FAKULTA APLIKOVANÝCH VĚD
ZÁPADOČESKÉ UNIVERZITY
V PLZNI

KATEDRA INFORMATIKY
A VÝPOČETNÍ TECHNIKY

Diplomová práce

Distribuovaná inteligence a její aplikace pro monitoring diabetického pacienta

Jakub Klimeš

PLZEŇ

2024

FAKULTA APLIKOVANÝCH VĚD
ZÁPADOČESKÉ UNIVERZITY
V PLZNI

KATEDRA INFORMATIKY
A VÝPOČETNÍ TECHNIKY

Diplomová práce

Distribuovaná inteligence a její aplikace pro monitoring diabetického pacienta

Jakub Klimeš

Vedoucí práce
Ing. Martin Úbl

PLZEŇ 2024

Citace v seznamu literatury:
KLIMEŠ, Jakub. *Distribuovaná inteligence a její aplikace pro monitoring diabetického pacienta*. Plzeň, 2024. Diplomová práce. Západočeská univerzita v Plzni, Fakulta aplikovaných věd, Katedra informatiky a výpočetní techniky. Vedoucí práce Ing. Martin Úbl.

ZÁPADOČESKÁ UNIVERZITA V PLZNI
Fakulta aplikovaných věd
Akademický rok: 2023/2024

ZADÁNÍ DIPLOMOVÉ PRÁCE

(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení:	**Bc. Jakub KLIMEŠ**
Osobní číslo:	**A22N0077P**
Studijní program:	**N0613A140039 Distribuované výpočetní systémy**
Specializace:	**Výpočetní technika**
Téma práce:	**Distribuovaná inteligence a její aplikace pro monitoring diabetického pacienta**
Zadávající katedra:	**Katedra informatiky a výpočetní techniky**

Zásady pro vypracování

1. Seznamte se s algoritmy distribuované inteligence, nutným teoretickým pozadím a možnostmi implementace. Seznamte se s frameworkem SmartCGMS.
2. Analyzujte možnosti implementace algoritmů distribuované inteligence v oblasti monitoringu diabetického pacienta.
3. Navrhněte softwarové řešení, které bude řešit vybraný problém v této oblasti a které bude zakládat na frameworku SmartCGMS.
4. Implementujte dané řešení na sadě nositelných zařízení s ohledem na efektivitu výpočtů, bezpečnost a spotřebu elektrické energie.
5. Otestujte implementované řešení a zhodnoťte dosažené výsledky s ohledem na výše uvedené aspekty.

Rozsah diplomové práce: **doporuč. 50 s. původního textu**
Rozsah grafických prací: **dle potřeby**
Forma zpracování diplomové práce: **tištěná/elektronická**

Seznam doporučené literatury:

dodá vedoucí diplomové práce

Vedoucí diplomové práce: **Ing. Martin Úbl**
Katedra informatiky a výpočetní techniky

Datum zadání diplomové práce: **8. září 2023**
Termín odevzdání diplomové práce: **16. května 2024**

L.S.

Doc. Ing. Miloš Železný, Ph.D.
děkan

Doc. Ing. Přemysl Brada, MSc., Ph.D.
vedoucí katedry

V Plzni dne 11. října 2023

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně a výhradně s použitím citovaných pramenů, literatury a dalších odborných zdrojů. Tato práce nebyla využita k získání jiného nebo stejného akademického titulu.

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 Západočeská univerzita v Plzni má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona.

Plzeň dne 16. května 2024

. . . . . .
Jakub Klimeš

V textu jsou použity názvy produktů, technologií, služeb, aplikací, společností apod., které mohou být ochrannými známkami nebo registrovanými ochrannými známkami příslušných vlastníků.

Abstrakt

Tato práce se věnuje návrhu a implementaci distribuovaného optimalizačního algoritmu pro framework SmartCGMS, ve kterém bude větší množství nezávislých zařízení řešit problém stejné třídy s občasnou výměnou informací o dosavadních výsledcích. Cílem je otestovat, jaký dopad bude tato výměna informací mít na kvalitu výsledného řešení. Výsledný program je zamýšlen pro využití v léčbě diabetu, konkrétně v rámci nositelné inzulinové pumpy, a jsou tedy zohledněny omezené technické prostředky, kterými tato zařízení disponují.

Abstract

This document describes the design and implementation of distributed optimalization algorithm intented for use under the SmartCGMS framework, in which numerous independent devices solve a problem of the same class with an occasional interchange of information about their progress. The goal is to evaluate the impact of this interchange on the quality of the resulting solution. The product is intented to be used for diabetes mitigation as a part of an insulin pump and therefore, the limited resources of these devices are taken into consideration.

Klíčová slova

diabetes • strojové učení • CGM • inzulinová pumpa • RaspberryPi • SmartCGMS • metaheuristika • federated learning

Obsah

Úvod 1

Diabetes je chronické onemocnění, způsobené nedostatečnou produkcí inzulinu slinivkou (diabetes 1. typu), nebo neschopností těla zpracovat dostupný inzulin (diabetes 2. typu). Důsledkem je nesprávná koncentrace glukózy v krvi, což může vést k poškození vnitřních orgánů pacienta. V roce 2014 trpělo diabetem 8,5 % dospělých [1]. Zatímco proti diabetu 2. typu se lze do určité míry bránit prevencí a částečně ho změnou životního stylu lze i léčit, proti diabetu 1. typu účinná prevence neexistuje a pacient je závislý na pravidelných injekcích inzulinu.

Tyto injekce si pacient může na základě manuálního měření koncentrace cukru v krvi aplikovat sám, v současnosti jsou ale k dispozici i automatizované metody, které tuto činnost zjednodušují:

- automatické senzory množství glukózy v podkoží, umožňující kontinuální měření (CGM – continuous glucose measurement), v praxi např. jednou za pět minut oproti jednotkám manuálních měření za den,

- inzulinové pumpy, umožňující průběžné, jemnější dávkování inzulinu.

Ideálním výsledkem by byl vznik tzv. *closed loop* systému, tedy takového, který by za pomoci inzulinu injektovaného inzulinovou pumpou množství glukózy v krvi reguloval tak, že by pacient nemusel do regulace nijak zasahovat. Pro dosažení tohoto cíle je nutné popsat dynamiku pacientova těla, nebo alespoň jeho části týkající se metabolismu a zpracování glukózy. Za tímto účelem se využívají různé matematické modely, např. Hovorkův, Sorensenův, UVA/Padova [2]. Zásadním problémem je nicméně jejich identifikace – každý model obsahuje podle jeho složitosti různé množství parametrů (od řádově jednotek v případě nejjednodušších – minimálních – modelů, až po stovky pro komplexní – maximální – modely), vystihujících fyziologii lidského těla, kterým je potřeba přiřadit konkrétní hodnotu pro konkrétního pacienta. Prakticky přitom nelze řešení získat analyticky, častěji je nutné modely identifikovat numericky, což je zpravidla náročnější na výpočetní výkon zařízení, které model identifikuje.

Samotný výpočet může být prováděn různými způsoby od centralizovaných po distribuované. Ideální by samozřejmě bylo pacientovi zasahovat do běžného života jen minimálně a zároveň poskytovat co možná nejlépe identifikovaný model v co možná nejlepších časech, což by pro pacienta znamenalo vyšší kvalitu života a krok směrem ke *closed loop* regulaci. Cílem této práce je v kontextu diabetu analyzovat různé možnosti identifikace souvisejících modelů, navrhnout program implementující v rámci frameworku SmartCGMS některou z analyzovaných metod a prakticky řešení otestovat a zhodnotit.

Teoretické pozadí 2

Tato kapitola stručně popisuje teoretické poznatky, na kterých zbytek práce zakládá. Jmenovitě nemoc diabetes mellitus, způsoby, jakými se diabetes modeluje, distribuované řešení problémů potažmo distribuovanou inteligenci, různé typy a metaheuristik, a nakonec framework SmartCGMS.

2.1 Diabetes mellitus

Každý živý organismus pro svou funkci vyžaduje průběžný přísun energie. V případě člověka to znamená především konzumaci potravy, kterou tělo během trávení rozkládá na jednodušší látky, se kterými dokáže pracovat a které jsou krví distribuovány po celém těle. Jednou takovou základní látkou je glukóza (typ cukru). Množství glukózy, které se v krvi vyskytuje, musí být nicméně poměrně striktně regulováno. Pokud by hladina glukózy v krvi klesla příliš nízko – v případě zdravého dospělého člověka pod 3,9 mmol/l [3] – nastává tzv. hypoglykémie, během které je postižen především mozek, který není dostatečně živen. Mezi symptomy patří bolesti hlavy, rozmazané vidění, ale i bezvědomí a v krajním případě až smrt [4]. Naopak stav, kdy je glukózy v krvi příliš, se nazývá hyperglykémie. U zdravého člověka nastává, pokud koncentrace glukózy v krvi přesáhne 5,6 mmol/l, krátce po jídle je hranice vyšší (až 7,8 mmol/l) [5]. Pokud tento stav přetrvává delší dobu (což se nejčastěji stane právě diabetikovi), může dojít k poškození očí, ledvin nebo kardiovaskulárního systému [6].

U zdravého člověka je množství glukózy udržováno mezi výše uvedenými hodnotami pro hypo- a hyperglykémii inzulinem, který vylučuje slinivka. Nadměrné množství glukózy v krvi (11 mmol/l a více) zpravidla znamená (neléčený) diabetes [7].

Inzulin je hormon vylučovaný slinivkou břišní, který stimuluje buňky ke vstřebávání glukózy a zároveň je za jeho podpory velké množství přebytečné glukózy ukládáno do jater ve formě glykogenu. Játra následně v době bez přísunu potravy uloženým glykogenem (opět transformovaným na glukózu) vyrovnávají hladinu glukózy v krvi tak, aby neklesla na nebezpečnou mez směrem k hypoglykémii. Po-

dobným způsobem mohou přebytečnou glukózu pro pozdější využití ukládat i svaly. Svaly glukózu taktéž využívají při vyšší fyzické námaze, a to i bez stimulace inzulinem [8].

Problém nastává, pokud systém regulace glukózy v krvi přestane fungovat – v tu chvíli pacient trpí oněmocněním diabetes mellitus. Rozlišovány jsou především dva základní typy: diabetes 1. a 2. typu [9]. Jejich důsledkem je zvýšená koncentrace glukózy v krvi, ale příčina se liší a způsob léčby se liší.

2.1.1 Diabetes 1. typu

Diabetes 1. typu (také anglicky nazývaný *insulin-dependent*) nastává zpravidla velmi prudce a nejčastěji propuká už v dětství, prakticky maximálně do třiceti let [10]. Proti tomuto typu neexistuje žádná prevence, konkrétní spouštěče nejsou známy (k rozvinutí může dojít například po virovém onemocnění) – jedná se o autoimunitní reakci (pacientovo tělo bojuje samo proti sobě), ale pacient musí mít vždy genetické predispozice.

Výsledkem je, že slinivka není schopna tělu dodávat dostatečné množství inzulinu a pacient si musí pravidelnými injekcemi zmíněného hormonu koncentraci glukózy v krvi regulovat ručně, případně (polo-)automaticky za pomoci inzulinové pumpy. S tím souvisí i potřeba koncentraci glukózy v krvi měřit a co nejlépe předvídat – to lze opět buď ručně (typicky píchnutím do prstu a vytlačením kapky krve), což je sice relativně přesné, ale nepříjemné na častější provádění, případně automaticky CGM systémem z podkoží – sice s potenciálně horší přesností, ale mnohem vyšší frekvencí. Zhruba 5 až 10 % ze všech diabetiků trpí tímto typem onemocnění [11].

2.1.2 Diabetes 2. typu

Diabetes 2. typu (anglicky také *non-insulin-dependent*) způsobuje odolnost těla proti inzulinu, který slinivka produkuje ve stejném množství, jako u zdravého člověka (alespoň zpočátku, při dlouhodobém přetěžování schopnost slinivky produkovat inzulin upadá) [10]. Největším rizikovým faktorem je obezita, obecně nezdravý životní styl, a stáří. Taktéž se předpokládá nutnost genetické predispozice.

Toto onemocnění se častěji vyskytuje u osob starších třiceti let, i když v současnosti se čím dál častěji vyskytuje i u dětí [1]. Projevy tohoto onemocnění nastávají velmi pozvolna i v řádech let, díky čemuž může k diagnostikování dojít až ve chvíli, kdy už došlo k nenávratnému poškození těla pacienta kvůli dlouhodobému stavu hyperglykémie. Diabetes 2. typu lze dlouhodobě léčit změnou životosprávy a snížením tělesné hmotnosti. Pro okamžitou léčbu jsou na trhu dostupné léky. Injekce inzulinu mohou být potřeba až v pozdějších stádiích onemocnění [8]. Pacienti s diabetem 2. typu tvoří až 95 % všech diabetiků.

Tato práce se dále bude zabývat především diabetem 1. typu, vzhledem k jeho typicky závažnějšímu průběhu a silnější motivaci léčbu maximálně automatizovat.

2.2 Modelování diabetu

Jak už bylo řečeno v odstavci 2.1.1, pro maximální komfort pacienta trpícího diabetickým onemocněním (především 1. typu) by bylo ideální mít možnost koncentraci glukózy v krvi co nejpřesněji předvídat, alespoň v krátkodobém horizontu (desítky minut). To je velmi komplikovaný úkol, jelikož každý pacient má odlišnou dynamiku metabolismu, klidně i proměnnou v čase. Bez podpory informačních systémů tuto činnost intuitivně provádí sám pacient tím, že zná své tělo – dokáže odhadnout, že po vydatném obědě se koncentrace glukózy v krvi bude chovat určitým způsobem, bude potřeba dodat určité množství inzulinu apod.

Mnohem příjemnější by bylo mít možnost regulaci přenechat automatickému systému, který by potenciálně mohl množství glukózy v krvi regulovat alespoň stejně dobře, jako sám pacient. Tento přístup by v ideálním případě vedl až k vývoji umělé slinivky (*artificial pancreas* – AP). Takový systém by se staral o regulaci glukózy v krvi zcela nezávisle a samostatně: na základě naměřené koncentrace glukózy v podkoží CGM systémem by pacientovi dávkoval inzulin inzulinovou pumpou.

Řídicí algoritmus umělé slinivky je založen na modelu. Základní podstatou modelování je vybraným více či méně zjednodušeným způsobem popsat důležité aspekty systému tak, aby bylo možné model použít jako jeho věrnou náhradu. Tím je pak také možné z chování modelu odvozovat předpoklady o tom, jak se bude chovat původní systém.

Modely lze dělit na black-box a white-box [12]. Black-box (nebo také data-driven) modely neuvažují apriori žádné zákonitosti mezi vstupem a výstupem – naopak na základě naměřených dat se tuto závislost teprve snaží stanovit.

Oproti tomu white-box modely vychází ze znalostí o modelovaném systému – v případě diabetu je potom model navrhován tak, aby reflektoval zákonitosti a funkcionalitu lidského metabolismu. Často se jedná o kompartmentové modely, kde kompartment značí část systému, která je z hlediska modelu homogenní. Vyjádřenybývají čsato soustavou diferenciálních rovnic. Takový model je nicméně následně potřeba identifikovat, tj. přiřadit parametrům konkrétní hodnoty, odpovídající konkrétnímu pacientovi. Množství parametrů se přitom může výrazně lišit podle určení modelu – minimální modely se snaží být co nejjednodušší a popsat jen ty nejzákladnější závislosti. Příkladem je např. Bergmanův minimální model [13]. Opakem jsou maximální modely, které se naopak snaží zachytit co nejširší spektrum znalostí, s čímž se ale samozřejmě váže mnohem větší složitost, a než pro regulaci se hodí spíše pro simulaci systému [14]. Příkladem je Sorensenův model [15] nebo UVA/Padova simulátor [16].

Pro ilustraci Bergmanův minimální model:

$$\frac{dG(t)}{dt} = (P_1 - X)G(t) - P_1 Gb$$

$$\frac{dX(t)}{dt} = P_2 X(t) + P_3 I(t)$$

obsahuje dvě rovnice a pět parametrů. Oproti tomu pro vypsání všech rovnic Sorensenova modelu by nestačila celá jedna strana, je totiž sestaven ze zhruba pěti desítek rovnic obsahujících přibližně 140 parametrů.

Jsou to dva extrémy. Na jedné straně malý model, nenáročný na identifikaci a výpočet, nicméně poskytující jen omezený vhled do systému a příliš hrubý na to, aby bylo možné jej použít v reguluaci. Na straně druhé obsáhlý model, popisující systém do nejmenších detailů, se všemi nevýhodami, které s tím souvisí – jako obtížnost identifikace, ale také např. validace.

2.3 Distribuovaná inteligence

Distribuovaná inteligence je podtypem umělé inteligence, ve kterém je schopnost systému řešit problémy rozprostřena mezi množství nezávislých zařízení. Očividnou a obecnou výhodou takového decentralizovaného přístupu spočívá v robustnosti – výpadek jednoho nebo i části participujících zařízení neohrozí funkčnost celku. V případě této práce je motivace k distribuci daná navíc tím, že inzulinová pumpa nemá sama o sobě příliš výpočetního výkonu, což by se dalo částečně zmírnit propojením většího množství těchto zařízení.

2.3.1 Inzulinová pumpa

Diabetem v roce 2014 trpělo 422 milionů lidí [1], část z nich trpí diabetem 1. typu a používá CGM systém s inzulinovou pumpou. Inzulinová pumpa je embedded zařízení skládající se z pumpy jako takové, infúzního setu, a procesoru s pamětí (typicky ve formě mikrokontroléru), který pumpu ovládá. Celé zařízení je napájeno z baterie (vyměnitelné tužkové, případně zabudované jiného typu) a hmotnosti oficiálních pump se pohybují řádově do 100 g [17] [18].

Je zřejmé, že výpočetní výkon zde není prioritou (a pro samotnou funkci pumpy ani není bezpodmínečně třeba), cílem je naopak minimální spotřeba, aby se pacient nemusel pumpě příliš často věnovat. O tom, jaké jsou konkrétní platformy, použité ve schválených pumpách, lze jen spekulovat, jelikož je výrobci neuvádějí. Použitý hardware lze nicméně částečně odhadnout: např. práce [19] a [20] se zabývají návrhem levné inzulinové pumpy a jako řídící jednotka byl v prvním případě využit mikrokontrolér ATSAMD21G18 od firmy Microchip, obsahující procesor ARM,

v druhém případě mikrokontrolér MSP430FG347 od firmy Texas Instruments. Samotné řízení vytlačování inzulinu není nikterak výpočetně náročné a obě zmíněné pumpy jsou konstruovány s ohledem na co nejnižší výrobní náklady a složitost, čemuž samozřejmě výběr řídících jednotek odpovídá. Poskytují jen minimalistické možnosti ovládání a integrace s CGM systémem není k dispozici vůbec.

Jiným příkladem na hardwarově vyšší úrovni může být projekt OpenAPS [21]. Jako „do-it-yourself“ („udělej si sám“) systém má oproti schváleným systémům značné nedostatky, ale pro tuto chvíli je podstatný hardware, na kterém autoři doporučují OpenAPS provozovat – kromě již nevyráběné platformy Intel Edison se jedná o Raspberry Pi [22], což je řada malých, plnohodnotných počítačů, postavených nad procesory ARM, v případě nejmenšího Raspberry Pi Zero se jedná o čip BCM2835. Tato zařízení už mají dostatečný výkon i na provozování některých „velkých“ operačních systémů, jako je GNU/Linux, což výrazně napomáhá tomu, aby si i netechničtí uživatelé dokázali řídící algoritmy zprovoznit, ale pro funkci pumpy jako takové interaktivní operační systém není podmínkou.

Obecně lze říct, že velmi jednoduché pumpy, kde jde jenom o průběžné dávkování inzulinu, si klidně vystačí i s těmi nejslabšími mikrokontroléry. Pokud je potřeba komunikace s dalšími prvky a sofistikovanější ovládání, výpočetní výkon musí být vyšší, ale jako myšlená hranice může sloužit právě například Raspberry Pi Zero – výkonnější vybavení už pro inzulinové pumpy nedává příliš smysl.

2.3.2 Cesta k distribuovaným algoritmům

Nabízí se tedy úvaha, zda by nebylo možné využít výpočetní výkon inzulinových pump k identifikaci modelů diabetu. Teoreticky je samozřejmě možné nasbíraná data od pacienta přenést tam, kde je k dispozici prakticky neomezený výpočetní výkon a elektrická energie, model identifikovat a zaslat zpět pacientovi. Tento centralizovaný přístup je lákavý pro svou potenciální jednoduchost, ale přináší dva zásadní problémy:

- Častý přenos potenciálně velkého množství dat – model by měl být ideálně přepočítáván po každé nově naměřené hodnotě, což může s CGM systémem být např. každých pět minut. Každých pět minut by tak bylo nutné periodicky serveru posílat nově naměřená data, v horším případě ve formě kompletní sady od začátku měření, nebo lépe inkrementálně s ukládáním předešlých hodnot na serveru. V obou případech je nutné mít stabilní připojení.

- Nasbíraná data podléhají přísným ochranám – jakákoliv medicínská data musejí být pečlivě chráněna proti zneužití a jejich sběr, přenos po síti a ukládání na dalších místech znamená pro systém další velmi přísné legislativní požadavky. Po jejich splnění může být ale navíc potřeba bojovat o důvěru uži-

vatelů, kteří, i přes všechny potvrzení o shodě, mohou zůstat skeptičtí a svá osobní data systému vůbec nesvěřovat, přestože k tomu není žádný objektivní důvod.

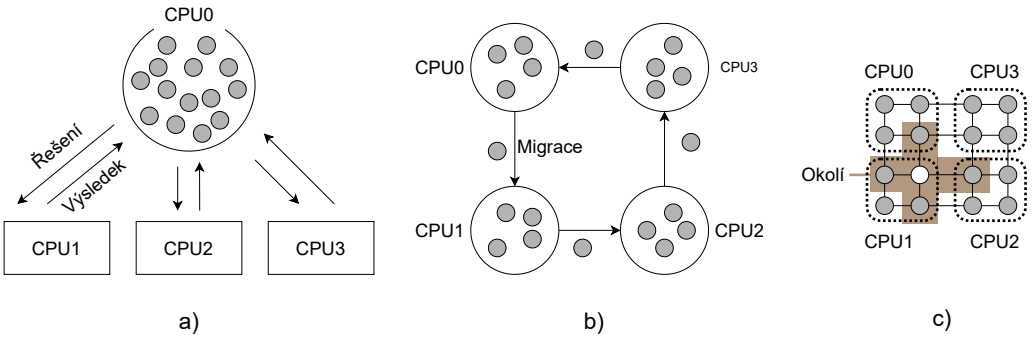
S ohledem na výše uvedené se nabízí data ponechávat lokálně a identifikaci provádět jen u pacienta, nicméně vzhledem k výpočetnímu výkonu pump se snižuje pravděpodobnost nalezení optimálního řešení. Důvodem je menší výkon embedded zařízení, kvůli kterému dokáží numerické metody prohledat jen menší část stavového prostoru.

Je ale nutné vzít v úvahu, že je těchto zařízení ale potenciálně velké množství, a vhodnou výměnou informací si mohou zařízení vzájemně předávat znalosti o prozkoumaných částech stavového prostoru – jinými slovy přejít od centralizovaného řešení k paralelnímu, resp. distribuovanému. V naprosto ideálním případě by se pak velké množství relativně slabých strojů dokázalo vyrovnat mnohem silnějšímu zařízení.

V případě diabetu nemá smysl hledat globální řešení – každý člověk je odlišný a tedy i stavové prostory budou pro každého pacienta odlišné. Nicméně to neznamená, že jsou znalosti ostatních v kontextu jednotlivce zcela bezvýznamné. Numerickým metodám obecně nezbývá mnoho jiných alternativ, než vybraným způsobem (pseudo-)náhodně prozkoumávat stavový prostor a snažit se konvergovat k optimu. Toto optimum bude s velkou pravděpodobností lokální, a právě informace od ostatních může pomoci algoritmus nasměrovat dále, místo toho, aby uvíznul na jednom místě nebo v celém regionu lokálních optim. Pokud by byla informace pro příjemce nezajímavá, může ji ignorovat.

Základní způsoby paralelizace na konkrétním příkladu genetických algoritmů jsou prezentovány například v [23]. Nejpřímějším způsobem je globální paralelizace, kdy jsou všechna řešení stále ukládána na společném centrálním prvku, ale pro jejich průběžné vyhodnocování je využito zároveň více výpočetních jednotek. Druhou možností je tzv. *ostrovní model*, kdy si každý procesor udržuje svou množinu řešení, se kterou pracuje nezávisle na ostatních. V určité části algoritmu dochází k vzájemné výměně některých řešení (tzv. migraci). Výhodou je nezávislost ostrovů a tedy možná nehomogenita při hledání řešení. Třetím zmiňovaným způsobem je celulární model, ve kterém jsou řešení rozprostřena do mřížky a jejich vyhodnocování je ovlivněno okolím. Každý procesor přitom vyhodnocuje určitou část celé množiny a stejně jako u ostrovního modelu lze zavést nehomogenní zpracovávání. Problematická ale může být komunikace, pokud je potřeba pracovat s řešením, které se nenachází pod správou lokálního CPU. Graficky jsou tři zmíněné varianty znázorněny na obrázku 2.1. Samozřejmě je možné různé modely kombinovat a vytvářet hybridní formy paralelismu.

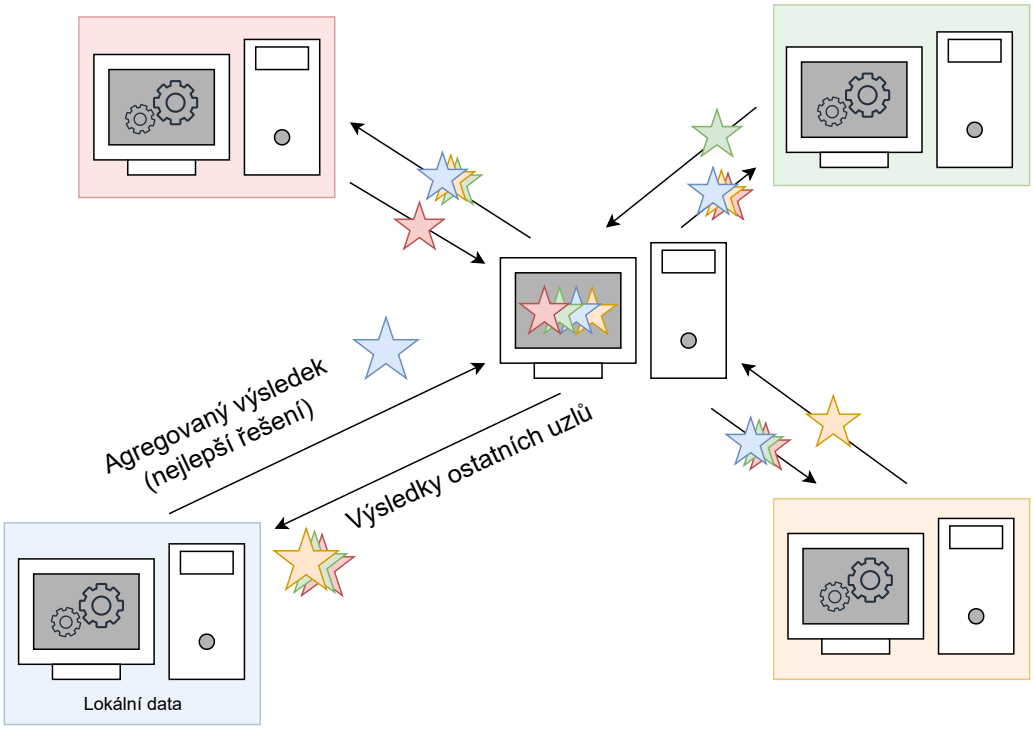
V kontextu výše zmíněného článku jsou všechny uvedené modely výpočtu lo-



Obrázek 2.1: Různé modely paralelismu podle [23], a) globální, b) ostrovní, c) celulární

kální ve smyslu, že běží na jednom stroji. Snadno lze ale přejít k distribuovanému výpočtu tak, že místo různých procesorů budeme uvažovat různé stroje, komunikující po síti. Pro tuto práci není příliš zajímavá globální paralelizace – prakticky nic nebrání jejímu použití i bez sdílení dat, ale pouze tehdy, pokud by všechny uzly systému pracovaly buď se stejnými daty, nebo se sice odlišnými daty, ale nezávisle vzorkovanými ze stejného problému (*independently and identically distributed – IID*). V opačném případě by vhodnost řešení, kterou jednotlivé uzly nad svými daty počítají, neměla žádnou spojitost s vhodností pro jiné uzly. Tento model je vhodný především pro urychlení výpočtu jako takového. Zajímavější je celulární model, kde by nemuselo být bezpodmínečně nutné vyžadovat, aby data v rámci uzlů měla vlastnost IID. Nové řešení, zakládající na řešeních od ostatních uzlů, po vyhodnocení na lokálních datech, nemusí být shledáno vhodným a vyloučeno. Problémem je ale jednak již zmíněná potenciálně zbytečně intenzivní komunikace, po síti navíc pomalá, a jednak propojení uzlů navzájem, resp. jeho regulární struktura, ve které lze jen těžko připustit výpadky, ke kterým v sítích běžně dochází. Nejvhodnějším modelem je model ostrovní. Také není potřeba trvat na vlastnosti IID distribuovaných dat a parametry migrace si může každý uzel řešit dynamicky podle aktuálních okolností. Tento model může navíc těžit ze zavedení jednoho centrálního uzlu (serveru), který může převzít roli příjemce migrovaného řešení a následně ho mezi uzly rozdistribuovat, případně řešení vybraným způsobem ukládat, agregovat a jinak s nimi pracovat.

Toto uspořádání (distribuované uzly s trvale lokálními daty, spolupracující při trénování modelu za podpory centrálního prvku) se v anglické literatuře označuje pojmem *federated learning* (FL), poprvé zmíněným společností Google při trénování modelů pro předvídání následujících slov na mobilních klávesnicích. Dnes už se tento přístup využívá v mnoha dalších odvětvích [24]. Graficky to zachycuje obrázek 2.2. Svým způsobem se jedná o vyšší úroveň tzv. *edge computing*, což je termín z oblasti internetu věcí, označující zpracování dat blíže jejich původu (senzorům).



Obrázek 2.2: Grafické znázornění konceptu federated learning.

Lze samozřejmě pokračovat ještě dále ve směru k decentralizovanosti a vypustit i centrální prvek. Účastníci systému pak ale musí komunikovat přímo mezi sebou, což do návrhu zavádí další složitosti. Článek [24] už takový systém označuje jako *peer-to-peer distributed learning* namísto federated learning.

Zásadním důvodem pro zabývání se tímto přístupem je napomáhání ochraně dat, která často podléhají i nadnárodním nařízením (např. GDPR v EU). Samozřejmě jen samotný koncept federated learning ochranu dat a bezpečnost aplikace jako takové nezajišťuje, jde jí ale více naproti. Kompletní analýza z tohoto pohledu je nad rámec této práce, informace lze nalézt např. v [25].

FL je často dělelno na cross-silo a cross-device, přičemž cross-silo označuje systém s relativně málo účastníky, kteří mají ale velké množství dat (např. několik nemocnic), oproti tomu cross-device označuje systém s potenciálně obrovským množstvím účastníků, ale menším množstvím dat na každém z nich. Z hlediska rozložení dat účastníků se může jednat o horizontální rozložení, kdy mají data stejné vlastnosti (*features*), ale jiný výběrový prostor (*sample space*), nebo o vertikální rozložení, kdy je výběrový prostor podobný, ale data mohou mít jiné vlastnosti. Horizontální rozložení je typické pro cross-device FL – každý účastník sbírá stejný typ dat, která jsou vzájemně nezávislá, oproti tomu vertikální rozložení je použito spíše v kontextu cross-silo, kdy mohou mít poskytovatelé dat různé informace o stejném

jevu [26].

Identifikaci modelů diabetu odpovídá cross-device FL.

2.3.3 Problémy a jejich distribuovanost

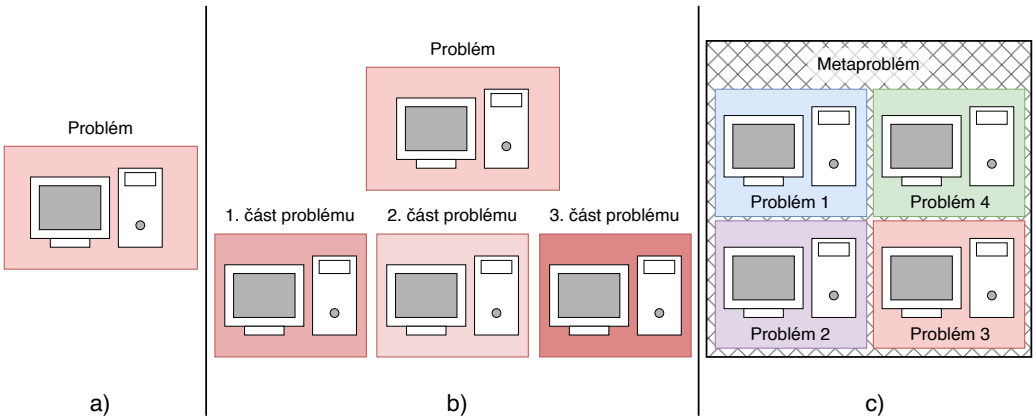
Na popsanou problematiku se dá pohlížet ještě z trochu jiného úhlu, a to sice z hlediska množství a rozprostřenosti problémů, které jsou řešeny – příkladem v tomto odstavci budiž třeba městské veřejné osvětlení, reagující dynamicky na úroveň denního světla.

Jednoduchý centralizovaný systém řeší na jednom stroji jeden problém: měří si intenzitu světla ve městě a pokud klesne pod stanovenou mez, zapíná lampy.

Systém může být distribuovaný, ale centrálně řízený: každá lampa po městě měří intenzitu nezávisle a tuto informaci sděluje centrálnímu uzlu. Pokud to centrální uzel uzná za vhodné, vydá všem lampám pokyn k rozsvěcení. Problém je v tomto případě stále jen jeden globální, ale je rozprostřen přes větší množství uzlů, které se podílí na jeho optimálním řešení.

Poslední možností jsou zcela nezávislé lampy, které se podle naměřených dat automaticky rozsvěcí a zhasínají. Samy o sobě tedy řeší stejný problém, o který se v předchozích případech staral výhradně centrální uzel, ale v menším měřítku. Zároveň se z principu málokdy stane, že se dvě nezávislé lampy rozsvítí ve stejnou chvíli – na různých místech města intenzita světla klesá různě rychle. Ideálně by spolu lampy ale měly alespoň částečně komunikovat – např. v případě poruchy snímání venkovního osvětlení se může porouchaná lampa stále rozsvítit ve chvíli, kdy svítí i ty v jejím nejbližším okolí. Problém už tedy není jen jeden, je jich naopak poměrně velké množství, ale všechny jsou stejného typu (jeden „metaproblém“).

Schematicky je popisované dělení znázorněno na obrázku 2.3.



Obrázek 2.3: Různé typy problémů vzhledem k jejich distribuovanosti, a) jeden problém, jeden stroj, b) jeden problém, více strojů, c) více problémů, více strojů.

Je zřejmé, že s rostoucím rozdrobením a volnějším vázáním uzlů na sebe roste jejich složitost (kterou na sebe v centralizovaných řešeních bere právě centrální prvek), na druhou stranu se může jednat o způsob, jak se dostat blíže k optimu systému jako celku a lepšímu využití zdrojů (paralela s internetem věcí a edge computing). Konkrétně v diabetu, jak už bylo zmíněno, je centralizované řešení použitelné jen složitě a globální řešení ani neexistuje, na druhou stranu ale pro řešení problému lze využít zařízení, které už pacient s určitou pravděpodobností stejně používá.

2.4 Metaheuristiky

Problémy, které se v reálném světě vyskytují, jsou často nelineární, komplexní a najít řešení analyticky by buď trvalo neprakticky dlouho, nebo pro to neznáme způsob. Metaheuristiky nabízí způsob, jak získat poměrně jednoduše a efektivně řešení pro širokou třídu problémů, která sice zpravidla nejsou optimální, ale jejich kvalita je dostatečně vysoká a dokáží si poradit s nepřesnostmi a nejistotami ve vstupních datech [27].

Metaheuristiky fungují iterativně – v každém kroku využijí dosavadní nejlepší řešení (nebo jejich množinu) k posunutí směrem k lepšímu řešení, které se stává výchozím pro další krok. Počáteční řešení může být zvoleno zcela náhodně, nebo, pokud existuje nějaká apriori znalost o daném problému, i jinou metodou v kontextu problému. Konec algoritmu může nastat, pokud už je nalezené řešení dost dobré nebo pokud už proběhlo zvolené množství iterací. Podle toho, jakým způsobem se v jednotlivých krocích dospěje k novému řešení, lze algoritmy dělit na založené na trajektorii (*trajectory-based*) a založené na populaci (*population-based*) [23].

Metody založené na trajektorii pracují vždy jen s jedním výchozím řešením, které je během každého kroku nahrazeno jiným (lepším) řešením, nacházejícím se v okolí výchozího. Také jsou nazývané *exploitation-oriented methods*, jelikož místo prozkoumávání stavového prostoru cílí na co nejlepší využití bezprostředního okolí. Tyto algoritmy jsou zpravidla rychlejší na konvergenci a úspornější z hlediska zdrojů než populační algoritmy (viz dále), ale za cenu větší závislosti na počátečním řešení, které v podstatě určuje, k jakému lokálnímu optimu bude metoda konvergovat. Tuto vlastnost se snaží zmírnit *multi-trajectory-based* algoritmy, které po nalezení lokálního minima hledání restartují v novém výchozím bodě.

Oproti tomu metody založené na populaci mezi iteracemi udržují potenciálních řešení hned několik. Během výpočtu dochází k modifikacím, případně křížení jednotlivých řešení mezi sebou podle stanovených pravidel. Tím je oproti metodám založeným na trajektorii docíleno prohledávání mnohem větší části stavového prostoru a proto se těmto metodám také říká *exploration-oriented*. Velké množství těchto algoritmů se inspiruje u přírodních procesů.

Metaheuristiky lze dále dělit například podle toho, jakým způsobem konstruují nová řešení (*local search-based* – prohledáváním okolí aktuálních řešení / *constructive* – jiným specifickým způsobem podle vlastností aktuálních řešení), a zda si ukládají historii průběhu (*memory-based*), nebo je každá iterace na předchozích krocích zcela nezávislá (*memory-less*) [27].

Řešení, získané jakoukoliv metodou, resp. jeho kvalitu, je potřeba vhodně kvantifikovat. Pro tento účel slouží různé *kriteriální funkce* (*objective functions*), často v podobě metriky, které číselně vyjadřují, jak velké chyby bychom se dopustili použitím nalezeného řešení oproti skutečnosti nebo ideálnímu stavu. Jednoduchou metrikou může být například průměrná chyba nebo maximální chyba. Volba metriky je závislá na konkrétní aplikaci – v případě diabetu je například podstatné, kolikrát byl pacient ve stavu hypoglykémie, o čemž jednoduchý průměr nic nevypovídá. Používanými metrikami jsou například *Time-in-Range*, udávající dobu, po kterou měl pacient určité množství glukózy v krvi, *RMSE* (Root Mean Square Error), což je odmocnina z průměru umocněných chyb nebo *variační koeficient*, což je podíl směrodatné odchylky a aritmetického průměru.

2.4.1 Metaheuristiky založené na trajektorii

2.4.1.1 Hill climbing

Hill climbing [28] je asi nejjednodušší způsob hledání (lokálního) optima. V jednom bodě stavového prostoru postupně zkouší odlišná řešení, nacházející se v blízkosti stávajícího, a za nové řešení je vybráno to s lepší metrikou. Jedná se o extrémně jednoduchý algoritmus, ale je vhodný pro diskrétní problémy nebo spojité problémy s malou dimenzí stavového prostoru, jinak je hledání sousedů obtížné.

2.4.1.2 Simulated annealing

Simulated annealing [29] je metaheuristika inspirovaná postupným chladnutím materiálů – po roztavení se atomy přesouvají do jiných poloh, ale jak materiál chladne, postupně se ustálí ve stavu, který odpovídá minimální energii.

Algoritmus je velmi podobný jako v případě hill climbing, ale s tím rozdílem, že je nové řešení zvoleno náhodně. Pokud je vybrané řešení lepší, je přijato. Pokud není, algoritmus ho může stejně přijmout s určitou pravděpodobností, která exponenciálně klesá s tím, jak špatné řešení to je:

$$P(\text{přijetí horšího řešení}) = 1 - exp(\Delta E/kT),$$

kde ΔE představuje rozdíl mezi původním a novým řešením, k je konstanta vázající teplotu s energií (Boltzmannova konstanta) [30], ačkoliv např. v [31] tento člen není uveden, a T je teplota. Na začátku algoritmu je uvažována vysoká teplota a tím je

umožněno častěji přijímat horší řešení, ale jak „materiál“ chladne, nakonec už se může pohybovat pouze směrem k nižší energii (tj. lepšímu řešení). To algoritmu dává šanci nalézt i jiné než nejbližší lokální optimum. Zásadní přitom je, jak rychle simulovaná teplota klesá (jak dlouho může algoritmus akceptovat i horší řešení).

2.4.1.3 Gradientní sestup

Gradientní sestup [32] vychází z faktu, že gradient funkce více proměnných udává směr největšího růstu. V opačném směru zákonitě platí, že dochází k nejrychlejšímu poklesu. Algoritmus tedy pro současné řešení zjistí gradient a nové řešení stanoví tak, že to současné posune o stanovenou délkou kroku ve směru záporného gradientu. Podstatné je zvolit vhodnou velikost kroku – pokud by měla být příliš velká, algoritmus může divergovat, naopak při příliš malých krocích bude konvergovat, ale jen pomalu.

Tato metoda je v současnosti často využívána pro trénování neuronových sítí, i když spíše v upravených variantách, které do výpočtu zavádí náhodu, setrvačnost, proměnlivou velikost kroku, a snižují výpočetní nároky, jako je *stochastický gradientní sestup* a *gradientní sestup s momentem* [33].

Aby bylo možné gradientní sestup využít, optimalizovaná funkce musí být z principu diferencovatelná.

2.4.2 Metaheuristiky založené na populaci

2.4.2.1 Genetický algoritmus

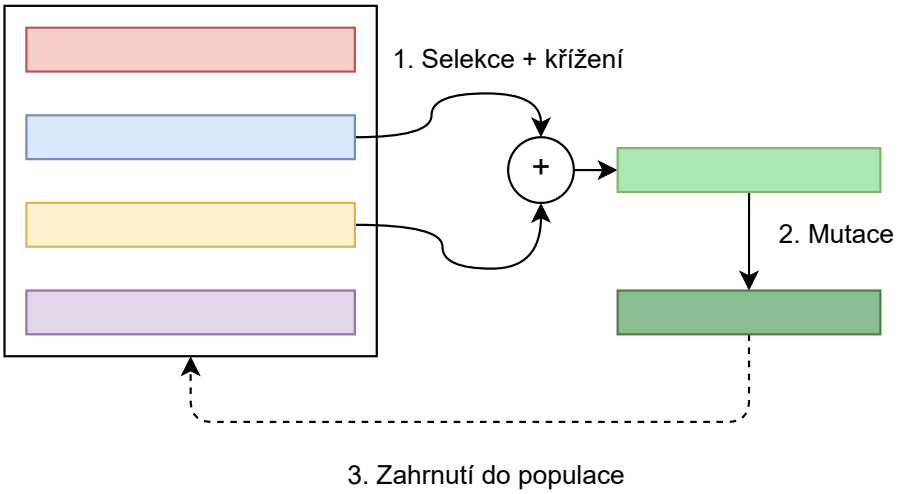
Genetický algoritmus [34] je podtypem evolučních algoritmů. Pro optimalizaci využívá (typicky konstantně početnou) populaci řešení, ve které je každý jednotlivec charakterizován svým genetickým kódem, což může být binární řetězec, nebo, především pro spojité problémy, pole reálných čísel.

Algoritmus nad populací iterativně provádí selekci (výběr rodičů), křížení (vytvoření potomků z rodičů) a mutaci (náhodné změny). Do nové iterace mohou postoupit buď všichni noví jedinci, nebo jen ti, kteří jsou lepší, než rodiče.

Na barvách tento proces demonstruje obrázek 2.4.

2.4.2.2 Diferenciální evoluce

Diferenciální evoluce [35] patří stejně jako genetický algoritmus mezi evoluční algoritmy a je pro ní společná selekce, křížení a mutace. Oproti genetickému algoritmu rovnou uvažuje parametry z oboru reálných čísel a křížení provádí vektorově – v základní podobě je nový zkušební vektor (*trial vector*) zkonstruován s využitím existujícího řešení, ke kterému je přičten (vektorový) rozdíl dvou odlišných řešení, ale možností je více.



Obrázek 2.4: Průběh genetického algoritmu.

2.4.2.3 Particle swarm optimization

Particle swarm optimization [36] se insipiruje chováním shluku zvířat (např. hejna ptáků). Každý jedinec populace (*particle* – částice) udržuje jedno řešení problému a svou rychlost. V každé iteraci algoritmu se jedinci stavovým prostorem posunou podle své rychlosti, jež je ovlivněna nejlepším řešením, které objevil každý sám, a globálním nejlepším řešením, které je společné pro celou populaci.

Prakticky to znamená vypočítat pro každého jedince novou rychlost podle vztahu:

$$v_{t+1} = w \cdot v_t + c_1 \cdot r_1 \cdot (B_{local} - X_t) + c_2 \cdot r_2 \cdot (B_{global} - X_t), \tag{2.1}$$

kde r_1 a r_2 jsou náhodná čísla mezi 0 a 1, w, c_1, c_2 jsou parametry optimalizace (setrvačnost, lokální ovlivnění, globální ovlivnění), X_t je aktuální pozice a B_{local} s B_{global} je lokální, resp. globální nejlepší řešení. Vypočtená rychlost jedince je následně přičtena k jeho pozici.

2.4.2.4 Artificial bee colony

Artificial bee colony [36] je metaheuristika napodobující hledání potravy včelami. Včelí kolonie rozlišuje tři druhy včel: zaměstnané (*employed*), přihlížející (*onlookers*), a výzvědné (*scouts*). Hledanou potravou je v tomto případě řešení optimalizačního problému.

Zpočátku jsou výzvědnými včelami nalezena náhodná řešení, která se pokusí vylepšit hledáním okolí – tím se z nich stanou zaměstané včely. Zároveň jsou lokace sděleny přihlížejícím včelám, které se podle vhodnosti konkrétního řešení rozhod-

nou, ke kterému se vydají, a následně se také pokusí v jeho okolí najít řešení s lepšími vlastnostmi. Pokud při lokálním prohledávání včela zdroj potravy „vyčerpá“, tj. po určitou dobu řešení nevylepší, stává se vyzvědačem a nové řešení stanovuje náhodně [37].

2.5 SmartCGMS

SmartCGMS [38][39][40] je framework pro obecnou analýzu a zpracování signálů, využívaný pro monitoring a léčbu diabetického pacienta, vyvíjený na Katedře informatiky a výpočetní techniky Fakulty aplikovaných věd Západočeské univerzity v Plzni. Všechny prvky systému, se kterými nástroj pracuje, jsou abstrahovány do podoby modulů, představujících dílčí kroky zpracování naměřených dat, které mohou být implementovány prakticky libovolně a je tak možné jednoduše zaměňovat a testovat čistě virtuální prvky za skutečná zařízení a data.

Framework se inspiruje architekturou High-Level-Architecture (HLA), která uvažuje množství nezávislých segmentů, které o sobě vzájemně nemusí nic vědět. Pro konkrétní segment jsou podstatné jen předem standardizované datové proudy, přes které probíhá výměna dat. Tento model se vším všudy byl ale příliš složitý pro efektivní implementaci na méně výkonných zařízeních, kvůli čemuž byla architektura zjednodušena na simplexní komunikaci (*pipe* – roura). Tato roura slouží jako blokující synchronizační primitivum mezi filtry, které běží každý ve vlastním vlákně.

V rámci dalších optimalizací pro méně výkonná zařízení byla architektura modifikována tak, že umožňuje synchronní předávání zpráv, čímž je redukován počet vláken (již neplatí, že co filtr, to samostatné vlákno) a tím pádem i přepínání do kontextu operačního systému [41]. Filtr si vlastní vlákno může vytvořit, pokud potřebuje vykonat náročnější výpočet.

Komunikace mezi filtry probíhá pomocí zpráv (nazývané *device event*). Tato struktura obsahuje identifikační pole, synchronizační značky a podle typu zprávy samozřejmě konkrétní data. Filtr může zprávu přečíst, modifikovat, nebo zahodit.

Entity definované ve frameworku SmartCGMS:

- Filtry – základní stavební blok, představující dílčí část výpočtu.
- Modely – matematické popisy lidského metabolismu, částečně popsáno v kapitole 2.2. SmartCGMS obsahuje předpřipravený např. Bergmanův rozšířený minimální model, nebo modely UVA/Padova S2013 a S2017.
- Signály – obecný kontejner udržující časově uspořádané hodnoty.
- Metriky – (statistické) nástroje pro porovnávání rozdílů mezi dvěma signály, např. průměrná chyba, maximální chyba, apod.

- Solvery – optimalizují parametry předaného modelu podle zadané kriteriální funkce.
- Aproximátory – aproximují diskrétní signál.

Díky vhodnému návrhu je možné aplikaci spouštět i na nízkopříkonových zařízeních beze změny zdrojového kódu. Aplikace je také velmi modulární a uživatel si může vyvinout své vlastní prvky v jakémkoli kompilovaném jazyce, podporovány jsou i Matlab skripty.

Řetěz filtrů lze konfigurovat pomocí grafického rozhraní, případně čistě za pomoci příkazové řádky.

Návrh programu 3

3.1 Technické prostředky

3.1.1 Hardware

Jak už bylo řečeno v kapitole o inzulinových pumpách 2.3.1, jejich hardware je, co se výkonu a přípustné spotřeby týče, značně omezený, a tato práce by to měla reflektovat. Zadáním je nicméně vyžadováno, aby výsledné řešení zakládalo na frameworku SmartCGMS, což znamená, že na vybrané platformě musí existovat podpora operačního systému, bez které se SmartCGMS zatím neobejde. Důsledkem bude možná vyšší výpočetní výkon, než by byl pro samotnou inzulinovou pumpu nutně potřeba, z čehož může benefitovat optimalizační algoritmus, ale cenou bude pravděpodobně vyšší spotřeba a menší schopnost ji výrazně ovlivnit, jelikož operační systém uživatele odstiňuje od hardwaru.

Z praktického hlediska by také mělo jít o platformu, kterou lze snadno a v dostatečném množství sehnat „off-the-shelf“, aby bylo možné jednoduše a uniformně program spustit na více zařízeních zároveň a ověřit tím tak koncept spolupráce při řešení společného problému.

Aby spolu zařízení mohla spolupracovat, musí existovat možnost komunikace jednak mezi sebou, a také s centrálním prvkem umístěným v Internetu, přičemž inzulinová pumpa má být mobilní, z principu by se tedy mělo jednat o bezdrátovou komunikaci, typicky ve formě Wi-Fi.

Výběr není příliš široký, k dostání jsou především minipočítače Raspberry a jejich obdoby Radxa s podobnými parametry. Pro účely této práce je asi nejzajímavější Raspberry Pi Zero W: je malé (65 x 30 mm), levné (v době psaní řádově do 500 Kč), obsahuje integrovanou Wi-Fi konektivitu a předpokládá se jeho provoz s distribucí operačního systému GNU/Linux. Navíc je velice rozšířené a se silnou uživatelskou základnou. Zachyceno je na obrázku 3.1.

Obrázek 3.1: Raspberry Pi Zero, zdroj RPishop.cz.

3.1.2 Programovací jazyk

SmartCGMS samotné je napsané v jazyce C++ (s rozhraním v jazyce C) s ohledem na širokou podporu překladačů a výkonnost, a přestože umožňuje vývoj vlastních součástí aplikace v libovolném kompilovaném programovacím jazyce, nemá příliš smysl pro tuto práci hledat jiný jazyk – využito bude opět C++, potažmo C.

3.2 Optimalizační algoritmus

V kapitole 2.4 byly popsány různé metaheuristiky, které lze použít k identifikaci modelu – optimalizaci jeho parametrů. Dvěma základními kategoriemi jsou metaheuristiky založené na trajektorii a metaheuristiky založené na populaci.

Pro tuto práci lze říct, že první jmenovaná skupina nebude nejvhodnější. Je tomu tak proto, že z principu dokáží prohledat jen malou část stavového prostoru a mají omezené možnosti úniku z lokálních optim.

Naproti tomu metaheuristiky založené na populaci dokáží prohledat větší část stavového prostoru naráz a lze si snáze představit jejich současný běh na několika oddělených zařízeních, která si mají o průběhu výpočtu předávat informace.

Pro tuto práci byly vybrány dvě konkrétní: genetický algoritmus a particle swarm optimization.

Genetický algoritmus byl vybrán s ohledem na předchozí autorovu zkušenost s touto metaheuristikou, což znamená možnost použít jen s nutnými úpravami již hotovou implementaci.

Druhá metoda byla zvolena především pro její snadnou uchopitelnost.

3.3 Síťová architektura

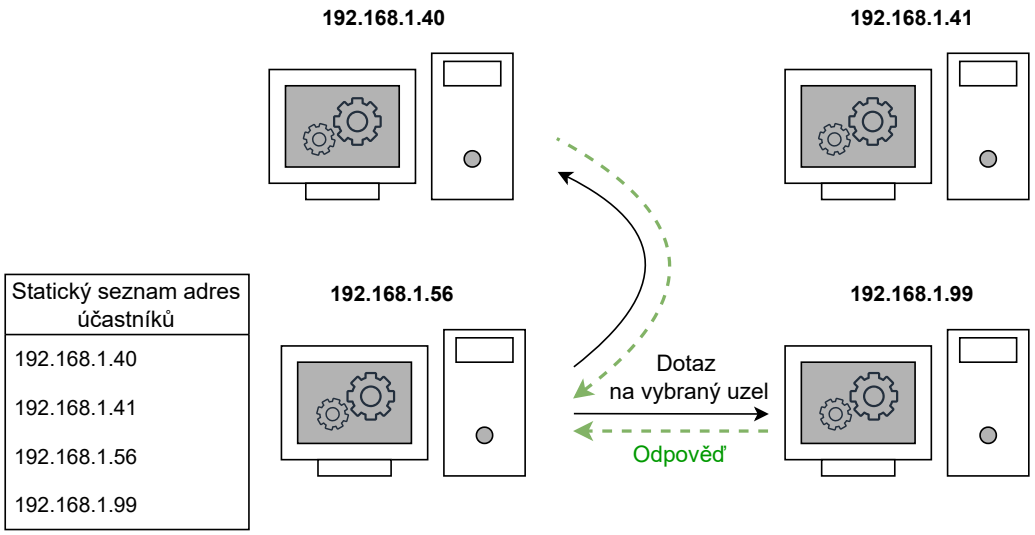
Záměrem je, aby aplikace byla distribuovaná přes potenciálně masivní množství zařízení – s ohledem na množství diabetiků klidně až stovky tisíc účastníků a více. Pro výměnu dat musí mít ale v jednu chvíli jedno zařízení informaci, jak komunikovat s alespoň jedním dalším účastníkem.

3.3.1 Bez centrálního prvku

Ideální by bylo mít aplikaci zcela decentralizovanou, ve které by klienti mezi sebou mohli komunikaci navazovat ad-hoc a neexistoval by žádný *single point of failure* (prvek, jehož selhání zapříčiní selhání celého systému).

3.3.1.1 Statická distribuce adres

Nejjednodušší způsob je pravděpodobně statický: adresy uzlů stanovit napevno a zanést je jako konstanty do programu. Každý uzel pak bude mít přinejmenším omezenou znalost o ostatních účastnících. Je ale zřejmé, že byť je tento způsob bez problému použitelný pro aplikace s několika málo účastníky, v kontextu této práce není příliš vhodný, jelikož adres může být obrovské množství a znát je všechny předem, zajistit jejich statičnost a distribuci před spuštěním výpočtu je nereálné uskutečnit. Graficky je tento přístup zachycen na obrázku 3.2.

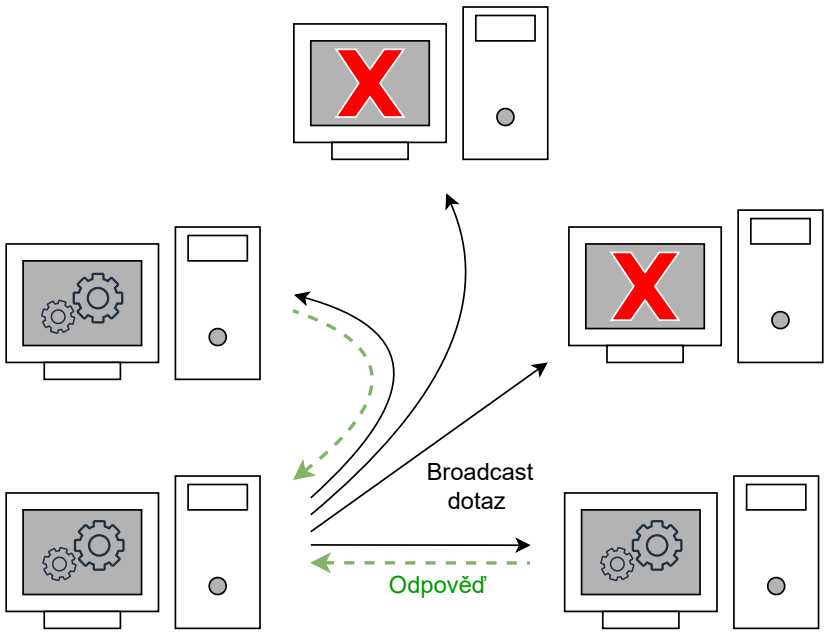


Obrázek 3.2: Statická distribuce adres – každý účastník má pevně daný seznam adres ostatních účastníků.

3.3.1.2 Objevování pomocí broad-/multicast zpráv

Dynamičtějším způsobem by bylo využít broadcast/multicast zprávy. Účastník, který si chce vyměnit znalosti, by s jejich pomocí dokázal během výpočtu najít dalšího účastníka tak, že se zeptá všech, zda se chtějí do výměny připojit. Potřebné směrovací informace jsou získány z přijatých odpovědí. Není tak potřeba řešit adresy předem a účastníci se mohou libovolně připojovat a odpojovat.

Opět se zde ale naráží na problém s množstvím zařízení – nelze předpokládat, že se budou všichni účastníci nacházet na jedné síti, a i kdyby ano, hromadný dotaz by způsobil buď její přetížení, nebo přetížení příjemce, kvůli (potenciálně) velkému množství odpovědí. Konkrétně broadcast zprávy navíc zbytečně zatěžují všechna připojená zařízení bez výjimky, ať už na nich konkrétní aplikace běží, nebo ne. Schematicky to znázorňuje obrázek 3.3.



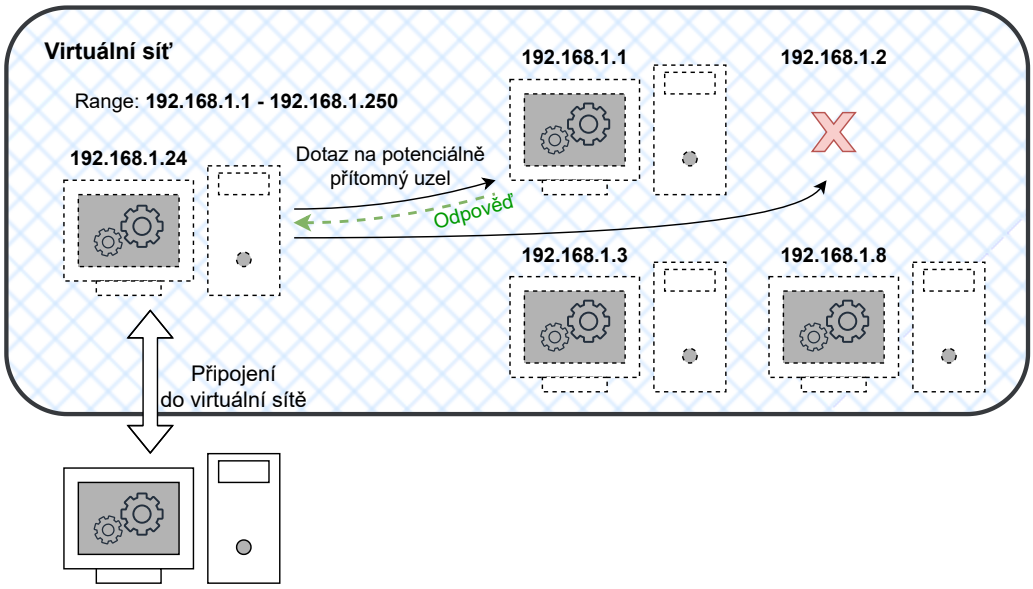
Obrázek 3.3: Objevování pomocí broadcast zpráv bez předchozí znalosti adres. Zprávu musí zpracovat i zařízení, na kterých cílová aplikace neběží.

3.3.1.3 Virtuální síť

Nedá se předpokládat, že budou pumpy fyzicky na stejné síti, ale bylo by možné využít technologii *Virtual Private Network* (VPN) k tomu, aby byly na stejné síti alespoň virtuálně. Adresy na této síti by se přidělovaly dynamicky, ale z předem dohodnutého rozsahu. Tím pádem by pak bylo bez problémů možné zkoušet kontaktovat náhodné adresy v rámci virtuální sítě, zda na nich zrovna není aktivní

účastník a pokud ano, navázat s ním komunikaci. Do určité míry je to podobný přístup jako v odstavci výše, ale VPN v tomto případě zaručuje, že pokud někde nějaká pumpa běží, bude možné s ní na virtuální síti navázat kontakt. Myšlenka je zachycena na obrázku 3.4.

Tato metoda ale přináší i některé očividné problémy, zejména přestože se z hlediska klientů dá mluvit o decentralizovaném řešení, reálně je potřeba mít další síťové vybavení, především VPN server, který bude virtuální síť spravovat, a který znamená (nechtěnou) centralizaci.



Obrázek 3.4: Řešení adresování za pomoci virtuální sítě s deterministickým přidělováním adres z pohledu pumpy.

3.3.2 S centrálním prvkem

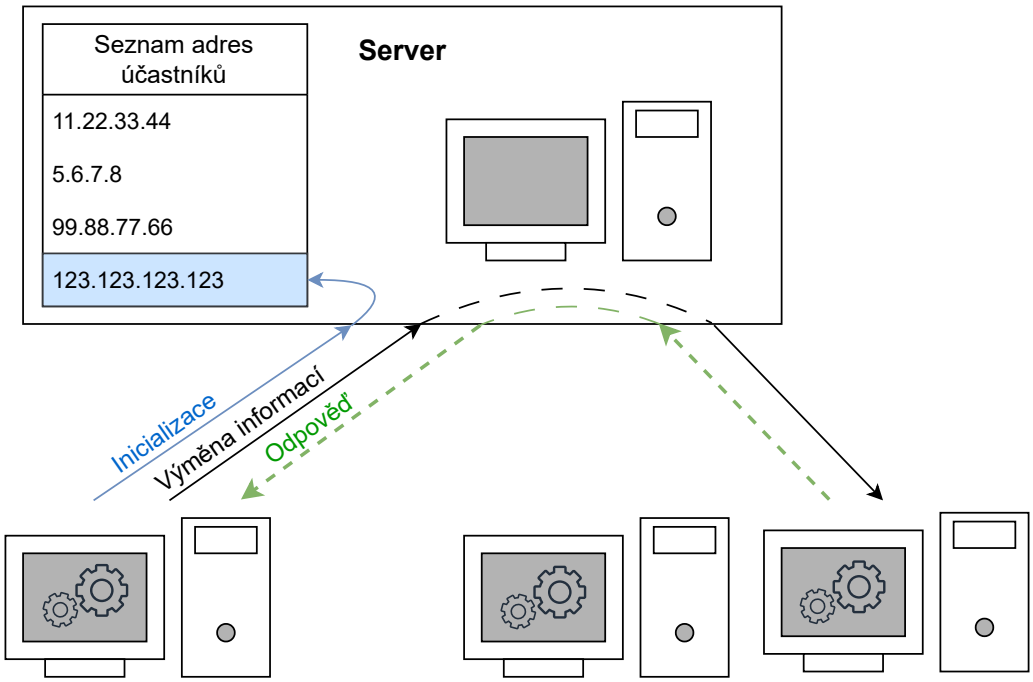
Striktní snaha o decentralizaci nevede k vhodným řešením. Nezbývá než požadavek na decentralizovanost uvolnit a zkusit problém vyřešit za pomoci (explicitního) ustanovení centrálního prvku, který bude komunikaci do určité míry řídit.

Tento centrální prvek už z principu nemůže fungovat se stejným omezeným hardwarem a napájením jako inzulinové pumpy, běžící u pacientů – bude se jednat o konvenční desktop či server, dostupný ideálně nepřetržitě s veřejnou IP adresou. Z toho vyplývá nutnost mít pro běh další (a z hlediska cíle aplikace, kterým je identifikace modelu, vlastně zbytečný) stroj a také obsáhlejší programové vybavení, jelikož kromě programu pro pumpy bude nutno mít specifickou logiku serveru. Je to nicméně přiměřená cena za přímočarou realizovatelnost.

3.3.2.1 Plně centralizovaná aplikace

Jednodušším způsobem z hlediska návrhu by bylo přes server směrovat veškerou komunikaci mezi pumpami. Server by musel znát adresy všech funkčních pump, čehož se dá v tomto případě dosáhnout jen ohlášením pumpy na server během inicializace – adresu odesílatele lze snadno získat z IP paketu.

Z pohledu pump by pak síťová část výpočtu byla výrazně jednodušší, jelikož by je server odstínil od distribuované povahy aplikace – síťovou topologií by byla hvězda se serverem uprostřed, pumpy by vůbec nepotřebovaly znát navzájem své adresy, veškerá komunikace by probíhala vždy jen mezi pumpou a serverem. Tuto koncepci znázorňuje obrázek 3.5.



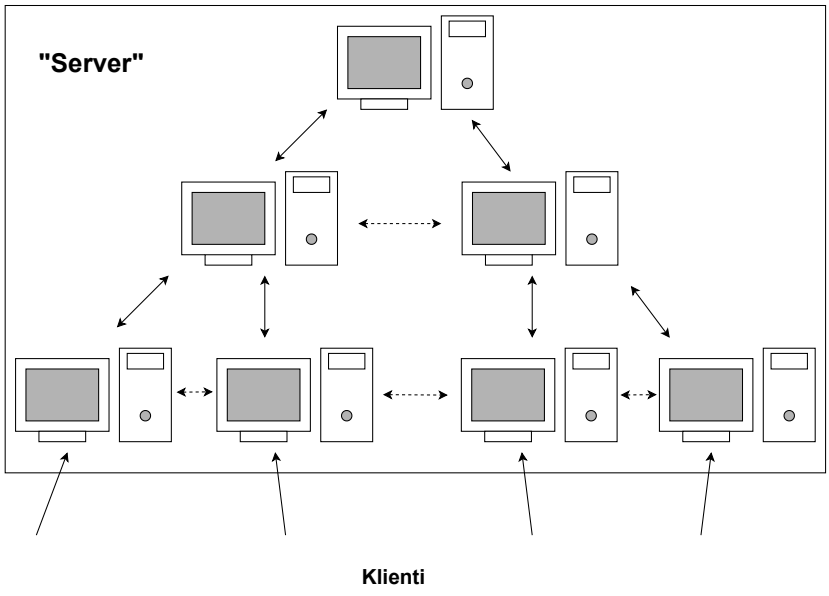
Obrázek 3.5: Využití serveru pro kompletní směrování zpráv mezi pumpami. Pumpa se v rámci inicializace musí serveru ohlásit, čímž server získá její adresu, později použitelnou pro směrování.

Zásadním důvodem, proč je tento způsobu nevhodný, je – opět – počet zařízení, který by se měl výpočtu účastnit – přestože bude server mít oproti pumpám poměrně dost výkonu, lze si představit, že budou statisíce až miliony dotazů příliš. Centralizace navíc nevyhnutelně vnáší do systému součást, jež svým selháním může ohrozit funkčnost celé aplikace. V tomto případě pokud by byl server nefunkční nebo nedostupný, každá pumpa už se může spolehnout jen sama na sebe a pravděpodobnost nalezení kvalitního řešení se snižuje. I to je samozřejmě do určité míry řešitelné replikací serveru, případně zálohováním, což by ideálně vedlo na aplikaci

běžící v cloudovém prostředí, kde je možné infrastrukturu dynamicky škálovat a reagovat tím na aktuální situaci.

3.3.2.2 Struktura serverů

Problémy striktní centralizovanosti by bylo možné eliminovat přidáním dalších strojů a jejich seskupením do předem dané struktury, například stromu. Klienti by mohli mít staticky definované adresy buď jen listů, nebo rovnou všech strojů struktury. Pak by se připojovali výhradně k listům, čímž by došlo k k rozprostření zátěže, a v případě výpadku zvoleného listu k nadřazenému stroji. Samotné stroje, tvořící server, by mohly požadavky od klientů předávat mezi úrovněmi stromu nebo i horizontálně a potenciálně by se tím dal do značné míry eliminovat vliv výpadků. Zjednodušeně to zachycuje obrázek 3.6.

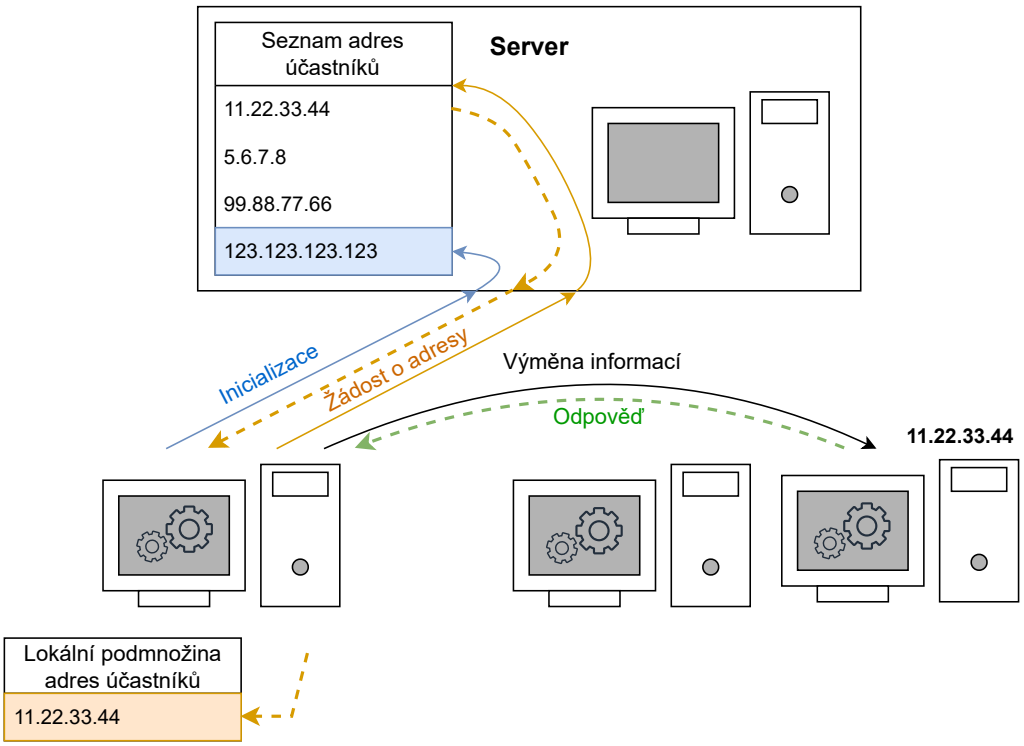


Obrázek 3.6: Server ve formě struktury několika strojů.

Tento přístup je nicméně v mnoha ohledech zbytečně maximalistický a přináší víc problémů, než řeší – namátkově je potřeba netriviální množství „nadbytečných“ výpočetních zdrojů, je nutné v rámci serveru distribuovat informaci, pod kterým konkrétním strojem je jaký klient zrovna připojen, a zprávy korektně směrovat. V kontextu této práce lze rovnou říct, že je uvedený způsob příliš složitý a nemá smysl se jím dále zabývat.

3.3.2.3 Server jako podpůrný prvek

Alternativou a úkrokem stranou by bylo centrální prvek zachovat, ale využít ho jen pro udržování informací o ostatních účastnících. Pumpa by se shodně s předchozí variantou po zahájení činnosti serveru ohlásila a následně by si podle potřeby zažádala o adresy některých ostatních účastníků, se kterými by navázala spojení napřímo, bez účasti serveru, tak, jak to naznačuje diagram 3.7.



Obrázek 3.7: Využití serveru jen pro udržování adres ostatních účastníků, kteří následně komunikují napřímo. Pumpa se v rámci inicializace serveru ohlásí, následně si může od serveru vyžádat určité množství adres dalších účastníků a s těmi komunikovat.

Selhání centrálního prvku je sice stále problém, ale mnohem méně závažný – především pro nově příchozí účastníky, kteří se nebudou schopni spojit s nikým dalším. Samozřejmě i pumpy s již známou podmnožinou ostatních účastníků mohou vlivem dynamické povahy světa (změna adres, trvalé výpadky) o své kontakty pomalu přicházet, ale lze logicky předpokládat, že bude server v konečné době opět zprovozněn a problém eliminován.

Ještě o krok dále by bylo zavedení možnosti zjišťovat adresy přímo od dalších účastníků, kdy by byl server potřeba jen pro inicializaci pumpy a poskytnutí prvních několika adres, následně by si pumpa rekurzivním dotazováním známých účastníků dokázala v extrémním případě obstarat kompletní sadu adres.

3.3.3 Shrnutí

Jako nejlepší možnost se jeví využití serveru v podpůrné roli jako „broker“ adres – nebude se nijak účastnit samotné výměny optimalizačních informací, čímž je výrazně omezen dopad jeho potenciálního selhání. Aplikace se dá vlastně rozdělit na dvě části:

- Zjišťování adres ostatních účastníků, ve které figuruje jeden účastník a server.
- Výměna znalostí o optimalizaci, které se účastní podle rozdaných adres v podstatě náhodná podmnožina všech zařízení kromě serveru (peer-to-peer).

Podstatné je, že jakmile má účastník adresy ostatních zařízení, funkčnost serveru už na optimalizaci nemusí nutně mít žádný vliv.

Toto uspořádání také může výrazně napomoci průchodu přes překladače adres (viz kapitola 3.4) a server nakonec nemusí udržovat jen samotné adresy, ale i další informace, které mohou být přínosné pro úspěšnost optimalizace.

3.4 NAT traversal

U navrhované aplikace se očekává posílání zpráv přes Internet, bude tedy využívat IP zásobník, konkrétně ve verzi 4, která je dnes stále dominantní a globálně podporovaná. Přestože jeho poslední verze IPv6 je specifikována již zhruba od počátku milénia, její adopce neprobíhá příliš prudkým tempem (podle dat Google konkrétně v České republice využívá IPv6 zhruba 25 % zařízení v Internetu a přibližně 40 % celosvětově [42]). Problémem je množství adres (2^{32} = 4294967296, z nichž je navíc část vyhrazena pro speciální účely), které IPv4 poskytuje, a které je již delší dobu nedostatečné. Místo přechodu na novější IPv6 je jedním z částečných řešení zavedení tzv. *Network Address Translation* (NAT). NAT je prvek sítě, který mění IP adresy v hlavičkách paketů, případně obecnější (a z pohledu této práce častější varianta) *Network Address/Port Translation* (NAPT) mění s IP adresami i porty (dále bude zkratkou NAT označován NAPT) a je uvažován provoz nad protokoly TCP/UDP, které pro identifikaci konkrétního cílového procesu používají číslo portu.

Paket směřující ven ze sítě obsahuje zdrojovou adresu/port a cílovou adresu/-port. NAT při překladu nahradí zdrojovou adresu svou vlastní adresou a zdrojový port je nahrazen nějakým jiným podle nastavení a funkcionality konkrétního NATu. Nový port je opět podle nastavení NATu na určitou dobu výhradně přiřazen dvojici zdrojová adresa/port – pokud uplyne daná doba, aniž by došlo k další výměně zpráv, port je uvolněn. Při příchodu paketu z vnějšku na základě cílového portu NAT zjistí, jaký stroj v podlehlé síti je skutečným adresátem, jeho adresu umístí včetně patřičného správného portu do paketu, a přepošle ho do podlehlé sítě. Tím je možné, aby se celá jedna síť (využívající typicky privátní IP adresy) zvenku tvářila

jako jediný stroj s plnohodnotnou veřejnou adresou. To umožňuje šetřit „cennými“ veřejnými adresami, ale zároveň je patrná komplikace, ke které dojde, pokud má být komunikace iniciována z Internetu směrem do privátní sítě – adresa stroje za NATem nemá z hlediska vnějšího světa žádný význam, vzdálený uzel může znát adresu NATu, ale ke směrování chybí port.

Jedním řešením může být v nastavení NATu trvale přiřadit konkrétní vnější port konkrétní vnitřní adrese a portu (tzv. *port forwarding*). Nelze ale od běžného netechnického uživatele očekávat, že bude schopný si sám zasáhnout do nastavení své sítě, leckdy nastavení ani měnit nelze (firemní sítě). Navíc nejde jen o NAT, staticky přiřazené musí být v takovém případě také IP adresy.

Dynamičtějším řešením je koncept tzv. *hole punching*, pro který je ale potřeba prostředníka – server s veřejnou adresou. Klient A zpoza NATu se před peer-to-peer provozem k serveru obvyklou cestou připojí. Serveru už zpráva přijde z přeložené adresy a portu, kterou si uloží a zároveň má možnost kontaktovat daného klienta A zpětně, jelikož už byla na NATu klienta A vytvořena se serverem relace. Stejnou proceduru provede každý klient. Pokud chce následně nějaký jiný klient B kontaktovat napřímo klienta A, vyžádá si asistenci serveru. Server klientu B poskytne (veřejnou) adresu klienta A a zároveň klientovi A pošle informaci o požadavku na spojení společně s adresou klienta B. Oba klienti na předané adresy protějšku začnou odesílat zprávy, čímž dojde na NATech v cestě k vytvoření relací pro nové spojení a následné komunikaci nic nebrání. Podstatné je nicméně mít na paměti, že po delší době neaktivity může NAT relaci zrušit a chování NATů se může mezi výrobci lišit. Techniku hole punching lze uplatnit jak na UDP, tak na TCP provoz, přestože v případě TCP je nutná podpora operačního systému. Více do hloubky se problematice věnuje článek [43]. Článek také upozorňuje na problém různorodosti NATů, což může způsobit, že implementace obcházení NATu v některých sítích funguje a v jiných ne, a navrhuje vlastnosti „P2P-Friendly“ NATu. Podobná doporučení pro výrobce obsahují také RFC od Internet Engineering Task Force, např. RFC4787 [44].

Navrhovaná aplikace bude zčásti využívat architekturu klient-server, kde NAT nezpůsobí žádné problémy (od serveru se veřejná adresa očekává a iniciátorem komunikace bude klient), ale větší část komunikace se bude odehrávat přímo mezi klienty navzájem, což může právě s ohledem na NAT přinášet komplikace. U klientů totiž dává smysl očekávat, že se budou nacházet za NATem – především na malých domácích sítích se běžně využívají privátní adresy, které ani jinak nejsou v Internetu směrovatelné.

Pokud by měla být technika hole punching implementována, výhodou je, že návrh aplikace už předem počítá s vyhrazeným veřejným serverem, který má sdílet adresy mezi účastníky. Bylo by akorát potřeba rozšířit funkcionalitu serveru o zasílání upozornění účastníkovi, jehož adresa byla poskytunuta někomu dalšímu, a na pumpách by výměně informací muselo předcházet ověření úspěšného navázání

komunikace.

Jak už bylo nicméně zmíněno, IP verze 4 byla již před nezanedbatelnou dobou překonána verzí 6, která kromě dalších úprav používá 128 bitové adresy namísto 32 bitových a poskytuje tak adresní prostor velikosti dostatečné pro opětovné přidělování veřejných a v Internetu směrovatelných adres každému připojenému zařízení. Přímá end-to-end konektivita je i jedním z principů Internetu, překládání adres mělo sloužit jen jako krátkodobé řešení před nástupem nové verze protokolu. Jediným problémem je relativně pomalý nástup IPv6. Každopádně za optimistického předpokladu, že IPv6 v (blízké) budoucnosti nahradí IPv4, bude problém s průchodem přes překladače adres vyřešen automaticky a v této práci není potřeba se jím explicitně zabývat.

3.5 Návrh komunikace

3.5.1 Omezení

Síť bude tvořit jen jakýsi doplněk k identifikaci modelu, což je numerický proces, který na slabších zařízeních (kterými inzulinové pumpy bezpochyby jsou) v závislosti na velikosti modelu zabere značné množství času (přinejmenším hodiny).

Výměnu zpráv lze realizovat prakticky kontinuálně, ale přijatá řešení je nutné také zpracovat – v extrému by to vedlo na situaci, kdy pumpy všechen svůj výpočetní čas stráví komunikací, místo toho, aby prováděly optimalizaci.

Během identifikace se zároveň předpokládá, že se lokální populace postupně přibližuje zajímavým bodům ve stavovém prostoru. V případě metaheuristik založených na populaci může být „zajímavých“ jedinců více najednou, každý s trochu odlišným řešením. Při výměně informací s dalšími pumpami tak není žádoucí ani získat nárazově velké množství parametrů, protože jejich začlenění do (konstantně velké) populace by mohlo znamenat nahrazení slibných jedinců zcela nevhodnými. Těžko lze navíc z místa porovnat kvalitu řešení – v jednom čase to samozřejmě lze porovnáním výsledků kriteriální funkce, ale v průběhu optimalizace výsledek nelze čekat konstantní.

Dalším podstatným limitujícím faktorem a důvodem, proč intenzitu komunikace držet při zemi, je spotřeba energie, která je úzce svázaná s výpočetním výkonem. Navíc i bezdrátový přenos přes Wi-Fi sám o sobě spotřebovává nezanedbatelné množství elektrické energie.

3.5.2 Protokol

Všechny výše zmíněné úvahy vedou na nespojovaný protokol s minimální režií, což případě Internetu znamená využití UDP na transportní vrstvě. S malou režií se ale pojí i nespolehlivost – UDP negarantuje doručení zpráv, zachování pořadí, a zprávy

mohou být při průchodu síti duplikovány. Tyto nedostatky budou adresovány definováním protokolu na aplikační vrstvě.

Z hlediska složitosti by bylo ideální mít protokol bezestavový, odpadla by pak nutnost řešit návaznost zpráv a do značné míry by to eliminovalo problémy s možným duplikováním. Klienti se ale mají u serveru při započetí výpočtu ohlásit a navázat relaci, což stavovost z principu vyžaduje. Stavy stačí nicméně jen dva – klient nepřipojen, klient připojen. Pro dosažení požadované funkcionality jsou potřeba následující zprávy:

- Inicializace – vytvoření relace se serverem, čímž dojde ke změně stavu z „nepřipojeno“ do „připojeno“.

  Odpovědí je identifikační číslo, kterým se klient prokazuje ve všech ostatních zprávách.

- Žádost o poskytnutí adres – připojený klient si může vyžádat adresy na některé další klienty.

  Odpovědí je pole adres a portů.

- Ping – periodické udržování relace připojeného klienta se serverem. Odpověď pouze potvrzuje úspěšný přenos.

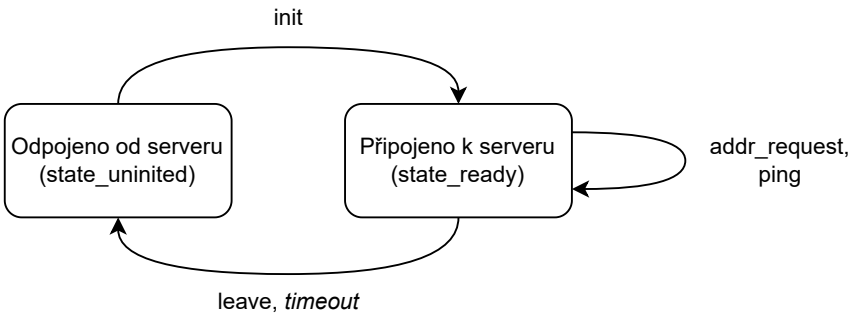
- Konec – připojený klient končí svou činnost, dojde ke změně stavu z „připojeno“ do „nepřipojeno“. Odpověď pouze potvrzuje úspěšný přenos.

U klientů je potřeba brát v potaz, že může dojít k výpadkům. Z toho důvodu je vhodné zavést zprávu typu *ping*, kterou klient serveru odešle, pokud by delší dobu nepotřeboval poskytnout adresy. V případě dlouhé nečinnosti (timeout) je připojený klient přesunut do stavu „nepřipojen“. Zpráva o explicitním ukončení činnosti ze strany klienta pak teoreticky není potřeba, ale s její pomocí lze proces odpojení výrazně urychlit a tím zajistit, že méně klientů v případě žádosti obdrží adresu již neaktivního klienta.

Přechody mezi stavy jsou graficky znázorněny na obrázku 3.8. Identifikátory jsou uvedeny tak, jak jsou definované v implementaci.

Při komunikaci mezi klienty se žádné apriori navázání spojení neočekává. Jakýkoliv klient může v podstatě kdykoliv přijmout požadavek o zaslání jeho řešení, což udělá, a poté je celá konverzace zapomenuta. Pokud žadatel odpověď neobdrží, zopakuje požadavek. Dotázaný může přitom odpovědět jiným řešením, aniž by to mělo zásadní dopad. Pro dosažení funkčnosti stačí jeden typ zprávy a její odpověď:

- Žádost o poskytnutí řešení – libovolný klient si může vyžádat parametry řešení. Odpovědí je jedno samostatné řešení (sada parametrů).



Obrázek 3.8: Stavy a přechody mezi nimi při komunikaci mezi klientem a serverem.

Kompletní tvar zpráv je zachycen v kódu 3.1. Typ zprávy je daný lidsky čitelným textovým řetězcem, zatímco samotná data jsou binární. Pokud se jedná o odpověď, předchází typu zprávy potvrzovací příznak. Každá zpráva navíc obsahuje hlavičku s pevně stanoveným textem pro případné odhalení zpráv, které nepatří implementované aplikaci, a patičku v podobě kontrolního součtu CRC32 pro odhalení chyb při přenosu. UDP nicméně ve své hlavičce již používá dvoubajtový kontrolní součet, který pro kratší zprávy bohatě dostačuje – bylo by tak možné nadbytečný CRC32 v některých typech vypustit.

Pokud by parametrů mělo být hodně (vyšší desítky a více) a byly by přenášeny ve formátu double precision, jehož velikost je 8 bajtů, velikost zprávy by mohla teoreticky vyžadovat fragmentaci, jelikož maximální velikost IP packetu, pro kterou je garantováno doručení bez fragmentace, je stanovena v RFC791 [45] na zhruba 500 bajtů. Fragmentace nemusí být všemi prvky sítě podporována, je tedy lepší fragmentaci řešit v případě velkých zpráv na úrovni aplikační vrstvy. Pro jednoduchost je zde předpokládáno, že zprávy nebude třeba fragmentovat.

Protokol používá pouze kladná potvrzování

Zdrojový kód 3.1: Definice formátu zpráv v jazyce ASN.1

```
DPProtocolDef DEFINITIONS AUTOMATIC TAGS ::= BEGIN

    IPv4_Endpoint ::= SEQUENCE {
        address INTEGER (0..4294967295),
        port    INTEGER (0..65535)
    }

    Packet_Body ::= CHOICE {
        init         SEQUENCE {
            name     VisibleString ("INIT")
        },
        address_request SEQUENCE {
            name     VisibleString ("ARQ"),
            id       INTEGER,
            amount   INTEGER (0..255)
        },
        ping         SEQUENCE {
            name     VisibleString ("PING"),
            id       INTEGER
        },
        leave        SEQUENCE {
            name     VisibleString ("END"),
            id       INTEGER
        },
        param_request   SEQUENCE {
            name     VisibleString ("PRQ")
        }
    }

    Response_Packet_Body ::= CHOICE {
        init_response   SEQUENCE {
            name     VisibleString ("INIT"),
            id       INTEGER
        },
        address_request_response    SEQUENCE {
            name     VisibleString ("ARQ"),
            amount   INTEGER (0..255),
            addresses   SEQUENCE OF IPv4_Endpoint
        },
        ping_response   SEQUENCE {
            name     VisibleString ("PING")
        },
        leave_response  SEQUENCE {
            name     VisibleString ("END")
        },
        param_request_response  SEQUENCE {
            name     VisibleString ("PRQ"),
```

```
48              amount   INTEGER,
49              params   SEQUENCE OF INTEGER
50          }
51      }
52
53      Packet ::= SEQUENCE {
54          header   VisibleString ("DBPUMP"),
55          body     Packet_Body,
56          crc32 INTEGER (0..4294967295)
57      }
58
59      Response_Packet ::= SEQUENCE {
60          header   VisibleString ("DBPUMP"),
61          ack      VisibleString ("OK"),
62          body     Response_Packet_Body,
63          crc32    INTEGER (0..4294967295)
64      }
65
66  END
```

Popis implementace 4

Zdrojové kódy jsou rozděleny na dvě části: implementace solveru pro framework SmartCGMS a implementace síťové komunikace.

4.1 SmartCGMS solvery

Implementace SmartCGMS solverů vychází z příkladu, který je uveřejněn na stránkách dokumentace projektu [46].

4.1.1 Deskriptory

Veškeré entity jsou do SmartCGMS přidávány ve formě dynamických knihoven. O této knihovně program při spuštění neví vůbec nic – aby bylo možné je správně rozpoznat a identifikovat, dynamická knihovna musí exportovat funkci `do_get_X_descriptors()`, kde X značí konkrétní entitu, v tomto případě solver. Tato funkce volajícímu předává ukazatele na pole struktur `TSolver_Descriptor`. Tato struktura, definovaná ve SmartCGMS SDK, udržuje konkrétní informace o implementovaném solveru, které jsou následně zobrazovány uživateli. Jedná se o globálně unikátní identifikátor GUID (resp. UUID) [47], lidsky čitelný textový řetězec s názvem, indikaci, zda je solver specializovaný pro konkrétní modely a pokud ano, množství použitelných modelů a ukazatele na jejich GUID.

Dále je potřeba konkrétně v případě solveru ještě exportovat funkci `do_solve_generic()`. Tato funkce je volána pro samotnou optimalizaci parametrů. Volanému je předáno GUID vybraného solveru a ukazatele na struktury `TSolver_Setup` a `TSolver_Progress`. První jmenovaná obsahuje všechny potřebné údaje pro běh solveru, především počet parametrů a odkaz na kriteriální funkci. Druhá jmenovaná slouží pro komunikaci solveru s okolním světem – skrz ní je předáváno výsledné (optimalizované) řešení, úroveň rozpracovanosti a příznak, zda se má solver ukončit. Funkce samotná jen zkontroluje, že je požadovaný solver skutečně součástí knihovny a pokud ano, spustí ho – konkrétně vytvoří instanci jeho třídy (`CDB_PSO_Solver` nebo `CDB_GE_Solver`), které předá `TSolver_Setup`, a zavolá její

metodu `Solve()`, které předá `TSolver_Progress` (dále popsány v kapitolách 4.1.3 a 4.1.4).

Naprostá většina výše uvedeného kódu je umístěna v `descriptor.cpp`, `descriptor.h` je využit pouze pro definování GUID implementovaných solverů. Tato GUID bylo vygenerována zcela náhodně online nástrojem.

4.1.2 Struktury particle swarm optimalizace

Paměťové struktury pro particle swarm optimalizaci jsou definovány v hlavičkovém souboru `pso_structs.h`. Uvedené názvy parametrů odpovídají těm, které byly uvedeny v rovnici 2.1 v kapitole 2.4.2.3 při popisu této metody optimalizace.

Struktura `pso_particle` představuje jednu částici. Kromě polohy a rychlosti si udržuje také nejlepší řešení. Její definici zachycuje zdrojový kód 4.1.

Zdrojový kód 4.1: Definice struktury `pso_particle`

```
typedef struct {
    /* Nejlepsi reseni tohoto particle */
    /* parametr B_local */
    double *my_best;
    /* Hodnota fitness pri pouziti reseni my_best */
    double my_best_fitness;
    /* Aktualni reseni - parametr X_t */
    double *current;
    /* Aktualni rychlost - parametr v_t */
    double *speed;
} pso_particle;
```

Struktura `pso_parameters` udržuje proměnné, které jsou při výpočtu společné pro všechny částice, tj. parametry w, c_1 a c_2 uvedené v rovnici 2.1 a navíc počet parametrů jednoho řešení, díky kterému lze určit délku polí, na které ukazuje `pso_particle`. Definice je uvedena v kódu 4.2.

Zdrojový kód 4.2: pso_structs.h

```
typedef struct {
    /* Setrvacnost - parametr w */
    double momentum;
    /* Lokalni ovlivneni - parametr c_1 */
    double local_influence;
    /* Globalni ovlivneni - parametr c_2 */
    double global_influence;
    /* Pocet parametru jednoho reseni */
    const size_t dimension;
} pso_parameters;
```

4.1.3 Logika particle swarm optimalizace

Vlastní particle swarm optimalizace je předmětem zdrojových souborů `distributed_pso_solver.cpp` a `distributed_pso_solver.h`.

V hlavičkovém souboru je umístěna jen definice třídy `CDB_PSO_Solver`, která obaluje většinu výkonného kódu. V rámci třídy jsou kromě jiného definovány defaultní hodnoty pro inicializaci optimalizace a pomocné řídicí proměnné pro výměnu dat se síťovou částí. Solver si také udržuje referenci na předanou strukturu `TSolver_Setup`.

4.1.3.1 Funkce `solve()`

Tato funkce je vstupním bodem, volaná aplikací SmartCGMS přes `descriptor.cpp`, popsaný v kapitole 4.1.1. V první části je provedena inicializace veškerých potřebných proměnných – na základě předaného `TSolver_Setup` nastaví počet generací, velikost populace, nechá alokovat a inicializovat struktury particle swarm optimalizace, a v neposlední řadě založí nové vlákno, ve kterém spustí síťového klienta a předá mu ve formě struktury `network_client_setup` všechny potřebné informace (dále popsáno v kapitole 4.3.1). Pokud byly skrz `TSolver_Setup` předány nějaké „hints“ (předpočítaná řešení), nahradí jimi ta náhodně vygenerovaná.

Následně jsou ve smyčce podle stanoveného počtu generací vyhodnocována aktuální řešení a vypočítávána nová podle algoritmu particle swarm. Po určitém množství iterací mohou být do populace zahrnuta řešení získaná po síti od ostatních, pokud byla nějaká přijata a jsou dostatečně kvalitní.

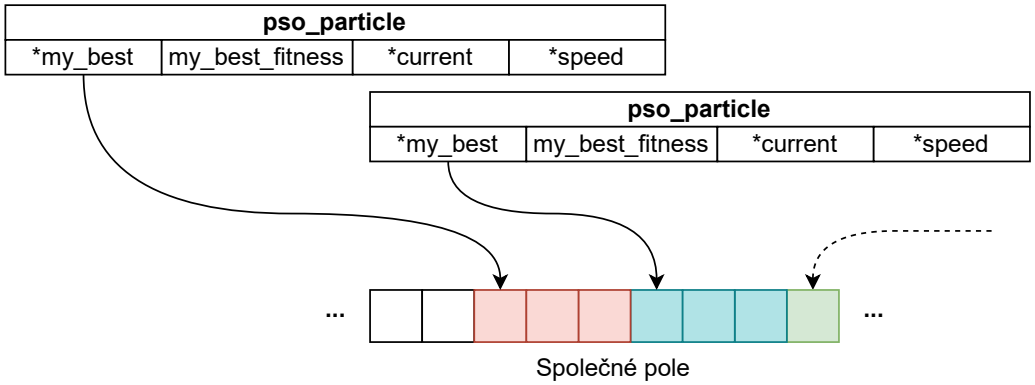
Solver končí po proběhnutí všech iterací, případně pokud byl zvenku nastaven příznak pro ukončení.

4.1.3.2 Funkce `pso_alloc()` a `pso_dealloc()`

Alokují a dealokují paměť pro použité struktury. Raději než paměť silně fragmentovat separátní alokací každého pole v `pso_particle` (tj. nejlepší řešení, aktuální řešení a rychlost), jsou tato pole alokována najednou s dostatečnou velikostí a následně jsou do jednotlivých `pso_particle` jen umístěny odkazy na správné místo globálních polí. Na obrázku 4.1 je to schématicky znázorněno například pro pole udržující parametry nejlepších řešení, v případě ostatních je princip stejný. V zásadě by bylo možné ušetřit paměť a strukturu `pso_particle` vůbec nepoužívat, ale její využití napomáhá čitelnost kódu.

4.1.3.3 Skupina funkcí `pso_init...()`

Zajišťují naplnění polí rychlostí a parametrů řešení (pseudo)náhodnými hodnotami před spuštěním algoritmu. Hodnoty jsou vymezeny konstantami třídy a zvenku



Obrázek 4.1: Odkazování na globální pole ze struktur `pso_particle`.

mezními hodnotami předanými v `TSolver_Setup`.

4.1.3.4 Skupina funkcí `vector_...()`

Slouží k provedení konkrétní matematické operace s číselným vektorem (sčítání, odčítání, násobení skalární hodnotou). Využity jsou při iteracích particle swarm algoritmu, který pracuje výhradně s vektory.

4.1.3.5 Funkce `pso_param_bounds_clip()`

Během průchodu částice skrz stavový prostor se může stát, že se v některé dimenzi dostane mimo stanovené meze. Tato funkce popsaný stav kontroluje a pokud k němu dojde, dotyčná částice je v problémové ose umístěn na její hranici s tím, že pokud by jeho rychlost mířila do „zakázané oblasti“, její směr bude obrácen.

4.1.3.6 Funkce `pso_iteration()`

Vlastní iterace particle swarm – pro všechny částice se přepočítá jejich rychlost podle vztahu 2.1 a tato rychlost se přičte k jejich poloze, čímž vznikne nové řešení.

4.1.3.7 Funkce `pso_solution_random_insert()`

Náhodně vybranou částici z populace nahradí novou, resp. přepíše její aktuální pozici tou, která byla předána v argumentu funkce, a vynuluje její rychlost. Funkce je využita pro zahrnutí řešení od ostatních pump.

4.1.3.8 Funkce `get_shared_solutions()`

Pokud byla síťovou částí aplikace přijata nějaká řešení od ostatních pump, tato funkce je potenciálně zahrne do lokální populace. Přijatá řešení se mezi vlákny pře-

dávají pomocí sdílené paměti (viz 4.3.1), je tedy nutné vzájemné vyloučení, pro které byl využit obyčejný mutex.

Po nakopírování dat do lokální proměnné a uvolnění kritické sekce je pro každé přijaté řešení vypočtena hodnota kriteriální funkce. Pokud je výsledek přijatého řešení lepší, než výsledek nejhorší částice v lokální populaci, je řešení přijato, tzn. je implementováno do náhodné částice funkcí `pso_solution_random_insert()`. Pokud je hodnota fitness horší, řešení je odmítnuto.

4.1.3.9 Funkce `Evaluate_Solutions()`

Vyhodocuje kvalitu aktuální populace. Přes kriteriální funkci, předanou skrz `TSolver_Setup` nechá pro každou částici vypočítat hodnotu fitness. Průchodem skrz výsledky pak zjistí, jaké je v aktuální populaci nejlepší a nejhorší řešení a případně upraví hodnoty lokálních nejlepších řešení pro jednotlivé částice. Pokud by navíc došlo k nalezení zatím globálně nejlepšího řešení, jsou jeho parametry a hodnota fitness uloženy a předány SmartCGMS aplikaci.

4.1.4 Logika genetické evoluce

Optimalizace pomocí genetické evoluce je implementována ve zdrojových souborech `distributed_ge_solver.cpp` a `distributed_ge_solver.h`.

Struktura je prakticky stejná jako pro particle swarm (popsaný v kapitole 4.1.3), tj. hlavičkový soubor obsahuje definici třídy `CDB_GE_Solver` s defaultními hodnotami a konstantami. Výkonný kód je umístěn v `.cpp` souboru. Kód je zde vzhledem k podobnosti s particle swarm popsán stručněji.

4.1.4.1 Funkce `solve()`

Vstupní bod solveru. Alokuje paměť, inicializuje proměnné, náhodně vygeneruje počáteční populaci (případně do ní zahrne předané „hints“) a spustí vlákno se síťovým klientem. Pak už probíhá samotná optimalizace – je vytvořena nová generace jedinců, vyhodnocena jejich fitness, a ti s lepší hodnotou fitness oproti předchozí generaci se stanou součástí výchozí množiny pro další iteraci. Po určitém množství iterací je možné do populace zahrnout řešení od ostatních.

4.1.4.2 Funkce `create_child_solutions()`

Z předaného pole rodičů vygeneruje potomky – s určitou pravděpodobností provede křížení vybraného (hlavního) jedince s náhodně vybraným jiným jedincem. Parametry výsledného potomka se ve stavovém prostoru posunou od hlavního rodiče směrem k vedlejšímu (délku posunu ovlivňuje globální konstanta). Pokud ke křížení nedojde, potomek je shodný s hlavním rodičem.

4.1.4.3 Funkce `mutate_solutions()`

S určitou pravděpodobností provede mutaci parametrů. Mutace spočívá v náhodném přičtení/odečtení části sebe sama. Pokud se hodnota parametru v absolutní hodnotě dostane pod stanovenou mez, dojde navíc k otočení znaménka.

4.1.4.4 Funkce `pick_better_solutions()`

Porovnává mezi sebou hodnotu fitness rodiče a potomka (tzn. dvou jedinců na stejném indexu mezi generacemi). Pokud dosáhl potomek zlepšení, je nakopírován na místo rodiče pro použití v další iteraci, v opačném případě se nestane nic a do další iterace pokračuje rodič.

4.1.4.5 Funkce `Evaluate_Solutions()`

Získá hodnotu fitness jedinců v aktuální populaci předáním jejich parametrů kriteriální funkci. V případě, že se dosáhne lepší hodnoty než byla dosavadní nejlepší, je tato hodnota uložena a předána SmartCGMS přes `TSolver_Setup`, a také je aktualizován ukazatel pro síťového klienta.

4.1.4.6 Funkce `get_shared_solutions()`

Převezme přijatá řešení ze síťového klienta. Narozdíl od particle swarm ale neposuzuje jejich kvalitu porovnáním fitness oproti nejhoršímu řešení současné populace – všechna přijatá řešení bez výjimky nahradí náhodně vybraná místní řešení. Před nahrazením je chráněn jedinec s nejlepší fitness v populaci, aby při importu „špatného“ řešení z vnějšku nedošlo ke kompletnímu znehodnocení dosavadních výsledků. Také se ze stejného důvodu předpokládá, že počet přijatých řešení je výrazně menší, než velikost populace.

4.2 Síťová komunikace – společný kód

Vzhledem k tomu, že byl pro naprogramování logiky serveru i klienta použit stejný programovací jazyk, velké množství kódu lze mít společné a beze změn využívat v obou částech aplikace.

4.2.1 Struktury síťové komunikace

Hlavičkový soubor `structs.h` obsahuje důležité struktury pro výměnu zpráv. Jsou zde především jako `enum` definovány všechny typy zpráv, aby se s nimi dalo ve zbytku kódu pracovat jednotným způsobem.

Dále jsou zde definovány struktury agregující data přenášená v některých typech zpráv, a také struktura `message_retry_info`, sdružující informace potřebné pro opakování přenosu, pokud by mělo dojít k výpadkům.

4.2.2 Konstanty protokolu

Hlavičkový soubor `protocol.h` obsahuje výhradně definice preprocesoru, týkající se protokolu – délky polí a texty.

4.2.3 Kontrolní součet CRC32

Implementace CRC32 je umístěna v souborech `crc_32.c` a `crc.h`, převzatých s drobnými úpravami z [48].

4.2.4 Čtení parametrů ze souboru

Zdrojové soubor `read_param_file.c` a `read_param_file.h` obsahují pomocné funkce pro čtení parametrů ze souboru, především IP adresy a portu, ale v případě potřeby je možné si nechat přečíst libovolný parametr.

Formát souboru se očekává co řádka to jedno key-value přiřazení: `<název parametru>:<hodnota>`.

4.2.5 Práce se zprávami

Zdrojové soubory `messaging.c` a `messaging.h` poskytují funkce pro práci se zprávami – serializace/extrakce dat, stanovení typu, kontrola integrity, odesílání.

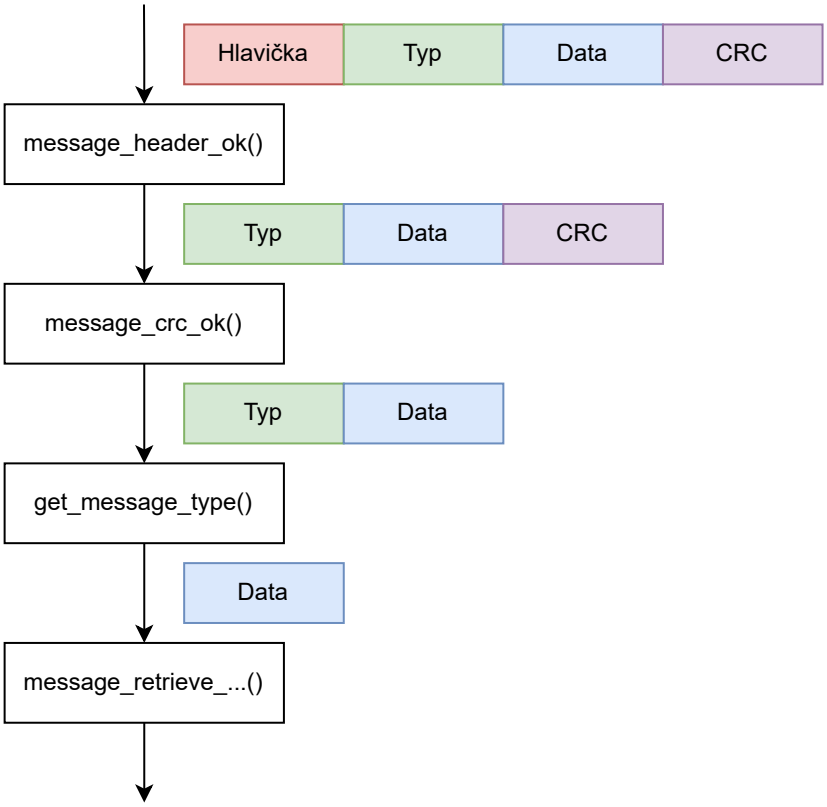
Vytvoření zprávy a její odeslání řeší jediná funkce `message_create()`. Přijatá zpráva se zpracovává postupně, průchodem skrz několik funkcí jsou postupně vyjímány řídicí a kontrolní prvky, dokud nezbydou samotná data. Schématicky to znázorňuje obrázek 4.2 – shora přijde kompletní zpráva přijatá například funkcí `recvfrom()`, vespod si přijatá data převezme vnější kód.

4.2.5.1 Skupina funkcí `message_fill_..._data()`

Tyto funkce tvoří nejpočetnější skupinu. Jejich úlohou je do předaného pole znaků serializovat data pro konkrétní typ zprávy. Všechny mají stejnou hlavičku, je tedy možné s nimi pracovat přes jeden ukazatel na funkci, čehož se využívá ve funkci `message_create()`.

4.2.5.2 Skupina funkcí `message_retrieve_...()`

Slouží k extrakci dat z přijatých zpráv.



Obrázek 4.2: Průběh zpracování přijaté zprávy.

4.2.5.3 Funkce `message_create()`

Obaluje základní funkci `sendto()`. Zajišťuje vytvoření validní zprávy na základě předaného požadovaného typu a dat s ní sdružených – umístění hlavičky, serializace dat zavoláním funkce pro stanovený typ, výpočet kontrolního součtu. Pokud je na zprávu očekávána odpověď, aktualizuje i informace pro opakování přenosu, reprezentované předanou strukturou `message_retry_info`.

4.2.5.4 Funkce `message_crc_ok()` a `message_header_ok()`

Kontrolují správnost zabezpečovacích prvků, tj. hlavičky a kontrolního součtu. Pokud jsou oba prvky v pořádku, dojde k jejich „odstranění“ ze zprávy formou posunutí ukazatelů na začátek zprávy a změnou hodnoty celkové délky.

4.2.5.5 Funkce `get_message_type()`

Identifikuje a vrací typ zprávy podle textového řetězce za hlavičkou.

4.2.5.6 Funkce `prepare_message_fill()`

Tyto funkce do pole ukazatelů na funkce umístí ukazatele na kód, který serializuje data do typu zprávy, který odpovídá jeho indexu. V `message_create()` je pak možné volat správnou funkci bez dalšího větvení. Ilustrují to úryvky zdrojového kódu 4.3.

Stejného výsledku lze docílit i staticky při inicializaci pole, nicméně ve formě výkonného kódu je možné rovnou využít `enum` zaštiťující typ zprávy.

Zdrojový kód 4.3: Inicializace a použití pole ukazatelů na funkce.

```
1  void prepare_message_fill(void) {
2      ...
3      message_fill[msg_type_init] = message_fill_init_data;
4      ...
5  }
6
7  int message_create(..., const enum msg_type type, ...) {
8      ...
9      message_fill[type](...);
10     ...
11 }
```

4.3 Síťová komunikace – klient

U klienta se předpokládá, že bude spuštěn ze SmartCGMS solveru v separátním vlákně. Znamená to lépe oddělené a nezávislé kódy, ale mírně to komplikuje výměnu dat mezi solverem a síťovým klientem, vzhledem k tomu, že obě vlákna přistupují do společné paměti a může dojít k souběhu. Pro eliminaci tohoto problému byly využity prosté mutexy, zajišťující vzájemné vyloučení.

4.3.1 Rozhraní síťového klienta

Hlavičkový soubor `client_iface.h` poskytuje implementaci solveru informace, jak pracovat se síťovým klientem.

Je zde definována hlavní funkce klienta `client_main()`, kterou solver předá funkci pro vytvoření nového vlákna jako jeho vstupní bod.

Také je zde definována struktura `network_client_setup`, která obsahuje všechny potřebné informace pro samostaný běh klienta. Jedná se především o odkazy na pole, ve kterých jsou mezi vlákny předávána řešení, včetně sdružených mutexů. Součástí je také informace o počtu parametrů, které tvoří jedno řešení, a indikace požadavku na ukončení. Kompletní podoba struktury je zachycena v kódu 4.4.

Struktura je označena jako `extern`, což překladači indikuje, že je její instance vytvořena v jiném zdrojovém souboru (konkrétně v `client_main.c`, viz 4.3.2.1).

Zdrojový kód 4.4: Struktura `network_client_setup`

```
1  typedef struct {
2      /* Pocet parametru jednoho reseni */
3      size_t dimension;
4      /* Priznak, zda ma klient ukonci svou cinnost */
5      uint8_t *client_terminate;
6
7      /* Odkaz na pole, kam maji byt umistena
8          prijata reseni od ostatnich */
9      double *received_solutions;
10     /* Aktualni pocet reseni
11         v poli received_solutions */
12     uint8_t *new_received_solutions_nr;
13     /* Maximalni pocet reseni v received_solutions */
14     uint8_t received_solutions_max_nr;
15     /* Vzajemne vylouceni pristupu
16         k poli received_solutions */
17     pthread_mutex_t *received_solutions_mutex;
18
19     /* Odkaz na pole s lokalnim resenim,
20         ktere chci sdilet s ostatnimi */
21     double *my_solution;
22     /* Vzajemne vylouceni pristupu
23         k poli my_solution */
24     pthread_mutex_t *my_solution_mutex;
25 } network_client_setup;
```

4.3.2 Kód síťového klienta

Ve zdrojovém souboru `client_main.c` je výkonný kód specifický pro síťového klienta.

4.3.2.1 Funkce `client_main()`

Vstupní bod vlákna – implementace funkce, definované v `client_iface.h` (4.3.1).

V první řadě alokuje pamět a inicializuje všechny potřebné proměnné, otevře socket pro síťovou komunikaci a ze souboru získá adresu a port, na kterém běží instance serveru.

Následuje nekonečná smyčka, ve které vlákno setrvává, dokud není vyžadováno jeho ukončení nastavením příznaku v `network_client_setup`. Ve smyčce je postupně zkontrolováno, zda nedošlo u nějakých předchozích odeslaných zpráv

k timeoutu (separátně pro server a ostatní účastníky) a pokud uběhla příliš dlouhá doba bez komunikace, serveru je poslán ping.

Zda má dojít k vyžádání adres od serveru nebo k vyžadání řešení od ostatních účastníků, indikují celočíselné proměnné `refresh_addresses` a `refresh_params`, se kterými se pracuje jako s booleovskými proměnnými. Obě jsou pro jednoduchost nastavovány zároveň, pokud je detekováno přečtení alespoň jednoho přijatého řešení ze strany solveru (detekováno snížením hodnoty `new_received_solutions_nr` ve struktuře `network_client_setup` oproti předchozí iteraci). Nastavovat je zároveň ale samozřejmě není nutné, pak by bylo možné víckrát získat nejlepší řešení od stejného účastníka.

Následně může být ve funkci `take_action()` iniciována nová výměna zpráv.

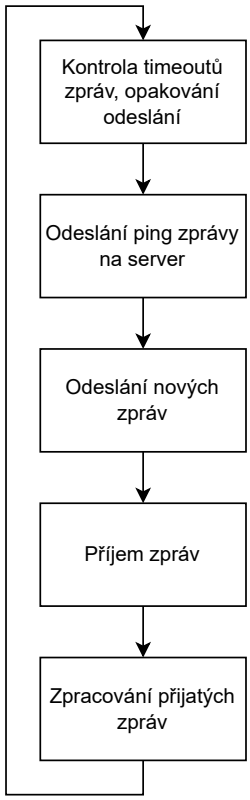
Zbytek smyčky je tvořen příjímáním zpráv od ostatních. Pro přenos byl zvolen protokol UDP, takže pokud byla nějaká zpráva přijata, je vždy operačním systémem předána celá (narozdíl od TCP, který se chová jako proud dat). UDP ale nezajišťuje zabezpečení přenosu, z tohoto důvodu je ke zprávě při odeslání přidán kontrolní součet CRC32. Tento kontrolní součet, stejně jako pevně stanovená hlavička zprávy, je po přijetí kontrolován a pokud se neshoduje s očekávanou hodnotou, zpráva je považována za neplatnou a zahozena. V opačném případě je stanoven její typ a podle něj dojde k zavolání funkce, která zprávu zpracuje.

Zjednodušený flowchart výpočtu je zachycen na obrázku 4.3.

Pokud je aktivní požadavek na ukončení klienta, lze smyčku na dvou místech opustit (a tím ukončit i běh vlákna) – pokud není navázáno spojení se serverem, dojde k ukončení při prvním průchodu smyčkou, kdy je požadavek aktivní. Pokud je spojení se serverem aktivní, nejprve je mu odeslána zpráva od odpojení a klient je ukončen, jakmile od serveru přijde potvrzení o přijetí této zprávy. Pokud by mělo dojít k timeoutu, server je v tu chvíli z hlediska klienta nedostupný, spojení není navázáno, a k ukončení dojde přes první podmínku.

4.3.2.2 Funkce `timeout_check_server()` a `timeout_check_peers()`

Jak už názvy napovídají, tat dvojice funkcí ověřuje, zda se už příliš dlouho nečeká na potvrzení od protistrany. Jejich obsah je z velké části podobný – pokud je potvrzení vůbec očekáváno, porovná se doba od odeslání poslední zprávy s limitem. Pokud je limit překročen, je několikrát zopakováno odeslání zprávy, a když ani to nepomůže, je protistrana považována za nefunkční. V případě dalšího účastníka výpočtu to znamená jeho odstranění z lokálního seznamu adres. V případě serveru to znamená návrat do neinicializovaného stavu.



Obrázek 4.3: Zjednodušený průběh nekonečné smyčky síťového klienta.

4.3.2.3 Funkce `info_consider_update()` a `info_update()`

Tyto funkce na základě přijaté zprávy zajišťují korektní úpravu struktur `message_retry_info` – pokud je přijatá zpráva odpovědí (nekontroluje se ale, zda správného typu) a daný adresát na nějakou odpověď čeká, je mu resetován příznak nepotvrzené zprávy a počet opakovaných odeslání. Struktura `message_retry_info` pro úpravu je vybrána podle adresy odesílatele.

4.3.2.4 Funkce `take_action()` a `take_action_server()`

Mohou zahájit novou výměnu zpráv - tj. odeslat zprávy, které nejsou jen reakcí na podnět od ostatních účastníků – jde o navázání/ukončení spojení se serverem, vyžádání adres od serveru, a vyžádání řešení od ostatních účastníků výpočtu. Pro přehlednost byl kód řešící zprávy na server umístěn do separátní funkce.

4.3.2.5 Skupina funkcí `message_process_...()`

Každá tato funkce je určena pro zpracování jednoho konkrétního typu přijaté zprávy. Všechny přebírají stejné argumenty, aby se daly abstrahovat do stejného ukazatele

na funkci, přes který jsou následně volány.

Klient implementuje zpracovávání následujících typů zpráv:

- `init response` – potvrzení inicializace u serveru, uloží přijaté identifikační číslo a změní stav serveru na `state_ready`,
- `addr request response` – přijetí adres ostatních účastníků výpočtu od serveru, adresy jsou uloženy do lokálního pole, je resetován příznak požadavku na aktualizaci adres `refresh_addresses`,
- `param request` – žádost o poskytnutí řešení od jiného účastníka výpočtu, reaguje odesláním odpovědi s řešením, překopírovaným ze solveru,
- `param request response` – odpověď na žádost o řešení, přijaté parametry jsou nakopírovány do sdíleného pole, odkud je může solver přečíst (pokud by bylo pole plné, přijaté řešení se zahodí).

4.3.2.6 Funkce `prepare_message_process()`

Aby mohly být během výpočtu bez dalšího větvení volány funkce pro zpracování zpráv podle typu, jsou použita pole ukazatelů na funkce, která mají tolik prvků, kolik je typů zpráv. Tato funkce pole inicializuje, jelikož je to přehlednější než v případě statické inicializace. Jedná se o stejný princip jako už byl popsán u funkce `prepare_message_fill()` v kapitole 4.2.5.6.

4.4 Síťová komunikace – server

Server je koncipován jako standalone aplikace pro podporu klientů. V současné implementaci to znamená, že pouze udržuje seznam všech aktivních klientů a v případě požadavku část adres poskytne pro navázání peer-to-peer komunikace bez jeho účasti.

Veškerá logika, vlastní serveru, je umístěná v souboru `server_main.c`.

4.4.1 Funkce `main()`

Hlavní funkce serveru. Provede inicializaci proměnných a následně velmi podobně jako v případě klienta běží v nekonečné smyčce. Narozdíl od klienta je ale zcela pasivní – nezahajuje nové výměny zpráv, jen reaguje na zprávy přijaté od ostatních. Smyčka je tak o něco jednodušší a střídá se v ní příjem a případně zpracování zpráv s kontrolou, zda jsou všichni klienti aktivní.

Nepředpokládá se, že by program měl smyčku opustit.

4.4.2 Skupina funkcí `message_process_...()`

Stejně jako v případě klienta, každá tato funkce je volána pro zpracování jednoho konkrétního typu zprávy.

Server implementuje zpracovávání následujících typů zpráv:

- `init` – inicializace nového klienta, je mu přiřazeno a v odpovědi posláno unikátní identifikační číslo, kterým se bude nadále prokazovat,
- `addr request` – požadavek na poskytnutí adres ostatních účastníků, pokud je validní přijaté id, reaguje odesláním zprávy s požadovanými adresami,
- `ping` – periodické udržování relace, aby server klienta nepovažoval za neaktivního, pokud není potřeba posílat jiné zprávy, server jen potvrdí přijetí a aktualizuje čas poslední komunikace,
- `leave` – indikace ukončení činnosti klienta, zpráva je potvrzena a klientovo identifikační číslo je uvolněno pro další použití.

4.4.3 Funkce `prepare_message_process()`

Vyplnění pole ukazatelů na funkce, které budou následně zpracovávat příchozí zprávy (viz stejná funkce v klientské části v kapitole 4.3.2.6).

4.4.4 Funkce `fill_addresses()`

Kopíruje adresy klientů z hlavního seznamu do pomocné struktury, aby je bylo možné odeslat.

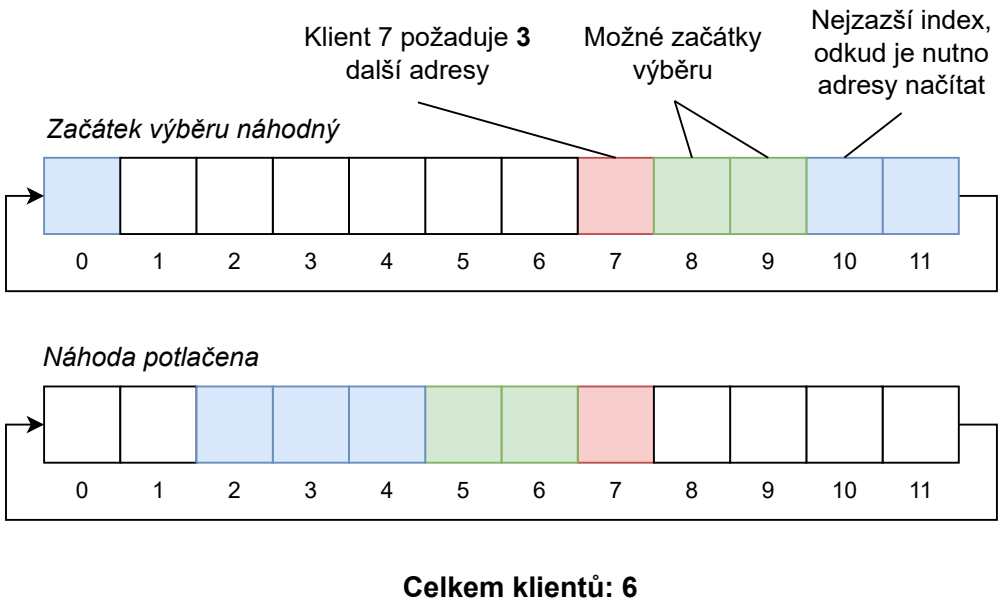
Nejprve je vybrán index, od kterého se začnou sbírat adresy pro odeslání na dotaz žadatele. Žádoucí je, aby se žadateli v odpovědi nevrátila jeho vlastní adresa. Toho se docílí tak, že je jako výchozí uvažován právě index žadatele, který je známý. Následně je k němu přičtena určitá hodnota podle okolností (viz dále) a od této hodnoty dále jsou vybírány adresy tak dlouho, dokud není splněn požadavek. K poli se přitom přistupuje jako k obtočenému (za posledním indexem se pokračuje počátečním).

Může se stát, že klient požaduje víc adres, než má server v danou chvíli k dispozici – v takovém případě je počátečním indexem výběru hned index následující po žadateli a množství požadovaných adres je upraveno tak, aby bylo rovno počtu připojených klientů (limitně až 0 – žadatel se nebere v potaz).

Pokud je přihlášených klientů více, server vybere počáteční index výběru náhodně tak, aby se v poli mezi ním a žadatelem nacházelo přinejmenším požadované množství adres. To znamená, že pokud je N celkový počet připojených klientů a P požadované množství adres, je možné se posunout v rozsahu $< 1; N - P >$. Pole

může být fragmentované, při přesunu na nový index se nejprve kontroluje, zda je slot vůbec využit, pokud ne, provede se posun o index dále. To také znamená, že pokud by mezi indexem žadatele a dalším obsazeným indexem byl dostatek nevyužitých slotů, náhoda bude negována a vždy dojde k výběru stejných adres (za předpokladu statičnosti klientů).

Graficky je to znázorněno na obrázku 4.4 pro, z hlediska náhody, nejlepší a nejhorší případ.



Obrázek 4.4: Výběr adres, které budou odeslány v reakci na požadavek od klienta.

4.4.5 Funkce `participant_list_init()`

Inicializuje kus seznamu účastníků – nuluje časy, kdy byl účastník naposledy aktivní, což ve zbytku programu indikuje, že je daný index nevyužitý.

4.4.6 Funkce `get_next_participant_id()`

Vrací identifikační číslo, které dostane další nově připojený klient. V první řadě se snaží plně naplnit kapacitu existujícího pole, pokud už ale žádné volné místo neexistuje, je velikost pole pomocí funkce `realloc()` zvětšena na dvojnásobek. V takovém případě je navíc nová část inicializována vrácené id je první index v nové části.

4.4.7 Funkce `check_client_inactivity()`

Projde celý seznam klientů, a pokud se inicializovaný klient odmlčel na příliš dlouhou dobu, je považován za neaktivního a odstraněn ze seznamu.

Testování 5

Navržený program byl otestován nad dodanými daty pro experimentální ověření funkčnosti a posouzení kvality výstupních řešení. Jako doplněk jsou v příloze 8 uvedené výstupní grafy pro některé testovací scénáře.

5.1 Optimalizace parametrů PID regulátoru

Pro testování byla využita čtveřice minipočítačů Raspberry Pi Zero s nainstalovaným operačním systémem Raspbian GNU/Linux 11 (Bullseye).

Oba implementované optimalizační algoritmy byly spuštěny na 100 generací s velikostí populace 20 jedinců. Velikost populace byla zvolena poněkud menší, než bývá zvykem (běžně spíše vyšší destíky jedinců), jako kompromis mezi kvalitou nalezených řešení a dobou běhu, která v některých případech přesáhla i deset hodin.

Testovány byly následující způsoby běhu:

1. Samotná pumpa bez sdílení informací se shodným pacientem – referenční výsledek pro možnost porovnávání oproti komunikujícím solverům. V kódu bylo za tímto účelem zcela vynecháno založení vlákna, obstarávajícího síťovou komunikaci.

2. Skupina pump se shodným pacientem se sdílením informací – syntetický scénář, kdy všechny pumpy řeší shodný problém. Důvodem jeho zařazení je naznačit, zda má výměna informací vliv alespoň při ideálních podmínkách.

3. Skupina pump s různými pacienty se sdílením informací – podmínky nejbližší skutečnému nasazení, každá pumpa řeší jiný problém.

Pro simulaci různých „pacientů“ byl vedoucím práce dodán různě parametrizovaný řetězec SmartCGMS filtrů, spouštěný nad stejným scénářem. Parametry filtrů byly stanoveny identifikací z OhioT1DM datasetu [49]. Cílem je identifikovat parametry PID regulátoru ověřeného na modelu pacienta GCT [50], jehož matematický

předpis je uveden v následující rovnici:

$$Y = B - S \cdot (K_p \cdot e(t) + K_d \cdot \frac{\Delta e(t)}{\Delta t} + K_i \cdot \int_0^t e(t)), \tag{5.1}$$

přičemž Y označuje dávkovanou hladinu inzulinu, B základní dávku, S citlivost na inzulin, $e(t)$ odchylku cílové od skutečné koncentrace glukózy v krvi, a K_x jsou parametry (zesílení proporcionální, derivační a integrační složky).

Pro bod 3 je směrodatný výsledek první pumpy, jelikož optimalizovala parametry stejného pacienta, jaký byl optimalizován v bodech 1 a 2.

5.1.1 Genetická evoluce

Kvalita výstupních parametrů v podobě průměrných chyb pro popsané scénáře pro algoritmus genetické evoluce je zachycena v tabulce 5.1. Hodnoty pro pumpu 2, 3 a 4 ve třetím sloupci nelze porovnávat s výsledky v ostatních sloupcích, jelikož optimalizace probíhala nad jiným pacientem!

	Průměrná chyba [mmol/l]		
	Bez sdílení, stejný pacient	Se sdílením, stejný pacient	Se sdílením, různí pacienti
Pumpa 1	1,47	1,37	1,32
Pumpa 2	1,55	1,42	*3,65*
Pumpa 3	-	1,45	*1,36*
Pumpa 4	-	1,46	*2,85*

	Průměrná chyba [%]		
	Bez sdílení, stejný pacient	Se sdílením, stejný pacient	Se sdílením, různí pacienti
Pumpa 1	22,07	20,58	19,89
Pumpa 2	23,23	21,37	*54,84*
Pumpa 3	-	21,77	*20,36*
Pumpa 4	-	21,87	*42,75*

Tabulka 5.1: Porovnání kvality optimalizovaných parametrů PID regulátoru metodou genetické evoluce.

5.1.2 Particle swarm optimization

Pro algoritmus particle swarm optimization jsou výsledky zachycené v tabulce 5.2. Stejně jako u genetické evoluce platí, s výjimkou pumpy 1 nelze srovnávat hodnoty s ostatními sloupci.

	Průměrná chyba [mmol/l]		
	Bez sdílení, stejný pacient	Se sdílením, stejný pacient	Se sdílením, různí pacienti
Pumpa 1	1,33	0,83	0,85
Pumpa 2	1,33	0,83	*2,64*
Pumpa 3	-	0,83	*1,02*
Pumpa 4	-	0,83	*2,39*

	Průměrná chyba [%]		
	Bez sdílení, stejný pacient	Se sdílením, stejný pacient	Se sdílením, různí pacienti
Pumpa 1	19,94	12,48	12,75
Pumpa 2	19,94	12,48	*39,59*
Pumpa 3	-	12,48	*15,25*
Pumpa 4	-	12,48	*35,93*

Tabulka 5.2: Porovnání kvality optimalizovaných parametrů PID regulátoru metodou particle swarm optimization.

5.2 Měření programem `perf`

`perf` je profiler pro operační systémy GNU/Linux, schopný sbírat detailní statistiky o běžicích procesech. V tomto případě byl využit příkaz `perf stat` pro rámcové porovnání implementovaných metod optimalizace.

Měření bylo provedeno opět na Raspberry Pi Zero s operačním systémem Raspbian GNU/Linux 11 pro jednu iteraci optimalizačního algoritmu s velikostí populace 20 jedinců se vším všudy. Ve statistikách jsou tak zahrnuty i výpočetní nároky samotné aplikace SmartCGMS, které by měly být nicméně mezi oběma algoritmy srovnatelné. Vybrané výstupní hodnoty pro particle swarm optimization a genetickou evoluci jsou pro porovnání zachyceny v tabulce 5.3.

	PSO	GE
Context switches [n]	9022	8331
Page-faults [n]	8416	3922
Branches [n]	8,84E+09	8,49E+09
Branch-misses [n]	4,09E+08	3,94E+08
Branch-misses [%]	4,6	4,6
Time elapsed [s]	580	559
System time [s]	2,51	2,57

Tabulka 5.3: Porovnání vybraných výstupů profileru `perf` pro particle swarm optimization a genetickou evoluci.

5.3 Identifikace pacientského modelu

Pro demonstrování vlastností nad větším modelem byla vyzkoušena identifikace „skutečného“ pacientského GCT modelu. Tato identifikace už oproti předchozímu testování probíhala na klasickém stolním počítači s ohledem na kratší doby výpočtu, ale různé způsoby běhu uvedené v kapitole 5.1 byly zachovány. Pumpy byly simulovány čtyřmi procesy na stejném počítači a s ohledem na výsledky předchozích testování byl pro optimalizaci využit pouze algoritmus particle swarm. Velikost populace byla stanovena opět na 20 jedinců, počet iterací na 50.

Výsledky zachycuje tabulka 5.4 – stejně jako výše, i zde není možné porovnávat hodnoty ve třetím sloupci s výjimkou prvního řádku s ostatními hodnotami v tabulce.

	Průměrná chyba [mmol/l]		
	Bez sdílení, stejný pacient	Se sdílením, stejný pacient	Se sdílením, různí pacienti
Pumpa 1	3,26	2,44	2,10
Pumpa 2	3,53	2,36	*1,50*
Pumpa 3	-	2,02	*1,72*
Pumpa 4	-	2,14	*2,10*

	Průměrná chyba [%]		
	Bez sdílení, stejný pacient	Se sdílením, stejný pacient	Se sdílením, různí pacienti
Pumpa 1	45,78	24,67	27,9
Pumpa 2	34,83	26,14	*18,21*
Pumpa 3	-	21,38	*16,54*
Pumpa 4	-	26,01	*42,1*

Tabulka 5.4: Porovnání kvality optimalizovaných parametrů pacientského modelu metodou particle swarm optimization.

Zhodnocení vlastností

6

Testování naznačuje, že výměna informací během optimalizace dokáže zvýšit kvalitu výsledného řešení. Konkrétně v případě genetické evoluce a identifikace parametrů PID regulátoru je mezi případem, kdy žádná komunikace neprobíhala, a případem, kdy spolu pumpy řešily stejný problém, rozdíl zhruba dvě procenta ve prospěch komunikace. Výsledek při řešení různých problémů byl sice ještě lepší, ale jen o necelé procento, což už lze s ohledem na množství dat označit za zanedbatelné.

Při použití particle swarm optimization pro stejný problém měly výsledky podobnou tendenci – se zavedením komunikace došlo ke zlepšení oproti nekomunikujícím pumpám, a to jak v případě řešení stejného problému, tak při řešení odlišných problémů. Výsledek těchto dvou situací byl přitom srovnatelný. Při řešení společného problému přitom cluster došel k jednomu nejlepšímu výsledku, který už se dále nepodařilo vylepšit, narozdíl od genetické evoluce, kde pumpa 1 našla kvalitnější výsledek, který se nedostal k ostatním.

Za zmínku stojí, že výsledky particle swarm optimization byly ve všech případech kvalitnější, než v případě genetické evoluce, a to až do té míry, že i samotná pumpa dosáhla s particle swarm optimization lepších výsledků, než pumpa využívající genetickou evoluci a spolupracující s ostatními. Testování bylo ale provedeno v tomto ohledu v omezené míře s pouze čtyřmi účastníky. Obecně komplexní testování této problematiky včetně zhodnocení dopadů intenzity výměny informací, způsobu jejich integrace a množství spolupracujících zařízení by vydalo na celou samostatnou práci.

Algoritmus particle swarm optimization byl dále otestován na identifikaci pacientského modelu s větším množstvím parametrů na výkonnějším hardwaru. V této úloze způsobilo zavedení komunikace napohled mnohem působivější výsledky – došlo ke zlepšení zhruba o deset procent. I zde ale platí, že testování spíše naznačuje možný trend, než že by se jednalo o vyčerpávající analýzu.

Co se výkonnosti týče, oba optimalizační algoritmy byly srovnány v rámci jejich jedné iterace nástrojem `perf`. Zatímco četnost špatné predikce větvení byla v obou případech prakticky stejná, stejně jako doba, kterou proces strávil obsluhou systé-

mových volání, odlišné jsou například doby běhu – particle swarm optimization běžel o něco pomaleji, ale jak bylo naznačeno výše, poskytuje kvalitnější výsledky.

Drastický rozdíl je potom v počtu výpadků stránek, kterých v případě genetické evoluce nastalo o polovinu méně oproti particle swarm optimization. Je těžké určit, čím přesně je tento úkaz zapříčiněn, nicméně particle swarm optimization potřebuje pro svou funkčnost udržovat oproti genetické evoluci větší množství dat. Zatímco genetická evoluce potřebuje pouze dvě sady řešení (rodiče a potomkové), particle swarm sice udržuje pouze jednu sadu řešení, ale každé se skládá ze samotných „okamžitých" parametrů, nejlepších parametrů za celou dobu optimalizace, okamžité změny parametrů (rychlosti částice) a navíc i struktury udržující ukazatele na zmíněné hodnoty. Jedná se ale stále o data s velikostmi v řádech kB a tedy potenciálně dostatečně malé pro umístění do několika jednotek stránek. Výrazně větší jsou soubory se spustitelným kódem – jak spustitelný soubor `scgms-console`, tak i sdílená knihovna s distribuovanými solvery a ostatní sdílené knihovny projektu mají velikost v řádech jednotek MB, což už by při velikosti stránky 4 kB znamenalo potřebu několika stovek stránek pro úplné načtení. Pokud operační systém stránky uvolňuje co možná nejdříve, častější výpadky mohou být způsobeny pomalejším výpočtem particle swarm, během kterého dojde k uvolnění stránek, jež je pak vzápětí opět potřeba načíst.

Komunikace mezi pumpami probíhá nešifrovaně, kdokoliv tak může vyměňované zprávy odposlechnout, nebo se dokonce vydávat za legitimního účastníka. Problémem by bylo především nadměrné zatěžování pumpy neustálým zasíláním zpráv, což by znamenalo plýtvání elektrickou energií a výpočetním výkonem – obojí je pro pumpu žádoucí investovat spíše do optimalizace. V žádném případě ale útočník nemůže přes síť získat surová pacientská data, pumpa poskytne maximálně agregovaná data v podobě parametrů modelu.

Raspberry Pi Zero odebírá minimalisticky zhruba 100 mA [51] při 5 V, tj. 0,5 W. Na dvě tužkové baterie AA s celkovou kapacitou 6 Wh (každá po 2000 mAh při 1,5 V) by ho tak bylo možné provozovat zhruba po dobu 12 hodin, což by mohlo postačovat na jeden den.

Možná rozšíření 7

Aplikace je funkční, tak jak byla navržena a implementována. Zdaleka to ovšem neznamená, že není co dále vylepšovat, spíše naopak.

7.1 Různé modely

V současné implementaci se předpokládá, že všichni účastníci výpočtu identifikují parametry stejného modelu a tento model je určen implicitně. V praxi se ale může stát, že různí pacienti používají modely různé.

K řešení lze přistoupit dvojím způsobem: buď na straně serveru, nebo na straně klientů.

Úprava na straně serveru by obnášela zavedení více seznamů účastníků – jeden pro každý model. Pumpa by při inicializaci musela předat informaci o tom, jaký konkrétní zrovna používá, podle které by jí server správně zařadil. Jako jednoznačný identifikátor lze využít opět GUID. Následně už by se pro pumpu nic neměnilo, vyžádala by si adresy dalších účastníků, se kterými by si mohla vyměňovat dosavadní výsledky. Server by pouze musel poskytnout adresy, které pochází ze stejného seznamu (tzn. adresy na pumpy, používající stejný model).

Alternativně by bylo možné nechat server tak, jak je, a pozměnit logiku klientů tak, že by při výměně parametrů uváděli navíc i identifikaci modelu, který používají. Bylo by pak možné odhalit nekonzistenci a případně zopakovat dotaz na jinou pumpu.

7.2 Výměna parametrů specifických pro optimalizační algoritmus

Všechny optimalizační algoritmy udržují jako svůj defacto stav aktuální řešení (tj. parametry modelu, se kterým pracují), nebo jejich sadu. Některé algoritmy ale mohou pracovat ještě s dalšími parametry, které nejsou přímo součástí modelu – příkladem je i implementovaný particle swarm optimization, kde si každá částice udržuje ak-

tuální polohu ve stavovém prostoru (tj. parametry modelu), ale navíc i svou aktuální rychlost, která s modelem přímo nesouvisí, ale optimalizace s ní pracuje.

Přenášení pouze parametrů modelu tak, jak je to v současnosti implementováno, zajišťuje nezávislost na konkrétní použité metodě optimalizace, protože ty musí být z principu přítomné za všech okolností. Prakticky ale nic nebrání přenášet i parametry navíc, naopak by mohla existovat šance, že si díky tomu předané řešení povede na „cizí" pumpě lépe. Pro běh optimalizace je stejně nutné tyto parametry vybraným způsobem stanovit, na příkladu particle swarm například tak, že je rychlost vynulována (viz kapitola 4.1.3.7).

7.3 Optimalizace využití sítě

Nevýhodou bezdrátové komunikace pomocí Wi-Fi je její poměrně nezanedbatelná spotřeba elektrické energie. Navíc nehraje příliš roli, jestli zařízení síť aktivně využívá, nebo ne – i při pasivní účasti je stále potřeba dodávat netriviální příkon. Např. [52] udává pro Raspberry Pi Zero W rozdíl 30 mA mezi vypnutou a zapnutou Wi-Fi při nečinnosti. Výměna informací ale trvá oproti identifikaci poměrně krátkou dobu, velká část energie, kterou Wi-Fi spotřebuje, je tedy zmařena.

Tento problém lze zcela eliminovat vypínáním Wi-Fi, pokud není zrovna používána, což ale za současného stavu nelze, protože pumpy mezi sebou nejsou časově synchronizovány a požadavky mohou posílat ve zcela náhodném čase.

Jednou možností je absence globálního času – tzn. hodiny každé pumpy v síti mají obecně odlišnou hodnotu a synchronizaci je nutné provádět pro každou dvojici komunikujících pump. Server navíc může udržovat synchronizační informace pro nově příchozí účastníky. Princip činnosti by mohl být následovný: pumpa se připojí k serveru, vyžádá si adresy dalších účastníků. Zároveň s nimi server poskytne informaci o době jejich aktivity (např. ve formě času do probuzení). Serveru je nazpět zaslána tatáž informace pro lokální pumpu. Následně je Wi-Fi vypnuta a zapínána vždy jen na určitý čas během předpokládané doby probuzení jiné pumpy, případně během svého udaného času probuzení, podle toho, co nastane dříve.

Přestože je tento způsob funkční a jeho idea použitelná například v bezdrátových senzorických sítích [53], kde závisí na každé miliampérhodině, pro tuto práci je zbytečně pracný a lze využít jiných nástrojů.

Jmenovitě je možné globální čas zavést pomocí protokolu NTP [54], který operační systémy umí zcela běžně využívat pro získání přesného času z Internetu. Pokud jsou všechny pumpy tímto způsobem synchronizované, lze předpokládat na všech dostatečně „stejný" čas pro řízení další komunikace. V tomto případě by nebylo potřeba výrazněji zasahovat do funkcionality programu – stačilo by doby aktivity omezit na předem stanovené časové intervaly, např. prvních pět minut hodiny. Během této doby by bylo možné kontaktovat libovolnou pumpu a vyměnit si

informace o průběhu identifikace. Po zbytek času by Wi-Fi zůstala vypnutá, i když v případě potřeby opětovné synchronizace nebo udržování relace se serverem je samozřejmě možné ji za tímto účelem zapnout.

7.4 Šifrování komunikace

Veškerá síťová komunikace byla navržena jako nešifrovaná, jelikož je takový přístup pro prvotní implementaci a ladění výrazně snažší. Pro ostrý provoz jinde než na izolované síti je nicméně vhodné šifrování zavést.

Hlavním problémem by byla distribuce klíčů, která je v mnoha ohledech podobná distribuci adres, popisované v kapitole 3.3. Stejně jako v případě adres je možné těžit ze zavedení serveru, který by navíc kromě adres mohl udržovat také veřejné klíče asymetrické šifry pro navázání spojení mezi pumpami.

Alternativou k plnohodnotnému šifrování by mohlo být pouze zajišťování integrity přenášených zpráv pomocí digitálních podpisů. Jak už bylo zmíněno, samotným odposlechem zpráv se útočník nedokáže dostat k citlivým datům, pouze jejich agregaci ve formě modelu. Zásadnějším problémem je možnost, že bude útočník uzly nadměrně zatěžovat, či injektovat nežádoucí informace do zpráv, čímž by snížil efektivitu algoritmu.

7.5 Inteligentní spojování klientů

Současná implementace nerozlišuje klienty mezi sebou, z hlediska serveru jsou všichni rovnocenní. Pokud si klient zažádá o adresy dalších účastníků, bude mu vrácena prakticky náhodná podmnožina všech aktivních klientů.

Výměna parametrů by ale teoreticky mohla být přínosnější, pokud by si byli navzájem komunikující pacienti podobní – např. věkem, pohlavím, hmotností. Server by tyto informace mohl využít pro výběr adresy, na které běží optimalizace pro co nejpodobnějšího pacienta, což by znamenalo složitější algoritmus pro procházení seznamu účastníků, případně nutnost účastníky dělit do skupin. Také by se mohlo stát, že by pacient nechtěl tyto osobní informace programu vůbec sdělit.

Závěr 8

Cílem této práce bylo prozkoumat, implementovat a na skutečných zařízeních otestovat způsoby sdílení informací a jejich dopadu na identifikaci/optimalizaci parametrů modelu.

Hlavní motivací je fakt, že identifikace modelu je výpočetně náročný numerický proces, u kterého není příliš vysoká šance úspěchu, pokud by měl běžet na výpočetně slabém nízkopříkonovém zařízení. Pokud by nicméně slabých zařízení bylo značné množství, společnými silami by mohly dosáhnout lepších výsledků.

Konkrétním uplatněním by v případě této práce byli pacienti trpící nemocí diabetes mellitus, kteří využívají inzulinovou pumpu. O tomto hardwaru platí to, co bylo zmíněno v odstavci výše, tj. je sice poměrně nevýkonný, ale vyskytuje se ho netriviální množství.

V rámci této práce byly navrženy dva „distribuované“ solvery pro framework SmartCGMS, jeden fungující na principu genetické evoluce a druhý na principu particle swarm optimization. Kromě samotné lokální optimalizace obsahují i síťovou část, která komunikuje s dalšími aktivními solvery a vyměňuje si s nimi parametry „zajímavých“ řešení. Tento způsob výměny je výhodný i z pohledu úspory přenášených dat a ochrany citlivých zdravotních informací.

Přestože jsou získaná řešení od ostatních vypočítávána pro jiného pacienta (a tedy samotný řešený problém je odlišný), jejich zanesení do populace lokálních řešení může algoritmu pomoci prozkoumat větší část stavového prostoru a případně uniknout z lokálních optim.

Součástí práce je také analýza a návrh síťové části, tj. síťové architektury a protokolu. V distribuované aplikaci, za jakou lze popisovaný problém označit, je především nutné vhodným způsobem získávat adresy ostatních účastníků. Jako nejvhodnější varianta bylo zvoleno ustanovení serveru, který bude tyto adresy sdružovat. Navržený protokol je minimalistický a zahrnuje komunikaci solveru se serverem a solverů mezi sebou.

Navržená aplikace byla otestována na několika minipočítačích Raspberry Pi Zero, výsledky přitom naznačují, že navržený přístup může dále vylepšit optimalizované parametry, přestože komplexní testování může být předmětem navazujících

prací.

Prezentováno je několik potenciálních rozšíření aplikace, jako úspora energie vypínáním Wi-Fi nebo podpora různých modelů.

Bibliografie

1. WORLD HEALTH ORGANIZATION. *Diabetes* [online]. 2023. [cit. 2024-01-01]. Dostupné z: https://www.who.int/news-room/fact-sheets/detail/diabetes.
2. POMPA, Marcello; PANUNZI, Simona; BORRI, Alessandro; GAETANO, Andrea De. *A comparison among three maximal mathematical models of the glucose-insulin system* [online]. 2021. [cit. 2024-01-01]. Dostupné z: https://journals.plos.org/plosone/article?id=10.1371/journal.pone.0257789.
3. SAUDEK, František. *Klinické Projevy | Cukrovka.cz* [online]. 2018. [cit. 2024-01-23]. Dostupné z: https://www.cukrovka.cz/hypoglykemie.
4. BRUTSAERT, Erika F. *Hypoglycemia - Endocrine and Metabolic Disorders - MSD Manual Professional Edition* [online]. 2023. [cit. 2024-01-01]. Dostupné z: https://www.msdmanuals.com/professional/endocrine-and-metabolic-disorders/diabetes-mellitus-and-disorders-of-carbohydrate-metabolism/hypoglycemia.
5. SAUDEK, František. *Klinické Projevy | Cukrovka.cz* [online]. 2018. [cit. 2024-01-23]. Dostupné z: https://www.cukrovka.cz/hyperglykemie.
6. CLEVELAND CLINIC. *Hyperglycemia* [online]. [cit. 2024-01-01]. Dostupné z: https://my.clevelandclinic.org/health/diseases/9815-hyperglycemia-high-blood-sugar.
7. SAUDEK, František. *Klinické Projevy | Cukrovka.cz*. 2018. Dostupné také z: https://www.cukrovka.cz/klinicke-projevy.
8. HALL, John E.; GUYTON, Arthur C. Textbook of Medical Physiology. In: Saunders Elsevier, 2011, kap. 78: Insulin, Glucagon, and Diabetes Mellitus. ISBN 978-1-4160-4574-8.
9. FOROUHI, Nita Gandhi; WAREHAM, Nicholas J. Epidemiology of diabetes. *Medicine*. 2019, roč. 47, č. 1, s. 22–27. ISSN 1357-3039. Dostupné z DOI: https://doi.org/10.1016/j.mpmed.2018.10.004.

10. MSD. *Diabetes Mellitus (DM) - Hormonal and Metabolic Disorders - MSD Manual Consumer Version* [online]. 2023. [cit. 2024-01-01]. Dostupné z: https://www.msdmanuals.com/en-gb/home/hormonal-and-metabolic-disorders/diabetes-mellitus-dm-and-disorders-of-blood-sugar-metabolism/diabetes-mellitus-dm.

11. CENTERS FOR DISEASE CONTROL AND PREVENTION. *What is Diabetes?* [online]. 2023. [cit. 2024-01-01]. Dostupné z: https://www.cdc.gov/diabetes/basics/diabetes.html.

12. SIMONE, Faccioli; ANDREA, Facchinetti; GIOVANNI, Sparacino; GIANLUIGI, Pillonetto; SIMONE, Del Favero. Linear Model Identification for Personalized Prediction and Control in Diabetes. *IEEE TRANSACTIONS ON BIOMEDICAL ENGINEERING*. 2022, roč. 69, č. 2, s. 558–568. ISSN 0018-9294. Dostupné z DOI: 10.1109/TBME.2021.3101589.

13. BERGMAN, R N; PHILLIPS, L S; COBELLI, C. Physiologic evaluation of factors controlling glucose tolerance in man: measurement of insulin sensitivity and beta-cell glucose sensitivity from the response to intravenous glucose. *The Journal of Clinical Investigation*. 1981, roč. 68, č. 6, s. 1456–1467. Dostupné z DOI: 10.1172/JCI110398.

14. COBELLI, Claudio et al. Diabetes: Models, Signals, and Control. *IEEE Reviews in Biomedical Engineering*. 2009, roč. 2, s. 54–96. Dostupné z DOI: 10.1109/RBME.2009.2036073.

15. SORENSEN, John Thomas. *A physiologic model of glucose metabolism in man and its use to design and assess improved insulin therapies for diabetes*. 1985. Dis. pr. Massachusetts Institute of Technology.

16. VISENTIN, Roberto et al. The UVA/Padova Type 1 Diabetes Simulator Goes From Single Meal to Single Day. *Journal of Diabetes Science and Technology*. 2018, roč. 12, č. 2, s. 273–281. Dostupné z DOI: 10.1177/1932296818757747. PMID: 29451021.

17. MEDTRONIC. *Medtronic MiniMed™ 630G* [online]. 2024. [cit. 2024-02-07]. Dostupné z: https://www.medtronicdiabetes.com/sites/default/files/library/download-library/user-guides/MiniMed-630G-System-User-Guide.pdf.

18. TANDEM DIABETES CARE, INC. *t:slim X2™* [online]. 2020. [cit. 2024-02-07]. Dostupné z: https://www.tandemdiabetes.com/docs/default-source/product-documents/t-slim-x2-insulin-pump/aw-1006690_b-user-guide-tslim-x2-basal-iq-6-4-mmoll-ces-artwork-web.pdf?sfvrsn=6f9817d7_147.

19. PAYNE, Matthew; POOKE, Francis; FULTON, Harrison; SHAW, Hamish; COULSON, Tom et al. Design of an open source ultra low cost insulin pump. *HardwareX*. 2022, roč. 12, e00375. ISSN 2468-0672. Dostupné z DOI: https://doi.org/10.1016/j.ohx.2022.e00375.

20. TENORIO, Fernanda Silva; MARTINS, Luiz Eduardo Galvão; CUNHA, Tatiana Sousa. Accuracy of a Low-Cost Continuous Subcutaneous Insulin Infusion Pump Prototype: In Vitro Study Using Combined Methodologies. *Annals of Biomedical Engineering*. 2021, roč. 49, č. 7, s. 1761–1773. ISSN 1573-9686. Dostupné z DOI: 10.1007/s10439-020-02721-8.

21. OPENAPS. *OpenAPS* [online]. [cit. 2024-01-28]. Dostupné z: https://openaps.org/.

22. OPENAPS. *OpenAPS Documentation* [online]. [cit. 2024-01-28]. Dostupné z: https://openaps.readthedocs.io/.

23. HARADA, Tomohiro; ALBA, Enrique. Parallel Genetic Algorithms: A Useful Survey. *ACM Comput. Surv.* 2020, roč. 53, č. 4. ISSN 0360-0300. Dostupné z DOI: 10.1145/3400031.

24. KAIROUZ, Peter; MCMAHAN, H. Brendan; AVENT, Brendan; BELLET, Aurélien; BENNIS, Mehdi et al. *Advances and Open Problems in Federated Learning*. 2021. Dostupné z arXiv: 1912.04977.

25. BLANCO-JUSTICIA, Alberto et al. Achieving security and privacy in federated learning systems: Survey, research challenges and future directions. *Engineering Applications of Artificial Intelligence*. 2021, roč. 106, s. 104468. ISSN 0952-1976. Dostupné z DOI: https://doi.org/10.1016/j.engappai.2021.104468.

26. LI, Qinbin et al. A Survey on Federated Learning Systems: Vision, Hype and Reality for Data Privacy and Protection. *IEEE Transactions on Knowledge and Data Engineering*. 2023, roč. 35, č. 4, s. 3347–3366. ISSN 2326-3865. Dostupné z DOI: 10.1109/tkde.2021.3124599.

27. NESMACHNOW, Sergio. An Overview of Metaheuristics: Accurate and Efficient Methods for Optimisation. *Int. J. Metaheuristics*. 2014, roč. 3, č. 4, s. 320–347. ISSN 1755-2176. Dostupné z DOI: 10.1504/IJMHEUR.2014.068914.

28. AL-BETAR, Mohammed Azmi. β-Hill climbing: an exploratory local search. *Neural Computing and Applications*. 2017, roč. 28, č. 1, s. 153–168. ISSN 1433-3058. Dostupné z DOI: 10.1007/s00521-016-2328-2.

29. DOWSLAND, Kathryn A.; THOMPSON, Jonathan M. Simulated Annealing. In: *Handbook of Natural Computing*. Ed. ROZENBERG, Grzegorz; BÄCK, Thomas; KOK, Joost N. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, s. 1623–1655. ISBN 978-3-540-92910-9. Dostupné z DOI: 10.1007/978-3-540-92910-9_49.

30. LEEMON@CS.CMU.EDU, Leemon Baird. *What is Simulated Annealing?* [online]. [cit. 2024-01-09]. Dostupné z: https://www.cs.cmu.edu/afs/cs.cmu.edu/project/learn-43/lib/photoz/.g/web/glossary/anneal.html.

31. XIONG, N.; MOLINA, D.; ORTIZ, M. L. et al. A Walk into Metaheuristics for Engineering Optimization: Principles, Methods and Recent Trends. *Int J Comput Intell Syst*. 2015, roč. 8, s. 606–636. Dostupné také z: https://doi.org/10.1080/18756891.2015.1046324.

32. SHARMA, Anuraganand. Guided Stochastic Gradient Descent Algorithm for inconsistent datasets. *Applied Soft Computing*. 2018, roč. 73, s. 1068–1080. ISSN 1568-4946. Dostupné z DOI: https://doi.org/10.1016/j.asoc.2018.09.038.

33. MEHTA, Pankaj et al. A high-bias, low-variance introduction to Machine Learning for physicists. *Physics Reports*. 2019, roč. 810, s. 1–124. ISSN 0370-1573. Dostupné z DOI: https://doi.org/10.1016/j.physrep.2019.03.001. A high-bias, low-variance introduction to Machine Learning for physicists.

34. WHITLEY, Darrell; SUTTON, Andrew M. Genetic Algorithms — A Survey of Models and Methods. In: *Handbook of Natural Computing*. Ed. ROZENBERG, Grzegorz; BÄCK, Thomas; KOK, Joost N. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, s. 637–671. ISBN 978-3-540-92910-9. Dostupné z DOI: 10.1007/978-3-540-92910-9_21.

35. SHI, Yujiao; GAO, Hao; WU, Dongmei. An improved differential evolution algorithm with novel mutation strategy. In: *2014 IEEE Symposium on Differential Evolution (SDE)*. 2014, s. 1–8. Dostupné z DOI: 10.1109/SDE.2014.7031540.

36. CORNE, David W.; REYNOLDS, Alan; BONABEAU, Eric. Swarm Intelligence. In: *Handbook of Natural Computing*. Ed. ROZENBERG, Grzegorz; BÄCK, Thomas; KOK, Joost N. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, s. 1599–1622. Dostupné z DOI: 10.1007/978-3-540-92910-9_48.

37. KARABOGA, Dervis; GORKEMLI, Beyza; OZTURK, Celal; KARABOGA, Nurhan. A comprehensive survey: Artificial bee colony (ABC) algorithm and applications. *Artificial Intelligence Review*. 2012, roč. 42. Dostupné z DOI: 10.1007/s10462-012-9328-0.

38. KOUTNY, Tomas; UBL, Martin. SmartCGMS as a Testbed for a Blood-Glucose Level Prediction and/or Control Challenge with (an FDA-Accepted) Diabetic Patient Simulation. *Procedia Computer Science*. 2020, roč. 177, s. 354–362. ISSN 1877-0509. Dostupné z DOI: https://doi.org/10.1016/j.procs.2020.10.048. The 11th International Conference on Emerging Ubiquitous Systems and Pervasive Networks (EUSPN 2020) / The 10th International Conference on Current and Future Trends of Information and Communication Technologies in Healthcare (ICTH 2020) / Affiliated Workshops.

39. FAV ZČU. *Portál diabetes.zcu.cz, SmartCGMS* [online]. [cit. 2024-01-08]. Dostupné z: https://diabetes.zcu.cz/smartcgms/.

40. KOUTNY, Tomas; UBL, Martin. Parallel software architecture for the next generation of glucose monitoring. *Procedia Computer Science*. 2018, roč. 141, s. 279–286. ISSN 1877-0509. Dostupné z DOI: https://doi.org/10.1016/j.procs.2018.10.197. The 9th International Conference on Emerging Ubiquitous Systems and Pervasive Networks (EUSPN-2018) / The 8th International Conference on Current and Future Trends of Information and Communication Technologies in Healthcare (ICTH-2018) / Affiliated Workshops.

41. UBL, Martin; KOUTNY, Tomas. SmartCGMS as an Environment for an Insulin-Pump Development with FDA-Accepted In-Silico Pre-Clinical Trials. *Procedia Computer Science*. 2019, roč. 160, s. 322–329. ISSN 1877-0509. Dostupné z DOI: https://doi.org/10.1016/j.procs.2019.11.084. The 10th International Conference on Emerging Ubiquitous Systems and Pervasive Networks (EUSPN-2019) / The 9th International Conference on Current and Future Trends of Information and Communication Technologies in Healthcare (ICTH-2019) / Affiliated Workshops.

42. GOOGLE. *Google IPv6 Statistics* [online]. 2024. [cit. 2024-03-02]. Dostupné z: https://www.google.com/intl/en/ipv6/statistics.html%5C#tab=ipv6-adoption.

43. FORD, Bryan; SRISURESH, Pyda; KEGEL, Dan. *Peer-to-Peer Communication Across Network Address Translators* [online]. [cit. 2024-03-07]. Dostupné z: https://bford.info/pub/net/p2pnat/.

44. AUDET, Francois; JENNINGS, Cullen Fluffy. *RFC 4787: Network Address Translation (NAT) Behavioral Requirements for Unicast UDP*. 2007. Dostupné také z: https://datatracker.ietf.org/doc/html/rfc4787.

45. *Internet Protocol* [RFC 791]. RFC Editor, 1981-09. Request for Comments, č. 791. Dostupné z DOI: 10.17487/RFC0791.

46. FAV ZČU. *Portál diabetes.zcu.cz, Documentation* [online]. [cit. 2024-01-08]. Dostupné z: https://diabetes.zcu.cz/docs.

47. LEACH, Paul J.; SALZ, Rich; MEALLING, Michael H. *A Universally Unique IDentifier (UUID) URN Namespace* [RFC 4122]. RFC Editor, 2005-07. Request for Comments, č. 4122. Dostupné z DOI: 10.17487/RFC4122.

48. STOUT, Bob. *The SNIPPETS C Source Code Archives* [online]. [cit. 2024-02-05]. Dostupné z: https://web.archive.org/web/20080217222203/http://c.snippets.org/.

49. MARLING, Cindy; BUNESCU, Razvan. The OhioT1DM Dataset for Blood Glucose Level Prediction: Update 2020. *CEUR Workshop Proc*. 2020, roč. 2675, s. 71–74.

50. UBL, Martin; KOUTNY, Tomas. A Novel Approach to Multi-Compartmental Model Implementation to Achieve Metabolic Model Identifiability on Patient's CGM Data. *Procedia Computer Science*. 2022, roč. 210, s. 116–123. ISSN 1877-0509. Dostupné z DOI: https://doi.org/10.1016/j.procs.2022.10.127. The 13th International Conference on Emerging Ubiquitous Systems and Pervasive Networks (EUSPN) / The 12th International Conference on Current and Future Trends of Information and Communication Technologies in Healthcare (ICTH-2022) / Affiliated Workshops.

51. GEERLING, Jeff. *Raspberry Pi Zero - Power Consumption Comparison | Jeff Geerling* [online]. 2015. [cit. 2024-05-13]. Dostupné z: https://www.jeffgeerling.com/blogs/jeff-geerling/raspberry-pi-zero-power.

52. HAWKINS, Matt. *Raspberry Pi Power Consumption Data - Raspberry Pi Spy* [online]. 2018. [cit. 2024-04-09]. Dostupné z: https://www.raspberrypi-spy.co.uk/2018/11/raspberry-pi-power-consumption-data/.

53. BHENDE, Manisha; WAGH, Sanjeev J.; UTPAT, Amruta. A Quick Survey on Wireless Sensor Networks. In: *2014 Fourth International Conference on Communication Systems and Network Technologies*. 2014, s. 160–167. Dostupné z DOI: 10.1109/CSNT.2014.40.

54. MARTIN, Jim; BURBANK, Jack; KASCH, William; MILLS, Professor David L. *Network Time Protocol Version 4: Protocol and Algorithms Specification* [RFC 5905]. RFC Editor, 2010-06. Request for Comments, č. 5905. Dostupné z DOI: 10.17487/RFC5905.

Seznam obrázků

Seznam tabulek

Seznam výpisů

Popis adresářové struktury příloh

- `/Aplikace_a_knihovny`
  - `/networking`: Adresář se zdrojovými soubory síťové části.
  - `/scgms_solver`: Adresář se zdrojovými soubory optimalizační části.
- `/Poster`: Adresář se soubory posteru.
- `/Text_prace`
  - `/DP.pdf`: Zkompilovaná práce.
  - `/main.tex, /references.bib`: Zdrojové soubory pro LaTeX.
  - `/img`: Adresář obsahující grafiku a zadání práce bez podpisů.
- `/Vstupni_data`: Adresář s konfiguracemi filtrů SmartCGMS scénáři.
  - `/Patient`: Adresář s konfigurací a scénáři pro idenfitikaci parametrů pacientského modelu.
  - `/PID`: Adresář s konfiguracemi a scénářem pro identifikaci parametrů PID regulátoru.
- `/Vysledky`: Adresář s výstupními (optimalizovanými) konfiguracemi pro různé scénáře.
  - `/pid_ge`: Adresář se soubory pro genetickou evoluci (identifikace PID).
    - `/ref`: Výsledky optimalizace samostatných pump bez komunikace.
    - `/equal`: Výsledky optimalizace komunikujících pump se společným pacientem.
    - `/full`: Výsledky optimalizace komunikujících pump nad různými pacienty.

  - `/pid_pso`: Adresář se soubory pro particle swarm optimization (identifikace PID) se stejnou struktuou jako adresář `/ge`.
  - `/patient_pso`: Adresář se soubory pro particle swarm optimization (identifikace pacientského modelu) se stejnou struktuou jako adresář `/ge`.

- `/Readme.txt`: Textový soubor s popisem adresářové struktury.

Uživatelská příručka

Pro jednoduchý překlad aplikace jsou mezi zdrojovými kódy dva skripty aplikace CMake CMakeLists.txt – jeden pro solvery v `./scgms_solver`, druhý pro server v `./networking`. CMake vygeneruje samotné soubory pro překlad (např. Makefile), kterými je možné projekt zkompilovat.

Při invokaci CMake pro vytvoření souborů solverů je nutné dodefinovat proměnnou `SMARTCGMS_COMMON_DIR` s cestou ke `common` adresáři SmartCGMS, např:

```
.../scgms_solver> cmake -B ../solver_build \\
-DSMARTCGMS_COMMON_DIR=/tmp/scgms/common
```

Při překladu serveru není potřeba definovat žádné další proměnné, tedy např.:

```
.../networking> cmake -B ../server_build
```

Následně je možné ve složce uvedené v předchozím kroku s přepínačem `-B` spustit kompilaci shodně pro server i solvery, pro Makefile např.:

```
.../solver_build> make
```

Výsledný přeložený soubor bude umístěn do stejné složky. V případě SmartCGMS solverů se jedná o sdílenou knihovnu, kterou je třeba zkopírovat/přemístit do složky `filters`, nacházející se ve složce se spustitelnou aplikací SmartCGMS.

V případě serveru je výstupem spustitelná aplikace.

Před spuštěním obou částí je třeba, aby se v aktivní složce nacházel soubor `params.txt`, ve kterém bude specifikováno, na jaké IP adrese a portu server běží (pro solvery – kam se připojovat, pro server – s jakou adresou a portem se svázat), např. ve tvaru:

```
server address: 127.0.0.1
server port: 10000
```

Server lze spustit z příkazové řádky tak, jak je, bez dalších parametrů, např.:

```
.../server_build> ./server
```

Pro spuštění SmartCGMS solveru je nejprve nutné překopírovat/přesunout konfigurační (.ini) soubory a scénáře (.csv) do složky se spustitelnou aplikací SmartCGMS.

Pro spuštění optimalizace je při spouštění konzolové aplikace potřeba specifikovat název konfiguračního souboru, přepínač `-o`, za přepínačem `-r` UUID požadovaného solveru (303ba959-1fbf-42f0-a95b-68a40efdf0b8 pro particle swarm optimization, 0f0f437e-2824-4877-a6ae-0576548a856c pro genetickou evoluci, tyto hodnoty je možné z příkazové řádky vypsat spuštěním konzolové aplikace SmartCGMS bez jakýchkoli parametrů) a za přepínačem `-p` index optimalizovaného filtru a jméno jeho konkrétního parametru. Dále je možné specifikovat velikost populace za přepínačem `-z` a počet iterací za přepínačem `-g` – pokud tyto hodnoty nebudou uvedeny, použijí se defaultní.

Spuštění optimalizace pro konfigurační soubor `gct2_001_bpid.ini` nad filtrem s indexem 5 a jeho parametry „Model_Bounds“ skrz příkazovou řádku se solverem na bázi genetické evoluce s velikostí populace 50 jedinců a 100 iteracemi:

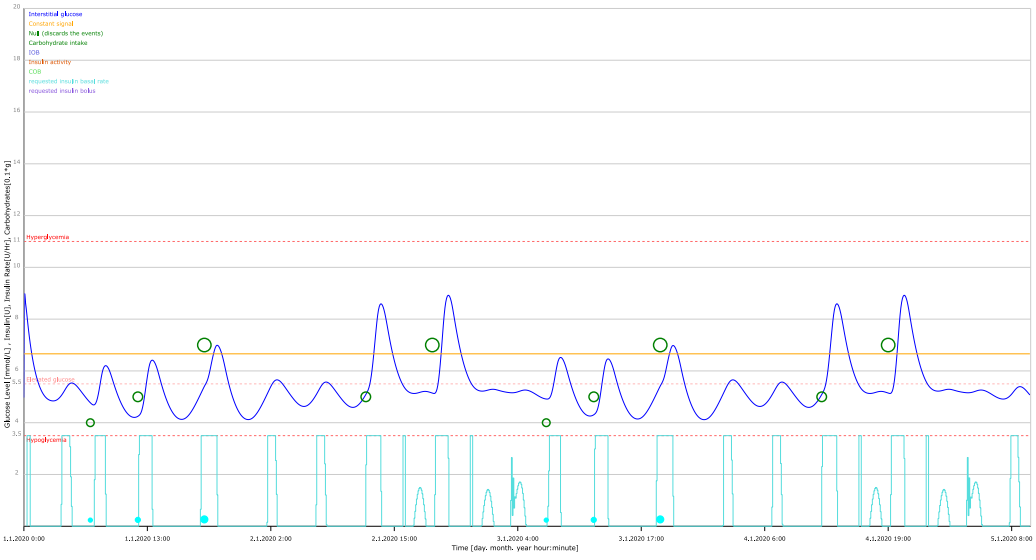
```
   .../scgms/build> ./scgms-console gct2_001_bpid.ini -o \\
-r0f0f437e-2824-4877-a6ae-0576548a856c -p5,Model_Bounds \\
-z50 -g1000
```

Výstupní grafy

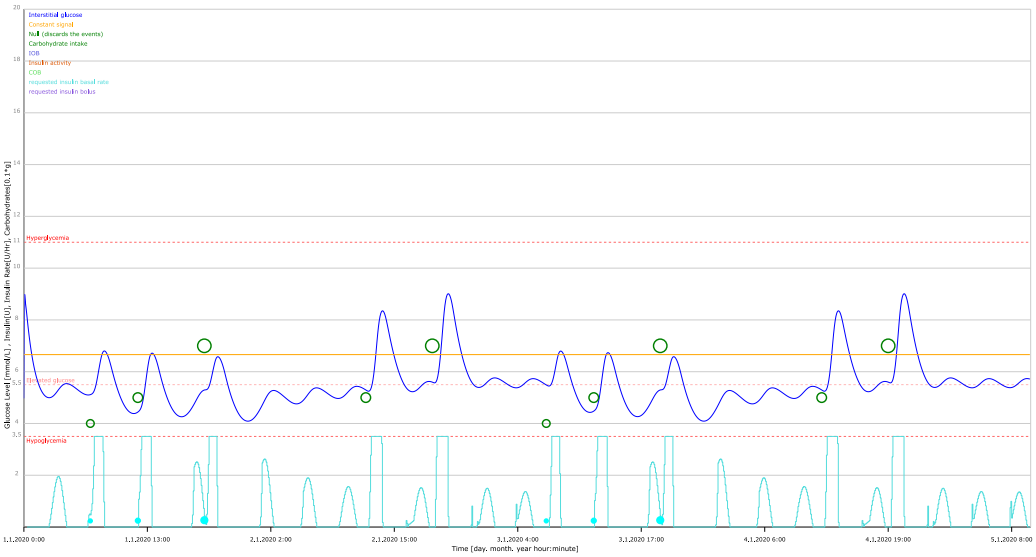
Grafy vygenerované simulací optimalizovaných .ini souborů ve SmartCGMS. Všechny grafy jsou pro pacienta 1.

Identifikace parametrů PID regulátoru

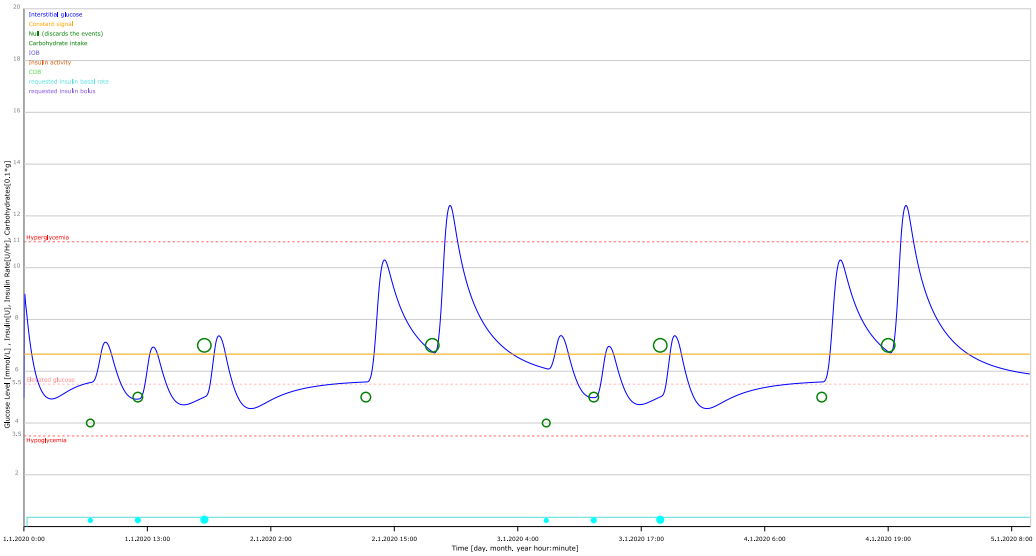
Grafy získané po identifikaci parametrů PID regulátoru na Raspberry Pi Zero. **Tmavě modrá čára odpovídá simulované koncentraci glukózy, žlutá čára (konstantní signál) představuje požadovanou úroveň**.



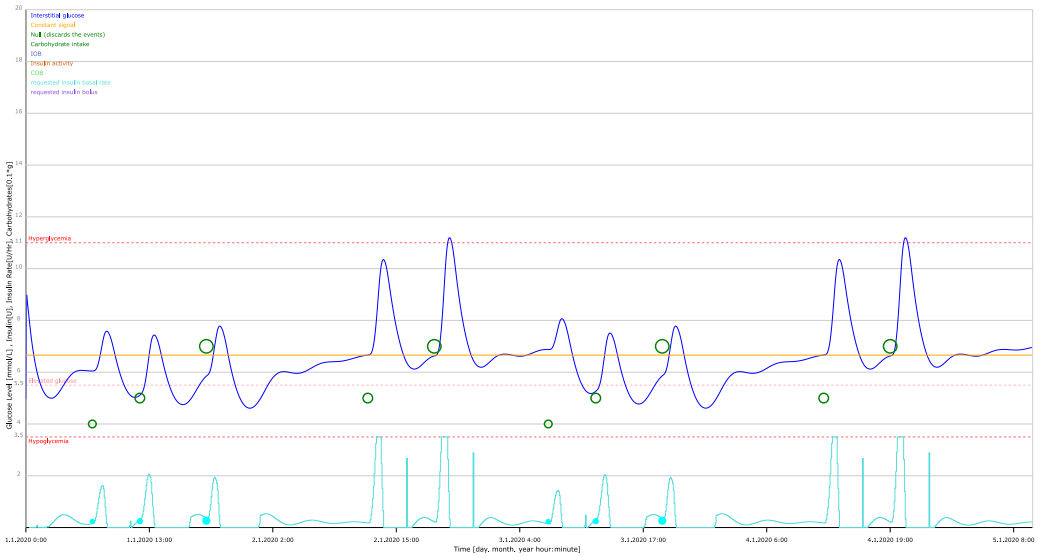
Obrázek 1: Výstupní graf při využití genetické evoluce, bez komunikace.



Obrázek 2: Výstupní graf při využití genetické evoluce s komunikací a jinými pacienty na každé pumpě.



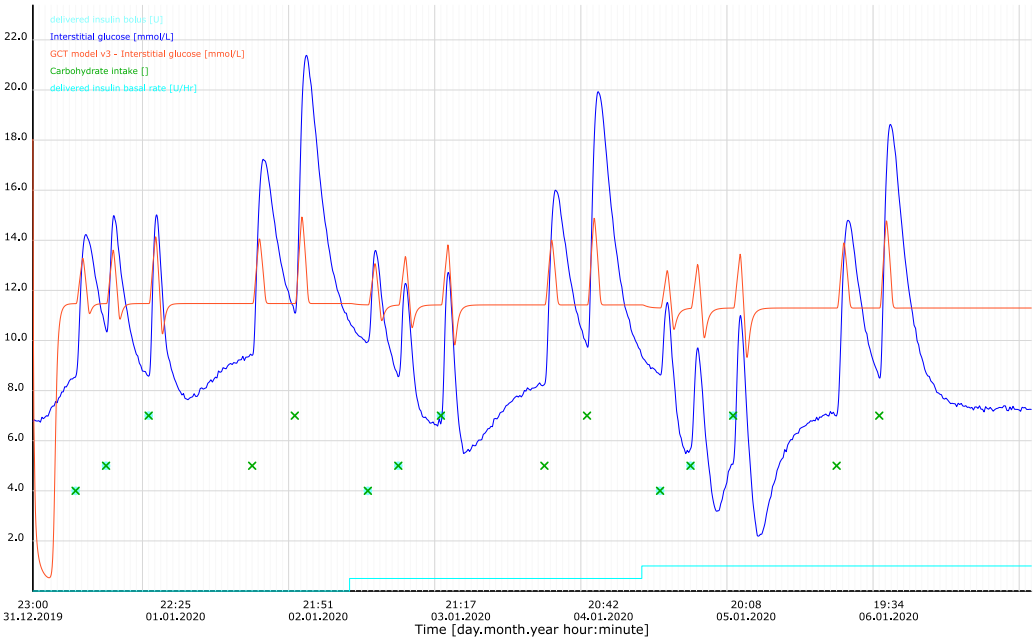
Obrázek 3: Výstupní graf při využití particle swarm optimization, bez komunikace.



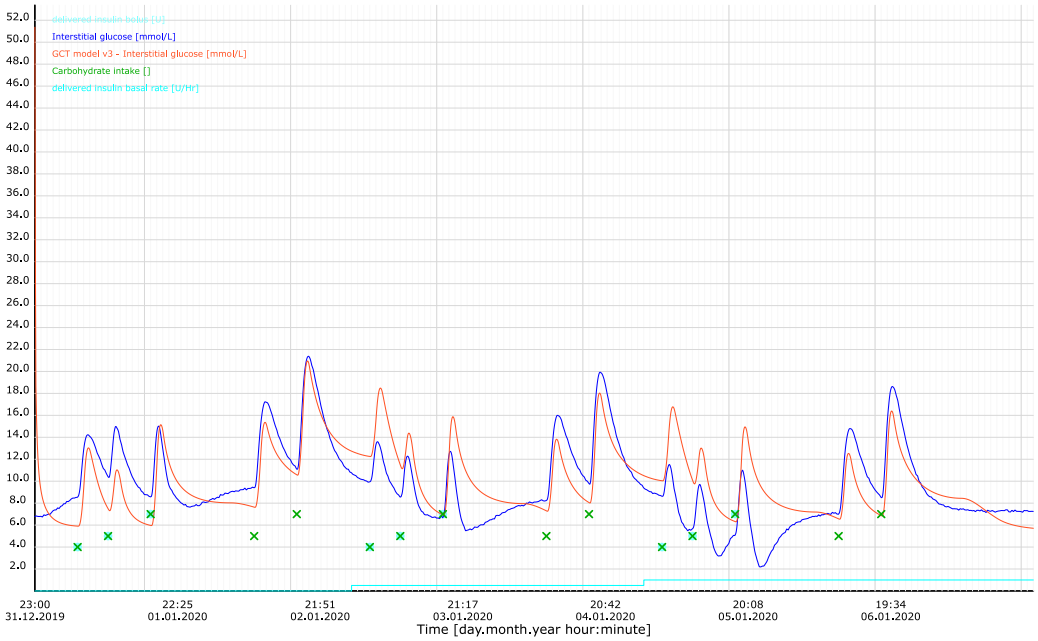
Obrázek 4: Výstupní graf při využití particle swarm optimization s komunikací a jinými pacienty na každé pumpě.

Identifikace parametrů pacientského modelu

Grafy získané po identifikaci pacientského modelu na stolním počítači. **Tmavě modrá čára odpovídá „skutečné“ koncentraci glukózy, červená čára představuje predikci modelu**.



Obrázek 5: Výstupní graf bez využití komunikace.



Obrázek 6: Výstupní graf při využití komunikace a jinými pacienty na každé pumpě.