FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

Zadání diplomové práce

Název:	Efektivní komunikace více vision systémů
Student:	Bc. Jindřich Kuzma
Vedoucí:	Ing. Jakub Novák
Studijní program:	Informatika
Obor / specializace:	Softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	do konce letního semestru 2023/2024

Pokyny pro vypracování

Cílem práce je navrhnout architekturu a vytvořit efektivní komunikaci více vision systémů s cílem efektivního poskytování a vyhodnocování obrazových dat. Aplikace cílí na komunikaci a předávání dat mezi více PC v jedné síti.

Existuje N zdrojů obrazových dat (kamer). Data jsou zpracovávána pomocí algoritmů pro rozpoznávání obrazu. Zpracovaná data je potřeba dále agregovat, uložit a zpřístupnit uživateli.

Úkoly:
1) Proveďte rešerši v oblasti komunikace více systémů s přihlédnutím k možnostem komunikace vision systémů.
2) Navrhněte architekturu systému, která bude zohledňovat počet kamer a umožní zpracování dat.
3) Implementujte navrženou architekturu s možností měřit vytíženost systému (výkonnost algoritmů pro rozpoznávání obrazu).
4) Otestujte funkčnost systému.
5) Zhodnoťte škálovatelnost a efektivitu implementovaného řešení (proveďte benchmark).

Elektronicky schválil/a Ing. Michal Valenta, Ph.D. dne 6. února 2023 v Praze.

Diplomová práce

EFEKTIVNÍ KOMUNIKACE VÍCE VISION SYSTÉMŮ

Bc. Jindřich Kuzma

Fakulta informačních technologií
Katedra softwarového inženýrství
Vedoucí: Ing. Jakub Novák
4. května 2023

České vysoké učení technické v Praze
Fakulta informačních technologií
© 2023 Bc. Jindřich Kuzma. Všechna práva vyhrazena.
Tato práce vznikla jako školní dílo na Českém vysokém učení technickém v Praze, Fakultě informačních technologií. Práce je chráněna právními předpisy a mezinárodními úmluvami o právu autorském a právech souvisejících s právem autorským. K jejímu užití, s výjimkou bezúplatných zákonných licencí a nad rámec oprávnění uvedených v Prohlášení, je nezbytný souhlas autora.

Odkaz na tuto práci: Jindřich Kuzma. *Efektivní komunikace více vision systémů.* Diplomová práce. České vysoké učení technické v Praze, Fakulta informačních technologií, 2023.

Obsah

Seznam obrázků

Seznam tabulek

Seznam výpisů kódu

Chtěl bych poděkovat Ing. Jakubu Novákovi za odborné vedení, pomoc a rady při zpracování této práce. Děkuji také své rodině a přátelům za jejich podporu.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací. Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů, zejména skutečnost, že České vysoké učení technické v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 citovaného zákona.

V Praze dne 4. května 2023 .....

Abstrakt

Cílem je navrhnout systém, který zvládne zpracovat data z více vision systémů. Řešení je postaveno na vztahu klient-server. Celkem vznikly 4 samostatné aplikace. Pro implementaci byl použit jazyk C++. Testy funkčnosti byly úspěšné. V oblasti škálovatelnosti zvládá řešení obsluhovat 30 zdrojů dat a zpracovávat vstupní data o velikosti 100 MB z 10 zdrojů za průměrně 4 s.

Klíčová slova vision systém, softwarový návrh, C++, síťová komunikace

Abstract

The goal is to design a system that can handle data from multiple vision systems. The solution is built on a client-server relationship. A total of 4 separate applications were created. The C++ language was used for the implementation. Functionality tests were successful. In terms of scalability, the solution can handle 30 data sources and process 100 MB of input data from 10 sources in an average of 4 seconds.

Keywords vision system, software design, C++, network communication

Seznam zkratek

TCP	Transmission Control Protocol
UDP	User Datagram Protocol
HTTP	Hypertext Transfer Protocol
REST	Representational State Transfer
XML	Extensible Markup Language
OCR	Optical character recognition
DRY	Don't repeat yourself
RPC	Remote procedure call

Kapitola 1
Úvod

Vision systém je nástroj pro automatizovanou kontrolu produktů. Kontrolovat lze produkty od léků, přes matičky a šroubky až po celá auta. Většinou jsou použita obrazová data z kamer, ale je možné použít i další senzory. Příkladem může být použití laserových senzorů pro měření vzdálenosti za účelem zjištění rozměrů produktu. Může se jednat i o spojité produkty, jako je například textil, kde vzniká konstantní proud obrazových dat.

Vision systém přispívá k zvýšení kvality zboží. Řešení mohou být komplexní a obsahovat desítky kamer. Firma A vyrábí tabulky čokolády o velikosti 1 x 2 metrů. Pro kompletní kontrolu potřebuje 10 řádkových kamer. Čokoláda je dále balena do dárkového balení. To se musí také zkontrolovat, zde stačí 3 kamery. Nakonec se přidá kartička s věnováním. Ta je zkontrolována pomocí OCR algoritmů. Celkem je systém složen z 14 kamer a výroba trvá 2 hodiny.

Cílem práce je poskytnou softwarové řešení, které ulehčí vytvoření softwaru pro tuto problematiku. Samotné zpracování obrazu je úzce spjaté s doménou. Zbytek procesu lze zobecnit za použití metod softwarového návrhu. Správný návrh usnadní práci expertům v oblasti strojového vidění. Pro uvedený příklad, může komplexní řešení nabídnout možnost archivace dat. Historická data mohou být použita při reklamacích, či jako zpětná vazba při zlepšování rozpoznávacích algoritmů. Posílání dat přes síť a ukládání na disk ale nijak nesouvisí s rozpoznáváním obrazu. Zde přichází na řadu softwarové inženýrství.

Kapitola 2
Rešerše

Vision systémy se používají ke kontrole kvality produktů. Prvním příkladem je firma Optel [1]. Ta se zaměřuje na oblast léčiv. Podle informací dostupných z [2] je zjevné, že kontrolovat se dá vše od samotného produktu až po jeho obal. Na obrázku 2.1 je vidět, jak tato kontrola může vypadat.

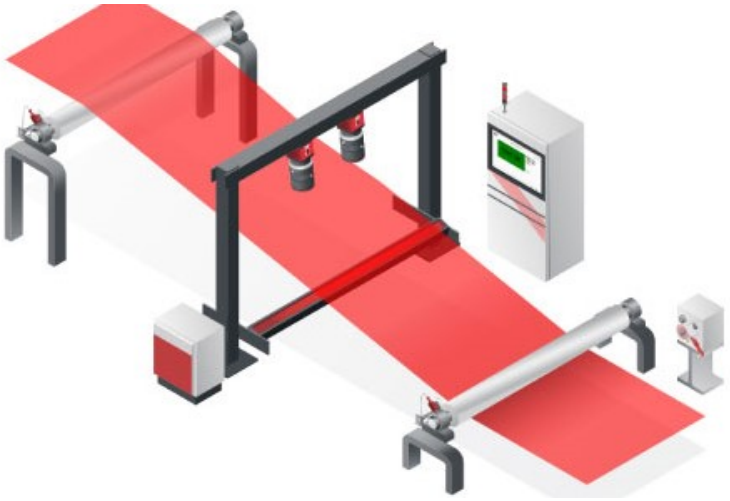
Obrázek 2.1 Použití vision systémů firmou Optel [2].

Optel uvádí, že může poskytnout systém zpracovávající až 1200 snímků za minutu a s neomezeným množstvím připojených kamer a skenerů.

Firma Industrial Vision Systems [3] nabízí velkou škálu řešení s využitím v různých průmyslových odvětvích. Pro potravinářský průmysl je k dispozici systém, který je schopný detekovat mimo jiné defekty, nežádoucí objekty, správnou porci a tvar zboží. Vadné produkty jsou automaticky zamítnuty a odstraněny z produkčního řetězce.

V automobilovém průmyslu lze využít kamery k vylepšení robotických rukou. Přidaná kamera může být použita k nafocení produktů s komplexními tvary z různých úhlů, nebo k ovládání robotické ruky. Vision systém lze také využít pro třídění produktů podle jakosti, zde je uváděna rychlost zpracování až 600 snímků za minutu.Kontrola velkých či spojitých objektů jako je například textil je řešena využitím řádkových kamer, znázorněno na obrázku 2.2. Pro snímkování za účelem archivace je uveden limit 30 kamer.

Firma Dr.Schenk [5] se specializuje na kontrolu kvality povrchů. Konkrétně se jedná například o sklo, textil, papír a železo. Tato řešení využívají řádkové kamery stejně jako na obrázku 2.2. Dr.Schenk využívá ve svých řešeních technologii MIDA [6]. Kamery zachycují stejný obraz v různých barevných spektrech. Například infračervené světlo může pomoci odhalit defekty na



Obrázek 2.2 Řádková kontrola produktů od Industrial Vision Systems [4].

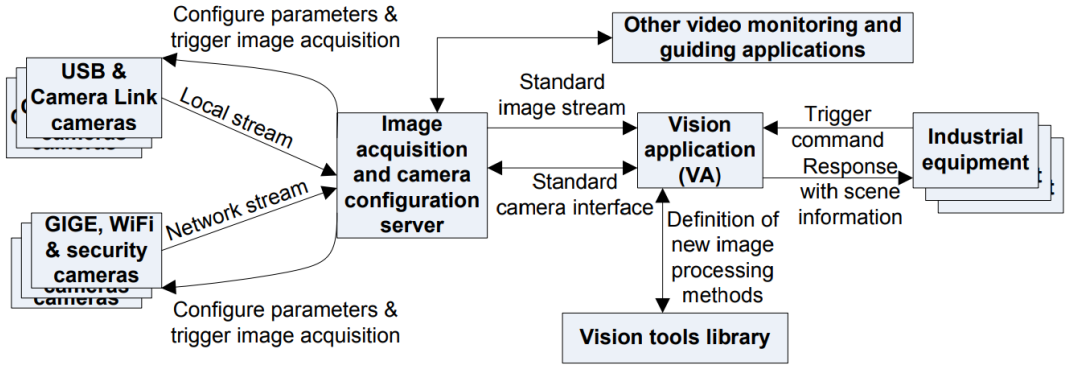
plastech, které je jinak obtížné odhalit za normálního osvětlení.

Firma ISRA Vision [7] má stejnou specializaci jako předchozí firma. Na svých webových stránkách zmiňují kontrolu automobilů, povrchů ale třeba i solárních panelů a baterií do elektrických aut. V [8] je uvedeno, že poskytnuté řešení zvládá zkontrolovat až 4500 waferů za hodinu.

EPROMI Live [9] je softwarové řešení, které integruje data z inspekčních systémů firmy ISRA. Data jsou k dispozici v rámci webového rozhraní. Systém nabízí možnost automatického upozorňování na problémy, které mohou při výrobním procesu nastat. Pokud se například zvýší počet defektů v krátkém období. Monitorování by mělo přispět k zvýšení výtěžnosti a kvality produkce.

Firma Cognex nabízí 1D , 2D a 3D vision systémy. Řešení Edge Intelligence [10] nabízí hromadný sběr z těchto systémů za účelem monitorování. Ve specifikaci je uvedeno, že model EI-700 zvládá sbírat data až z 20 zařízení. Jediný dostupný údaj o datech je popsán u výpočtu kapacity úložiště, kde je použita frekvence 1 snímek za sekundu při rozlišení 3 MP ve formátu JPEG.

V článku [11] je popsáno, jak lze použít vision systém k ovládání robotů. Na obrázku 2.3 je znázorněna použitá architektura. Řešení využívá vztah klient-server. Kromě samotného běhu programu je uveden také proces offline konfigurace. Ten spočívá v kalibraci kamer.



Obrázek 2.3 Diagram komponent z článku [11].

Pro komunikaci více aplikací se kromě klient-server dá použít i vztah peer to peer. Hlavním rozdílem jsou zodpovědnosti mezi aplikacemi. U klient-server je server zodpovědný za vše. Oproti tomu u peer to peer má každá aplikace svou vlastní autoritu. [12]

Na uvedených konceptech se dále staví vysokoúrovňové návrhy. Příkladem je vývojová platforma Rasa [13] umožňující tvorbu chatbotů. Ta pro komunikaci mezi jednotlivými částmi nabízí použití RabbitMQ. RabbitMQ [14] umožňuje výměnu zpráv mezi aplikacemi za použití front.

Firma Netflix používá pro škálování svého softwaru microservices. Snaží se tím docílit vysoké dostupnosti služeb. [15]

Kapitola 3
Technologie

K pochopení problematiky jsou rozpracovány následující knihovny a postupy.

3.1 Knihovny

Zajímavé jsou především knihovny související s obsluhou kamer a dále s následným zpracováním jejich výstupu.

3.1.1 Práce s kamerami

Dostupné programové rozhraní úzce souvisí s výrobcem kamery. Pro rozhraní dostupné napříč vícero kamerami se používá GenICam.

GenICam GenIcam [16] je knihovna, která normalizuje rozhraní pro kamery od různých výrobců. Základem je soubor ve formátu XML, který popisuje, jak s kamerou zacházet. Tuto specifikaci dodává výrobce kamery. Knihovna se skládá z několika rozhraní.

GenApi slouží pro práci s kamerou a využívá zmíněný XML soubor.

GenTL se používá pro získání obrázků z kamery. Specifikace používá programovací jazyk C. Hlavní důvody použití C jsou snadné napojení do jiných programovacích jazyků, dynamické načítání knihovny a možnost aktualizovat knihovnu bez nutnosti rekompilace pokud je zachována binární kompatibilita.

SFNC obsahuje popis jednotlivých funkcí, které kamery mohou mít a přiřazuje jim jednotné jméno. Funkce dále mohou mít specifikováno, jak jsou ovládány, například že na vstupu očekávají celočíselnou hodnotu.

Basler Firma Basler nabízí pro obsluhu svých kamer Pylon API. API je pro platformu .NET a jazyky C a C++. Pylon Viewer je grafické rozhraní, které umožňuje měnit parametry kamery. Pro práci s kamerou je potřeba instalovat ovladače na cílovém zařízení. [17]

Baumer NeoAPI se používá k ovládání kamer od firmy Baumer. Dostupné je pro jazyky C++, C# a Python. Knihovna podporuje operační systémy Windows a Linux. Dokumentace k API není volně dostupná na internetu. [18]

Zivid Zivid nabízí produkty zaměřené na pořizování 3D obrázků. API je dostupné v jazycích C++, Python a .NET. Pro nastavení kamery lze použít nástroj Zivid Studio s grafickým rozhraním. [19]

3.1.2 Práce s obrazem

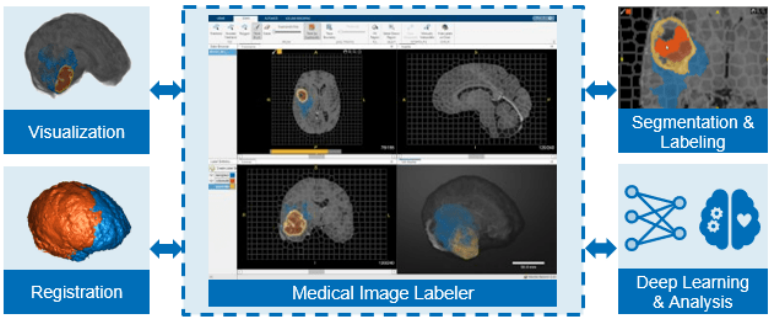
OpenCV OpenCV je od verze 2.x, postavené na programovacím jazyce C++. Verze 1.x podporující jazyk C je zastaralá. Dokumentace zmiňuje využití v dalších jazycích. OpenCV-Python je mířené na využití Pythonu jako vysokoúrovňového jazyka s tím, že se volají procedury v C++ pro zajištění výkonu. OpenCV.js míří na použití ve webových prohlížečích pomocí JavaScriptu a WebAssembly.

Knihovna nabízí vlastní rozhraní pro získávání obrazových dat za použití třídy `VideoCapture`. Dále knihovna definuje vlastní datové typy a pro operace s pixely je použita výpočetní metoda `Saturation Arithmetics`. Informace jsou z dokumentace. [20]

TensorFlow TensorFlow je knihovna zaměřené na strojové učení. V ukázce [21] je popsáno jak se dá použít k rozpoznávání obrazu. V tomto případě je použit dataset obsahující 70000 kusů oblečení k naučení modelu. Knihovna má rozhraní dostupné v jazyce Python. Další jazyky jsou buď podporované komunitou nebo přímo dodavatelem knihovny, ale bez záruky kompatibility mezi verzemi. [22]

MathWorks MathWorks nabízí řešení zvané Image Processing Toolbox. Nejedná se přímo o samostatné API jako v předešlých případech, ale o komplexní řešení s vizualizací a prostředím pro práci se strojovým viděním. K dispozici je možnost následně převést kód do C/C++. [23]

Za zmínění také stojí Medical Imaging Toolbox, ukázán na obrázku 3.1. Jedná se o nástroj, který používá grafické rozhraní pro usnadnění zpracování obrazových dat z oblasti medicíny. [24]



Obrázek 3.1 Ukázka Medical Imaging Toolbox. [24]

NVIDIA Nvidia nabízí knihovnu VPI, která implementuje algoritmy pro zpracování obrazu a počítačové vidění. Knihovna je dostupná v jazce C/C++ a Python. VPI umožňuje spouštět algoritmy na procesoru, grafické kartě a akcelerátorech jako je třeba PVA. Knihovna zvládá spolupracovat s jinými knihovnami jako je OpenCV a klade si za cíl být výkonnější variantou. [25]

BoofCV BoofCV je knihovna s otevřeným zdrojovým kódem dostupná pod licencí Apache 2.0. Knihovna používá jazyk Java a je rozdělena do několika logických částí. Například část `Image processing` obsahuje algoritmy pro práci s pixely. [26]

PCL PCL je knihovna dostupná pod BSD licencí. Použitý jazyk je C++. Knihovna umí zpracovat kromě 2D také point cloud data použitý ve 3D. [27]

3.2 Architektonické vzory

V softwarovém návrhu se používají vzory, o kterých je známo, že efektivně řeší nějaký problém. Tyto vzory je pak možné použít jako inspiraci pro návrh řešení jiných problémů. Pod architektonickými vzory jsou chápány vzory, které umožňují návrh vysokoúrovňové architektury okolo jednotného konceptu.

Monolit Aplikace je označována za monolitickou pokud všechny funkcionality jsou navržené jako celek. Hlavní předností je jednoduchost. Proto je tato architektura vhodná pro jednoduché projekty nebo za účelem prototypování. Nevýhodou je špatná možnost integrace rozdílných technologií. Pokud v kterékoliv částí aplikace dojde k chybě, ohrožuje to chod celého systému. [28]

Microservices Microservices neboli mikroslužby je návrh opačný monolitu. Každá funkce systému je rozdělena do jedné samosprávné aplikace. Ke komunikaci mezi aplikacemi se používají protokoly jako je HTTP. Mezi výhody patří flexibilita, jelikož každá aplikace může používat různé technologie a není ovlivněna během ostatních aplikací. Další výhodou je možnost dobrého škálování.

Nevýhody jsou vyšší náklady a komplexita. Je potřeba klást důraz na komunikační rozhraní aby nedošlo k fragmentaci a duplicitě funkcionalit. Komplexita také znesnadňuje hledání chyb a chyby se mohou projevit v jiné části systému než odkud pochází. [29]

Server-Klient Klient se připojuje k serveru za použití síťových protokolů a server zprostředkovává funkcionalitu. Výhodou je centralizace a možnost škálovat počet klientů. Klient a server mohou používat jiné technologie dokud oba používají stejné komunikační rozhraní. Nevýhodou je závislost klienta na server. Jakmile není server dostupný tak klient přichází o funcionality. Při komunikaci přes externí síť se musí dbát na bezpečnost posílaných dat a ochranu proti kybernetickým útokům. [30]

Message Queue Message Queue lze použít k vytvoření vztahu producent-konzument. To přispívá k oddělení závislostí jednotlivých částí a umožňuje asynchronní fungování obout částí. Vztah je zobrazen na obrázku 3.2. Výhodou je možnost škálovat producenta i konzumenta. Oproti server-klient není potřeba, aby obě aplikace běželi současně, jelikož fronta může zprávy ukládat k pozdějšímu zpracování. Nevýhodou je, že každá zpráva je přijata pouze jedním konzumentem. [31]

Obrázek 3.2 Ukázka Message Queue. [31]

Model-View-Controller MVC rozděluje aplikaci na 3 logické celky. Model obsluhuje business logiku, controller se stará o vstup a view je prezenční vrstvou. Výhodou je snadná údržba jednotlivých částí a to, že každá část má jasně definovanou roli. Nevýhodou je přidaná složitost v controlleru. [32]

Část I

Teoretická část

Kapitola 4
Sběr požadavků

Cílem je navrhnout software, který umožní automatizovat obsluhu algoritmů pro počítačové vidění. Samotné algoritmy nejsou součástí řešení a budou nahrazeny zjednodušenými variantami pro účely testování. Požadavky vychází především ze zadání práce a dále jsou ovlivněny zjištěními z rešerše. Software musí archivovat zpracovaná data. Počítá se s nasazením vision systému na kontrolu kusových produktů a ne s nekonečnou smyčkou(např. textil). Na diagramu 4.1 je znázorněné použití systému. Systém zde reprezentuje automatizovanou část a také hlavní funkcionalitu řešení.

Obrázek 4.1 Diagram případů užití.

4.1 Funkční požadavky

Vypracování softwaru, který na vstupu má data zpracovaná algoritmy pro počítačové vidění a tato data:

1. Agreguje z více zdrojů.
2. Persistentně uloží.

3. Zpřístupní dál.

Systém dále musí splňovat následující:

- Dostupnost informací o vytíženosti strojů.
- Možnost propagace dat k jednotlivým kamerám.

4.2 Nefunkční požadavky

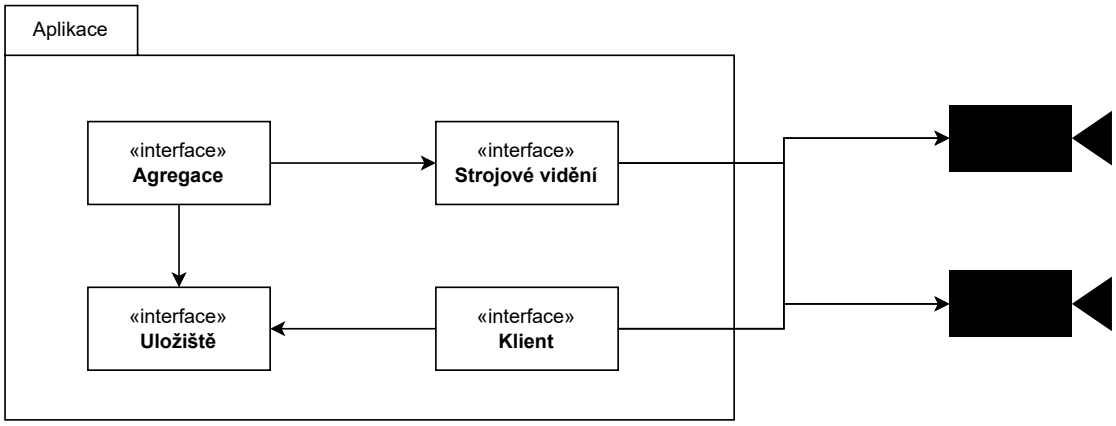
Řešení budou dále využívat lidé specializující se v oblasti strojového vidění. Proto musí být jasné, jak software dále modifikovat. Z rešerše vyplývá, že vision systém může využívat desítky kamer a zpracovávat tisíce snímků za minutu. Tyto data lze použít pro zadefinování pojmu efektivní řešení. Nefunkční požadavky lze shrnout takto:

- Důraz na zpracování velkého objemu dat.
- Rozšířitelnost celého řešení.
- Škálovatelnost na více strojů.

Kapitola 5
Obecný návrh

Na základě uvedených požadavků lze vytvořit vysokoúrovňový návrh architektury. Správná volba architektury může řešení problému výrazně ulehčit. Řešení zmíněná v rešerši mají uzavřený zdrojový kód a nelze učinit žádný předpoklad o tom, jaká je použita architektura. Proto je vytvořeno několik návrhů a jsou porovnány pozitiva a negativa. Každý návrh je doplněn o diagram.

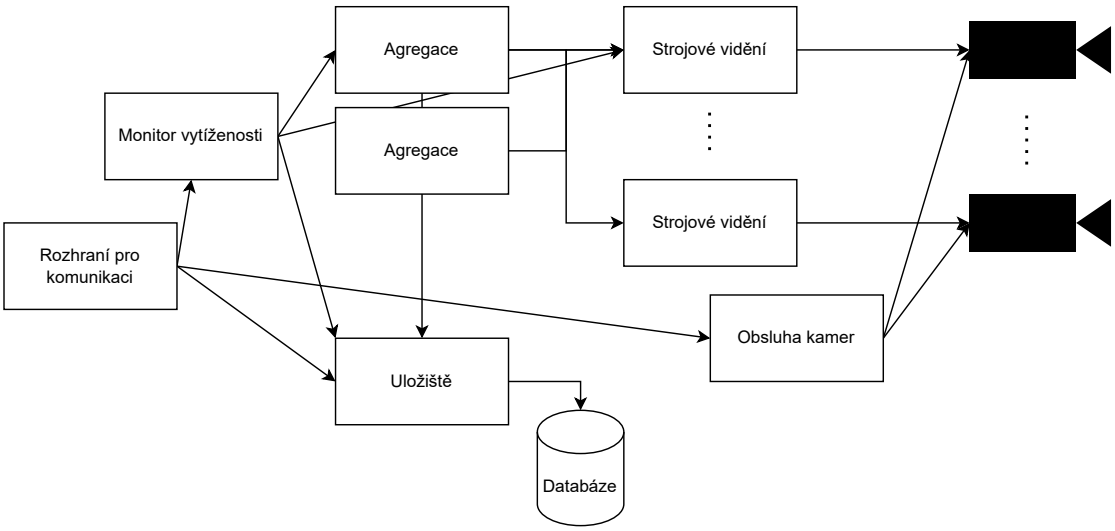
Monolit, viz obrázek 5.1 Z návrhu je vidět, že v rámci monolitu je velice snadné předávat data. Informace o vytíženosti systému je také snadné získat, jelikož uživatelské rozhraní se nachází přímo v aplikaci. Problém ale nastává při pokusu o škálování. Pokud se spustí více instancí aplikace, dojde k problému s agregací, jelikož každá instance bude mít vlastní agregační část.



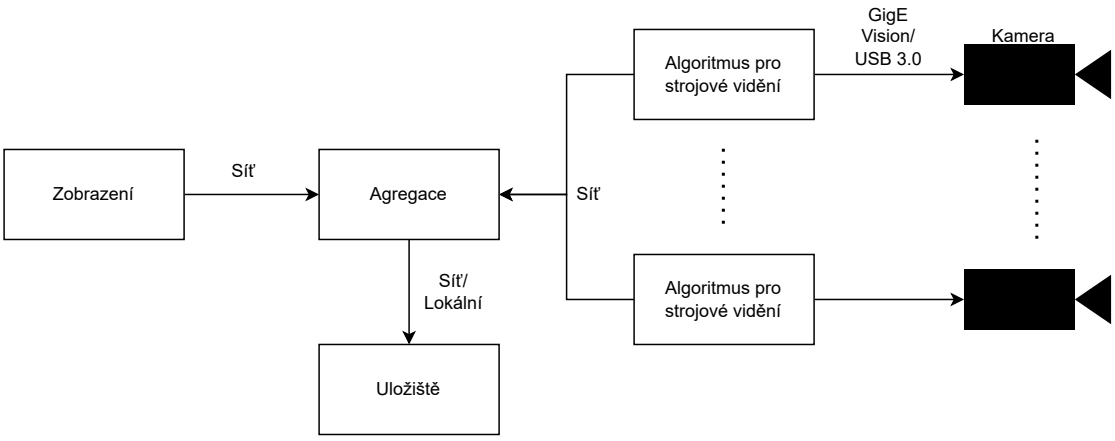
Obrázek 5.1 Obecný návrh Monolit.

Microservices, viz obrázek 5.2 Oproti monolitu je zde vše oddělené a každou část lze volně replikovat. Pokud nějaká část systému nebude stíhat, je možné spustit další instanci. Nevýhodou je ale hustá propojenost jednotlivých komponent.

Klient-server, viz obrázek 5.3 Každá kamera, či skupina kamer může být brána jako samostatný klient. Následně lze vytvořit centrální prvek, který data z klientů agreguje. Tento postup zajišťuje možnost obsluhovat několik vision systémů a svést výstup do centrálního bodu. Agregační část v tomto případě představuje úzké hrdlo a bylo by náročné až nemožné zajistit běh agregační části na více strojích bez změny návrhu.



Obrázek 5.2 Obecný návrh Microservices.

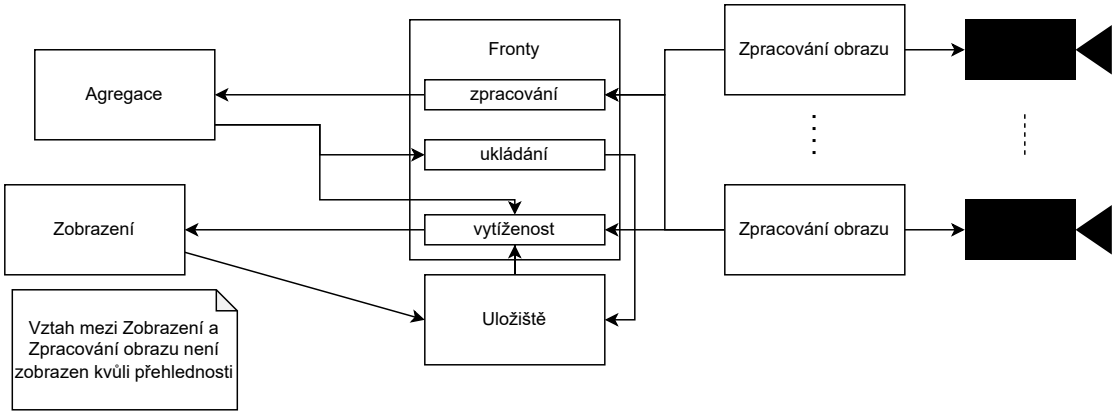


Obrázek 5.3 Obecný návrh Klient-server.

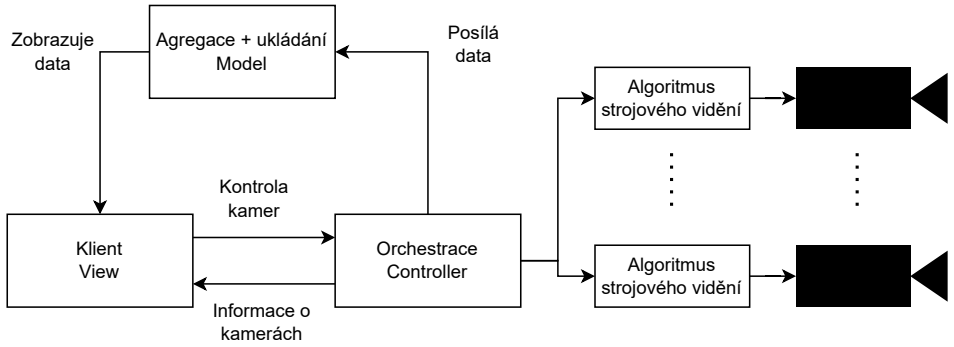
Message Queue, viz obrázek 5.4 Návrh je soustředěn na využití aplikace obsluhující frontu jako centrálního bodu. Fronta nabízí relativně jednoduchou možnost škálovat jednotlivé části a to i včetně agregace. Jedinou překážkou je, aby každá instance agregace dostala všechna potřebná data pro sestavení celku. Diagram neobsahuje vztah, který by umožnil ovládat kamery z venčí. Pro zajištění této funkcionality je potřeba přímo adresovat jednotlivé instance obsluhující kamery. Není vhodné řešit tento problém přidáním další fronty, protože se nejedná o případ producent konzument.

Model-view-controller, viz obrázek 5.5 Pro MVC je spojena funkce agregace a persistence dat v modelu. Controller se stará o získávání zpracovaných dat a předává je do modelu. Data se v modelu uloží a zobrazí ve view. View přes controller může komunikovat s kamerami. Takto lze získat informace o vytíženosti strojů, které zpracovávají obrazová data.

Volba architektury Z uvedených možností je vybrán návrh klient-server, viz obrázek 5.3. Role každé části je zde jasně definovaná a jedinou slabinou je škálovatelnost agregace. Pokud se ukáže, že škálování je potřeba, může agregační část delegovat výpočty do nové aplikace. Vyřeší



Obrázek 5.4 Obecný návrh Message Queue.



Obrázek 5.5 Obecný návrh Model-view-controller.

se tím i problém, jak správně rozhodnout, která instance má dostat jaká data. V praktické části je tento návrh dále rozpracován.

Kapitola 6
Volba technologií a postupů

V požadavcích nejsou uvedeny žádné závazné technologie. Knihovny pro zpracování obrazu jsou rozšířeny napříč populárními programovacími jazyky. Jediné omezení mohou představovat samotné kamery a jejich API. V tomto případě se lze obrátit na knihovnu GenICam, která explicitně zmiňuje, že je navržena pro usnadnění použití v různých programovacích jazycích.

6.1 Volba programovacího jazyka

Dříve zmíněné knihovny pro strojové vidění jsou dostupné např. v jazycích C++, Python a Java. Všechny tři jazyky lze použít pro tvorbu velkých systémů. Na základě dříve určených požadavků a charakteru technologie byl zvolen jazyk C++.

V rámci C++ lze zařídit komunikaci s kódem psaným jak v Pythonu, tak i v Javě (popř. jiných). Dále přináší vysoký výkon, který je nezbytný pro zpracování velkého objemu dat v rozumném čase. Článek [33] uvádí, že C++ se používá k tvorbě efektivního softwaru. Používá se k implementování databází, videoher, strojového učení nebo třeba operačních systémů. Proto je vhodnou volbou.

6.2 Komunikace mezi procesy

Z návrhu vyplývá potřeba pro zajištění komunikace mezi procesy. To lze řešit čistě lokálně pomocí souborů či sdílené paměti. Nebo to lze řešit vzdáleně přes síť za použití síťových protokolů. Síťová komunikace je nezbytná pro využití více výpočetních jednotek v případě škálovatelnosti řešení. I v případě nutnosti lokálního řešení by bylo možné využít síťovou komunikaci.

Existuje spousta možností, jak program může síť využít, například TCP, HTTP, REST, RPC a další. TCP je zprostředkováno přímo systémovými knihovnami a nemá žádnou přidanou režii na aplikační vrstvě. Byl vybrán protokol TCP, jelikož umožňuje návrh vlastního protokolu.

6.3 Měření vytíženosti

Pro měření vytíženosti lze použít následující postupy:

- Profilování
- Externí dedikované programy
- Periodické měření využití systémových zdrojů

Pro potřeby řešení postačí periodicky zjistit využití systémových zdrojů. Hlavním účelem je, aby systém zvládal zpracovávat data včas a nevznikala fronta. Pokud dojde k přetížení stroje, tak to zvolená metoda zvládne odhalit, jelikož ukazatele vytížení budou blízko 100/100.

6.4 Volba knihoven

Standardní knihovna C++ v nejnovějších variantách obsahuje velké množství funkcionalit. Knihovny jsou spjaté s verzí jazyka. Pro potřeby řešení nejlépe vychází verze C++17 a novější. V této verzi byla přidána podpora pro obsluhu souborového systému. Tato funkcionalita bude využita pro perzistentním ukládání. Mezi další potřebné funkcionality patří práce s vlákny a jejich synchronizace. K tomu lze použít knihovny `thread` a `mutex`. Informace byly získány z [34].

Obsluha síťové komunikace není dostupná ve standardní knihovně a je řešena využitím systémových knihoven, proto bude použita knihovna asio [35]. Ta je dostupná ve dvou verzích. Samostatně nebo jako součást knihovny boost. Pro potřeby řešení postačí samostatná varianta. Asio kromě práce se sítí také nabízí funkce zaměřené na asynchronní zpracování kódu. Použitá licence je Boost Software License.

Pro serializaci datových struktur byla vybrána knihovna cereal [36]. K dispozici je serializace do binárního formátu, XML a JSON. Knihovna vyžaduje verzi jazyka alespoň C++11, což je v tomto případě splněno. Distribuována je pod licencí 3-clause BSD, což nijak neomezuje její využití v této práci.

Poslední použitou knihovnou je CpuMem Monitor [37]. Knihovna nabízí rozhraní použitelné k zjištění využití systémových prostředků napříč různými operačními systémy. Knihovna je distribuována pod licencí MIT.

Část II

Praktická část

Kapitola 7
Návrh řešení

Návrh je rozdělen do několika částí. Každá část se zaměřuje na jeden problém, který ovlivňuje implementaci celého řešení.

7.1 Architektura

Jako základ pro návrh architektury je použit koncept klient-server. Definice jednotlivých částí je následující:

Master Stará se o agregaci dat a celkové řízení řešení.

Klient Posílá požadavky na data a poskytuje přístup z venčí.

Úložiště Poskytuje perzistentní uložení dat.

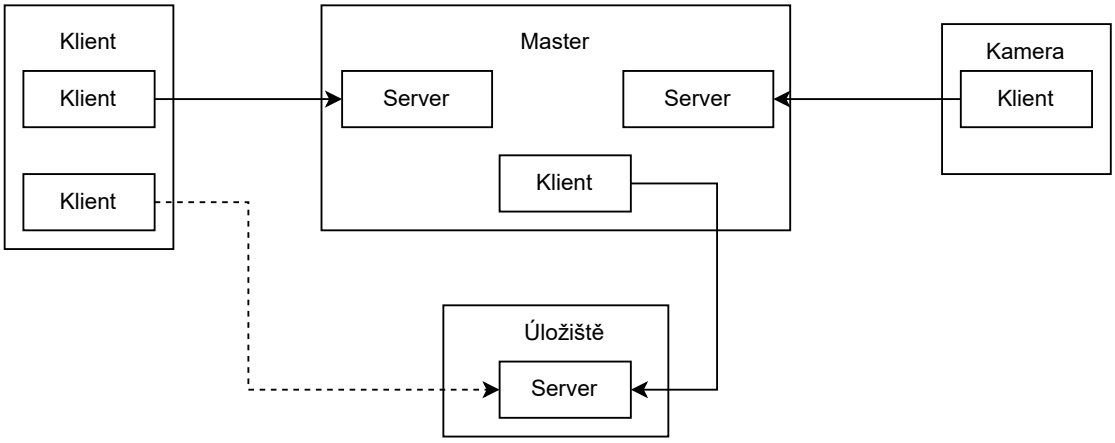
Kamera Přestavuje zdroj zpracovaných dat a zprostředkovává prostředí pro běh algoritmu počítačového vidění.

Každá část je zodpovědná za svůj vlastní chod a s ostatními částmi komunikuje přes síť. Tento vztah je znázorněn na diagramu 7.1. Diagram odpovídá originálnímu diagramu z teoretické části, ale je doplněn o vztah Klienta a Úložiště. Jedná se o přidanou hodnotu s nulovými náklady. Klient může využít stejný kód, jako je dostupný pro Master pro připojení. Příklad využití může být, že si uživatel chce prohlížet historická data a nemusí spouštět zbylé 2 části řešení.

7.2 Reprezentace dat

Řešení je založeno na práci s daty, která nejsou předem definovaná. Formát dat by měl co nejlépe odpovídat výstupům algoritmů, které zpracovávají obraz. Tyto algoritmy mohou ale produkovat jakákoliv data. Například:

- Pozice defektu a obrázek defektu.
- Celý zaznamenaný obraz kamery.
- Řetězce znaků z OCR (tištěné ID produktu).
- Úhel mezi stranami produktu.
- Rozměry produktu.



Obrázek 7.1 Reprezentace vztahů klient-server

Výpis kódu 7.1 DataFragment struktura

```
struct DataFragment
{
        FragmentIdentifier fragment_identifier;
        unsigned int
                source_type,
                source_device;
        std::vector<unsigned char> data;
};
```

Je možné, že bude zapotřebí agregovat výstup z dvou různých zdrojů. Jediný unifikační prvek v takovém případě je použití proměnné typu boolean reprezentující nalezení defektu. Tím se ale ztratí dodatečné informace, například přiložený obrázek defektu, sloužící jako důkaz.

Jako příklad lze uvést tabuli skla. U takového produktu lze hledat defekty na povrchu, kontrolovat správnou velikost a číst sériové číslo. V tomto případě jsou generovány 3 různé výstupy, které musí být agregovány. Vzhledem k okolnostem bylo zvoleno nejobecnější možné řešení.

Obecné řešení je založené na odstranění typů na úrovni programovacího jazyka. Pro výstup z algoritmů byla zvolena struktura `DataFragment` s předpisem 7.1 a pro výstup z agregace struktura `DataBlock` s předpisem 7.2. K uložení dat jsou použita pole znaků, neboli binárních dat. Algoritmy nyní mohou mít na výstupu jakákoliv data, pokud jsou převedena do binární či textové podoby. Jednou takovou textovou podobou může být formát JSON. Agregační část je nyní zodpovědná za správné převedení dat.

Takto obecné řešení ale znamená, že na úrovni zdrojového kódu se ztrácí informace o typech a přichází se tak o metody kontroly jako je typová kontrola na straně kompilátoru. Pokud je zvolená reprezentace nevyhovující, je možné řešení modifikovat dvěma způsoby.

Výpis kódu 7.2 DataBlock struktura

```
struct DataBlock
{
        BlockIdentifier identifier;
        bool is_defect;
        std::vector<unsigned char> data;
};
```

Konkretizace Pro práci s pouze jedním typem, lze nahradit typ proměnné `data` konkrétním typem, který nejlépe vyhovuje řešení. Tím by mohlo být například pole pixelů. Postup lze silně doporučit jakmile bude řešení použito v konkrétní doméně. Především pro strukturu `DataBlock`.

Polymorfismus Polymorfismus umožňuje různým strukturám se stejným chováním, aby byly použity ve stejném kódu. To ale znamená, že se na ven tváří jako jedna stejná struktura. Pokud má být zachován typ, je potřeba použít techniku `Double Dispatch`. V opačném případě bude program muset provádět přetypování stejně jako tomu je u aktuálního návrhu. Použitím `Double Dispatch`[1] se ale výrazně zkomplikuje práce s daty kvůli nutnosti přidat extra logiku a zhorší se tím i postup případné integrace jiných programovacích jazyků.

7.3 Bezpečnost

V rámci sběru požadavků nebyly definovány žádné požadavky na bezpečnost. Řešení nepracuje s žádnými citlivými daty, i tak je vhodné problematiku nezanedbat. Pro navrženou aplikaci lze vést polemiku na úrovní lokální a síťové.

Existující řešení jsou uzavřené systémy, které se používají pouze k běhu vision systému. Lze předpokládat, že v reálném nasazení je lokální stroj plně pod správou technického personálu. Pokud někdo má zájem na škodném jednání, je fyzický útok velmi snadnou variantou a jakákoliv ochrana na úrovni aplikace se stává neúčinnou. Pozornost si zaslouží uložení dat, především jejich integrita. Pokud je potřeba zaručit dlouhodobé a bezporuchové uložení, je vhodné zvolit dedikovaná řešení, které se problematikou zabývá.

Použitím protokolu TCP pro síťovou komunikaci je k dispozici mechanismus kontrolní sumy pro zaručení integrity a pořadové číslo, aby se předešlo ztrátě paketů. Samotná komunikace není nijak šifrována. Pro navržené řešení je možné, že Úložiště bude řešeno například ukládáním dat ve vzdáleném datovém centru. V takovém případě je potřeba zvážit, zda je potřeba data šifrovat.

7.4 Protokol pro síťovou komunikaci

Protokol TCP nerozumí typům, které jsou použity ve zdrojovém kódu. Hlavním cílem je předat dříve definované datové struktury mezi jednotlivými částmi. Dále se budou posílat informace o měření výkonu a Klient může posílat data do části Kamera.

Je možné vytvořit jedno spojení mezi každým párem pro každý typ dat. Takové řešení je ale neefektivní a implementace bude nepřehledná. Proto je vhodnější nadefinovat vlastní síťový protokol pracující nad TCP. Pro návrh vlastního protokolu se nejdříve zadefinuje, jaká data budou posílána a poté bude zvolena vhodná podoba.

Protokol je orientován na předávání datových struktur. Proto je zaveden koncept požadavků (Request). Ty příjemci říkají, jaká data má odeslat.

DataBlock Agregovaná data.

DatalessBlock Pouze metadata z DataBlock.

DataFragment Zpracovaná data určená k agregaci.

HealthData Informace o aktuální vytíženosti.

CommandData Data určená k obsluze Kamera části.

Request Rozhraní obsahující identifikátor požadavku.

[1] Double Dispatch je technika, která během běhu programu volá funkci v závislosti na typech obou objektů zahrnutých do volání.

Výpis kódu 7.3 Header struktura

```
struct Header
{
        unsigned int data_type;
        size_t item_count;
};
```

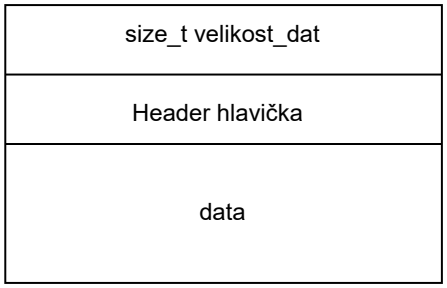
HealthRequest Požadavek na všechna dostupná HealthData.

BlockRequets Požadavek na DataBlock.

BlockRequestQuery Požadavek na DatalessBlock obsahující dotaz.

Struktury jako je `DatalessBlock` dává smysl posílat jako pole objektů. Pro zjednodušení bude uvažováno že každá výměna dat reprezentuje pole objektů. Dále bude každé pole obsahovat data pouze jednoho typu. Nově zmíněná struktura `DatalessBlock` má umožnit zpřístupnit informace o tom, jaké `DataBlock` jsou uložené v Úložišti bez toho, aniž by bylo potřeba posílat i objemnou datovou část. `DatalessBlock` si lze vyžádat posláním `BlockRequestQuery`. Tímto je shrnuto, co se bude posílat.

Zvolená knihovna `Cereal` umožňuje serializaci dat do binární podoby. Při této serializaci se ale ztrácí informace o typu. Proto je definována struktura `Header` s předpisem 7.3, která bude informaci uchovávat. Následně je uložena velikost celého balíčku. Výsledná podoba celého paketu je znázorněna na obrázku 7.2



Obrázek 7.2 Datový formát paketu pro navrhovaný protokol.

Práce s paketem bude probíhat následovně:

1. Vytvoření hlavičky.
2. Serializace hlavičky a dat.
3. Výpočet velikosti serializované hlavičky a dat
4. Odeslání velikosti + binárních dat.

Přijetí paketu:

1. Přijetí velikosti.
2. Přijetí množství dat uvedených ve velikosti.
3. Deserializace hlavičky.

4. Zpracování zbytku dat podle typu uvedeného v hlavičce.

Posílání velkých dat není problém, jelikož TCP zajišťuje správné pořadí paketů a příjemce ví, kolik dat má přijmout. Tím je vyřešen problém s fragmentací, která nastane, když jsou data příliš velká, aby se vešla do jednoho TCP paketu.

7.5 Měření výkonu

Pro měření výkonu byla vybrána technika průběžného monitorování systémových prostředků. Každá část řešení kromě Klient bude schopna poslat data o tom, jak moc běh aplikace zatěžuje procesor. Jelikož Klient komunikuje pouze s Master, využije možnost posílat pole objektů přes síť a Klient obdrží data, která Master posbírá z přidružených aplikací.

Kapitola 8
Implementace

K implementaci je použit jazyk C++ doplněn o dříve zmíněné knihovny. Samotná implementace je rozdělena podle jednotlivých částí řešení. Jelikož je cílem vytvořit 4 různé aplikace, je zde část kódu, která je sdílena. Text je doplněn o ukázky kódu, které nemusí odpovídat skutečné implementaci. Kód byl pozměněn pro lepší čitelnost a kompaktnost. Pořadí jednotlivých podkapitol bylo zvoleno na základě závislosti vyplývajících z návrhu.

8.1 Sdílená část

Prvky, které lze použít mezi více částmi byly vyčleněny do samostatného celku podle principu DRY. Aby bylo možné co nejvíce kódu sdílet, jsou použity templaty pro tvorbu generického kódu. Proto byla pro sdílenou část zvolena forma `hlavičkových souborů` místo samostatné knihovny. V předchozích kapitolách byly již zmíněny datové struktury, proto jsou zde přeskočeny.

První implementovanou třídou je `ConfigParser`, viz kód 8.1. Pro práci se sítí je potřeba mít k dispozici IP adresu a port. Dále bude potřeba konfigurovat jednotlivé instance Kamera. Aby nebylo potřeba kompilovat aplikace při změně těchto parametrů, lze použít několik technik. Například registry, čtení konfigurace ze souboru nebo argumenty příkazové řádky. Byla zvolena kombinace čtení ze souboru a argumentů příkazové řádky. Z příkazové řádky je přečtena adresa souboru a ten pak obsahuje hodnoty jednotlivých proměnných. Samotný formát souboru používá řádky k oddělení proměnných a znaménko = k určení klíče a hodnoty. Příklad formátu:

```
master_port=1000
master_ip=127.0.0.1
```

Výpis kódu 8.1 Předpis Třídy ConfigParser

```
class ConfigParser
{
public:
    ConfigParser(const std::filesystem::path & file_path)
    template <typename T>
    T get(const std::string & key) const;
private:
    std::map<std::string, std::string> m_data;
};
```

Výpis kódu 8.2 Předpis jmenného prostoru Network

```
namespace Network {
class Connection {
    Connection(std::shared_ptr<INetworkExecutor> executor, ...);
    template<typename T>
    void send(T a);
};
class Server {
    Server(int listening_port,
        std::shared_ptr<INetworkExecutor> connection_executor);
    void start();
    void stop();
};
class Client {
public:
    static std::shared_ptr<Connection> CONNECT(
        const std::string & ip_adress, int port,
        std::shared_ptr<INetworkExecutor> executor);
}; };
```

Výpis kódu 8.3 Předpis třídy HealthMonitor

```
class HealthMonitor
{
public:
    double cpu_usage();
    double mem_usage();
    HealthStruct* get_health();
    void set_data(std::string name, unsigned int app_id);
};
```

Další vyčleněnou částí je práce se sítí. Jmenný prostor `Network` znázorněný v předpisu kódu 8.2 zabaluje volání knihoven asio a cereal. Implementace odpovídá dříve navrženému protokolu. Protože je komunikace čistě asynchronní, je přidáno rozhraní `INetworkExecutor`. Ze soketu smí číst pouze jedno dedikované vlákno, to se stará o deserializaci dat. Jakmile jsou data deserializována a jejich typ je zjištěn, zavolá se odpovídající metoda v `INetworkExecutor`. Pokud aplikace chce obsloužit daný typ objektu, implementuje třídu která dědí rozhraní, přetíží metodu a data může zpracovat. Jinak se s daty nic nestane.

Aby bylo možné obsloužit `Request` a vrátit data správnému vláknu, používá se identifikátor. Ten je členskou proměnnou struktury `Request` a jejích potomků. V odpovědi na `Request` se uvede identifikátor zkopíruje do hlavičky a příjemce ví, komu data předat. Pro předání se použije knihovna `future` a třída `std::promise`. Takto definované postupy umožní použití jednotné implementace napříč všemi 4 aplikacemi.

Poslední sdílenou částí je `HealthMohitor` 8.3. Ten zprostředkovává data o vytíženosti. K dispozici jsou metody pro získání informací o CPU a paměti ram. Ve třídě je dále možné nastavit jméno a identifikátor aplikace. Tyto informace lze využít při výpisu. Identifikátor je také potřeba, pokud Klient chce odeslat data do Kamera. Předpokládá se, že Klient pošle dotaz na stav zařízení a tím zjistí identifikátor potřebný k adresování správné instance aplikace.

Výpis kódu 8.4 Header struktura

```
class IStorage
{
public:
    virtual void store(DataBlock* data) = 0;
    virtual DataBlock* retrieve(BlockIdentifier identifier) = 0;
    virtual vector<DatalessBlock*> retrieve_by_date(string date) = 0;
};
```

8.2 Úložiště

Pro snazší práci s ukládáním dat, bylo navrženo rozhraní, které má za cíl oddělit implementační detaily jednotlivých způsobů persistentního ukládání dat. Předpis rozhraní je ve výpisu 8.4. Kromě samotných metod pro uložení a načtení byla přidána i metoda, která umožňuje úložiště prohledávat. Vybrán byl způsob dotazování na konkrétní den uložení. Používá se řetězec ve tvaru YYYY/MM/DD. Zvolená návratová hodnota je pole obsahující objekty typu `DatalessBloock`. Struktura obsahuje ID na které je možné se dotázat a získat odpovídající data.

Aktuální řešení obsahuje dvě různé implementace zmíněného rozhraní. Jedna varianta je `Local Storage`, ta využívá k ukládání lokální disk. Druhou variantou je `NetworkStorageClient` a přidružený `NetworkStorageServer`. Tato varianta umožňuje ukládat data vzdáleně za použití navrženého síťového protokolu a také poskytuje data o vytíženosti. Pro lokální uložení byla zvolena následující hierarchie složek.

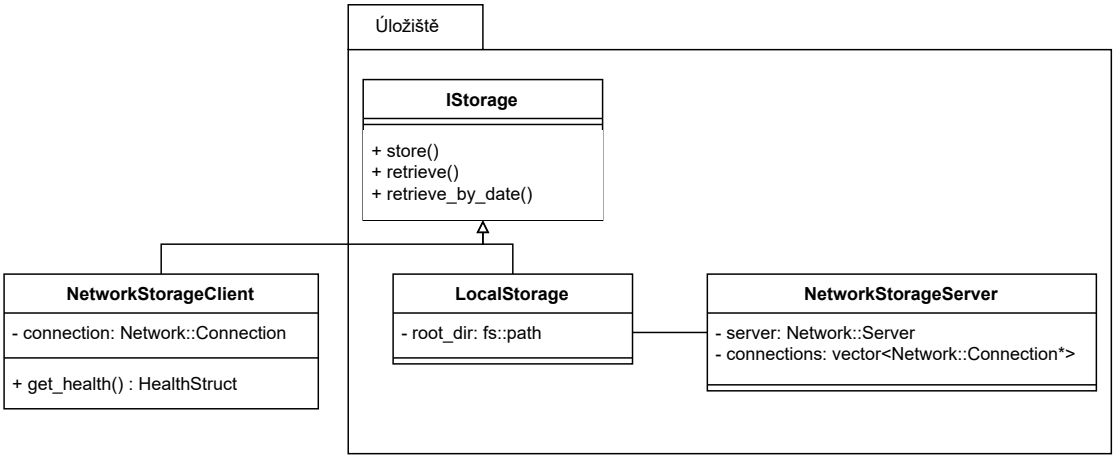
```
root
└── 2023
    ├── 3
    │   └── 15
    │       └── block1
    └── 4
        ├── 20
        │   ├── block2
        │   ├── block3
        │   └── block4
        └── 21
            └── block5
```

Aplikace Úložiště používá obě dvě implementace rozhraní. Server pro síťovou variantu přímo předává požadavky lokální implementaci. Vztah je zachycen na diagramu 8.1.

8.3 Master

Master má za úkol agregovat `DataFragment` a obsluhovat požadavky. Pro agregaci bylo navrženo jednoduché rozhraní `IAgregationAlgorithm` 8.5. Návrh spočívá ve spuštění agregace v samostatném vláknu. Při inicializaci jsou na vstupu dodány dvě fronty, které jsou navrženy tak, aby podporovali synchronní přístup. Jedna fronta složí k předávání `DataFragment` a druhá přebírá zpracované `DataBlock`. Metoda `run` je blokující a slouží k spuštění agregačního algoritmu. Metodou `close` se pak oznámí ukončení. Po zavolání `close` jsou do front poslány prázdné objekty, aby se čekající vlákna odblokovala a mohla se ukončit. Na obrázku 8.2 je znázorněn sekvenční diagram při použití 2 Kamera instancí.

Pro zajištění obsluhy Kamera části jsou v Master nadefinované rozhraní `IDeviceManager` a `ISourceDevice`. Implementace umožňuje připojit více nehomogenních zařízení a pracovat s nimi pomocí jednotného rozhraní.



Obrázek 8.1 Diagram tříd Úložiště

Výpis kódu 8.5 Rozhraní pro agregaci

```
class IAgregationAlgorithm
{
public:
    IAgregationAlgorithm(Queue<DataBlock*>* block_queue,
        Queue<DataFragment*>* fragment_queue)
    : m_block_queue(block_queue), m_fragment_queue(fragment_queue){}
    virtual void run() = 0;
    virtual void close() = 0;
protected:
    Queue<DataBlock*>* m_block_queue;
    Queue<DataFragment*>* m_fragment_queue;
};
```

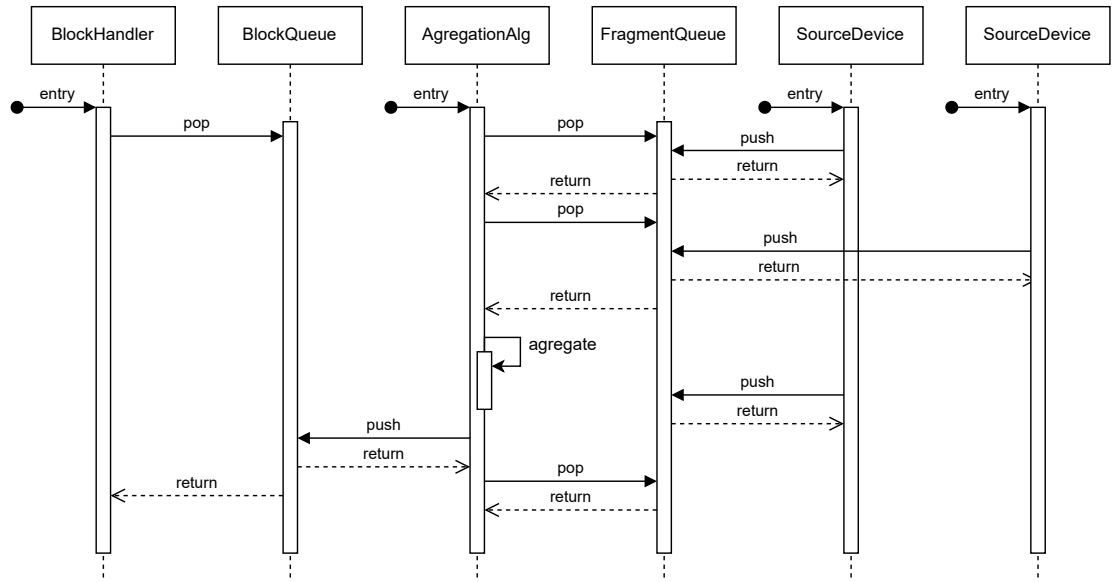
8.4 Kamera

Hlavní funkcionalitou části je získat zpracovaná data a odeslat je do Master. Hlavní problematikou je práce s kamerou a potřebné knihovny. Výběr jedné knihovny ovlivní flexibilitu řešení. Dále je potřeba umožnit algoritmu strojového vidění nastavovat parametry kamery a dohodnout formát pro výměnu obrazových dat. Pro tento postup je příliš málo informací. Proto je upuštěno od jakékoliv komunikace s kamerou.

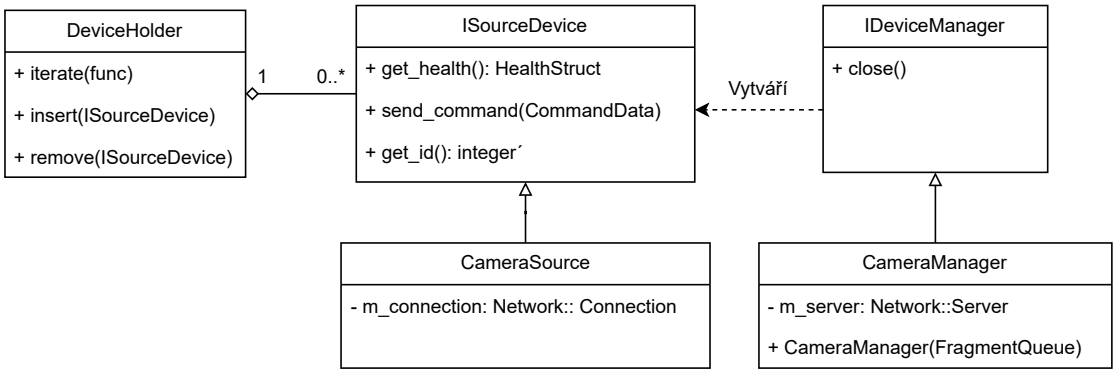
Master očekává `DataFragment` jako vstup pro agregaci. Stejné řešení jde použít i zde. Pro algoritmus bude existovat pouze jediný kontrakt a to ten, že na konci každého cyklu musí vyplnit strukturu `DataFragment` a umístit jí do fronty. Aplikace se pak postará o dopravení objektu do Master.

Tento postup výrazně zjednodušuje řešení a nabízí vysokou flexibilitu. Nelze odhadnout, jak bude algoritmus kameru využívat, proto v tomto případě spadá zodpovědnost za obsluhu kamery na samotný algoritmus. Rozhraní je ukázáno na výpisu 8.6. [1]

[1] Agregace a algoritmus nyní používají kompatibilní rozhraní. Pro řešení, která nepotřebují škálovatelnost na více strojů tak vzniká příležitost triviálně odstranit síťovou komunikaci a přiblížit se monolitickému řešení. Myšlenka může být dále rozvinuta využitím LocalStorage.



Obrázek 8.2 Sekvenční diagram komunikace během agregace.



Obrázek 8.3 Třídní diagram pro správu zařízení

8.5 Klient

Klient má za úkol zobrazovat data. Pro některá použití se hodí grafické rozhraní a pro jiná stačí konzole. Proto je Klient pojat jako rozhraní, které zabaluje volání síťového protokolu. Rozhraní je implementováno třídou `ClientInterface`, viz kód 8.7. Klient se připojuje na Master, který požadavky zpracovává. Před zavoláním metody `send_command`, by se Klient měl ujistit o existenci příjemce, k tomu lze použít `request_health_update`.

Výpis kódu 8.6 Předpis rozhraní pro algoritmus strojového vidění

```
class IAlgorithm
{
public:
    IAlgorithm(Queue<DataFragment*>& fragment_queue)
        : m_fragment_queue(fragment_queue) {}
    virtual void run() = 0;
    virtual void stop() = 0;
    virtual void process(CommandData* command_data) = 0;
protected:
    Queue<DataFragment*>& m_fragment_queue;
};
```

Výpis kódu 8.7 Předpis třídy ClientInterface

```
class ClientInterface
{
public:
    std::vector<HealthStruct*> request_health_update();
    std::vector<DataBlock*> request_data_block(
            BlockIdentifier block_identifier);
    void send_command(std::string command, unsigned int receiver);
    std::vector<DatalessBlock*> block_query(std::string query);
private:
    Network::Connection connection;
};
```

Kapitola 9
Testování

U testování musí být zajištěna replikovatelnost. Další problém je, že vision systém používá kamery, to zvyšuje nároky na přípravu testu. Řešení je navrženo tak, že vision systém je abstrahovaný na rozhraní `IAlgorithm`. Je tedy možné jej nahradit libovolným algoritmem implementujícím dané rozhraní. Byl proto vytvořen pseudo případ použití pro vision systém, který lze otestovat lokálně a průběh testu lze replikovat.

Jako experiment bylo zvoleno zpracování velkého množství obrázků, v obrázcích jsou náhodně rozmístěny zelené pixely. Úkolem je, tyto zelené pixely najít a zaznamenat, na jakém indexu se nacházejí. Formát obrázků je definovaný jako posloupnost RGB dat bez oddělovačů, kde se pro každý barevný kanál používá 1 bajt. Obrázky jsou pojmenovány unikáním číslem a data pro každý obrázek jsou rozděleny do N složek. Z každé složky smí číst pouze jedna instance Kamera. Pro nalezení všech zelených pixelů, je potřeba přečíst data ze všech složek a agregovat je. Obrázky jsou fixní velikosti známé předem. Při agregaci musí být index pixelu správně posunut podle toho, v jakém pořadí je daná část obrázku.

Tento problém je zaměřen na otestování zpracování dat. Data budou přečtena z částí Klient a porovnána s daty získanými při generování obrázků. Pro otestování předávání funkčnosti předávání dat z Klient do Kamera se problém dále rozšíří o podmínku, že Kamera začne zpracovávat data až poté, co od Klient dostane povel `start`. Testování výkonnosti a škálovatelnosti bude provedeno zvlášť po ověření funkčnosti řešení.

9.1 Testovací implementace

Při návrhu datové reprezentace, byla uvedena možnost konkretizace datových struktur. Pro řešení zadaného problému je vhodné takto postupovat a proto je u struktur `DataBlock` a `DataFragment` změněn typ proměnné `data` na `vector<unsigned int>`[1]. Nový typ reprezentuje pole indexů zelených pixelů.

Pro zpracování obrázků je použit algoritmus uvedený v 9.1. Každá instance Kamera nyní dostane ve svém konfiguračním souboru cestu ke složce obsahující přidělené obrázky. Algoritmus je dále doplněn o proměnou `can_start`. Ta je nastavena na `false` a za použití aktivního čekání se čeká na zprávu od Klient, že se může začít.

V Master byl přidán agregační algoritmus a upraven konfigurační soubor. Pro agregaci je využita datová struktura mapa. Pseudokód je uveden v ukázce 9.2. Hodnota N odpovídá počtu instancí Kamera a společně s `PIXELS_PER_FILE` je součástí konfigurace.

Tímto je shrnuto přidání vision systému do navrženého řešení. Při řešení reálných problému

[1] Změna datového typu nevyžadovala žádnou další změnu kódu, kromě upravení testovacích výpisů. Použitá knihovna pro serializaci změnu aplikovala v rámci síťové komunikace a při ukládání dat.

Výpis kódu 9.1 Pseudo algoritmus zpracování obrázku

```
for each file in folder
    DataFragment df {identifier = file.name}
    unsigned char R,G,B
    unsigned int iter = 0
    while !file.eof
        file >> R >> G >> B
        if R == 0 && G == 255 && B == 0
            df.data.add(iter)
        iter++
    fragment_queue.push(df)
```

Výpis kódu 9.2 Pseudo algoritmus agregace

```
while run == true
    DataFragment df = fragment_queue.pop
    if fragment_map.contains(df.identifier) == false
        fragment_map.add(df.identifier, array)
    fragment_map[df.identifier ].add(df)
    if fragment_map[df.identifier].size != N
        continue
    DataBlock db {identifier = df.identifier}
    unsigned int i
        for each fragment in fragment_map[df.identifier]
            for each index in df.data
                db.data.push_back(index + i * PIXELS_PER_FILE)
            ++i
    block_queue.push(db)
```

se dá očekávat, že bude potřeba rozšířit konfigurační soubory o více proměnných. Všechny změny byly zcela izolované v rámci dané části řešení a navržená rozhraní umožnila snadnou implementaci. I když neexistují žádné metriky a jedná se o subjektivní závěr, rozsah aplikovaných změn naznačuje možnost snadného rozšíření systému a splnění odpovídajícího nefunkčního požadavku.

9.2 Test funkčnosti

Řešení je testováno na operačním systému Windows. Vzhledem k použitým knihovnám by mělo být možné program zkompilovat na řadě dalších systémů a chování by mělo být stejné. Pro ověření funkčnosti je test rozdělen na dvě části. První test používá problém o rozsahu 2 složek a 3 souborů. Cílem je ověřit správnou funkčnost síťové komunikace a proto je Master spuštěn na jiném zařízení. Všechny ostatní částí komunikují s Master a proto každý požadavek musí jít přes síť a ne lokálně. Druhý test testuje větší instanci problému s 3 složkami a 10 soubory a neobsahuje kontrolní výpisy síťové knihovny. Pozornost se klade na splnění funkčních požadavků.

9.2.1 Síť

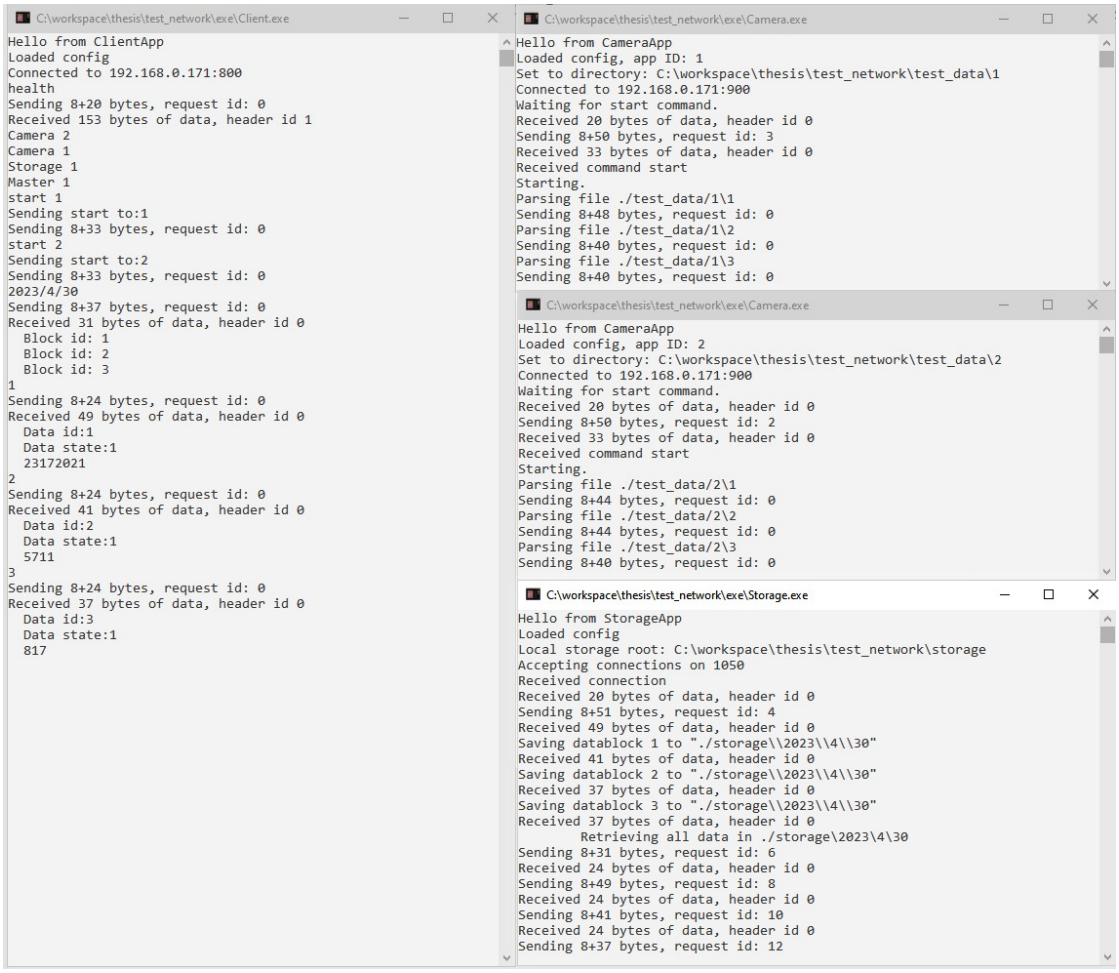
Na obrázku 9.1 je vidět výstup testu sítě. Test proběhl v pořádku a všechna data byla správně předána mezi aplikacemi. Pro lepší vizualizaci je přiložena časová osa jednotlivých událostí.

```
T0: Inicializace všech částí
T1: Klient žádá o data o stavu jednotlivých částí.
T2: Master přebírá požadavek.
T3-T6: Kamera 1, Kamera 2 a Uložiště odpoví Master.
T7: Master predá data Klientovi.
T8: Klient posílá příkaz start mířený na Kamera 1.
T9: Master předává zprávu do Kamera 1.
T9: Klient posílá příkaz start mířený na Kamera 2.
T10: Kamera 1 započíná činnost.
T10: Master předává zprávu do Kamera 2.
T10: Kamera 1 odesílá DataFragment.
T11: Kamera 2 započíná činnost.
T11: Master přijíma DataFragment z Kamera 1 a ukládá k agregaci.
T11: Kamera 2 odesílá DataFragment.
T12: Master přijíma DataFragment z Kamera 2 a ukládá k agregaci.
T13: Master agreguje data a posílá první DataBlock do Uložiště.
T14: Uložiště přijímá DataBlock a ukládá jej.
... Stejné operace pro druhý a třetí DataBlock.
T30: Klient žádá o data zpracována 2023/4/30.
T31: Master předává požadavek.
T32: Uložiště odpovídá.
T33: Master odpovídá.
T34: Klient zobrazí data.
T35: Klient se ptá na DataBlock 1.
... Následuje stejná posloupnost výměny dat.
```

9.2.2 Splnění požadavků

Pro splnění požadavků je potřeba otestovat následující funkce:

- Zpracování a uložení dat.



Obrázek 9.1 Log testování síťové komunikace.

- Zpřístupnění dat.
- Propagace dat ke kamerám.
- Informace o vytíženosti.

V Předešlém testu již bylo potvrzeno několik funkcí. Lze propagovat data ke kamerám, když je potřeba poslat start příkaz. Uložení dat bylo také ukázáno. Lze se tedy soustředit na správnost zpracovaných dat a měření vytíženosti.

Nyní je potřeba přímo pracovat s testovacími daty. Vygenerovaná data jsou zobrazena v obrázku 9.2. Dále byl pro účel testu upraven algoritmus pro zpracování souborů tak, aby při každém zpracování aktivně čekal 2 sekundy.

Na obrázku 9.3 jsou výsledky testu. Data, která řešení vyprodukovalo jsou shodná s daty nagenerovanými. Byly nalezeny 4 soubory se zelenými pixely. Výpis byl formátován pro lepší čitelnost.

Během běhu testu byl dále spuštěn příkaz pro získání dat o vytíženosti 9.4. Instance Kamera zde ukazují hodnotu 4,16 %, mezitím co Master a Úložiště 0 %. Hodnota okolo 4 % dává smysl, jelikož test probíhal na stroji s 24 logickými jádry. 4 % odpovídá plnému využití jednoho jádra aktivním čekáním. Tento údaj se ale může na první pohled jevit jako mylný a vzniká otázka zda by nebylo lepší použít jiné metriky.

```
Green pixels:
    File:1 index:4
    File:1 index:16
    File:1 index:25
    File:3 index:0
    File:3 index:1
    File:3 index:14
    File:3 index:15
    File:5 index:2
    File:5 index:6
    File:5 index:17
    File:5 index:28
    File:6 index:8
Generation successful.
```

Obrázek 9.2 Výpis z generátoru testovacích dat.

```
2023/4/30
  Block id: 1
  Block id: 10
  Block id: 2
  Block id: 3
  Block id: 4
  Block id: 5
  Block id: 6
  Block id: 7
  Block id: 8
  Block id: 9
1
  Data id:1
  Data state:1
  4 16 25
2
  Data id:2
  Data state:0
3
  Data id:3
  Data state:1
  0 1 14 15
4
  Data id:4
  Data state:0
5
  Data id:5
  Data state:1
  2 6 17 28
6
  Data id:6
  Data state:1
  8
7
  Data id:7
  Data state:0
8
  Data id:8
  Data state:0
9
  Data id:9
  Data state:0
10
  Data id:10
  Data state:0
```

Obrázek 9.3 Výsledek testu na přiložených datech.

Tímto je předvedena funkčnost celého řešení. Všechny funkční požadavky byly splněny.

9.3 Test škálovatelnosti

Pro otestování škálovatelnosti byly vybrány dvě metriky. Počet kamer a velikost dat. Pro každý případ bude pozměněna implementace algoritmu a agregace.

9.3.1 Počet kamer

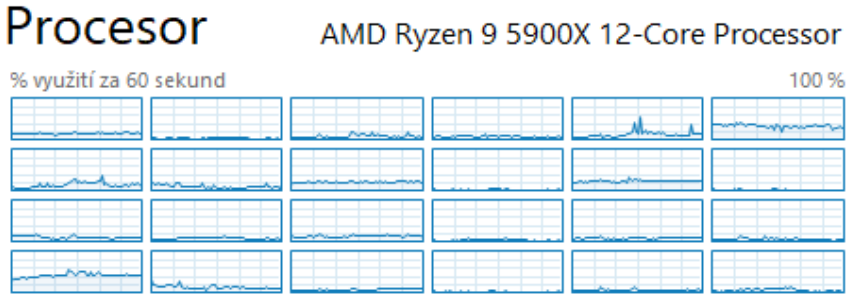
Pro měření je použit algoritmus, který každých 20 ms vygeneruje 400 bajtů dat. Cílem je zatížit systém a zjistit jak se chová vzhledem k rostoucímu počtu připojených zařízení. První návrh testu počítal s použitím vytíženosti procesoru jako metriky. Takto vypadá vytížení při použití 30 Kamer 9.5. Celkové vytížení se pohybovalo okolo 14% a z toho byla nezanedbatelná část procesy na pozadí. Při pokusu dále zvýšit počet zařízení bylo limitujícím faktorem přepínání kontextu procesoru.

Byla zvolena metrika odpovídající době potřebné pro zpracování 1000 bloků. Toto měření probíhá na straně Úložiště. Tabulka 9.1 obsahuje naměřená data.

Drobná prodleva u první měřené hodnoty je čas, který systém má na svůj start. Hodnoty odpovídají rozdílu 20 s s přidanou režií. Režie je nejvyšší při použití 30 Kamer a to okolo 2 s. Hodnota 30 kamer je také shodná s horním limitem zjištěním v rešerši. Rostoucí prodleva může být způsobena přepínáním kontextu. Podle dat z testu lze očekávat, že řešení se bude škálovat dobře podle počtu připojených vision systémů.

```
health
Camera 1 CPU: 4.16568%
Camera 2 CPU: 4.16343%
Camera 3 CPU: 4.16359%
Storage 1000 CPU: 0%
Master 1 CPU: 0%
```

Obrázek 9.4 Informace o vytíženosti.



Obrázek 9.5 Obrázek zobrazuje 24 logických a jejich vytížení při použití 30 Kamer.

9.3.2 Velikost dat

Velikost dat se měří, aby se zjistilo, jak systém reaguje na rostoucí velikost zpracovaných dat. Test by měl především zatížit implementovaný protokol a metrikou je potřebný čas pro agregaci 10 bloků při použití 10 kamer. Výsledky měření jsou zachyceny v tabulce 9.2.

V naměřených datech a při pozorování chování systému se vyskytl trend, že nejdříve jsou všechna data odeslána do Master a až poté jsou posílána do Úložiště. Ve variantě s 100 MB je potřeba přes síť přenést celkem 20 GB dat. Kamera zvládá data odeslat výrazně rychleji než je Master zvládá přijmout, a to by mohlo způsobit, že i když je již připraven prvním blok, tak kvůli plánování se dostane až na konec fronty a je odeslán poté, co jsou přečteny všechny příchozí data z Kamera.

Lze říct, že řešení podporuje práci s velkými daty. Jestli je rychlost dostačující, záleží v jaké doméně je nasazeno. Test lze interpretovat tak, že systém zvládne postupně zpracovat objemná data, pokud je dostatečný čas mezi jejich příchodem.

Tabulka 9.1 Výsledky měření škálovatelnosti vzhledem k počtu kamer.

Identifikátor bloku	5 Kamer	10 Kamer	20 Kamer	30 Kamer
1000	26,8 s	27,1 s	27,9 s	29,1 s
2000	47,3 s	47,8 s	49,1 s	50,9 s
3000	67,9 s	68,7 s	70,2 s	72,4 s
4000	88,5 s	89,5 s	91,7 s	94,3 s

Tabulka 9.2 Výsledky měření škálovatelnosti vzhledem k velikosti dat.

Identifikátor bloku	1 MB	10 MB	100 MB
1	5.7 s	7,3 s	17,2 s
2	5,7 s	7,6 s	20,5 s
3	5,8 s	7,9 s	23,2 s
4	5,8 s	8,1 s	25,7 s
5	5,8 s	8,3 s	27,8 s
6	5,9 s	8,6 s	30,4 s
7	5,9 s	8,8 s	32,9 s
8	5,9 s	9,0 s	35,2 s
9	5,9 s	9,2 s	37,3 s
10	6,0 s	9,5 s	39,7 s

Kapitola 10
Diskuze

Na závěr praktické části je k dispozici seznam možných změn a vylepšení, kterými lze práci dále rozšířit.

Konkretizace řešení Řešení je postavené na velkém množství abstrakce. Abstrakci lze v mnoha případech lehce odstranit nahrazením jednotlivých rozhraní konkrétními implementacemi napříč systémem. Změna je vhodná, pokud je žádoucí mít úzce specializované řešení.

Kontejnerizace Pro využití v cloudových řešeních je vhodné jednotlivé části zabalit do kontejnerů pro použití v prostředích jako je například Docker.

Pokročilé monitorování Na trhu existují sofistikované řešení, která lze použít místo postupu implementovaného v této práci. Řešení poskytují vlastní grafické rozhraní, které zahrnuje informace o stavu jednotlivých částí systému.

Implementace jiného úložiště Úložiště lze nahradit za databázové úložiště či cloudové. Pokud je potřeba data uchovávat mimořádně dlouho, lze zvážit uložení na magnetické pásky, kde neověřené zdroje uvádějí životnost 10 až 20 let.

Implementace jiného síťového protokolu Protokol TCP nemusí být nejvýhodnějším řešením pro každé zadání. Pro lokální sítě s nízkou ztrátovostí a nutností přenášet extrémně objemná data by mohl například protokol UDP poskytnout vyšší výkon.

Dynamické načítání knihoven C++ umožňuje načtení knihovny při startu programu ale i později za běhu. Algoritmy pro strojové vidění nejsou součástí práce a je potřeba jednotlivé části celé zkompilovat, pokud dojde ke změnám nejen v rozhraních. Algoritmy je možné zabalit do knihovny a tu načíst buď při startu, nebo jít o kus dále a umožnit provádět změny algoritmu za běhu.

Vytvoření rozhraní pro jiné programovací jazyky Aby bylo možné použít i jazyky jako je Python nebo Java pro implementaci algoritmů, je potřeba zpřístupnit důležitá rozhraní a přidat logiku pro obsluhu virtuálních strojů, které uvedené jazyky používají.

Tvorba metrik Pokud je každý vytvořený produkt zaznamenán pomocí systému, lze vytvořit aplikaci, která se připojí na úložiště a bude vytvářet metriky jako je počet vyrobených kusů za časový úsek, či chybovost výrobního procesu.

Kapitola 11
Závěr

Bylo představeno několik návrhů, které by umožnily vyřešit zadaný problém. Byl vybrán návrh založený na architektuře klient-server. Návrh byl dále rozšířen a implementován. Jako implementační jazyk bylo zvoleno C++.

Výsledné řešení se skládá ze 4 aplikací. Aplikace spolu komunikují pomocí navrženého protokolu. Ten je postaven nad protokolem TCP. Základem je část pojmenovaná Master, ta komunikuje s ostatními aplikacemi a stará se o agregaci dat. Vision systém je reprezentován částí Kamera. Úložiště je řešeno lokálně na disk. O zobrazování dat se stará aplikace Klient, která je postavena na navrženém rozhraní. To lze použít v jiných aplikacích, pokud by příkazová řádka nebyla vyhovující.

Řešení bylo otestováno. Test funkčností ukázal, že jsou splněny všechny požadavky. Dále proběhl test škálovatelnosti. Zde se simulovalo chování při připojení více vision systémů a při zpracování velkých dat. Jediným pozorovaným problémem byla schopnost přesunout data po síti při použití balíčků o velikosti 100 MB, zde se čas potřebný na zpracování pohybuje okolo 4 s při použijí 10 zdrojů dat.

Příloha A

Návod na spuštění řešení

Řešení je spustitelné na platformě Windows. Otestována byla verze Windows 10 64-bit.

Popis adresáře exe
Zkompilované aplikace jsem k dispozici ve složce exe. Složka configs obsahuje konfigurační soubory potřebné pro běh aplikací. Scripts obsahuje další skripty použité k ulehčení spuštění. Ve složce test_data jsou předgenerována data, která jsou použita v defaultní konfiguraci. Storage se používá k ukládání dat, je vhodné obsah složky smazat mezi spuštěními.

Skripty
K dispozici jsou 3 skripty určené k usnadnění práce s aplikacemi.

run - Spustí celé řešení za použití konfiguračních souborů ze složky configs

generate_data - Spustí generátor testovacích dat pro hledání zeleného pixelu

generate_camera_configs - Předgeneruje konfiguraci pro N instancí Kamera

Konfigurace
Kromě nastavení koncových bodů pro síťovou komunikaci je potřeba, aby Master aplikace věděla kolik kamer poběží, tato hodnota se předává v konfiguraci hodnotou cameras_count. Dále je potřeba aby Master a Kamera měli nastavenou stejnou hodnotu algoritmu.

Spuštění
Jakmile jsou nastaveny konfigurace je možné vše spustit skriptem run.

```
run [počet kamer]
```

Příkaz postupně spustí jednotlivé aplikace. Pro možnost otestování je připravena konfigurace pro 3 kamery.

```
run 3
```

Pokud je použit algoritmus pro hledání zelených pixelů, je potřeba rozeslat start příkazy.

Obsluha
Aplikace Client.exe přijímá následující vstupy od uživatele.

health - Vypíše vytíženost jednotlivých zařízení.

start [id kamery] - Pošle start příkaz do zvolené kamery.

YYYY/MM/DD - Vypíše id bloků uložených k danému datu, například 2023/4/20.

[id bloku] - Vypíše informace o daném bloku.

Bibliografie

1. GROUP, OPTEL. *Vision Inspecion System for the Pharmaceutical Industry*. 2023. Dostupné také z: https://www.optelgroup.com/en/solution/vision-systems/.
2. GROUP, OPTEL. *VISION SYSTEMS TYPICAL APPLICATIONS* [online]. [B.r.]. Dostupné také z: https://www.optelgroup.com/app/uploads/2022/07/catalogue_vision_systems_analyser_en_mkt_4553.pdf.
3. LTD, Industrial Vision Systems. *Industrial Vision Systems— Machine Vision Technology*. 2023. Dostupné také z: https://www.industrialvision.co.uk/.
4. LTD, Industrial Vision Systems. *Linescan & Web Inspection*. 2023. Dostupné také z: https://www.industrialvision.co.uk/products/linescan-web-inspection.
5. SCHENK, Dr. *Dr. Schenk GmbH*. 2023. Dostupné také z: https://www.drschenk.com/.
6. SCHENK, Dr. *Drschenk - MIDA – Superior Defect Detection and Classification*. 2023. Dostupné také z: https://www.drschenk.com/news/news-article/mida-superior-defect-detection-and-classification-15.html.
7. GMBH, ISRA VISION. *Isra Vision - Leading Machine Vision Systems*. 2023. Dostupné také z: https://www.isravision.com/en/.
8. GMBH, ISRA VISION. *Inspection solutions for the solar industry from ISRA VISION*. 2023. Dostupné také z: https://www.isravision.com/en/solar/.
9. GMBH, ISRA VISION. *EPROMI Live: System and quality status accessible from anywhere*. 2023. Dostupné také z: https://www.isravision.com/en/landing-pages/epromi-live/.
10. CORPORATION, Cognex. *Cognex Edge Intelligence*. 2023. Dostupné také z: https://www.cognex.com/solutions/edge-intelligence.
11. RĂILEANU, Silviu; BORANGIU, Theodor; ANTON, Florin; ANTON, Silvia. Open source machine vision platform for manufacturing and robotics. *IFAC-PapersOnLine*. 2021, roč. 54, č. 1, s. 522–527. ISSN 2405-8963. Dostupné z DOI: https://doi.org/10.1016/j.ifacol.2021.08.060. 17th IFAC Symposium on Information Control Problems in Manufacturing INCOM 2021.
12. ANDREA, Harris. *Comparison of "peer-to-peer"vs "client-server"Network Models*. 2023. Dostupné také z: https://www.networkstraining.com/peer-to-peer-vs-client-server-network/.
13. GMBH, Rasa Technologies. *Introduction to Rasa Open Source & Rasa Pro*. 2023. Dostupné také z: https://rasa.com/docs/rasa/.
14. VMWARE. *Messaging that just works - RabbitMQ*. 2023. Dostupné také z: https://www.rabbitmq.com/.

15. MESHENBERG, Ruslan. *Microservices at Netflix Scale: Principles, Tradeoffs & Lessons Learned • R. Meshenberg • GOTO 2016*. 2016. Dostupné také z: https://www.youtube.com/watch?v=57UK46qfBLY.
16. ASSOCIATION, EMVA - European Machine Vision. *Introduction - EMVA*. 2023. Dostupné také z: https://www.emva.org/standards-technology/genicam/introduction-new/.
17. AG, Basler. *Software — Basler Product Documentation*. 2023. Dostupné také z: https://docs.baslerweb.com/software.
18. BAUMER. *Baumer neoAPI — Baumer USA*. 2023. Dostupné také z: https://www.baumer.com/us/en/product-overview/industrial-cameras-image-processing/software/baumer-neoapi/c/42528.
19. ZIVID. *Zivid SDK for developers, by developers*. 2023. Dostupné také z: https://www.zivid.com/zivid-sdk.
20. OPENCV. *OpenCV modules*. 2022. Dostupné také z: https://docs.opencv.org/4.7.0/.
21. MORONEY, Laurence. *Build a computer vision model with TensorFlow — Google Developerss*. 2023. Dostupné také z: https://developers.google.com/codelabs/tensorflow-2-computervision.
22. DEVELOPERS, Google. *API Documentation : TensorFlow v2.12.0*. 2023. Dostupné také z: https://www.tensorflow.org/api_docs.
23. THE MATHWORKS, Inc. *Image Processing and Computer Vision*. 2023. Dostupné také z: https://www.mathworks.com/help/overview/image-processing-and-computer-vision.html.
24. THE MATHWORKS, Inc. *Medical Imaging Toolbox*. 2023. Dostupné také z: https://www.mathworks.com/help/medical-imaging/index.html.
25. NVIDIA. *VPI - Vision Programming Interface Documentation*. 2023. Dostupné také z: https://docs.nvidia.com/vpi/.
26. *BoofCV*. 2023. Dostupné také z: https://boofcv.org/index.php?title=Main_Page.
27. RUSU, Radu Bogdan; COUSINS, Steve. 3D is here: Point Cloud Library (PCL). In: *IEEE International Conference on Robotics and Automation (ICRA)*. Shanghai, China: IEEE, 2011.
28. SARAWAGI, Shivang. *Monolithic Architecture Simplified*. 2023. Dostupné také z: https://scaleyourapp.com/monolithic-architecture/.
29. VOIDONICOLAS, Roxanne. *Don't believe the hype: 5 disadvantages of microservices*. 2023. Dostupné také z: https://www.shopify.com/enterprise/disadvantages-microservices.
30. TERRA, John. *What is Client-Server Architecture? Everything You Should Know: Simplilearn*. 2023. Dostupné také z: https://www.simplilearn.com/what-is-client-server-architecture-article.
31. AMAZON WEB SERVICES, Inc. *What is a Message Queue? - aws.amazon.com*. 2023. Dostupné také z: https://aws.amazon.com/message-queue/.
32. MARTIN, Matthew. *MVC Framework Tutorial for Beginners: What is, Architecture & Example*. 2023. Dostupné také z: https://www.guru99.com/mvc-tutorial.html.
33. LANDICHO, Esa. *Who Uses C++*. 2022. Dostupné také z: https://careerkarma.com/blog/who-uses-c-plus-plus/.
34. *C and C++ reference*. 2023. Dostupné také z: https://en.cppreference.com/w/.
35. *Asio C++ Library*. 2023. Dostupné také z: http://think-async.com/Asio/.
36. GRANT, W. Shane; VOORHIES, Randolph. *cereal - A C++11 library for serialization*. 2017. Dostupné také z: http://uscilab.github.io/cereal/.

37. $CpuMem_Monitor : Crossplatformcpuandmemorymonitor$. 2018. Dostupné také z: `https://github.com/smasherprog/CpuMem_Monitor`.

Obsah přiloženého média

Dostupné na adrese https://gitlab.fit.cvut.cz/kuzmajin/dp-visionsystems.

- `README.md` ..... stručný popis obsahu média
- `exe` ..... adresář se spustitelnou formou implementace
  - `README.md` ..... návod na spuštění
- `src`
  - `impl` ..... zdrojové kódy implementace
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
- `text` ..... text práce
  - `thesis.pdf` ..... text práce ve formátu PDF