prostředích. Úlohy, které vyžadují velký výpočetní výkon, zpracovává na vzdálených serverech s velkým množstvím prostředků. Zároveň úlohy, které potřebují co nejnižší latenci, zpracuje na uzlech poblíž koncových zařízeních.

1.1 Cíle práce

Cílem práce je implementace hlavních prvků Edge-Cloud Continuum v Javě. A to formou frameworku, který umožňuje implementaci a nasazení distribuovaných služeb. Funkčnost se dokáže na jednoduchém demonstračním příkladu.

Hlavními prvky ECC, které budeme implementovat jsou:

- Migrace zpracování služeb – mezitím, co se klient (například chytré auto) pohybuje v reálném světě, tak se místo zpracování dat klienta plynule přesouvá mezi servery.
- Rozhodování o přesunu – přesun musí být automaticky řešený frameworkem bez vnějšího zásahu.

[1] IoT (Internet-of-Things) je síť fyzických objektů, které jsou vybaveny senzory, softwarem a dalšími technologiemi pro účel propojení a výměny dat s jinými zařízeními a systémy přes internet.[1]

[2] IoE (Internet-of-Everything) je širší koncept IoT, který zahrnuje nejen fyzická zařízení, ale také data, procesy a lidi, které jsou propojeny a komunikují prostřednictvím internetu.[1]

- Přenos dat mezi klientem a nějakou službou – framework umožňuje běh distribuovaných služeb, které nějak zpracovávají data, klient s nimi musí umět komunikovat.

1.2 Struktura práce

V druhé sekci je definice Edge-Cloud Continuum – jeho přínosy a rozdíly od ostatních přístupů. Dále popisuje motivační příklad a využití frameworku.

V sekci třetí je popsaná implementace a architektura frameworku v Javě.

Ve čtvrté sekci je demonstrováno využití systému na reálném zjednodušeném příkladu, který používá implementovaný framework.

V poslední sekci je zhodnocen výsledek práce, co bylo dosaženo a čemu by se dalo v budoucnu věnovat.

2 Analýza problému

Tato kapitola popisuje, co přesně znamená *Edge-Cloud Continuum* (dále také jako *ECC*) a porovnává ji s ostatními používanými architekturami.

2.1 Specifikace Edge-Cloud Continuum

První definice Edge-Cloud Continuum se objevily v roce 2017 [4, 5]. Obecně se rozděluje do tří úrovní hierarchie.

Nejvyšší úrovní je *cloud*. Tato úroveň má k dispozici velké množství zdrojů (jak výpočetní sílu, tak i úložiště a síťovou kapacitu). Vyznačuje se tím, že je geograficky vzdálená od koncových zařízení a to obvykle přináší vyšší latenci. Je ideální pro úlohy, které nevyžadují nízkou odezvu, ale potřebují větší množství prostředků. Mezi takové úlohy patří například analýza velkých dat nebo trénování AI modelů, které se učí z dat poskytnutých z *edge* vrstvy, tento koncept distribuovaného učení byl popsán v [6]. V reálném světě by tato úroveň mohla být reprezentována například velkými datovými centry nebo virtuálními stroji.

Další úroveň se označuje *edge* nebo také *fog*. Skládá se z jednotek, které mají menší výkon. Tyto jednotky jsou umístěny poblíž koncových zařízení a jejich použití je nutné pro zajištění nejnižší latence. Ideální úlohou pro tuto vrstvu je analýza dat v reálném čase – příkladem jsou přijímače rozmístěné po městě, které za běhu zpracovávají živý stream z kamer autonomního vozidla. Data ze streamu jsou následně použitá k okamžité analýze prostředí a dopravní situace v okolí vozidla.

Poslední a nejnižší úrovní jsou koncová zařízení. Takové zařízení může být cokoliv, od telefonu po IoT zařízení jako třeba termostat. Důležitým znakem je, že toto je jediný prvek architektury, který poskytuje rozhraní pro komunikaci s vnějším světem, ostatní prvky jsou v architektuře za touto vrstvou skryty. [4].

2.1.1 Přínosy a výhody

K popsání výhod ECC oproti *cloud* a *edge computing* modelům, je třeba si tyto dvě architektury podrobněji vysvětlit. Shrnutí porovnání je v tabulce 2.1.

Cloud Computing

Tento způsob řeší všechny výpočetní problémy a uložení dat v centralizovaných serverech. Počáteční definice a popsání tohoto přístupu bylo v 20. století [7], realizace se uskutečnila až v 21. století, kdy firmy jako IBM, Google, nebo Microsoft začaly spouštět své vlastní *cloudové* služby[8]. V dnešní době je tato architektura často využívaná, příkladem mohou být velká datová centra [9].

Tento přístup se časem vyvíjel a dnes *cloud computing* používá technologie jako distribuovaný výpočet, balancování, paralelní výpočet, síťová úložiště, virtualizace a mnoho dalších.

Mezi hlavní výhody patří vertikální škálovatelnost[1] a zároveň i flexibilita, jelikož

[1] **Vertikální škálovatelnost** se týká schopnosti zvýšit výkon jednotlivé části systému – přidání více výpočetních prostředků.

	Pro situace	Potřebná šířka pásma	Použití v reálném čase	Výpočetní režim
Cloud Computing	Globální	Velká	Slabší	Pro výpočty z velkého počtu dat
Edge Computing	Lokální	Malá	Dobré	Pro analýzu dat z méně vzorků
Edge-Cloud Continuum	Globální	Malá	Dobré	Zpracování menších vzorků dat, tak velké výpočty v cloudu

Tabulka 2.1 Hlavní rozdíly mezi Edge, Cloud computing a ECC

aplikace a služby nevyžadují specifickou implementaci, aby využily možnosti této architektury.

Edge computing

Edge computing je rozšířením *cloud computing*, oba přístupy mají v dnešním světě své místo a vzájemně se doplňují. *Edge computing* se využívá tam, kde je kritická rychlost. Namísto centrálního zpracování se data zpracují na okraji sítě[10]. Příkladem je analýza dat z nějaké lokální služby nebo jednoduchých senzorů (teploměr, senzor pohybu, kamery, ...). Díky zpracování menšího vzorku dat tímto způsobem se sníží zátěž na datová centra.

Další výhodou *edge computing* je lepší zabezpečení, jelikož data nejsou zpracovávaná na centrálním serveru a jsou uložená pouze na lokálním uzlu.

Edge-Cloud Continuum

Edge-Cloud Continuum dále rozvíjí a spojuje myšlenky *edge* a *cloud computing*. Opět snižuje latenci tím, že výpočetní sílu přibližuje ke koncovým zařízením a zároveň umožňuje v nutných případech použít výkon cloudového serveru.

Výhodou je kombinace vlastností předchozích dvou architektur – nízká latence s vysokou škálovatelností (jak vertikální, tak horizontální[2]). Další vlastností je vysoká flexibilita, tedy schopnost se dynamicky přizpůsobit různým úlohám a zatížením.

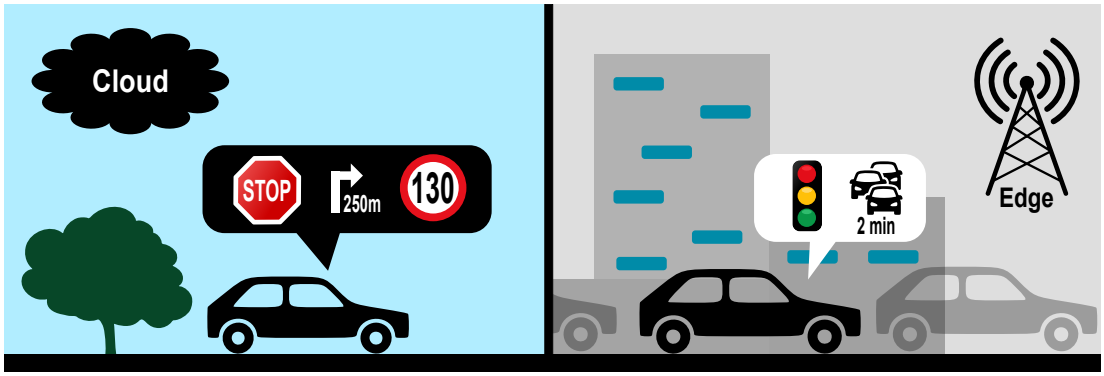
2.1.2 Liquid Software

Kvůli architektuře ECC je nutné, aby na všech úrovních mohly běžet stejné softwarové komponenty [4]. Pojem *liquid software* označuje software, který se může snadno a plynule přesouvat mezi různorodými zařízeními a aplikacemi [11, 12].

V ECC je tímto myšlen software, který může plynule migrovat mezi uzly stejného typu.

Zároveň se ale také dá uvažovat o migraci mezi zařízeními různého typu. Software, který zvládne oba druhy této migrace, se nazývá *fully liquid software*.

[2]**Horizontální škálovatelnost** označuje schopnost systému zvládat zvýšené zatížení rozšiřováním počtu jeho částí (instancí).



Obrázek 2.1 Vizualizace motivačního příkladu. Vlevo je auto na méně rušné silnici. Vpravo ve městě.

Implementace *fully liquid software* zatím nebyla provedena. Je totiž nutné, aby migrace z jednoho zařízení na druhé proběhla zcela plynule a aktivní aplikace nebyla přesunem přerušena. Zároveň musí software běžet na zařízeních různého typu (jiný hardware, různé operační systémy, …).

Momentálně je potřeba, aby každá komponenta byla na jednotlivých typech zařízeních implementována od základu s jeho podporou, což komplikuje vývoj *fully liquid software*, jeho správu a nasazení [4].

2.2 Analýza stávajícího stavu

Tato kapitola rozebírá různé use-case a k čemu se dá ECC využít. Pro analýzu je uveden motivační příklad využití architektury.

Představí-li se situace, kde nejsou žádné dopravní značky a všechny informace o aktuálních pravidlech, rychlostních omezeních a semaforech jsou poskytovány pouze virtuálně, všechna vozidla budou napojena do sítě, která jim tyto informace zasílá. Zároveň vozidla sbírají za pomoci různých senzorů data ze svého okolí a ta musí být zpracovaná co nejdříve, aby vozidlo mohlo učinit situační rozhodnutí. Na tento příklad lze použít architekturu ECC, která splňuje tyto požadavky.

Je potřeba mít co nejnižší latenci při zpracovávání dat, to se zajistí přiblížením výpočetních uzlů ke koncovým zařízením – v tomto případě je koncové zařízení autonomní vozidlo.

Zároveň se zpracování dat může přizpůsobit aktuální situaci. Pojede-li auto po nějaké méně rušné silnici, stačí použít vzdálený uzel s vyšší latencí nebo si data stáhnout a zpracovat vše lokálně. Vjede-li do velkoměsta, kde se toho na silnici a okolí děje mnohem více a data je pak zapotřebí zpracovávat rychleji, může analýzu přesunout na nejbližší výpočetní jednotku. Tento přístup je znázorněn na obrázku 2.1.

2.2.1 Uživatelé

Po analýze naší služby lze rozpoznat tři hlavní aktéry:

- **Řidič:** Uživatel aplikace v autonomním vozidle. Očekává, že mu služba poskytne aktuální a přesné dopravní informace za všech okolností.

- **Vývojář aplikace:** Je odpovědný za vývoj a údržbu služby. Využívá framework k její implementaci a nasazení.
- **Administrátor:** Dohlíží na servery a síť. Monitoruje aktuální stav služeb.

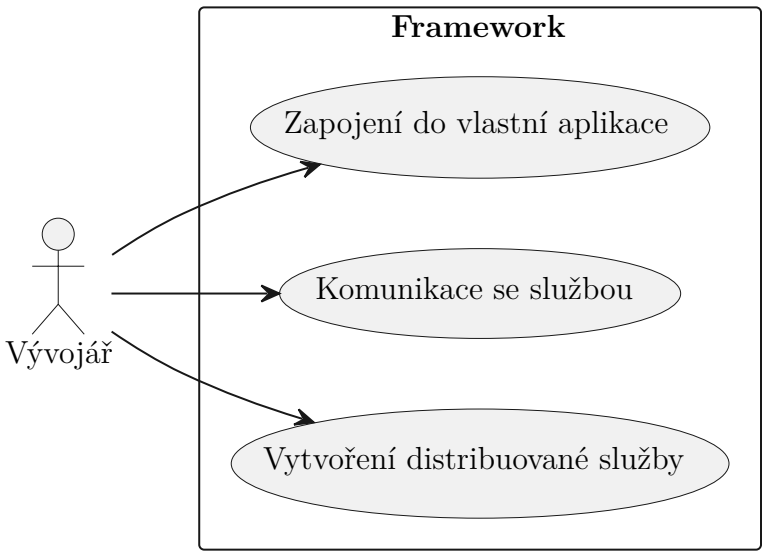
2.2.2 Use-case

Vývojář

Vývojář požaduje od frameworku nástroj pro vytvoření a nasazení služby, která zpracovává data pro výslednou aplikaci. Vyžaduje jednoduchou možnost komunikace se službou ze strany klienta.

Výčet požadavků (viz obrázek 2.2):

- Nástroj k vytvoření distribuované služby: Framework musí umožňovat vytvořit distribuovanou službu, kterou lze následně nasadit na servery architektury
- Rozhraní pro komunikaci klienta se službou: Framework musí umožňovat komunikaci mezi klientem
- Zapojení logiky do aplikace



Obrázek 2.2 Use case diagram pro vývojáře

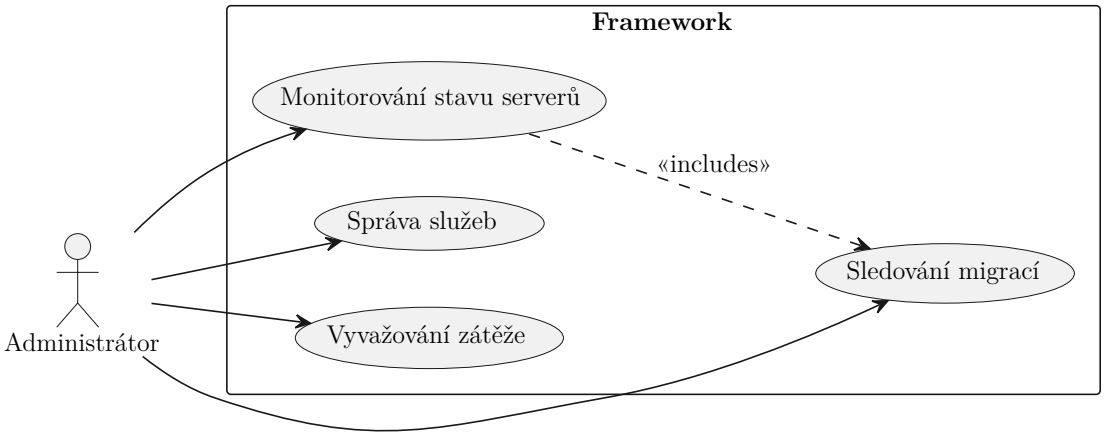
Administrátor

Administrátor systému požaduje možnost spravovat a monitorovat běžící služby a servery v rámci ECC. Framework by měl poskytnou nástroje pro sledování výkonu a umožnit přidávat nebo odebírat aktivní servery.

Výčet požadavků (viz obrázek 2.3):

- Monitorování stavu serverů: Získání informací o vytížení CPU, paměti, síťovém provozu, popřípadě další relevantní metriky jednotlivých serverů
- Správa služeb: Možnost vzdáleného spouštění, zastavování a restartování služeb na jednotlivých serverech

- Sledování migrací: Záznamy o provedených migracích služeb mezi servery, včetně důvodu migrace a jejím výsledku (zda migrace proběhla úspěšně, či pokud selhala nebo server odmítl migraci uskutečnit)
- Vyvažování zátěže: Umožnit přesunutí zátěže/výpočtu na jiný server



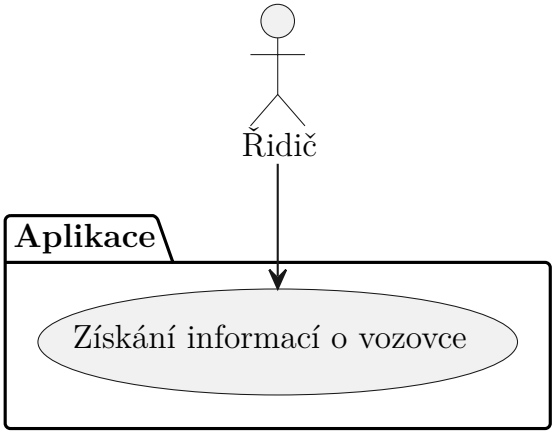
Obrázek 2.3 Use case diagram pro administrátora

Řidič

Pro řidiče je hlavní nerušený provoz aplikace. Ze systému přebírá informace o aktuálním dění na vozovce.

Výčet požadavků (viz obrázek 2.4):

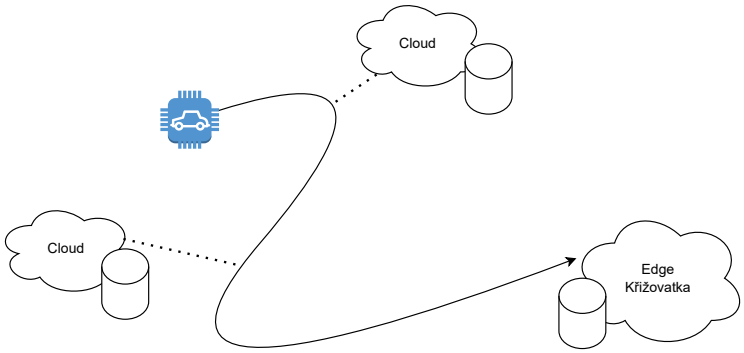
- Zobrazení informací o aktuálním stavu vozovky: Na HUD se zobrazuje aktuální stav vozovky a situace kolem



Obrázek 2.4 Use case diagram pro řidiče

2.3 Implementace ECC v Javě

Implementace v Javě přinesla svoje vlastní výzvy a spoustu možností jak architekturu implementovat. Tato část popisuje, jak lze implementaci vytvořit a co je potřeba vyřešit.



Obrázek 2.5 Vizualizace auta a přepínání mezi servery během trasy

2.3.1 Rozdělení komponent

Z požadavků je vidět, že bude potřeba nějaký server, který zpracovává požadavky, a na kterém běží samotná služba řízení dopravy. Dále je potřeba klientská část, přes kterou se budou odesílat požadavky na servery.

ECC používá více-vrstvou architekturu, ale v rámci implementace je účel *cloud* a *edge* vrstvy poměrně stejný, jsou to výpočetní uzly, liší se hlavně tím, kolik prostředků mají k dispozici. Ve smyslu implementace budeme zamýšlet *cloud* a *edge* jako uzel na stejné úrovni.

Zároveň každý server musí mít vlastní úložiště, kam si bude ukládat data. Data z těchto úložišť bude potřeba při migraci úlohy nějak sdílet, aby mohla úloha pokračovat nepřerušovaně dál na jiném serveru.

Vizualizace průběhu trasy je znázorněna na obrázku 2.5. Je zde znázorněno auto, které po dobu své trasy přepíná mezi vzdálenými cloud servery do té doby, než se dostane ke křižovatce, kde se nachází server v bezprostřední blízkosti vozidla.

Zároveň bude potřeba zjistit, které servery jsou aktivní. Buďto bude mít klient napevno nastavené adresy serverů, které může využít, nebo lze vytvořit centrální rejstřík aktuálně aktivních serverů. Jelikož je potřeba, aby vše fungovalo automaticky, je rejstřík serverů lepší volba. Pro plnou automatizaci je rejstřík serverů lepší volbou, klientovi stačí mít pouze jednu nastavenou adresu a vše ostatní se postupně bude stahovat.

2.3.2 Výzvy

Je několik problémů, které je třeba pro implementaci vyřešit. První z nich je, jak vyřešit distribuci balíčků na server.

Distribuce a inicializace služeb

Je potřeba navrhnout způsob, jak distribuovat a inicializovat služby na serveru při prvotním nasazení. Služby se nainstalují jednou a pak zůstanou na serveru. Šel by navrhnout způsob, jak služby instalovat vzdáleně, například přenesením balíčků .JAR.

Pro naší architekturu ale bude stačit, aby byly balíčky připravené na serveru a následovně se při startu serveru všechny načetly. Všechny služby by měly dědit nebo alespoň implementovat nějaké společné rozhraní.

Komunikace mezi klienty a službou

Existuje několik způsobů, jak lze realizovat přenos dat mezi klientem a službou. Níže jsou uvedeny tři zkoumané přístupy.

První varianta je použití Java Serializable. Tento přístup využívá vestavěnou podporu pro serializaci objektů v Javě. Výhodou je poměrně snadná implementace s využitím existujících mechanizmů. Nevýhodou je potřeba přesně definovat typ, který se odesílá a ten musí souhlasit s typem při deserializaci. To přináší problémy v distribuovaném prostředí, kdy server nemusí znát přesnou implementaci a vše nemusí být v classpath.

Další variantou je použití textové serializace například pomocí knihovny Gson. Ten používá formát JSON (JavaScript Object Notation). Výhodou je poměrně malý overhead při přenosu dat. Nevýhodou je, že pro deserializaci je opět potřeba znát přesný typ přijímaného objektu. Kvůli tomu by deserializaci musela dělat samotná služba, to nás vede k poslední variantě.

Univerzální řešení je nechat komunikační protokol na službě. Každá služba by mohla tak použít co vyhovuje jí a nebyla by tak závislá na knihovně. Jediné co musí framework vyřešit, je odesílat data s co nejnižší latencí.

Sdílená databáze

Služba si musí někam ukládat informace, zvlášť když bude docházet k migraci služby z jednoho serveru na druhý. Služba si musí umět načíst aktuálně zpracovávané informace. Přesněji řečeno, po migraci musí data z počátečního serveru doputovat v co nejkratší dobu na server konečný.

Jsou tu 2 relevantní řešení – aktuální data se budou neustále přesouvat mezi servery s nějakým nastavením granularity. Nebo mít nějakou externí službu, která to bude řešit za programátora – distribuovanou databázi, která by automaticky přesouvala data podle toho, kde jsou právě potřeba.

První možností je neustále přesouvat ručně data z jednoho serveru na druhý, to by bylo zajištěno nějakým systémem škály - stačily by 3 stupně:

1. **Nemigrovat**
   (a) Tento stav bude aktivní po většinu času. Znamená to, že se nemají odesílat žádná data na jiný server.
2. **Blížení migrace**
   (a) Blížení migrace je stav, kdy se předpokládá, že se bude výpočet přesouvat na jiné servery. Nemusí být přesně určeno, na který server se budou data přesouvat, ale měly by se už postupně odesílat na ty nejbližší servery.
3. **Migrovat**
   (a) V tuto chvíli už je rozhodnuto na který server se bude služba migrovat a je tedy nutné, aby data dorazila co nejdříve.
   (b) Služba je migrována nezávisle na datech, ty se mohou přesunout později. Ideální situací by bylo zároveň se službou.

Tento přístup je pro ECC vhodnějším řešením. Problematické je, že služba musí být na toto připravena a vědět, že pokud něco uloží a bude migrovat na jiný server, všechna data jí po přesunu nemusí být k dispozici.

Dalším přístupem je použití již existující služby. Mohla by se využít distribuovaná databáze, například MongoDB Sharding[13]. *Sharding* je metoda horizontálního škálování databází. Data jsou rozdělená napříč několika servery, jeden server se nazývá *shard*. Každý tento server obsahuje podmnožimu celkových dat. Tím se zátěž čtení a zápisu dat rozloží na více strojů, díky čemuž se zvýší výkon i kapacita databáze.

V kontextu MongoDB, *sharding* funguje tak, že rozdělí data na základě *shard key* pro každý dokument[3]. K databázi se přistupuje přes *MongoDB router*, který dotazy směruje na správný *shard* na základě hodnoty *shard key*. Takto funguje distribuce operací v MongoDB.

Ve frameworku by tedy distribuci dat mohla provádět databáze MongoDB. Poblíž každého serveru by byl umístěn *shard* pro snížení latence k přístupu datům. Jakmile by proběhla migrace, data by se přesunula z jednoho shardu na druhý.

Pro výběr mezi těmito možnostmi se na to lze podívat z programátorského hlediska. Buď je možnost naprogramovat vlastní mechanismus pro migraci s postupným přesouváním dat, nebo lze využít již existujícího způsobu, který bude pravděpodobně efektivnější se všemi operacemi. Zároveň bude pro programátora jednodušší použít knihovnu, co by s databází komunikovala.

Výběr způsobu ukládání dat tedy připadl na externí databázi.

Migrace služeb

Zde je popis migrace z jednoho serveru na druhý. Prvně je třeba definovat několik termínů:

- **Úloha** – Jednotka práce, která musí být vykonána. Například zpracování 1 snímku z nahraného videa nebo rozhodnutí, zda auto má nebo nemá zastavit.
- **Uzel** – V kontextu této kapitoly tím bude myšleno zařízení, které může zpracovávat úlohy.
- **Služba** – Proces, který zpracovává na uzlu své dané úlohy.
- **Klient** – Koncové zařízení využívající framework ke komunikaci se službou.

Migrace je rozdělena do několika kroků:

1. **Identifikace nutnosti migrace**
   (a) Tento krok zahrnuje určení nutnosti migrace úlohy z původního uzlu do nového, to může být z různých důvodů:
      i. Původní uzel je přetížený a nedokáže úlohu efektivně zpracovat

[3]Dokument je v MongoDB základní jednotkou dat, jedná se o pár *klíč:hodnota*. Kde hodnotou mohou být různé datové typy, například čísla, řetězce, pole nebo vnořené dokumenty.

ii. Je k dispozici vhodnější uzel, například je blíže nebo naše úloha vyžaduje speciální požadavky (třeba zpracování videa pomocí umělé inteligence).

(b) Pomocí různých aspektů se vyhodnotí, který další uzel je pro úlohu nejvhodnější a započne migrace.

2. **Pozastavení úlohy**

   (a) Služba by měla dostat pokyn, že bude probíhat migrace a tím by měla co nejrychleji ukončit svoji aktuální činnost. Pokud to daná úloha umožňuje, uloží si mezivýsledky a pozastaví výpočet.

3. **Uložení aktuální stavu úlohy**

   (a) Stav úlohy uložen do distribuovaného úložiště.

4. **Přesunutí úlohy**

   (a) Cílový uzel se kontaktuje s minimálním počtem informací o požadavku k migraci.

   (b) V tuto chvíli je již aktuální stav uložen v distribuované databázi a měl by být k dispozici na cílovém uzlu.

5. **Pokračování v úloze**

   (a) Nová úloha by měla být vytvořena na cílovém uzlu a měla by pokračovat nepřerušeně.

   (b) Připojení klienta (zařízení) je přesunuto na nový server.

Klient by o tomto procesu neměl vědět, celý by měl proběhnout bez interakce s klientem. Framework by měl celý proces před klientem skrýt.

2.3.3 Použité technologie a knihovny

Klíčovým bodem rozhodování je výběr komunikačního protokolu mezi klientem a servery. Je nutné, aby pokryl základní požadavek ECC, tím je minimální latence při komunikaci s klientem.

Tato sekce rozebírá zvažované technologie a důvody výběru gRPC pro komunikaci. Zvažované technologie jsou Java RMI, REST API a gRPC. Hlavními cíli je rychlost požadavků a univerzálnost implementace.

Java RMI (Remote Method Invocation)

Java RMI je technologie, která umožňuje volání metod na objektech umístěných ve vzdálených JVM (Java Virtual Machine). Tato technologie je součástí standardní Java knihovny a nevyžaduje tedy žádné externí závislosti. RMI umožňuje vytvořit distribuované aplikace, kde klient může volat metody vzdálených objektů, jako by byly lokální.

Mezi hlavní výhody patří dostupnost ve standardní knihovně a objektově orientovaná komunikace. Lze jednoduše posílat komplexní objekty, což může urychlit vývoj.

Nevýhodou je naopak závislost na jazyce Java. Kvůli tomu by byl klient nucen tento jazyk také používat a to snižuje flexibilitu pro programátory. Další nevýhodou může být nižší výkon oproti jiným, modernějším technologiím.

Vzhledem k výše uvedeným nevýhodám je tato technologie pro projekt nevhodná.

REST API

REST (Representational State Transfer) je softwarová architektura pro návrh síťových aplikací. Základní princip spočívá v tom, že komunikace je realizována pomocí standardních HTTP metod (GET, POST, PUT, DELETE, ...).

Pro implementaci REST API lze využít framework *Spring Boot*, který nabízí jednoduchou cestu k vytvoření *RESTful* služeb v Javě.

Největší výhodou tohoto řešení je rozšiřitelnost a na rozdíl od Java RMI je univerzální - klient není vázaný na jazyk Java, může být napsaný v jakémkoliv jiném jazyce. V dnešní době jsou REST API poměrně rozšířená, díky tomu je s nimi spoustu vývojářů již obeznámeno a existuje velké množství nástrojů a knihoven pro jejich podporu.

gRPC

Poslední zvažovanou technologií je gRPC (gRPC Remote Procedure Calls). Opět nelze využít standardní knihovnu, ale na druhou stranu gRPC nabízí komplexní framework pro mnoho programovacích jazyků, včetně jazyku Java.

Hlavní výhodou gRPC je využití *Protocol Buffers* (protobufs), což je mechanismus serializace dat. Komunikační rozhraní je definované v *.proto* souborech, které se následně kompilují do kódu ve zvoleném jazyce. Lze tedy napsat serverovou část v Javě a klientskou část v jakémkoliv dalším podporovaném jazyce.

gRPC navíc podporuje streamování, což umožňuje odesílání a přijímání proudů dat a obousměrnou komunikaci v reálném čase. To je užitečné například pro přenos obrázků ve vysokém rozlišení nebo streamování videa.

Protože se dopředu neví, jak objemná budou přenášená data, je tato vlastnost gRPC pro projekt velkou výhodou. Právě z tohoto důvodu je pro projekt zvolena technologie gRPC.

3 Architektura a implementace frameworku

Tato kapitola rozebírá architekturu frameworku a jeho implementaci v jazyce Java.

3.1 Základní principy

Framework spojuje několik hlavních částí, první z nich je server, ten obstarává běh služeb, které byly přidané do *ECC* a umožňuje s nimi komunikovat. Další částí je registr serverů, to je rejstřík všech aktivních a dostupných serverů. Poslední částí je klientská část, přes kterou lze vyvíjet aplikace používající *ECC*.

Na obrázku 3.1 je vidět znázornění architektury. Jedná se o její zjednodušenou verzi a jak vypadá po nasazení. Je zde centrální registr, klienti (koncové zařízení) a servery. Servery se zaregistrují do registru. Klienti si získají seznam těchto serverů a navážou s nimi spojení. Nakonec přes servery komunikují s nainstalovanými službami.

Služba

Služba je distribuovatelný kód, který se instaluje na servery.

Pro službu je k dispozici sdílená databáze, kam si může ukládat informace, které budou následně replikovány na ostatní servery, kde jiné instance této služby mohou k těmto informacím přistupovat.

Server

Server je nejdůležitější částí architektury. V rámci ECC jsou tím zamýšleny všechny tři hlavní výpočetní uzly (*edge-fog-cloud*). Je prostředím pro běh služeb.

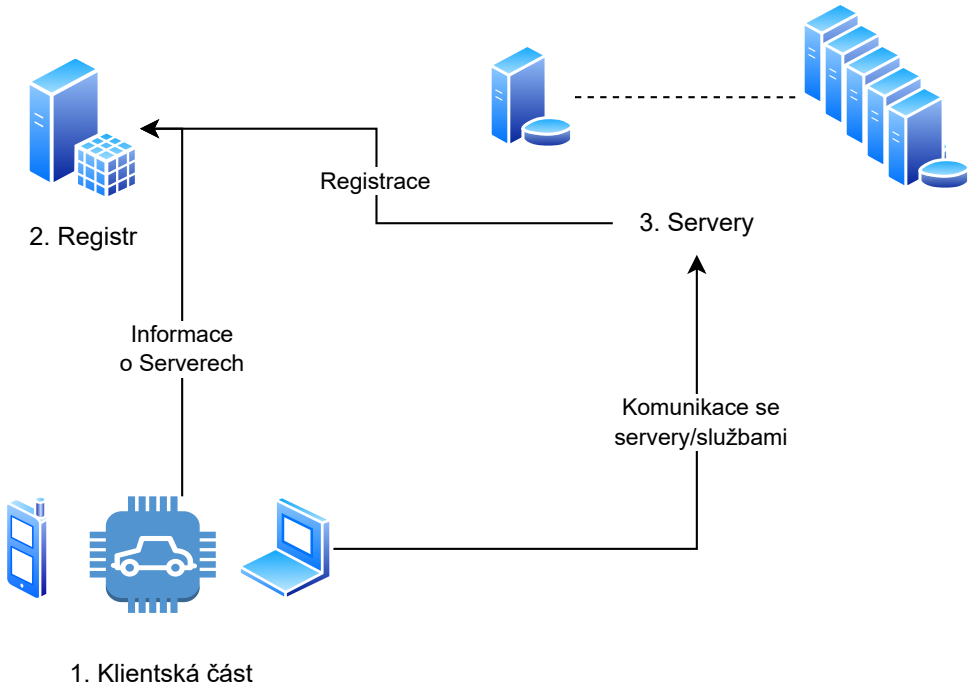
Se serverem lze komunikovat přes veřejné gRPC API, jak bylo rozhodnuto v sekci 2.3.3. Toto API obsahuje jen několik málo koncových bodů, mezi nimiž je zahrnuta možnost migrace klienta na server a ze serveru. Žádost o migraci nezávisí jen na klientovi, ale také na aktuálním stavu serveru, jeho zatížení a specifických požadavcích samotné služby, která v některých případech může migraci odmítnout.

V poslední řadě je má rozhraní pro komunikaci mezi klientem a službou. Samotnou specifikaci komunikace si určuje klient a služba, server nemusí vědět co si posílají.

Registr serverů

Registr je seznam serverů, které jsou aktuálně zapojené do architektury. Každý server se musí zaregistrovat, aby ho zbytek frameworku mohl použít. Knihovna si z registru vždy bere soupis aktuálních serverů, na které se následně připojí.

Nutností každého klienta je si ověřit, zda server vyhovuje jeho požadavkům - na serveru může běžet jiná služba než klient ve skutečnosti vyžaduje. Server je navržen tak, aby při požadavku na migraci pro službu, kterou u sebe nemá zaregistrovanou, odmítl.



Obrázek 3.1 Nasazená architektura frameworku

Klientská knihovna

Rozhraní knihovny umožňuje komunikaci s ostatními částmi frameworku. Základní myšlenkou je, že si uživatel (v tomto kontextu je uživatelem myšlen programátor aplikace využívající framework) vytvoří spojení pro každou službu, než aby vytvářel spojení se specifickým serverem. Ve většině případů se bude komunikovat pouze s jednou službou v rámci celé aplikace.

Jak bylo uvedeno v kapitole 2.3.2, protokol komunikace se službou není definován a její způsob záleží na implementaci samotné služby. Knihovna slouží pouze jako brána pro komunikaci.

Další věcí, co knihovna řeší, je migrace služeb na jiný server. Přesněji řečeno, migrace neprobíhá jen v rámci knihovny, ale celého frameworku. Tento proces je před klientem skrytý a probíhá automaticky.

3.2 Implementace v Javě

Tato kapitola popisuje hlavní části implementace.

3.2.1 Systém služeb

Služby jsou jedním ze základních principů frameworku. Jak už bylo popsáno v kapitole 2.3.2, služby jsou distribuovány přes *.JAR* balíčky a server si je při spuštění načítá. Přesně takto funguje implementace – server při spuštění prohledá nakonfigurovanou složku se službami a pokusí se je postupně načíst. Pro tento

úkol byl použit vestavěný *Class* a *Service* loader.

Všechny služby zároveň musí implementovat stejné rozhraní. Pokud ho neimplementují nebo nedědí od pomocné třídy, nejsou použity.

3.2.2 Sdílené úložiště služeb

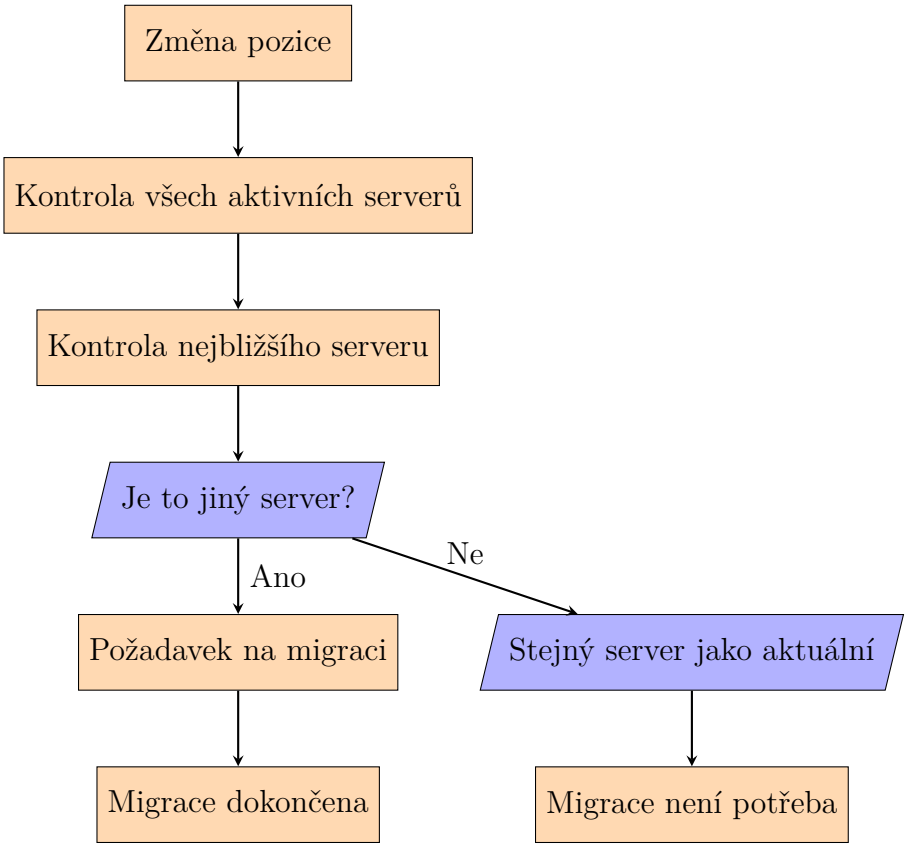
Jak se rozhodlo v kapitole 2.3.2, každá služba bude mít přístup k distribuované databázi. Pro jednoduchost je sdílená databáze implementovaná vestavěnou hašovací tabulkou, ke které mají přístup ostatní servery.

3.2.3 Rozhodování migrace

Rozhodnutí, kdy a na jaký server migrovat probíhá pokaždé, kdy se změní poloha aktuálního klienta.

Jakmile se poloha klienta změní, zkontroluje se, který server se nachází nejblíž. Nadále se daný server zkontaktuje a zjistí se, zda souhlasí s migrací. Aby server souhlasil s migrací, musí být splněny 2 podmínky. První podmínkou serveru je mít nainstalovanou službu, ke které chce klient migrovat – migraci jinak nelze uskutečnit. Druhou podmínkou je, že služba, ke které se migruje, požadavek neodmítne. Taková situace nastane například v případě, že službě dochází prostředky a nemůže zpracovat další klienty.

Migrace tedy může být provedena, souhlasí-li všechny strany. Tento proces je znázorněn na obrázku 3.2.



Obrázek 3.2 Flowchart pro rozhodování výběru serveru

4 Použití frameworku

Tato část popisuje implementaci motivačního příkladu a ukazuje funkčnost frameworku.

4.1 Implementace motivačního příkladu

Implementace je zjednodušená verze příkladu, která byla uvedena v kapitole 2.2. Cílem je splnění bodů definovaných v kapitole 1.1.

Aplikace slouží k ovládání semaforů na křižovatkách ve městě. Řidiči dostávají informace přes HUD (Head-Up Display[1]) v autě, který jim prezentuje informace o nadcházející křižovatce, například za jak dlouho se na semaforu objeví zelená.

Zatímco se vozidlo nachází mimo město, jsou data zpracovávány v cloudu nebo lokálně – podle potřeby. Jakmile vjede do města a potřebuje rychlejší zpracování dat, použije se síť uzlů ve městě. U křižovatky, kde se informace zpracovávají ze stovek až tisícovek vozidel najednou, se tato data využijí pro směrování dopravy po celém městě.

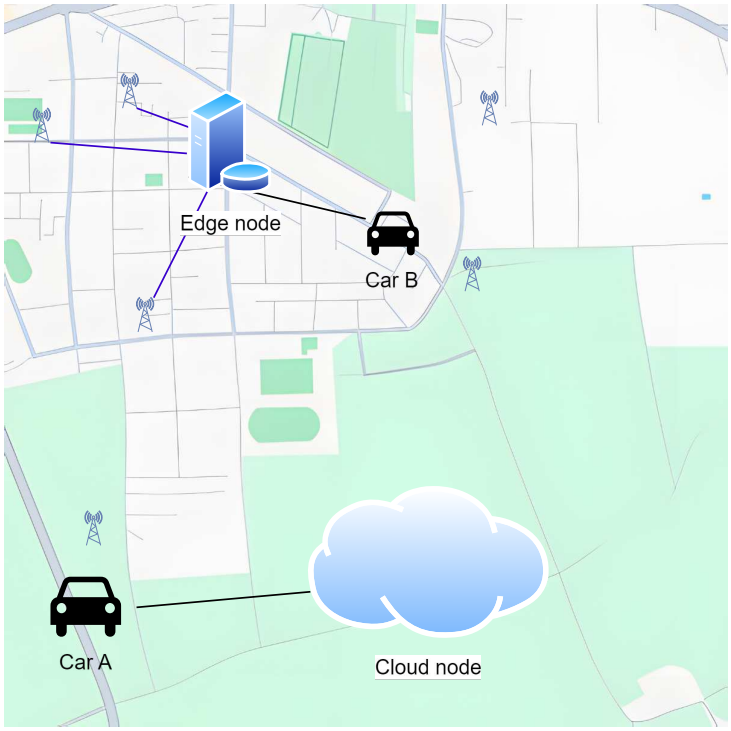
4.1.1 Tok dat

Z pohledu auta (koncové zařízení):

1. Auto není ve městě:
   (a) Data jsou zpracována v nejvyšší úrovni - *cloud*.
   (b) Pokud auto není připojené k síti, data jsou zpracovávána lokálně.
2. Auto vjede do města.
3. Auto se blíží ke křižovatce:
   (a) Výpočet migruje k nejbližšímu *edge* zařízení, preferovaně k tomu, které je na křižovatce.
4. Data ze všech aut v okolí se zpracovávají na křižovatce.
5. Zpracovaná data se odesílají zpět na zařízení:
   (a) HUD v autě ukazuje aktuální stav semaforu.
6. Auto projede křižovatkou.

Z pohledu uzlu na křižovatce jsou nová data ze senzorů aut neustále zpracovávány. Kontinuálně se zpracované výsledky odesílají zpět do aut.

[1] **HUD (Head-Up Display)**: Průhledový displej promítající informace na čelní sklo, aby řidič nemusel spouštět oči ze silnice.



Obrázek 4.1 **Auto A**: Není ve městě, data zpracovává lokálně, popřípadě využije Cloud. **Auto B**: Jede ve městě a blíží se ke křižovatce, data jsou odesílaná a zpracovávaná na uzlu u křižovatky

4.1.2 Demonstrace prvků ECC na tomto příkladu

1. Služba běžící na více úrovních. Data jsou generována ve vozidle, kde mohou být také zpracována. Jakmile je potřeba, výpočet se přesune na nejbližší uzel.

2. Více úrovní. Koncové zařízení je vozidlo, dále jsou tu *edge/fog* uzly, které se nacházejí ve městě. Na nejvyšší úrovni je *cloud*.

3. Nutnost nízké latence. Aby měl řidič aktuální přehled, zpracovaná data musí být dodána co nejdříve, jinak se nezajistí požadavek plynulé dopravy.

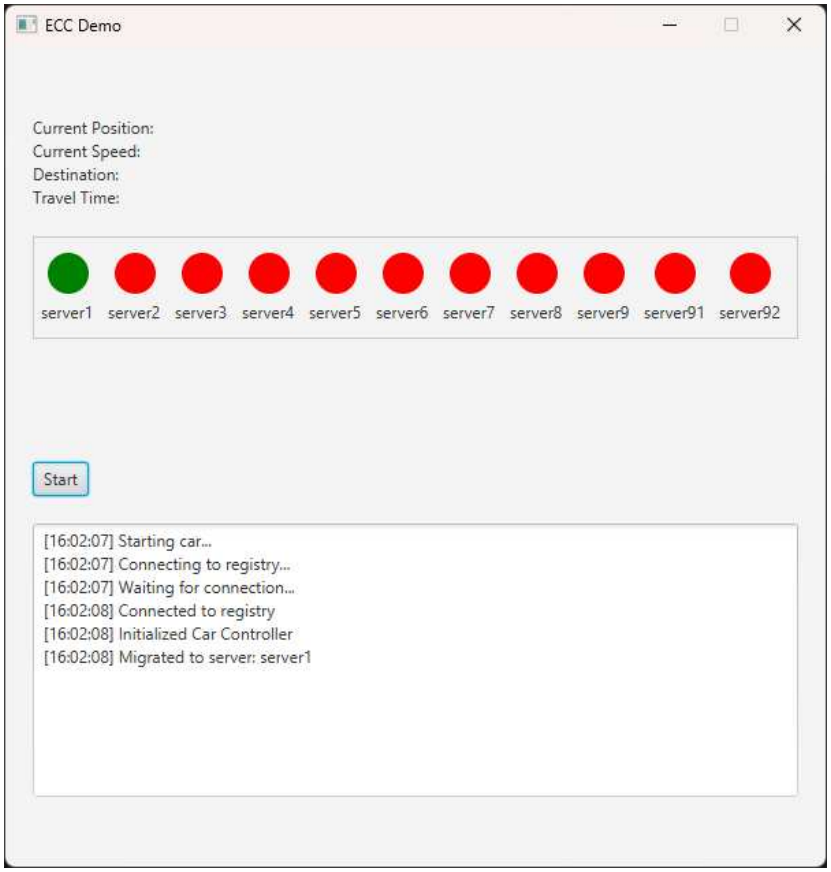
4.2 Testovací prostředí

Pro demonstraci implementace byla vytvořena zjednodušená verze zadání.

V ECC je nainstalovaná služba zpracovávající pohyb z auta. Tato služba určuje, zda má vozidlo jet, zpomalit nebo zastavit v závislosti na svých datech.

Ve vozidle běží jednoduchá aplikace, která každou chvíli vyžaduje informace ze vzdálené služby.

Ve stejnou chvíli zde probíhá migrace služby z jednoho serveru na druhý, v závislosti na aktuální poloze vozidla vůči serverům. Migrace proběhne plynule bez toho, aniž by se narušil chod služby. K této implementaci byla vytvořená jednoduchá vizualizace těchto procesů.



Obrázek 4.2 Okno aplikace po spuštění

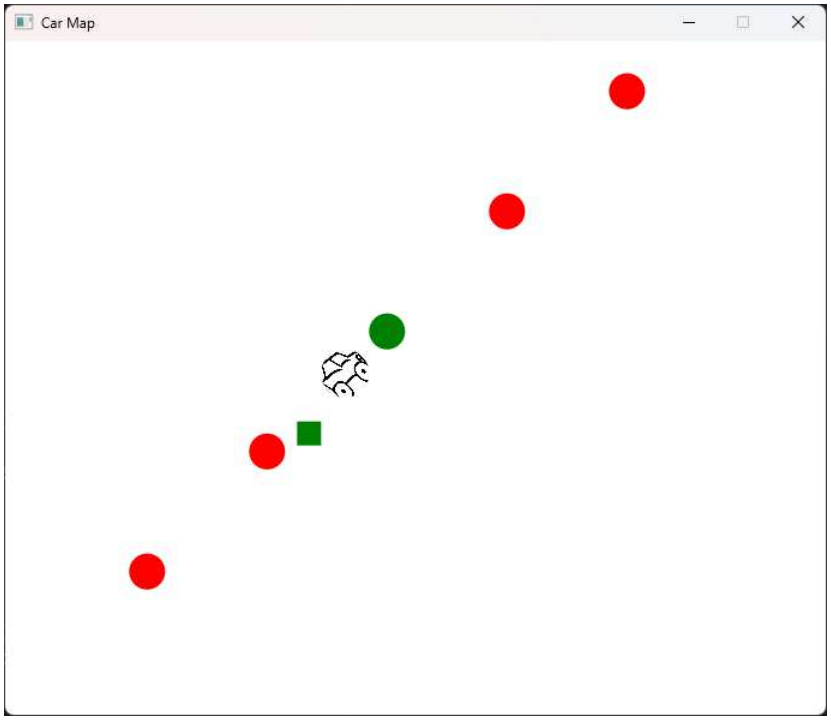
4.2.1 Popis aplikace

Aplikace po spuštění ukazuje základní stav našeho klienta. Toto okno lze vidět na obrázku 4.2.

Pokud je spuštěn registr serverů, připojí se k němu a stáhne si aktuální seznam serverů. Každý server je pak vyobrazen jako jedna tečka s jeho názvem. Barvy teček znázorňují, na jaký server je vozidlo aktuálně připojeno (zelená - připojeno, červená - nepřipojeno).

4.2.2 Nastavení simulace

- Auto
  - Maximální rychlost nastavená na 15 jednotek za vteřinu.
  - Vyjíždí z bodu (0,0) a jeho destinace je v bodě $(1000, 1000)$
- Servery
  - Serverů je celkově 10
  - Rozmístěné rovnoměrně po trase auta, tedy (0,0), $(100, 100)$, $(200, 200)$, ...
- Křižovatky
  - Pro ukázku předávání informací a komunikace s autem je zde křižovatka v bodě $(125, 125)$.



Obrázek 4.3 Vizualizace pohybu auta.

Migrace mezi servery se řídí standardní implementací uvnitř frameworku a poloha se aktualizuje při každém pohybu vozidla. Rozhodovací algoritmus je popsán v 3.2.3.

Simulace vozidla se aktualizuje každou vteřinu. Servery a registr byly spuštěny odděleně v Docker kontejnerech.

4.2.3 Úloha služby

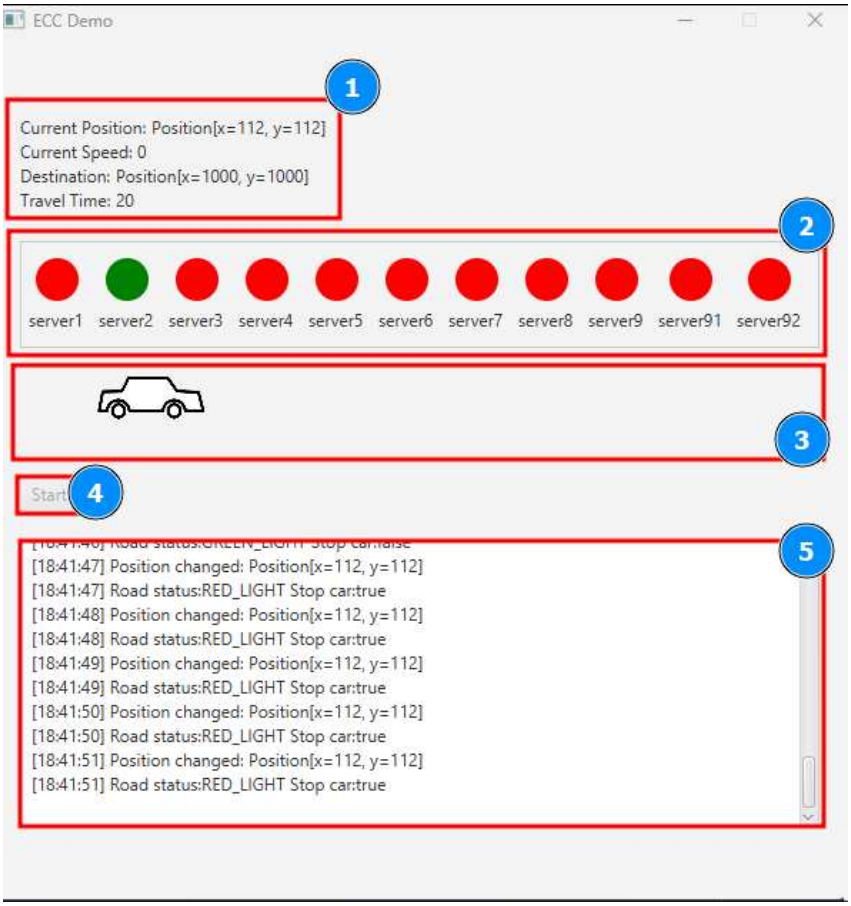
Služba, která je distribuovaná na všech serverech, má za úkol odesílat vozidlu informaci o stavu vozovky před ním. Vozidlo podle toho patřičně reaguje.

4.2.4 Spuštění simulace

Okno aplikace s vysvětlením je na obrázku 4.4.

Simulace se spustí stisknutím tlačítka *Start* (**(4)** na obrázku 4.4), tím se začne vozidlo pohybovat. Zároveň se otevře druhé okno s mapovou vizualizací pohybu vozidla. To můžeme vidět na obrázku 4.3. Na mapě jsou červené/zelené tečky, ty znázorňují jednotlivé servery. Barvou se rozeznává, zda je vozidlo k serveru aktuálně připojené nebo ne. Čtvereček je křižovatka, zde barva indikuje, zda může auto jet (zelená) nebo musí stát na křižovatce (červená). Posledním prvkem je kolečko s ikonou auta, to znázorňuje aktuální polohu vozidla.

Postupnou cestou vozidla se také mění server, na kterém se zpracovávají data. Jakmile vozidlo vjede do okruhu křižovatky, služba mu oznámí, jestli je na semaforu červená nebo zelená barva. V simulaci je nastaveno, že musí vozidlo čekat 5 vteřin než bude moct z křižovatky vyjet. Doba čekání se mezitím ukládá do sdílené databáze.



Obrázek 4.4 Popis Aplikace: 1. Aktuální informace o autě. 2. Zapnuté servery, barva indikuje, kde aktuálně probíhá zpracovávání dat. 3. Pohyb auta relativně k serverům. 4. Tlačítko k zapnutí simulace. 5. Logy aplikace

Křižovatka je záměrně umístěna na okraji dvou uzlů. Na zpracování dat o této křižovatce se tedy podílí oba tyto uzly. Je zde nutné, aby si mezi sebou včas předali informace o tom, jak dlouho vozidlo stálo a zda se smí rozjet.

4.2.5 Zhodnocení implementace

Tato zjednodušená implementace zadání se snaží demonstrovat klíčové prvky *ECC*. Služba je distribuována na všech serverech a je možné zpracovávání dat plynule přesunout na jiný server.

Splnění požadavků:

- Hierarchické rozdělení – je tu několik uzlů, které mohou data zpracovávat. Nejnižší je koncové zařízení - vozidlo. Poté tu jsou servery, ty reprezentují *edge/fog/cloud*.
- Nutnost nízké latence – vozidlo potřebuje co nejrychleji dostat zpět data o tom, zda se může pohybovat nebo ne. Framework díky použitým technologiím těmto požadavkům vyhovuje.
- Zpracování na více úrovních – data mohou být zpracována jak ve vozidle tak na samotných serverech.

4.2.6 Srovnání s existujícími pracemi

Koncept *Edge-Cloud Continuum* je poměrně nový a není pro něj tolik implementací. Existují především návrhy na implementace, třeba koncept *Vehicular Edge Computing* [14], kde *edge* uzly jsou samotná vozidla a výpočet probíhá na nich. Dalším návrhem je například strojové učení na vozidlech, které probíhá v reálném čase [15].

Existuje *CODECO Experimentation Framework*[16], což je otevřený software, který slouží k experimentování s nasazením *edge-cloud* architektury pomocí technologie Kubernetes. Tato práce není tolik o implementaci ECC, ale více o testování a vyhodnocení různých konfigurací architektury.

5 Závěr

V této práci byl prozkoumán koncept Edge-Cloud Continuum (ECC) a jeho výhody oproti jiným architekturám.

Cílem práce bylo implementovat framework v jazyce Java, který by splňoval základní požadavky ECC a umožňoval nasazení distribuovaných služeb. Vyvinutý framework umožňuje plynulou migraci služeb mezi servery a rozhodování o umístění zpracování dat. Zároveň umožňuje komunikaci mezi klienty a službami s nízkou latencí. Tímto se splnily cíle, které byly stanoveny v kapitole 1.1.

Práce se zaměřovala na základní principy ECC, existuje tedy prostor pro vylepšení a rozšíření. V budoucnu by bylo vhodné se soustředit na zabezpečení komunikace mezi uzly nebo implementaci pokročilejších algoritmů na rozhodování o migraci služeb. V neposlední řadě by se mohl výzkum zaměřit na koncept *fully-liquid software*, který byl popsán v kapitole 2.1.2.

Literatura

1. Miraz, Mahdi H.; Ali, Maaruf; Excell, Peter S.; Picking, Rich. A review on Internet of Things (IoT), Internet of Everything (IoE) and Internet of Nano Things (IoNT). In: *2015 Internet Technologies and Applications (ITA)* [online]. 2015, s. 219–224 [cit. 2024-07-03]. Dostupné z doi: 10.1109/ITechA.2015.7317398.

2. Cao, Keyan; Liu, Yefan; Meng, Gongjie; Sun, Qimeng. An Overview on Edge Computing Research. *IEEE Access* [online]. 2020, roč. 8, s. 85714–85728 [cit. 2024-06-20]. issn 2169-3536. Dostupné z doi: 10.1109/ACCESS.2020.2991734.

3. Milojicic, Dejan. The Edge-to-Cloud Continuum. *Computer* [online]. 2020, roč. 53, č. 11, s. 16–25 [cit. 2024-07-09]. issn 1558-0814. Dostupné z doi: 10.1109/MC.2020.3007297.

4. Khalyeyev, Danylo; Bureš, Tomas; Hnětynka, Petr. Towards Characterization of Edge-Cloud Continuum. In: Batista, Thais; Bureš, Tomáš; Raibulet, Claudia; Muccini, Henry (ed.). *Software Architecture. ECSA 2022 Tracks and Workshops* [online]. Cham: Springer International Publishing, 2023, sv. 13928, s. 215–230 [cit. 2024-06-18]. isbn 9783031368882 9783031368899. Dostupné z doi: 10.1007/978-3-031-36889-9_16.

5. Carmo, Maxweel S.; Jardim, Sandino; Neto, Augusto V.; Aguiar, Rui; Corujo, Daniel. Towards fog-based slice-defined WLAN infrastructures to cope with future 5G use cases. In: *2017 IEEE 16th International Symposium on Network Computing and Applications (NCA)* [online]. Cambridge, MA: IEEE, 2017, s. 1–5 [cit. 2024-06-18]. isbn 9781538614655. Dostupné z doi: 10.1109/NCA.2017.8171397.

6. Arzovs, Audris; Judvaitis, Janis; Nesenbergs, Krisjanis; Selavo, Leo. Distributed Learning in the IoT–Edge–Cloud Continuum. *Machine Learning and Knowledge Extraction* [online]. 2024, roč. 6, č. 1, s. 283–315 [cit. 2024-07-06]. issn 2504-4990. Dostupné z doi: 10.3390/make6010015.

7. Surbiryala, Jayachander; Rong, Chunming. Cloud Computing: History and Overview. In: *2019 IEEE Cloud Summit* [online]. 2019, s. 1–7 [cit. 2024-06-20]. Dostupné z doi: 10.1109/CloudSummit47114.2019.00007.

8. Buyya, Rajkumar; Yeo, Chee Shin; Venugopal, Srikumar; Broberg, James; Brandic, Ivona. Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility. *Future Generation Computer Systems* [online]. 2009, roč. 25, č. 6, s. 599–616 [cit. 2024-07-16]. issn 0167-739X. Dostupné z doi: 10.1016/j.future.2008.12.001.

9. Armbrust, Michael; Fox, Armando; Griffith, Rean; Joseph, Anthony D.; Katz, Randy; Konwinski, Andy; Lee, Gunho; Patterson, David; Rabkin, Ariel; Stoica, Ion; Zaharia, Matei. A view of cloud computing. *Communications of the ACM* [online]. 2010, roč. 53, č. 4, s. 50–58 [cit. 2024-07-16]. issn 0001-0782, issn 1557-7317. Dostupné z doi: 10.1145/1721654.1721672.

10. Satyanarayanan, Mahadev. The Emergence of Edge Computing. *Computer* [online]. 2017, roč. 50, č. 1, s. 30–39 [cit. 2024-07-07]. ISSN 1558-0814. Dostupné z DOI: 10.1109/MC.2017.9.

11. Gallidabino, Andrea; Pautasso, Cesare; Mikkonen, Tommi; Systa, Kari; Voutilainen, Jari-Pekka; Taivalsaari, Antero. ARCHITECTING LIQUID SOFTWARE. *Journal of Web Engineering* [online]. 2017, 433{470–433{470 [cit. 2024-07-05]. ISSN 1544-5976. Dostupné z: https://journals.riverpublishers.com/index.php/JWE/.

12. Taivalsaari, Antero; Mikkonen, Tommi; Systä, Kari. Liquid Software Manifesto: The Era of Multiple Device Ownership and Its Implications for Software Architecture. In: *2014 IEEE 38th Annual Computer Software and Applications Conference* [online]. 2014, s. 338–343 [cit. 2024-07-05]. Dostupné z DOI: 10.1109/COMPSAC.2014.56. ISSN: 0730-3157.

13. *Sharding - MongoDB Manual v7.0* [online]. [B.r.]. [cit. 2024-07-04]. Dostupné z: https://www.mongodb.com/docs/manual/sharding/.

14. Liu, Lei; Chen, Chen; Pei, Qingqi; Maharjan, Sabita; Zhang, Yan. Vehicular Edge Computing and Networking: A Survey. *Mobile Networks and Applications* [online]. 2021, roč. 26, č. 3, s. 1145–1168 [cit. 2024-07-15]. ISSN 1572-8153. Dostupné z DOI: 10.1007/s11036-020-01624-1.

15. Peltonen, Ella; Sojan, Arun; Päivärinta, Tero. Towards Real-time Learning for Edge-Cloud Continuum with Vehicular Computing. In: *2021 IEEE 7th World Forum on Internet of Things (WF-IoT)* [online]. 2021, s. 921–926 [cit. 2024-07-15]. Dostupné z DOI: 10.1109/WF-IoT51360.2021.9595628.

16. Koukis, Georgios; Skaperas, Sotiris; Kapetanidou, Ioanna Angeliki; Tsaoussidis, Vassilis; Mamatas, Lefteris. *An Open-Source Experimentation Framework for the Edge Cloud Continuum* [online]. arXiv, 2024 [cit. 2024-07-07]. Dostupné z DOI: 10.48550/arXiv.2403.10977. arXiv:2403.10977 [cs] version: 1.

Seznam obrázků

Seznam tabulek

Seznam použitých zkratek

A Přílohy

A.1 Popis struktury přiloženého projektu

Příloha je kopie veřejného repositáře, který lze najít na stránce: https://-gitlab.mff.cuni.cz/lamplotm/ecc-demonstration.

Soubor README.md obsahuje základní informace a instrukce jak si spustit testovací architekturu.

V podsložce ./src/ECC jsou zdrojové kódy všech spustitelných součástí.