Západočeská univerzita v Plzni
Fakulta aplikovaných věd
Katedra informatiky a výpočetní techniky

Diplomová práce

Analýza a úpravy architektury systému SensLog

Plzeň 2020 Bc. Lukáš Černý

Místo této strany bude zadání práce.

Prohlášení

Prohlašuji, že jsem diplomovou práci vypracoval samostatně a výhradně s použitím citovaných pramenů.

V Plzni dne 21. května 2020

Bc. Lukáš Černý

Poděkování

Děkuji panu Doc. Ing. Přemyslu Bradovi, MSc., Ph.D., za odborné vedení mé diplomové práce a za cenné rady, které mi pomohly tuto práci vytvořit. Dále bych rád poděkoval panu Ing. Michalovi Kepkovi, Ph.D. za konzultace o systému SensLog.

Abstract

The aim of this thesis is load testing of sensor data management system SensLog. The goal is to prove whether the system is ready for use in scenarios involving the Internet of Things within the concept of smart farming. Test cases based on real data from real tractors operation are created to verify it. There are localized bottlenecks of the system, analyzed sources of the problem and proposed suitable solutions to solve some of the problems in this paper. The result of the thesis are a repaired system and a proof of concept, which implements some basic functions of the system SensLog and verifies basic requirements and technologies for new SensLog 2.0.

Abstrakt

Práce se věnuje výkonnostnímu testování systému pro správu senzorických dat SensLog s cílem zjistit, zda je připravený na použití ve scénářích zahrnující internet věcí v rámci konceptu chytrého zemědělství. Pro ověření jsou vytvořeny testovací sady založené na reálných datech z testovacího provozu traktorů. V práci je poukázáno na slabá místa v systému, jsou analyzovány příčiny a navrhnuta vhodná řešení oprav. Výsledkem je systém s opravenými nejzásadnějšími výkonnostními problémy a dále proof-of-concept, který implementuje základní funkcionalitu systému SensLog a ověruje základní požadavky a technologie pro SensLog 2.0.

Obsah

1 Úvod

Trend průmyslu 4.0 mění výrobní schopnosti všech průmyslových odvětví včetně zemědělské oblasti. Konektivita je základním kamenem této transformace a internet věcí (IoT) je pro to klíčovou technologií. Tento trend je postaven na řadě technologií: IoT, velká data (Big Data), umělá inteligence a digitálních praktik: kooperace, mobilita a otevřené inovace [15]. Za těchto okolností vznikl i systém SensLog, který se zaměřuje na sběr senzorických dat.

S narůstající plochou pro zemědělské využití nebo narůstajícím množstvím pracujících strojů, dochází i ke zvyšování množství osazených senzorů pro měření různých vlastností. Takový objem dat klade vysoké nároky na server, který data musí přijmout a zpracovat. V opačném případě může dojít k zatížení systému nebo až k jeho úplnému selhání. Tomuto scénáři je snaha se vyhnout, a proto jsou mezi mimofunkčními požadavky na systém uváděny například minimální počty uživatelů, které by systém měl obsloužit za danou časovou jednotku.

Tato práce vzniká za účelem ověření, zda systém SensLog dokáže zpracovat zátěž danou reálnými scénářemi použití a jeho chování pod velkou zátěží. Cílem je nejen ověření, ale také poukázání na případné chyby a jak jejich odstranění může vést ke zvýšení propustnosti celého systému. Práce je uspořádána do následujících kapitol:

- Kapitola **Analýza výkonnosti podnikových aplikací** se zabývá obecnými postupy pro získání informací o výkonnosti aplikací, možných problémů a vhodných nástrojů.
- Kapitola **Systém SensLog** popisuje aktuální podobu systému a jeho použití.
- Kapitola **Měření systému SensLog** obsahuje konfigurace testů a výsledky měření výkonnosti systému SensLog.
- Kapitola **Analýza příčin a úprava systému** se zaměřuje na analýzu přičin vycházející z měření, návrhem vhodných úprav, implementací proof-of-concept (PoC) a porovnání s opravenou verzí.

2 Analýza výkonnosti podnikových aplikací

V dnešním světě existuje mnoho rozsáhlých systémů čelících náporu tísíce nebo milionů uživatelů. Studie ukazují, že selhání bývá způsobeno jejich neschopností dostatečně škálovat podle požadavků uživatelů, než podle funkčních chyb [36]. Neschopnost škálovat často vede ke katastrofickým selháním nebo ztrátě uživatelů. Jednou z metod, jak předejít těmto scénářům, je provést zátěžové testování [37].

Příklady z praxe ukazují, že tyto katastrofické scénáře nejsou vzácnou výjimkou, ale v informačním světe se stále takové případy najdou. Příkladem je americký HealthCare.gov, který nechal instalovat nový systém. Po spuštění tohoto systému se začaly objevovat výkonnostní problémy a došlo ke zhroucení celého systému [31]. Stejnou negativní zkušeností si prošli i občané České republiky. Ministerstvo dopravy spustilo nový systém registrace vozidel a od prvního spuštění zaznamenával velké výkonnostní problémy a systém se stal nepoužitelným [29]. V obou těchto případech se jedná o systémy vytvořené ministerstvy vlády a přestože vznikaly velké finanční ztráty, neznamenalo to ztrátu uživatelů.

Další skupinou, kde se negativním způsobem projeví finanční ztráty v důsledku ztráty uživatelů, je rychlost načítání webu. Rychlost načítání stránky je měřítkem výkonnosti, se kterým se zákazníci nejčastěji setkají. Když lidé přicházejí na web, musí být schopni co nejrychleji provést akce, které na webu chtějí provádět. Ať už je účel webu jakýkoliv - sociální sítě nebo online nakupování, výkon webu je důležitý pro udržení zákazníků. Sociální síť Pinterest přestavěla své stránky pro lepší výkon a zaznamenala snížení časové prodlevy pro načtení stránky a v důsledku toho se zvýšil počet registrací o 15 %. Pouze zlepšením výkonu si společnost zařídila růst uživatelů [35]. Stejnou metriku lze použít i pro systémy, které nemají uživatelské rozhraní, ale komunikace probíhá přes veřejné rozhraní aplikace (API). Takové systémy si poté mohou najít zákazníci vyžadují rychlou odezvu a očekávají velký datový tok.

Optimalizovat aplikaci pro lepší propustnost není tak jednoduchý úkol, jak by se mohlo na první pohled zdát. Je-li aplikace vyvíjena zcela od začátku, snadno se může stát, že zvyšující se složitost systému povede k prohlubujícím se problémům s výkonem, které mohou být odhaleny až v posledních fázích vývoje. Pro již dokončený systém je situace jiná. Je na zvážení

společnosti, zda se vyplatí investovat do optimalizace již hotových částí systému nebo implementovat nové funkcionality pro zhodnocení investovaných prostředků.

2.1 Testování

Testování softwaru je velmi široká oblast, která zahrnuje mnoho dalších technických a netechnických oblastí, jako je specifikace, návrh, implementace, údržba, proces a řízení v softwarovém inženýrství. Testy je možné rozdělit podle přístupu na testování černé a bílé skříňky. Testerovi je při přístupu černé skříňky k dispozici pouze specifikace funkčnosti bez znalosti implementace (bez přístupu ke zdrojovému kódu). V případě přístupu bílé skříňky je situace opačná, je k dispozici implementace a tester může vytvořit sady testů pokrývající větší část zdrojového kódu.

Testování se účastní každé fáze životního cyklu softwaru, ale v každé fázi má odlišné cíle [20]. Existuje mnoho druhů testů, které mají své opodstatnění pro danou fázi vývoje. Mezi známé a často používané testy se řadí:

- **Jednotkové testování**: Provádí se na nejnižší úrovni a testuje metodu, modul nebo komponentu.
- **Integrační testy**: Jsou-li dvě a více testovaných jednotek sloučeny do větší struktury, testy se provádí na komunikační rozhraní a konstrukci. Možné je také provádět testy na správnou komunikaci mezi aplikací a operačním systémem nebo hardwarem.
- **Testování systému**: Ověření kvality celého systému. Test je často založen na specifikaci funkčních požadavcích.
  - **Zátěžové testy**: Ověření funkcionality systému pod větší zátěží (viz sekce 2.1.1).
  - **Stresové testy**: Uvedení systému do extrémních podmínek a sledování různých parametrů (viz sekce 2.1.2).
- **Akceptační testy**: Provádí se v době, kdy je systém dokončený a je předán uživatelům nebo zákazníkovi. Účelem je poskytnout jistotu, že systém funguje podle představ zákazníka.

Tato práce je zaměřena na testování systému SensLog, konkrétně ověření mimofunkčních požadavků. Systém je v aktuální verzi považovaný za dokončený a případné další verze budou spíše servisního charakteru. Z mimofunkčních požadavků je zaměřeno především na využití zdrojů, stabilitu a

propustnost. Pro ověření těchto vlastností jsou vhodné typy testů, které jsou založeny na generování větší zátěže.

2.1.1 Zátěžové testování

Zátěžové testování je proces hodnocení chování systému pod zátěží a detekcí případných problémů, které se zatížením souvisí. Provádí se na prototypu nebo plně funkčním systému. Míra, se kterou jsou požadavky doručovány na testovaný systém, se nazývá zátěž [10]. Test může být zaměřen také na účinnost různých návrhových architektonických rozhodnutí, různých algoritmů nebo konfigurací systému. Hlavním cílem testování je odhalit funkční a výkonnostní problémy při zatížení a jejich hlavní příčiny. V praxi se testování provádí za provozních podmínek, to znamená, že testování je obvykle založeno na očekávaném využití systému [8].

Problémy související se zatížením se dělí na funkční, které se objeví pouze při zátěži, a nefunkční, které souvisí s porušením mimofunkčních požadavků na kvalitu systému. Mezi funkční problémy se často řadí typy chyb, které vlivem vysoké zátěže vedou k porušení funkčních požadavků, nebo-li k rozdílným výsledkům. Jako příklad lze uvést synchronizaci vláken při paralelním zpracování, kdy vlivem zátěže může dojít k jinému plánování vláken v operačním systému a výsledkem může být nesprávná hodnota.

Mezi mimofunkční požadavky se často řadí spolehlivost nebo stabilita, jejíž parametry jsou definovány maximální očekávanou zátěží na dané platformě (cloud, vlastní server atd.). V případě chybějících mimofunkčních požadavků jsou kritéria zátěže odvozena z předchozího testování nebo charakteristiky dat, které sloužít jako výchozí bod (anglicky baseline) pro další testy, které by neměly dosahovat horších výsledků. Tato zásada stanoví, že nová verze je minimálně tak dobrá, jako ta předchozí.

Další kategorií mohou být průzkumné testy, které neobsahují žádná jasná kritéria o úspěchu či neúspěchu. Příkladem mohou být testy typu co se stane když: bude zvýšen počet vláken zpracovávající požadavky nebo navýšen počet připojení do databáze.

2.1.2 Stresové testování

Stresové testování je proces uvádění systému do extrémních podmínek za účelem ověření robustnosti systému a zjišťování různých problémů souvisejících s maximálním zatížením, které se neprojevily při zátěžovém testování. Příklady takových podmínek mohou být vztažené k zátěži (extrémně vysoká zátěž), omezené výpočetní zdroje, omezené komunikační schopnosti (nesta-

bilní připojení) nebo selhání (např. hw komponenty). V ostatních případech se stresové testování používá k vyhodnocení účinnosti návrhů softwaru.

2.1.3 Příprava scénářů pro zátěžové testování

Cílem fáze návrhu testů je navrhnout zatížení, které může odhalit případné problémy. Existují dvě obecné skupiny pro návrh zátěže k dosažení těchto cílů.

Návrh realistického zatížení

Hlavním cílem testování je zajistit, aby systém správně fungoval, bude-li nasazen do produkčního nasazení.

- Přístup založení na agregovaném vytížení. Cílem je generovat individuální cílově zaměřené požadavky - například systém musí být schopný zpracovat určitý počet záznamů přes API a určitý počet požadavků na zobrazení webové stránky.
- Přístup založený na případech užití, kde cílem je sestavit testovací scénáře tak, aby na sebe navazovaly. Například v online nakupování se jedná o přihlášení, přidání zboží do košíku a provedení platební transakce.

Zátěž vyvolávající chyby

Cílem je navrhnout zátěž, která pravděpodobně způsobí nějaké funkční nebo nefunkční problémy.

- Nalezení slabých míst analýzou zdrojového kódu.
- Systematickou analýzou architektonických modelů systému.

Druhy testovacích dat

Ať už se jedná o zatížení simulující reálný provoz systému, nebo je snaha systém napadnout, každý scénář vyžaduje jinou charakteristiku dat. Typy testovacích dat mohou být rozděleny do následujících bodů:

- **Reálná**: Data z běžící aplikace, která zachycují reálné použití v praxi. Takový typ dat dává představu o tom, jaké chování vykazují uživatelé při používání systému. Získání dat lze provést několika způsoby: exportem z databáze, které obsahují časové razítko; z logů s trasovacími

značkami; monitorováním průtoku dat ať už v aplikaci, nebo předřazenou bránou. Použití reálných dat s sebou nese i nevýhody. Systém může běžet pouze v testovacím režimu a množství získaných dat nemusí být dostatečné nebo data nedostatečně pokrývají funkcionalitu systému. Je-li k dispozici dostatek reálných dat z běžného provozu, je výhodné extrahovat vlastnosti a využít je při sestavování dat syntetických.

- **Syntetická**: Jedná se o generovaná data, která obsahují definované vlastnosti chování. Nejčastěji se používají v oborech, kde je vyžadována kontrola nad daty, pro které je očekávaný nějaký určitý výsledek (například systémy pro umělou inteligenci). V testování se tento typ dat používá pro simulace uživatelů, jejiž chování může být nakonfigurované tak, aby pokrývalo hraniční hodnoty pro veškerou funkcionalitu systému. Může se jednat o data, která jsou zatížena určitým procentem chybovosti nebo integrované vlastnosti se snahou napadnout systém. Takový typ dat se dá použít nejen k simulace většího časového úseku, ale i generování předpokládané zátěže, jedná-li se například o systém ve vývoji [9].

2.2 Profilování

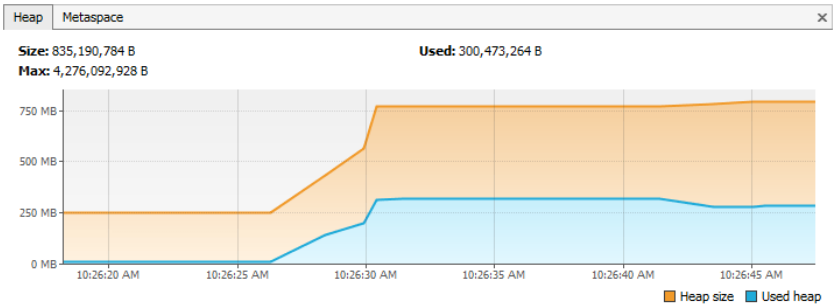
Profilování výkonu je prostředek pro určení, kde systém tráví nejvíce času. Ke shromažďování údajů o událostech se používají trasovací instrukce. Z dat lze získat různé typy informací, jako je například vstup a výstup komponent, volání funkcí, stav, komunikace zpráv nebo využití zdrojů [22]. Sledování výkonu má ovšem dopad na výkon systému, je dobré tedy provádět profilování v testovacím prostředí, které odpovídá reálnému provozu. Analýzou nasbíraných statistických dat lze navrhnout potřebné kroky, které danou část kódu optimalizují.

Důvody vedoucí k optimalizace nemusejí být pouze zrychlení aplikace. Může se jednat také o požadavek na přesunutí celé infrastruktury do cloudu. V takovém případě jde primárně o ušetření nákladů za provoz systému, kdy aplikace využívající efektivně své zdroje mohou běžet za menší provozní náklady. Jelikož každý poskytovatel nabízí jiný ceník služeb, optimalizace v těchto případech se týká například počtu požadavků na databázový server, využití procesorového času a paměti nebo počet dotazů na API.

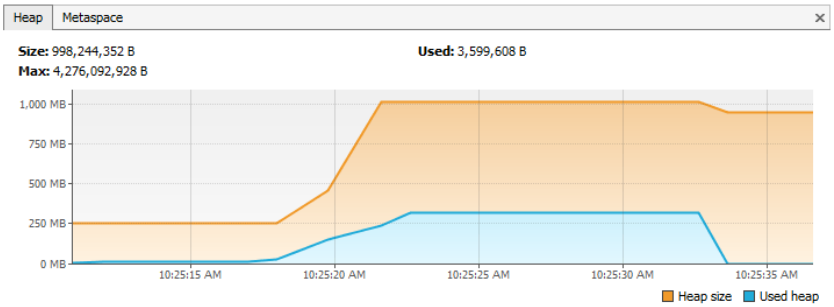
Testovaný systém je vytvořený v programovacím jazyce Java, a proto je i tato kapitola zaměřená především na vzory, které se objevují v Java světě. Možným problémem může být práce se zdroji, kde nesprávné použití programátorských konstrukcí povede například k možnému úniku paměti.

2.2.1 Únik paměti

Velkou výhodou jazyka Java je správa paměti přes garbage collector (GC), který se implicitně stará o její přidělování a uvolňování. Zatímco GC účinně zpracovává část paměti, která již není využívána žádným objektem, nezaručuje spolehlivé řešení úniků paměti. Stále mohou nastat situace, kdy aplikace generuje nadbytečné množství objektů, čimž dochází ke spotřebování paměťových prostředků a může dojít k selhání aplikace. Únik paměti je situace, kdy se na haldě nacházejí objekty, které se již nepoužívají a GC je nemůže odstranit, protože je stále na ně držena reference. Bez použití specializovaných nástrojů je odhalení úniků nesnadný úkol, je ale možné sledovat určité symptomy, které se v aplikaci mohou objevovat - zhoršení výkonu po delším běhu, podivné selhání aplikace nebo ukončení výjimkou *OutOfMemoryError*. Na obrázku 2.1 je cca v čase 10:26:30 vidět přidělení paměti pro statickou proměnnou, která v průběhu běhu aplikace není již uvolněna. Vyřešení úniku paměti pro stejný příklad zobrazuje obrázek 2.2, kde v čase 10:25:33 je vidět uvolnění prostředků.



Obrázek 2.1: Ukázka úniku paměti ve VisualVM. Zdroj [21].



Obrázek 2.2: Ukázka bez úniku paměti ve VisualVM. Zdroj [21].

Základní chybou je nesprávné použití statických proměnných pro ukládání objektů. Takové proměnné mají typicky životnost po celou dobu běhu aplikace. Staticky uložené jsou také objekty, které jsou vytvořené podle návrhového vzoru *jedináček*. U takových tříd, zvlášť drží-li nějaké zdroje, je důležité dbát na to, aby byly načítány podle vzoru *lazy loading*. Nejedná se ovšem o řešení problému, ale o minimalizaci příčin. Vytváření tříd až v době potřeby může ušetřit značné množství prostředků.

Nemálo častou a ne úplně na první pohled zřejmou chybou je nepřekrytí metod *equals()* a *hashCode()* pro nově definované třídy. Výchozím chováním je použití čísla adresy, na které se objekt nachází. Vytvářením více stejných objektů dojde k tomu, že v paměti se budou tyto objekty hromadit. Problém je dobře viditelný při použití mezipaměti, kam se ukládají objekty pro zvýšení výkonu. S přibývajícím množstvím objektů je nebude možné porovnat mezi sebou a určit, zda se jedná o ty samé. Nové objekty se budou považovat vždy za nové, ukládat se a prostor mezipaměti bude neustále růst. Výsledkem bude zhoršení výkonu celé aplikace.

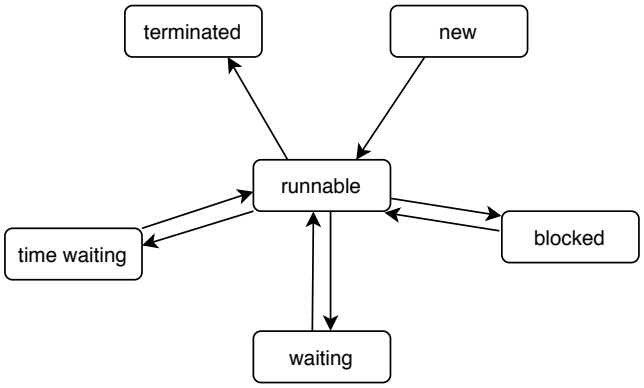
Úniky paměti jsou obtíženě řešitelné a jejich nalezení vyžaduje odbornou znalost jazyka Java. Při jejich řešení neexistuje univerzální řešení, protože k nim může docházet v řadě různých situací [21].

2.2.2 Stav vlákna

Sledovat stav vlákna je důležité pro efektivní využití procesorového času. Ideální stav je, jsou-li všechna vlákna ve stavu běžícím a maximálně využijí svůj přidělený čas. Vlákno v jazyce Java může existovat v jednom z následujících stavů, viz též diagram 2.3 [2].

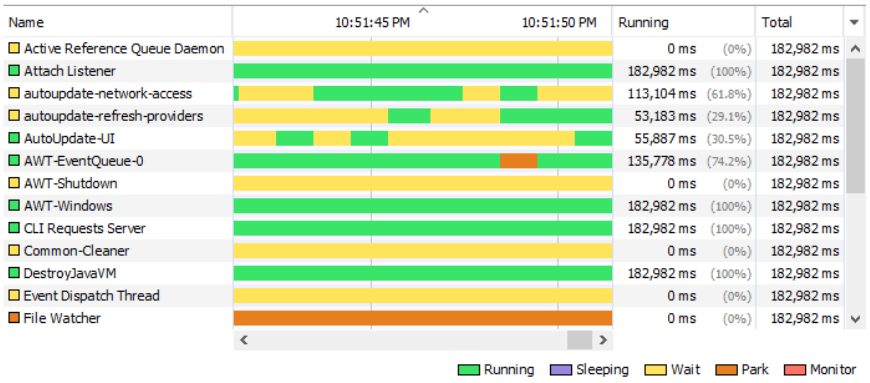
Vlákno se může nacházet v následujících stavech:

- **Nové**: Nově vytvořené vlákno, jehož kód ještě nebyl spuštěn.
- **Běžící**: Běžící vlákno je ve stavu připraveném ke spuštění a je na plánovači, aby dal vláknu čas na provedení operací, nebo již provádí svůj kód. Ve vícevláknových aplikací je vláknům přiděleno pevné množství času a postupně se střídají o procesorový čas. Ostatní vlákna čekají a jsou připravené ke spuštění.
- **Blokované**: Obsahuje-li vlákno práci s I/O, čekáním na dokončení operace se přesune do stavu blokované. Zodpovědností plánovače je, aby blokované vlákno znovu uvedl do běžícího stavu a naplánoval.
- **Čekající**: Vlákno se nachází v čekajícím stavu, přistupuje-li do chráněné sekce, kde se již nachází jiné běžící vlákno. Jakmile je chráněná sekce uvolněna, plánovač vybere jedno z čekajících vláken a uvede jej do běžícího stavu.
- **Časované čekání**: Volá-li vlákno metodu s parametrem časového limitu, vlákno se nachází v časovém čekání dokud není časový limit vypršen nebo pokud nedostane oznámení. Do tohoto stavu se vlákno dostane voláním metody *sleep* nebo čekáním na podmínkové proměnné.
- **Ukončené**: Ukončené vlákno je takové, které dokončilo svůj kód nebo nastala neočekávaná událost, která vedla k předčasnému ukončení vlákna (např. neošetřená výjimka).



Obrázek 2.3: Stavy vlákna v jazyku Java. Zdroj [2].

Na obrázku 2.4 je ukázka výstupu z programu VisualVM s vizualizací stavu vláken. Výstup v takové podobě je vhodný pro rychlou analýzu běžícího programu a může vést k rychlé lokalizaci připadného problému v aplikaci.

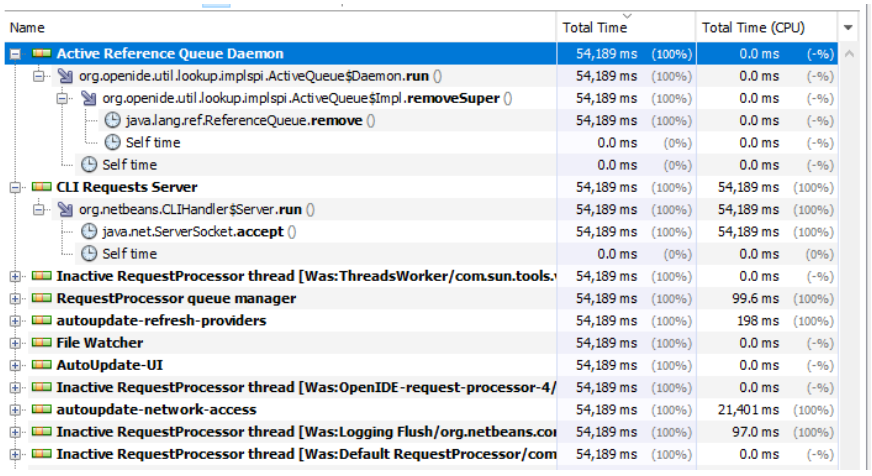


Obrázek 2.4: Ukázka stavu vláken ve VisualVM.

2.2.3 Vzorkování procesoru

Během profilování aplikace je vhodné využít i vzorkování procesoru. Odběr vzorků CPU ukazuje, kolik času aplikace tráví v jaké metodě. Nebo-li kolik času potřebuje metoda pro vykonání vlastního kódu. Na obrázku 2.5 je ukázka výstupu ze vzorkování CPU, kde je v prvním sloupci zobrazený název vlákna a názvy metod aktuálně vykonávaných.

Tuto metodu profilování je užitečné použít v kombinaci se stavem vlákna. Z výstupů obou metod lze zjistit, v jakém stavu se určité vlákno nacházelo a která metoda byla v daném časovém úseku prováděna.



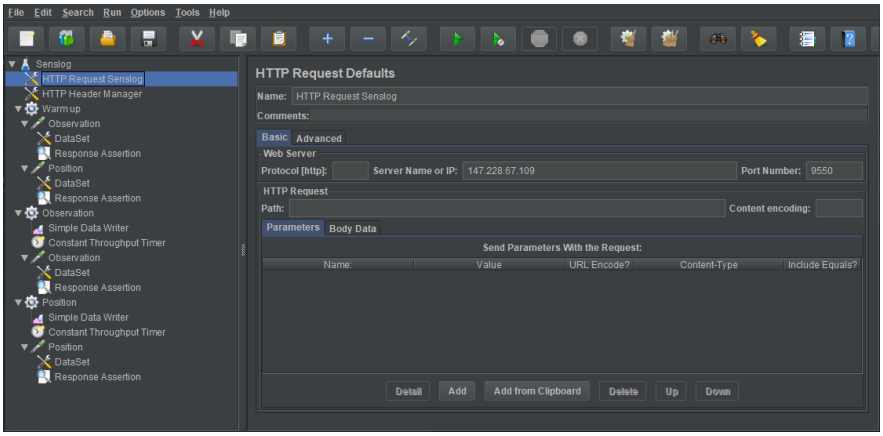
Obrázek 2.5: Ukázka vzorkování procesoru ve VisualVM.

2.3 Nástroje

Na základě charakteristiky testů ze sekce *Testování* byly vybrány vhodné nástroje, které jsou zde představeny. Vybrané nástroje slouží ke konfiguraci zátěžových testů, sběr záznamů o aktuálním vytížení serveru a profilování aplikace.

2.3.1 JMeter

Apache JMeter je aplikace s otevřeným zdrojovým kódem vytvořená v programovacím jazyce Java. Je navržena tak, aby umožňovala návrh zátěžových testů, jejich spouštění a sběr statistických výsledků z měření. Původně byl nástroj zaměřen pouze na testování webových aplikací, ovšem jeho funkcionalita byla rozšířena o další protokoly. Nástroj se používá k simulaci zátěže na server, skupinu serverů nebo sítě. Podporovány jsou nejběžnější protokoly a konektory pro komunikaci se serverem: HTTP(S), FTP, JDBC a další [6]. Aplikace nabízí grafické uživatelské rozhraní pro modelování testů (viz obr. 2.6), ale možný je i přístup z příkazové řádky, který je doporučovaný při reálném testování.

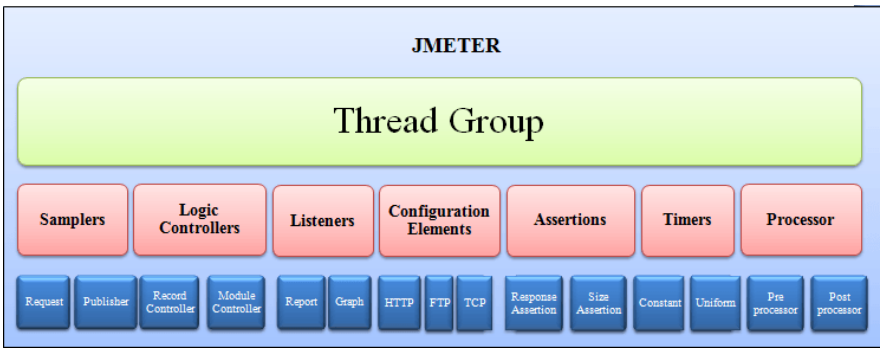


Obrázek 2.6: Ukázka uživatelského rozhraní JMeter.

Testovací plán (Test Plan)

Vytvoření testovacích scénářů následuje stromovou strukturu, kde vstupní bod je vytvoření testovacího plánu. Tento plán nastavuje výchozí chování např. definování uživatelských proměnných nebo přidání knihoven pro testo-

vací scénář. Architektura plánu je rozdělena do hiearchických modulů, které představují chování plánu (viz obr. 2.7) [7].



Obrázek 2.7: Architektura JMetru. Zdroj [1].

Skupiny vláken (Thread Group)

Skupina uživatelů se simuluje přes vlákna. Všechny kontrolery a vzorkovače musí být potomkem této skupiny vláken. Ostatní elementy např. posluchači mohou být potomkem testovacího plánu a budou se vztahovat k celému testovacímu scénáři (pro všechny uživatele/jádra). Dané skupině vláken lze nastavit počet, potřebný čas pro spuštění všech vláken a počet opakování. Každé vlákno spustí testovací plán nezávisle na ostatních vláknech [7].

Vzorky (Samplers)

Vzorky představují vytvoření vlastního dotazu na testovací subjekt a čekání na odpověď. Spuštění jednotlivých vzorů je dané uspořádáním ve stromu, je například možné jednoduše deterministicky simulovat průchod webovou stránkou.

JMeter obsahuje implementaci pro následující technologie:

- FTP
- HTTP
- JDBC
- Java object
- JMS
- JUnit Test
- LDAP
- Mail

- Příkaz operačního systému
- TCP

Každý vzorek má několik vlastností, které mohou být nastaveny. Při větším množství testovacích vzorků, jako jsou například dotazy na webový server, je doporučeno využít konfigurační prvky [7].

Logické kontrolery (Logic Controllers)

Pro organizaci testovacích vzorků se používají logické kontrolery. Je možné zvolit od jednoduchých kontrolerů, které neobsahují žádnou přidanou funkcionalitu (*Simple Controller*) a slouží především pro logické uskupení, až po složitější, které přídávají další funkcionalitu dané skupině. Často používaným je *Once Only Controller*, který je vhodný použít pro přihlášení uživatele. Tento kontroler je spuštěn pouze jednou pro uživatele/jádro dané testovací sady. Na výběr jsou další v podobě podmínek, smyček atd.

Posluchači (Listeners)

Posluchači poskytují přístup k informacím, které generuje JMeter během testování. Výsledky mohou být prezentovány v různých formách - graf, tabulka, strom nebo soubor. Využívání grafických výstupů přímo v grafickém uživatelském rozhraní aplikace se podle oficiální dokumentace nedoporučuje, jelikož prezentace těchto dat zatěžuje testovací stroj. Grafické vizualizace dat jsou vhodné především pro odladění funkčnosti testovacích scénářů a pro reálné testování použít posluchače, který generuje získané výsledky do souboru.

Konfigurační prvky (Configuration Elements)

Jak už název napovídá, konfigurační prvky slouží pro obecnou konfiguraci vzorků. Běžnou situací je, že testovaný server se nachází na určité IP adrese nebo doméně. Testovací scénář je složen z poslání několika dotazů na různé části webové aplikace včetně například přihlášení, kde získaný token musí být přiložen pro všechny následující požadavky. Jelikož je tento nástroj používaný především pro testování webových stránek, mezi užitečné konfigurační prvky patří ty, které začínají HTTP.

- HTTP Request Defaults: obecná konfigurace cílového serveru pro vzorky
- HTTP Header Manager: konfigurace hlaviček HTTP požadavku
- HTTP Cookie Manager: správa cookie pro uložení ID sezení (session)
- CSV Data Set Config: načtení testovacích dat z CSV souboru

Tvrzení (Assertions)

Pro otestovaní správnosti dotazu na server je důležité také znát odpověď z testovaného serveru. Tvrzení umožňují přidat logiku do odpovědi dotazu a je možné testovat, zda server odpověděl očekávanou zprávou, nebo zda odpověď prohlásit za odmítnutou. Příkladem může být HTTP dotaz pro vložení nových dat na webový server. Server vrátí odpověď s HTTP kódem o úspěchu, ovšem v těle se nachází data ve formátu JSON se zprávou o neúspěšné operaci. V této chvíli je vhodné použít tvrzení, kde je možné nastavit regulární výraz, a tím zjistit skutečný stav transakce.

Časovače (Timers)

Jednotlivé dotazy se v základním nastavení spouštějí bez časování, což může způsobovat zahlcení testovacího serveru. V manuálu je doporučováno použít časovače, které umožňují kontrolovat generovanou zátěž pro vlákna [7].

Zpracovávač (Processor)

Zpracovávače se dějí na dvě skupiny - před a po. Z logiky názvu vyplývá, že se jedná o nějakou úpravu dat před tím, než je zavolán testovací vzorek a následně získání nějakých dat z odpovědi. Nejčastěji se tato funkcionalita provádí právě při posílání HTTP požadavků, kde po přihlášení je potřeba získat autorizační token pro další požadavky.

2.3.2 BlazeMeter

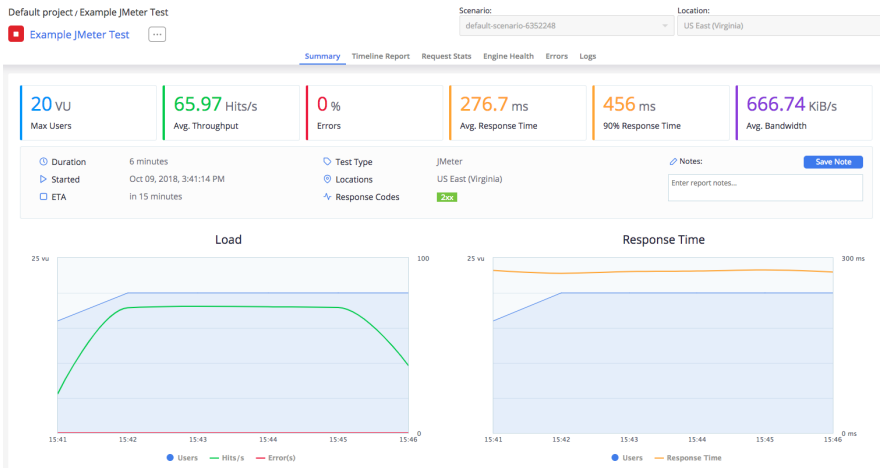
BlazeMeter je cloudové řešení pro testování webové aplikace. Hlavní předností je 100% kompatibilita s konfiguračním souborem z aplikace JMeter. Služba nabízí funkcionalitu v podobě funkcionálního testování, které dovoluje vytvářet HTTP požadavky na RESTové API. BlazeMeter také podporuje služby pro mokování a monitorování API.

Výkonnostní testování

Služba pro testování podporuje nejrozšířenější nástroje pro vytváření testovacích scénářů. Kromě JMeter lze použít Selenium, Gatling a mnoho dalších. Hlavním benefitem je možnost určit lokalitu, ze které se má testování spustit. Služba využívá Google Cloud Platform, Amazon Web Services a Microsoft Azure, je tedy možné zvolit lokality, ve kterých se nachází poskytovatelé těchto služeb. Na obrázku č. 2.8 je vidět výstup z testování, který agreguje

základní údaje o výkonnosti serveru a pod těmito údaji se nachází vizualizace průběhu dotazů.

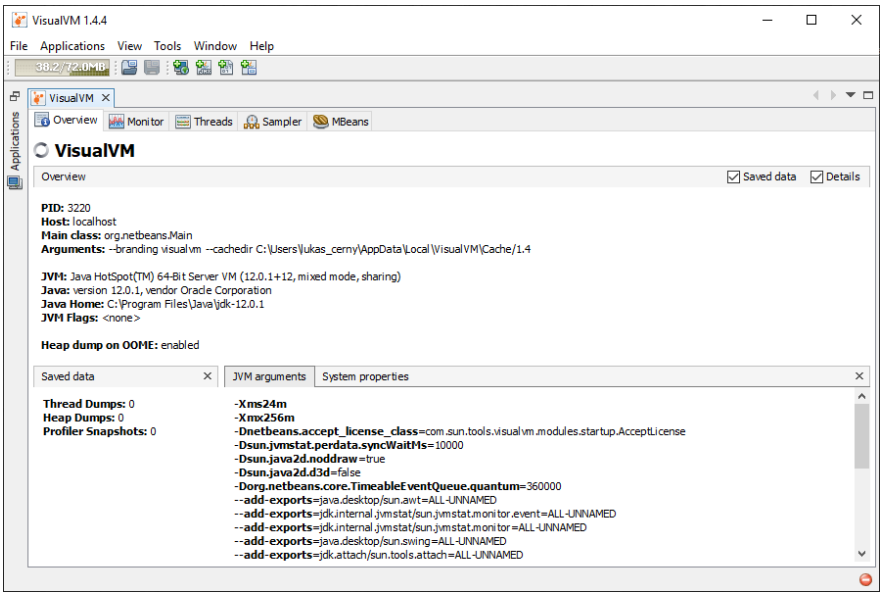
- **Maximálně uživatelů:** maximální počet uživatelů, kteří běželi souběžně v určitý okamžik
- **Průměrná propustnost:** průměrný počet HTTP požadavků za sekundu
- **Míra chyb:** procentuelně vyjádřená míra špatných požadavků na server
- **Průměrná doba odezvy:** průměrná hodnota počtu požadavků na server vyjádřená v jednotkách milisekund
- **90 percentil z doby odezvy:** průměrná doba odezvy pro 90% testovaných vzorků
- **Průměrná propustnost:** průměrná propustnost serveru



Obrázek 2.8: BlazeMeter: grafický report

2.3.3 VisualVM

VisualVM je nástroj poskytující vizualizační rozhraní pro profilování aplikací vytovřených v Java technologiích a běžícím na JVM (Java Virtual Machine). Nástroj organizuje data poskytnutá JVM, které jsou načtena nástrojem Java Development Kit (JDK) a umožňuje reprezentaci dat z více Java aplikací. Nabízí možnost analyzovat aplikace běžící lokálně a vzdáleně [34]. Pro každou aplikaci je následně možné vizualizovat data poskytnuté JVM v několika záložkách (viz obrázek č. 2.9).



Obrázek 2.9: Grafické uživatelské prostředí VisualVM

Přehled (Overview)

Záložka zobrazuje obecné informace o analyzované aplikaci a její běhové prostředí - JVM argumenty, systémové proměnné a informace následující struktury [34]:

- **PID:** identifikátor procesu aplikace
- **Host:** umístění běžící aplikace
- **Main class:** hlavní třída pro spuštění aplikace
- **Arguments:** argumenty aplikace při spuštění
- **JVM:** verze JDK pro virtuální stroj
- **Java Home:** umístění JDK na hostovaném stroji
- **JVM flags:** zobrazuje značky JVM aplikace při spuštění JDK
- **Heap dump on OOME:** zobrazuje status při překročení dostupné paměti na haldě

Sledování (Monitor)

Následující záložka pro sledování aplikace v reálném čase obsahuje čtyři grafy zobrazující:

- **CPU:** vytížení procesoru

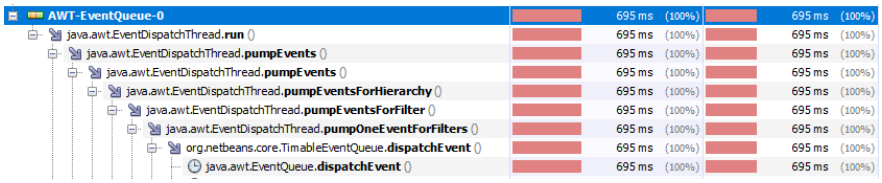
- **Heap/Metaspace:** celková velikost paměti a kolik je aktuálně využíváno
- **Classes:** počet načtených a sdílených tříd
- **Threads:** přehled uživatelských a démonových vláken aplikace

Vlákna (Threads)

Záložka s vlákny zobrazuje časovou osu s jejich stavem v reálném čase. Je možné zobrazit aktuálně běžící, ukončené nebo čekající vlákna [34]. Praktické využití je především v tom, že je možné sledovat činnost každého vlákna včetně doby jejich běhu a analyzovat, zda se program chová efektivně a nedochází k zastavování vláken. Užitečnou funkcí pro potřeby analýzy je výpis aktuálního stavu vláken (thread dump).

Vzorkovač (Sampler)

Vzorkování nabízí dvě možnosti - procesorový čas a množství paměti. Obě funkcionality se zaměřují na základní profilování a bohužel je nelze použít současně. Vzorkování procesoru sbírá informace z JVM a periodicky je zobrazuje do hiearchicky seřazené struktury (viz obr. 2.10). Každý záznam obsahuje celkový čas strávený v dané metodě. Možné je zobrazit i procesorový čas pro každé vlákno. Při tomto zobrazení se ovšem nezobrazují volané metody. Podobná funkcionalita je zobrazena při vzorkování paměti (viz obr. 2.11), ovšem zobrazuje se využití paměti pro jednotlivé datové struktury na haldě.



Obrázek 2.10: Ukázka výstupu vzorkování procesoru ve VisualVM.

Obrázek 2.11: Ukázka výstupu vzorkování paměti ve VisualVM.

2.3.4 Top

Pro měření zdrojů využívaných jednotlivými aplikacemi je vhodné zvolit nástroj *top*, který je dostupný na linuxových platformách. Program poskytuje dynamický pohled na běžící systém v reálném čase. Může zobrazit souhrnné informace o systému a seznam procesů, které jsou aktuálně spravovány jádrem Linuxu [4]. Ačkoliv je program dynamický, včetně jednoduchého grafického prostředí, nabízí přístup přes příkazovou řádku.

Užitečné parametry při spuštění aplikace:

- **-b**: dávkový přístup, výstupy mohou být přesměrovány do souboru
- **-c**: obsluha přes příkazový řádek
- **-u**: filtr procesů podle uživatele
- **-p**: filtr procesů podle PID procesu
- **-d**: interval pro aktualizaci výstupu v sekundách

```
top -bc -u <user> -d 2
top -bc -p <PID> -d 2
```

Ukázka výstupu programu je zobrazena v boxu níže. V hlavičce jsou obsaženy souhrnné informace o stroji, například využití CPU v režimu jádra nebo v uživatelském režimu, nebo využití paměti RAM. Pro účely měření jsou využity sloupce **%CPU** a **%MEM** u jednotlivých procesů.

```
top - 09:56:13 up  1:30,  1 user,  load average: 0.24, 0.12, 0.16
Tasks: 102 total,   2 running, 100 sleeping,   0 stopped,   0 zombie
%Cpu(s): 18.8 us,  6.2 sy,  0.0 ni, 70.3 id,  4.7 wa,  0.0 hi,  0.0 si,  0.0 st
MiB Mem :   3850.4 total,   2899.8 free,    400.2 used,    550.3 buff/cache
MiB Swap:    952.0 total,    952.0 free,      0.0 used.   3184.5 avail Mem

PID USER      PR  NI  VIRT    RES    SHR    S  %CPU  %MEM  TIME+    COMMAND
658 postgres  20  0   235456  35872  30436  R  31.2  0.9   0:00.05  postgres: 11/main
518 postgres  20  0   216160  27224  25272  S   6.2  0.7   0:00.39  /usr/lib/postgresql
```

2.3.5 Iotop

Velice podobný předchozímu nástroji je *iotop*. Taktéž poskytuje pohled na běžící procesy v reálném čase, ovšem zaměřuje se na měření vstupních a výstupních operací. Využití tohoto nástroje přináší výhodu především při monitorování komunikace s databází.

Užitečné parametry pro monitorování databáze:

- **-b**: dávkový přístup, bez interaktivního módu
- **-u**: filtr procesů podle uživatele
- **-d**: interval pro aktualizaci výstupu v sekundách

```
iotop -b -u <user> -d 2
```

Uživatelské rozhraní je velice podobné a je také rozděleno do dvou částí - hlavička s obecnými informacemi a podrobné údaje pro jednotlivé procesy. V hlavičce je uvedena celková a aktuální rychlost zápisu a čtení na disk. Jednotlivé procesy obsahují podrobnější informace, ovšem sledovaným atributem je především IO. Tato hodnota zobrazuje procento času, které vlákno nebo proces tráví čekáním na tyto operace [3]. Bude-li proces obsahovat hodnotu blížící se ke 100% znamená to, že většinu svého času běhu tráví čekáním na I/O operace. Stane-li se tak, neznamená to, že by byla využita veškerá přenosová kapacita disku, ale že proces tráví většinu svého času čekáním na disk.

```
Total DISK READ:         0.00 B/s | Total DISK WRITE:      1292.40 K/s
Current DISK READ:       0.00 B/s | Current DISK WRITE:    1132.56 K/s
TID    PRIO  USER      DISK READ  DISK WRITE  SWAPIN  IO       COMMAND
25025  be/4  postgres  0.00 B/s   245.61 K/s  0.00 %  21.42 %  postgres: 11/main
25018  be/4  postgres  0.00 B/s   206.63 K/s  0.00 %  17.23 %  postgres: 11/main
```

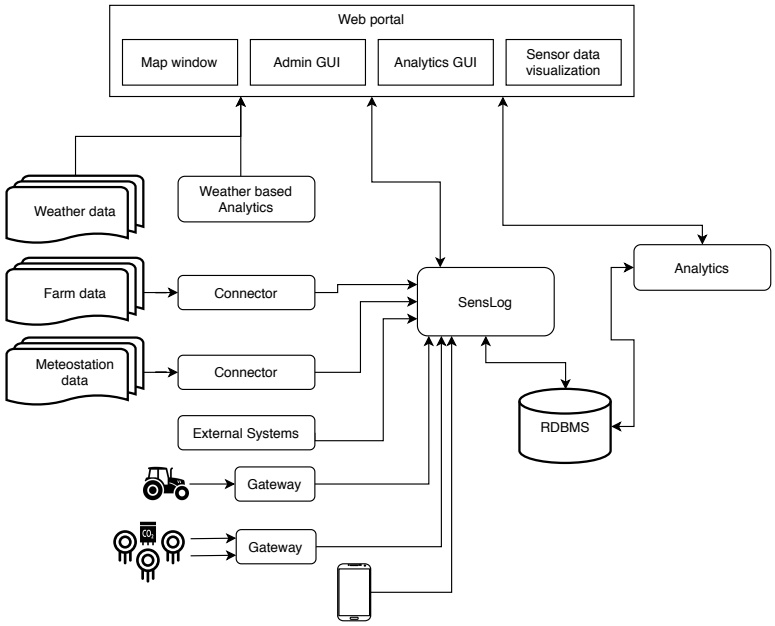
3 Systém SensLog

SensLog je webový systém pro správu senzorových dat. Umožňuje přijímat surová data z různých typů zařízení. Projekt byl zahájen na Fakultě Aplikovaných věd v Plzni v roce 2015 a během vývoje byl otestován na projektech SDI4Apps [12], FOODIE [11] a OTN [32]. Část projektu je vedena s otevřeným zdrojovým kódem na portálu GitHub [18], která je použita pro účely této práce, a část projektu není k dispozici, protože je využívána pro komerční použití. Systém je aktuálně ve verzi *1.4.* a není již dále aktivně rozšiřován.

Obecné úkoly SensLogu lze shrnout do následujících bodů:

- Příjem měřených dat (pozorování) buď přímo ze senzorového zařízení, nebo nepřímo z jakékoliv brány, do které je zařízení připojeno.
- Ukládání dat ze senzorů do relačního modelu.
- Předběžné zpracování dat pro potřeby analýz.
- Publikování dat prostřednictvím systému webových služeb.

Systém SensLog slouží jako centrální prvek pro ukládání senzorických dat z různých zdrojů a publikování přes webové služby ve formátu JSON a standardu OGC Sensor Observation Service verze 1.0.0. Infrastruktura kolem systému byla postupně rozšiřována a vznikly další podpůrné aplikace rozšiřující funkcionalitu (viz obrázek 3.1). Jednou z nich je aplikace *Connector*, která zajišťuje integraci dat z různých systémů. Systém je tak schopný dostávat další data z různých zdrojů, které lze využít pro komplexnější analýzy. Pro všechny přijímané hodnoty platí, že jsou naměřena z fyzického zařízení a obsahují jednotku měřené veličiny [19].



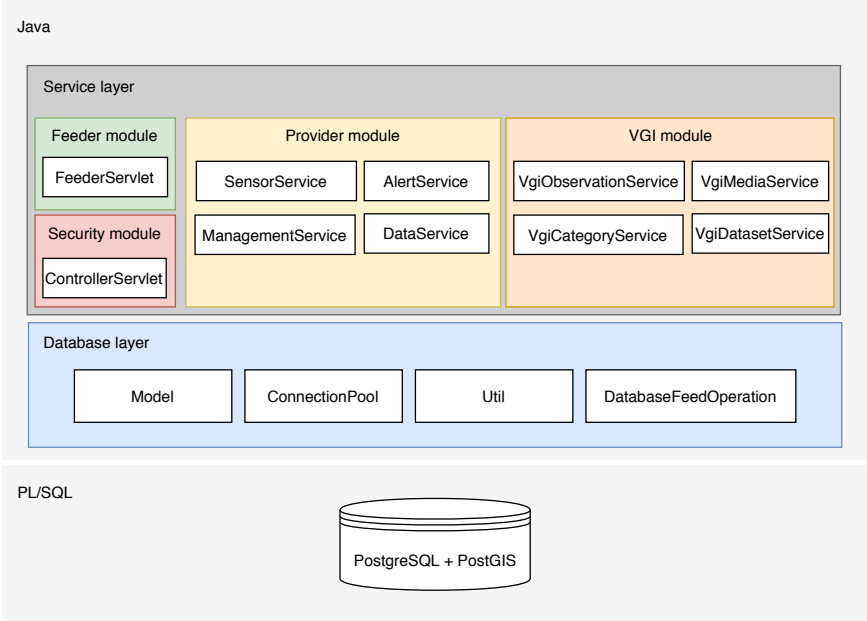
Obrázek 3.1: Pohled na využití systému SensLog.

Senzorická zařízení jsou součástí jednotek a dělí se do dvou kategorií - statické a mobilní. Do statických jednotek jsou instalovány senzory, které nemění svou polohu v čase. Jejich poloha je dána jednorázově při instalaci a mění se pouze, dojde-li k fyzickému přemístění jednotky. Příkladem mohou být senzory pro měření vlhkosti nebo teploty v půdě.

Pro mobilní jednotky je charakteristické, že je v reálném čase zaznamenávána i poloha zařízení. Pozice zařízení je ve většině případů určována globálním družicovým systémem (GNSS). Příklady tohoto systému jsou americký GPS, ruský GLONASS a evropský Galileo. Pozorované polohy jsou přijímány primárně z mobilních zařízení a zpracovávány do souvislé trajektorie. Mobilní zařízení mohou být vybavena libovolným počtem dalších senzorů, například teplota, světelné podmínky, stav jednotek nositele (pro člověka, například srdeční tlak, pro automobily, například tlak v pneumatikách). Tato pozorování jsou zpracovávána stejně jako pozorování od statických senzorů, jsou ale vázáné na pozici, kde bylo pozorování provedeno [30].

3.1 Architektura

Struktura systému je rozdělena do dvou částí. Serverová část je implementovaná v programovacím jazyce Java a zabývá se přijímáním a publikací dat. Databázová část, která využívá konstrukcí jazyka PL/SQL v systému řízení báze dat (DBMS), zajišťuje manipulaci s daty a jejich perzistenci. Aplikace je rozdělena do jednotlivých vrstev, jak je znázorněno na obrázku 3.2, a příchozí požadavky procházejí při zpracování jednotlivými vrstvami.



Obrázek 3.2: Modulární architektura systému SensLog.

3.2 Serverová aplikace

Serverová aplikace využívá aplikační server Apache Tomcat a je vytvořená v jazyce Java verze 7. Komunikace se systémem je zajištěna veřejným aplikačním rozhraním poskytované aplikačním serverem (Web API). Implementováné je také grafické uživatelské rozhraní (GUI), které je vytvořené v jazce JavaScript a Java technologii JavaServer Pages (JSP). Tento způsob ovládání systému ovšem není funkční v testovací verzi a komunikace probíhá výhradně přes veřejné Web API. Systém využívá standardní webové knihovny dané specifikací Java Enterprise Edition a Java Database Connectivity (JDBC)

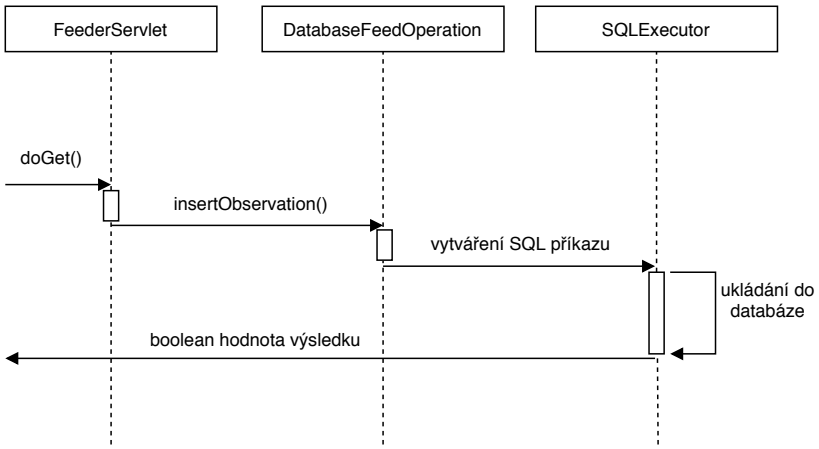
pro připojení do databáze. Dalšími využívanými knihovnami jsou pro práci s JSON a XML formáty (Jackson), XSLT formátem (Saxon), grafy (Jfreechart).

3.2.1 Servisní vrstva

Servisní vrstva, složená z jednotlivých modulů, zajišťuje komunikaci se systémem a obsahuje strukturu podle veřejného API (viz sekce 3.4). Jelikož se jedná o vrstvenou architekturu využívá databázovou vrstvu pro ukládání a načítání perzistentních dat. Všechny moduly používají webové technologie a komunikace probíhá přes protokol HTTP.

Feeder modul

Modul zajišťuje stěžejní funkcionalitu systému, a to přijímání surových senzorových dat - observace a pozice. Je založen na technologii *Java Servlet* a využívá třídu *HTTPServlet*. Příjem dat je obsluhován metodou *doGet()*, kde na základě parametru dotazu jsou data správně převedena na příslušné datové typy, provede se validace dat a následně uloží do databáze. Celý proces zpracování požadavku je znázorněn na sekvenčním diagramu 3.3.



Obrázek 3.3: Sekvenční diagram zpracování observace.

Provider modul

Modul obsahuje jednotlivé služby, které umožňují nahlížet na data a reprezentovat je ve formátu JSON. I zde je použita technologie *Java Servlet*

a zpracování požadavku zajišťuje metoda *doGet()*. Metoda obsahuje převod parametrů na správné datové typy a podle rozhodovacího stromu zavolá příslušnou metodu databázové vrstvy pro načtení dat. Celkový průběh zpracování celého dotazu je velice podobný zmiňovanému Feeder modulu.

VGI modul

VGI je zkratka pro geografické informace získané od neprofesionálů (dobrovolníků). Charakteristika těchto dat sdílí část atributů s body umístěných na mapě nazvané jako body zájmů (POI). Jedná se o termín používaný v kartografii pro reprezentaci určitého bodu pomocí ikony nebo kategorie. Myšlenka je taková, že na rozdíl od lineárních prvků, jako jsou silnice nebo pozemky, mohou být některé georafické souřadnice vhodné označit jako bod v konkrétním kontextu. Příkladem jsou budovy pošty, které jsou označeny obálkou a lze je tak snadněji lokalizovat na mapě [25].

Tento modul implementuje funkcionalitu, která nabízí rozhraní pro sběr takto strukturovaných dat. Struktura dat není pevně definovaná a je možné přidávat rozšiřující atributy, které mají omezení pouze v datovém typu, který je možný přenést v JSON formátu. Příjem požadavků je implementován v technologii *JAX-RS* a architektonickém stylu REST.

Security modul

Modul implementující zabezpečení je zde z důvodu webové aplikace. Obsahuje jednoduchou funkcionalitu autentizace, autorizace a nastavuje potřebné atributy do uživatelského sezení (session), které jsou potřebné pro procházení aplikace. Stejně jako v předchozích modulech je využita technologie *Java HTTPServlet*. Více o bezpečnosti systému je popsáno v sekci **Zabezpečení**.

3.2.2 Databázová vrstva

Databázová vrstva zajišťuje komunikaci s databází. Obsahuje třídy datového modelu (viz 3.3.1), správu připojení, sadu nástrojů pro složení jednotlivých SQL příkazů z datových objektů a spouštěče těchto příkazů. Zajímavostí této vrstvy je vlastní implementace správy připojení přes JDBC konektor. Zmíněná sada nástrojů pro vytváření dotazů může být potencionální místo k napadení systému, jelikož jednotlivé SQL příkazy se skládají jako textové řetězce.

3.3 Perzistence dat

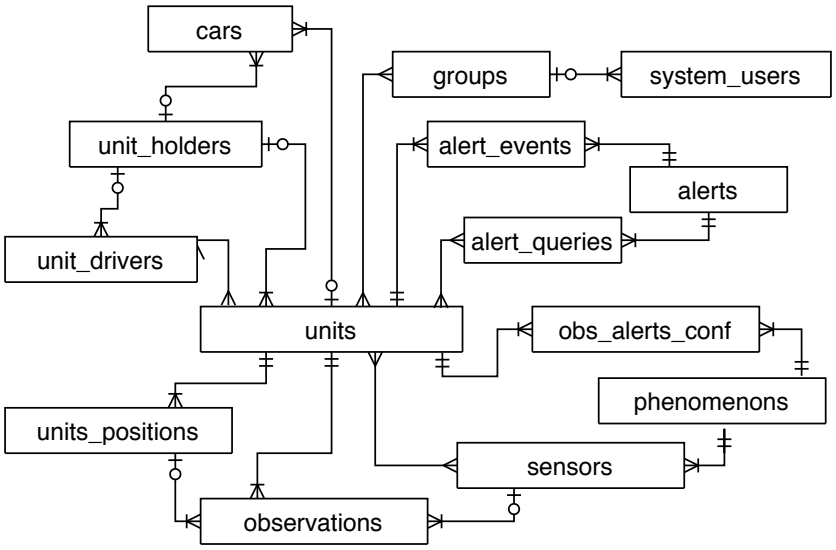
Pro perzistenci dat slouží relační databáze PostgreSQL s rozšířením PostGIS. Toto rozšíření rozšiřuje standardní funkcionalitu o prostorová a geografická data [26]. Tento typ dat se v databázi nachází u pozice a objektů zájmů, nad kterými probíhají různé transformace.

3.3.1 Datový model

Datový model přestavuje nejdůležitější část celého systému. Musí být schopný ukládat přijatá data tak, aby nedocházelo ke ztrátě informace a při načítání se zpráva dala zpětně sestavit. Systém obsahuje dva hlavní moduly - zpracování observací dle standardu OGC a VGI. Tato skutečnost musí být promítnuta i do datového modelu, který je rozdělen do dvou schémat - *public* a *vgi*. Na diagramu 3.4 je zobrazeno jádro datového modelu ze schématu *public* pro ukládání dat ze statických a mobilních jednotek. Celý model je uveden v příloze a obsahuje 31 tabulek. Pilíře modelu jsou tabulky *units*, *sensors* a *observations*.

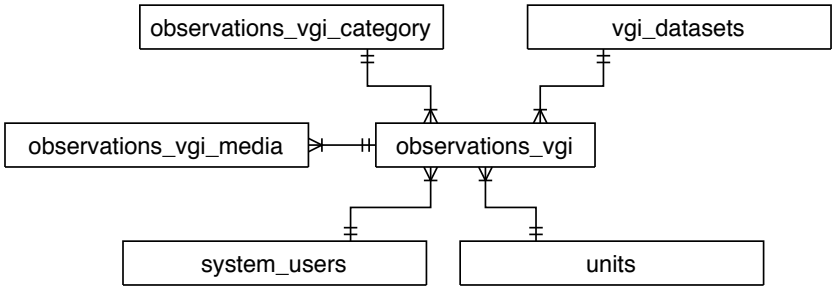
Díky narůstajícímu počtu informací, především v tabulkách uchovávající observace a pozice jednotek, jsou v databázi vytvořena další dvě schémata s názvem *maplog_obs_child* a *maplog_upos_child*. Ta jsou využívána při seskupování záznamů do takzvaných oddílů, které přináší výhody v načítání a hledání v uložených datech. Se zvyšujícím se množstvím dat v tabulkách *observations* a *units_positions* se tyto operace stávaly časově náročnými a toto řešení pomohlo problém vyřešit. Funkcionalita je implementována procedurami v PL/SQL (viz sekce 3.3.2) a provádí se před každým uložením nového záznamu. Připravená schémata obsahují automaticky generované tabulky, do kterých se duplicitně ukládají záznamy. Název tabulky je složen z přijatých informací, konkrétně se jedná o název jednotky a rok. V rámečku níže je uveden vzor pro sestavování názvu tabulky a ukázka.

```
Vzor:
    maplog_obs_child.observations_<rok>_<id_jednotky>
    maplog_obs_child.units_positions_<rok>_<id_jednotky>

Ukázka:
    maplog_obs_child.observations_2019_12345
    maplog_obs_child.units_positions_2019_12345
```



Obrázek 3.4: Jádro datového modelu systému SensLog.

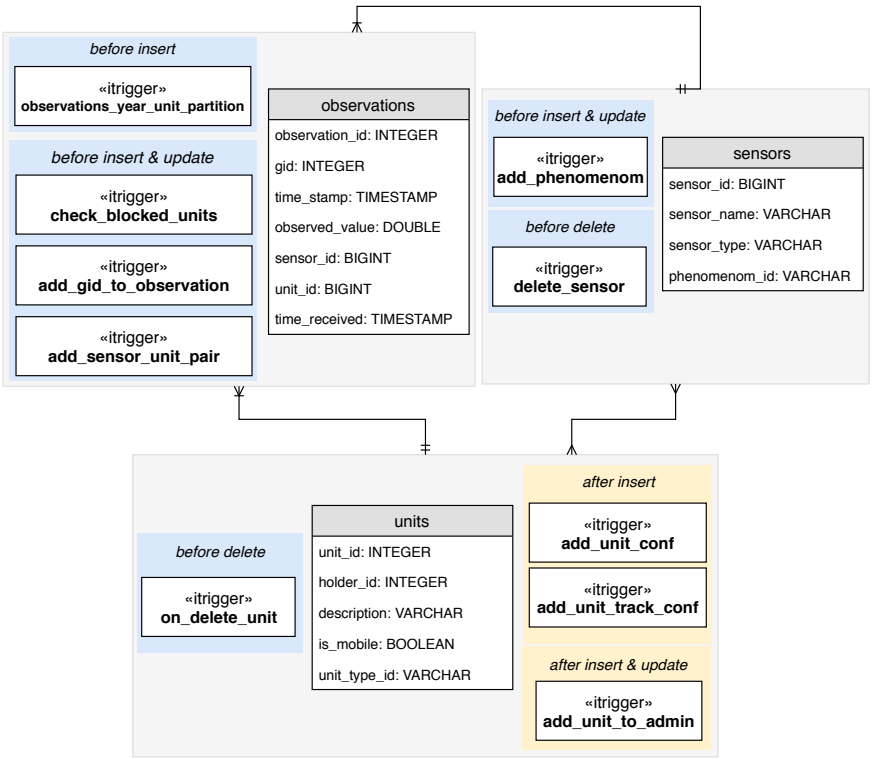
Pro modul VGI byl vytvořen datový model zobrazený na diagramu 3.5, který představuje rozšíření hlavního modelu. Celkem obsahuje 5 tabulek, ale využity jsou i další tabulky z hlavního modelu. Stežejní funkcí je zde možnost ukládat multimédia do tabulky *observations_vgi_media* a přiřazení k observaci. Média jsou ukládána jako bytová pole.



Obrázek 3.5: Datový model VGI modulu.

3.3.2 Funkce PL/SQL

Druhá část logických operací, která zajišťuje především manipulaci s daty, je implementována funkcemi v jazyce PL/SQL. Tyto funkce jsou přiřazeny jako spouštěče (tzv. trigger) před nebo po vykonání nějaké operace nad tabulkou (např. vložení nového záznamu nebo úprava již existujícího). Funkce obsahují logiku pro již zmíněné ukládání do oddílů, částečnou implementaci časových databází, validaci správnosti dat a geografické výpočty (např. vzdálenost dvou bodů využitím funkcí z rozšíření PostGIS).



Obrázek 3.6: Ukázka spouštěčů nad hlavními tabulkami.

Před uložením nové observace dochází ke kontrole, z jaké jednotky a senzoru je záznam pořízen. Pokud senzor nebo jednotka neexistuje, je automaticky vytvořen záznam do příslušné tabulky. Při ukládání záznamu do tabulky se senzory je provedena kontrola na existenci fyzikálního jevu (například pro teplotní senzor se jedná o stupně Celsia). Neexistuje-li, je opět automaticky vytvořen. Další kontrolou je, zda jednotka nepatří mezi blokované. Dojde také k přiřazení poslední polohy jednotky (u statických jednotek je poloha

zadaná ručně a jedná se vždy o stejnou hodnotu, u mobilních jednotek se periodicky mění). Poslední operací před uložením je zmíněné oddílování, které může znamenat největší zátěž databáze při ukládání. Název je složen z textových řetězců, které se načítají z příchozích dat, a následně podle složeného názvu oddílu dojde k jeho vyhledání a přiřazení observace. Na obrázku 3.6 jsou znázorněny zmíněné funkce pro hlavní tabulky.

3.4 Veřejné rozhraní (API)

Ovládání systému je možné plně provádět přes veřejné Web API. Struktura odpovídá modulům v servisní vrstvě. Umožňuje práci s observacemi (vkládání a publikování) a obsluhu bodů zájmů. Jedná se o webové služby, přenos dat využívá protokol HTTP a data se posílají jako parametry dotazu nebo ve formátu JSON. Celkový přehled všech služeb je k dispozici na webových stránkách systému www.senslog.org.

3.4.1 Vložení observace

Vložení nového pozorování je nejčastěji využívanou službou. Z historických důvodů je vložení možné provést pouze přes dotaz *GET* a všechny hodnoty jsou přiloženy jako parametry dotazu. Ukázka je zobrazena v boxu níže.

```
Požadavek: GET <domain>/FeederServlet?
        Operation=InsertObservation&
        value=15.5&
        date=2015-07-15 12:00:00+0200&
        unit_id=1234567&
        sensor_id=7654321

Odpověď: true
```

V ukázce je vidět, že typ operace je uveden přímo v dotazu u parametru *Operation*. Podle tohoto parametru se zpracovává i zbytek dotazu. Následující parametry jsou hodnoty z pozorování - *value* značí vkládanou hodnotu; *date* časové razítko; *sensor_id* typ senzoru, ze kterého hodnota pochází; *unit_id* obsahuje identifikátor jednotky, ke které je senzor přiřazen. Návratová hodnota je datového typu *boolean* - *true* při úspěšném uložení a *false* při neúspěšném. Při neúspěchu bohužel není uvedeno k jaké chybě došlo a nelze příslušně zareagovat.

3.4.2 Získání observací

Další častou operací je zobrazení dat. Veřejné API zpřístupňuje data všech jednotek nebo senzorů, poslední pozice jednotek nebo zobrazení observací. Observace je možné získat intervalově pro danou jednotku a senzor, nebo poslední observaci ze všech jednotek (viz ukázka níže).

```
Požadavek: GET <domain>/SensorService?
    Operation=GetLastObservations&
    group=<group>&
    user=<user>

Odpověď: [{
    "gid": 56349,
    "observedValue": 15.5,
    "sensorId": 1234567,
    "timeStamp": "2015-07-15 12:00:00+02",
    "unitId": 1234567
}, ...]
```

V boxu je ukázka dotazu pro načtení poslední vložené observace ze všech jednotek. Parametry dotazu jsou jméno uživatele a skupiny, do které uživatel patří. Tyto parametry slouží ke kontrole oprávnění uživatele zobrazovat data jednotek. Výsledek dotazu je pole objektů v JSON formátu. Každý objekt reprezentuje poslední pozorování pro daný senzor. Mezi atributy navíc přibyl identifikátor poslední polohy *gid*.

3.5 Zabezpečení

Zabezpečení systému je na velice slabé úrovni. Veřejné API je veřejné v pravém slova smyslu a volání jednotlivých služeb nevyžaduje předchozí autentizaci. Systém se může stát snadným cílem pro potencionálního útočníka. Některé služby disponují jednoduchou formou autorizace v podobě uživatelského jméno a názvu skupiny (viz 3.4.1). Služby pro vkládání nových záznamů nedisponují touto formou autorizace a přijaté hodnoty jsou rovnou zpracovávány.

Zmíněná webová aplikace disponuje formulářem pro autentizaci a při procházení jednotlivých stránek dochází k ověřování práv uživatele. Jelikož ovládání systému probíhá primárně přes webové API, a aplikace nebyla v testované verzi zcela funkční, nelze bezpečnost zcela ověřit.

3.5.1 Ohrožení systému

Neautorizovaný přístup je nebezpečný především při manipulaci s daty. Budou-li do systému posílána data z měření, která budou použita pro řízení dalších systémů, může dojít k fatálním následkům. Nebezpečí se skrývá také v pouhém čtení dat, ikdyž se nejedná o citlivé osobní údaje, poskytované výsledky mohou být zneužity konkurencí.

Napadení systému je možné provést i bez povšimnutí provozovatele. Jedná se o útok, při kterém dojde k znefunkčnění některých částí systému nebo k jeho zhroucení. Právě útok pro zhroucení systému, nebo-li útok odepření služby (DoS útok) je jednoduše proveditelný přes veřejné API. Stačí k tomu generátor zpráv se zvyšující se časovou značkou a zprávy periodicky posílat do systému. Při dostatečné dlouhé době dojde využití veškerého paměťového místa pro databázi a systém se stane nedostupný. Může se stát, že při útoku dojde k využití chyby v systému, která způsobí jeho zhroucení. Odhalení takových chyb je i součástí této práce.

Zmíněný formulář webové aplikace obsahuje dvě vstupní pole (pro přihlašovací jméno a heslo), které představují další bezpečnostní riziko v podobě SQL Injection. Jedná se o typ útoku, kdy útočník vloží sérii SQL příkazů do SQL dotazu a pokusí se zmanipulovat data [5]. Tento typ chyby je velice nebezpečný, protože umožňuje útočníkovy přístup k databázi a spouštět příkazy pro modifikaci dat popřípadě mazání celých tabulek. Jedním z předpokladů je, že útočník musí znát schéma databáze, aby mohl s tabulkami manipulovat.

Pro komerční použití se používá upravená verze, která umožňuje příjem požadavků pouze ze stroje na kterém aplikace běží (localhost). Před systémem je předřazena aplikace (firewall), která monitoruje, od jakého klienta mohou být požadavky přijaty. Tento firewall není k dispozici při vypracování této práce, takže není možné posoudit, jak moc je celé řešení bezpečné v komerční sféře.

3.6 Verze 2

Paralelně s touto prací vzniká i SensLog 2.0, který má přinést výrazné zvýšení rychlosti a umožnit jednoduché škálování. Kromě rychlosti je kladen důraz i na zabezpečení, odolnost a spolehlivost systému. Zabezpečené budou nejen jednotlivé služby, ale přenos dat bude šifrovaný. Jednou z hlavních změn je nový datový model, který bude kompatibilní se standardem OGC SensorThings API [23], a také s původním modelem systému. Z toho vychází požadavek na zpětnou kompatibilitu API.

4 Měření systému SensLog

Cílem testování je ověřit, zda je systém připravený na využití ve scénářích zahrnujících internet věcí, například v rámci konceptů Průmyslu 4.0. a chytrého zemědělství (smart farming). Pro testovací sady jsou využity teoretické znalosti z kapitoly *Analýza výkonnosti podnikových aplikací* a je v nich promítnuto zátěžové i stresové testování. Testovací sady jsou nakonfigurovány v nástroji *JMeter* a pro měření výkonu a profilování aplikace jsou využity zbylé nástroje ze sekce 2.3.

4.1 Charakteristika dat

Důležitým kritériem pro testování výkonnosti jsou reálné charakteristiky dat vycházející ze zákaznických požadavků a simulovat tím reálné zatížení. Pro tyto účely jsou vybrány dva zákazníci, pro které je/bude systém SensLog instalován a na kterých lze ukázat minimální a předpokládanou maximální zátěž na systém. Charakteristika dat vychází z interní dokumentace k projektu AFarCloud a SensLog.

4.1.1 Senzory v poli

Pro účely sběru dat o kvalitě půdy na pozemku je vytvořena síť statických senzorů zapojených do uzlů. Každý uzel obsahuje řídící jednotku a několik senzorických zařízení pro sběr fyzikálních veličin. Naměřená data jsou periodicky odesílána přes bránu do systému ke zpracování. Senzorová sít obvykle obsahuje 15 - 20 uzlů, maximálně však 100 uzlů. Pro každý uzel jsou periodicky po 15 minutách (někdy po 60 minutách) měřeny veličiny z půdy, vzduchu a vody - celkově se jedná o 11 veličin. Tabulka 4.1 ukazuje, kolik observací je potřeba vygenerovat pro zpracování všech dat z pole za daný časový interval.

	Maximálně	Průměrně
Počet senzorů	11	11
Počet uzlů	100	20
Frekvence [min]	15	15

Tabulka 4.1: Přepokládané parametry pro senzory v poli.

4.1.2 Flotila vozidel

Majitel zemědělských vozidel (traktor, kombajn atd.) by rád sbíral data o pozici vozidel, aktuální zátěži a statistických informacích poskytnuté vozidlem (průměrná spotřeba paliva, ujetá vzdálenost) v reálném čase. Každé vozidlo je již z továrny vybaveno potřebnými senzory a jednotkami, je ale možné tento stav dále rozšířit. Připojení je zajištěno základní sběrnicí (například CAN bus), ze které jsou data čtena, některé ale komunikují přes přípojný uzel ve vozidle. V tabulce 4.2 jsou předpokládané parametry pro flotilu.

	Maximálně	Průměrně
Počet senzorů	30	15
Počet vozidel	1 000	500
Frekvence [s]	1	30

Tabulka 4.2: Předpokládané parametry flotily aut.

Jelikož se jedná o zemědělská pracovní vozidla, které obsluhuje řidič v pracovní době, počet záznamů, které vozidlo za den může vygenerovat, je cca 420. Jeden záznam obsahuje hodnoty z jednoho měření a vztahuje se vždy k jednomu vozidlu. Systém SensLog ovšem není schopný přijmout záznam s více hodnotami v jednom požadavku (viz sekce *Veřejné rozhraní (API)*), je tedy nutné záznam z vozidla rozdělit do více dotazů dle typu senzoru.

Přepočtem lze získat teoretický odhad, jakou zátěž by flotila vozidel představovala pro systém SensLog. Jednalo by se ovšem o případ, když by se všechna vozidla sesynchronizovala a poslání záznamů by proběhlo ve stejné sekundě. Jaká je pravděpodobnost této situace lze ověřit až v praxi.

4.2 Měření

Úvod do analýzy v kapitole 2 obsahuje ukázky katastrofických scénářů, které vedly ke zhroucení státních systémů. V obou případech se jednalo o zátěž způsobenou souběžným přístupem větším množstvím uživatelů. Aby se podobná katastrofa nestala i u zákazníka systému SensLog, jedním z cílů měření je ověřit, jak systém reaguje na zátěž, budou-li na něj odesílána data z většího množství jednotek. Sledovaným parametrem je nejen vytížení hardwaru, ale také, zdali dojde, vlivem souběžnému přístupu jednotek, k odepření přístupu nějaké služby.

Dalším cílem je zjistit, jaká je maximální propustnost. Nebo-li ověřit chování systému při nadměrné zátěži z malého množství jednotek. Za příklad lze zvolit menší flotilu jedoucích vozidel osazenou větším počtem senzorů,

u kterých jsou sledovanými parametry stabilita systému a garance maximální odezvy.

V případě senzorů umístěných v poli je jednoduché vypočítat očekávanou zátěž, protože konfigurace na jednotlivých jednotkách je statická a data ze senzorů jsou získávána v pravidelných intervalech. Opačná situace nastává při jezdících vozidlech, kdy nelze odhadnout, kolik jednotek bude v daném čase aktivních a jakou rychlostí se budou pohybovat. I to je jeden z cílů zjistit, jak na danou situaci bude SensLog reagovat.

4.2.1 Testovací prostředí

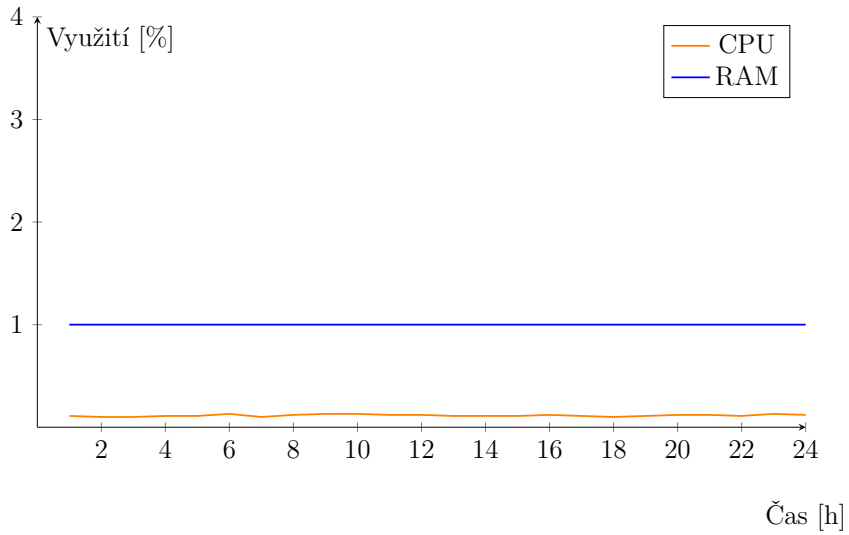
K měření byly využity laboratorní počítače poskytnuté univerzitou. V tabulce 4.3 je kompletní specifikace serveru použitá pro serverovou aplikaci. Server nabízí dostatek výpočetního výkonu (CPU a RAM) a podobná konfigurace se může objevit i u zákazníka. Problém by mohl nastat u použitého pevného disku, protože se jedná o typ vhodný spíše pro domácí použití a do serverového prostředí se nehodí. Počítač použitý pro generování zátěže je stejný jako v případě použitého serveru a propojení je zajištěno 1 Gbps switchem. Toto zapojení má přinést minimalizaci latence mezi stroji, aby nedocházelo ke zkreslení výsledků. Aplikace i databáze běží na jednom stroji a jedná se o stav v jakém je systém provozován u zákazníků.

CPU	Intel Xeon E3-1246 3.5 GHz
RAM	16GB
OS	Debian 10 (buster)
Úložiště	WDC WD10EZEX-08M (WD Blue 1TB)
Síťová karta	1 Gbps

Tabulka 4.3: Parametry testovacích strojů.

4.2.2 Měření klidového stavu serveru

Pro získání co nejpřesnějších výsledků měření je zapotřebí omezit běžící služby na minimum, aby nedocházelo k vytěžování stroje jinými službami. Úplné vypnutí je potřeba provést u služeb, které nemají periodické chování, ale jsou závislé na vnějším vstupu a nelze odhadnout, kolik výpočetního času by bylo potřeba pro jejich zpracování. K získání informací o vytížení stroje, nebo-li šumu na pozadí, v klidovém stavu, byl použit nástroj *top*. Test měření šumu proběhl po dobu 24 hodin a na grafu 1. je vykreslen průběh CPU a RAM.



Graf 1: Průběh využití zdrojů stroje po dobu 24 hodin.

V tomto případě se jedná o izolované prostředí a z naměřených hodnot je patrné, že na stroji neběží žádná aplikace, která by značným způsobem využívala zdroje serveru. Průměrná hodnota využití **CPU** je **0.11 %** a využití **RAM** je **1.00 %** z celkového výkonu. Spuštěny jsou pouze služby, které jsou nezbytné pro běh operačního systému, vzdálené správy a vzdáleného připojení. Na stroji se nenachází žádná aplikace, která by mohla být spuštěna během testů a ovlivnit výsledky.

4.2.3 Příprava testovacích dat

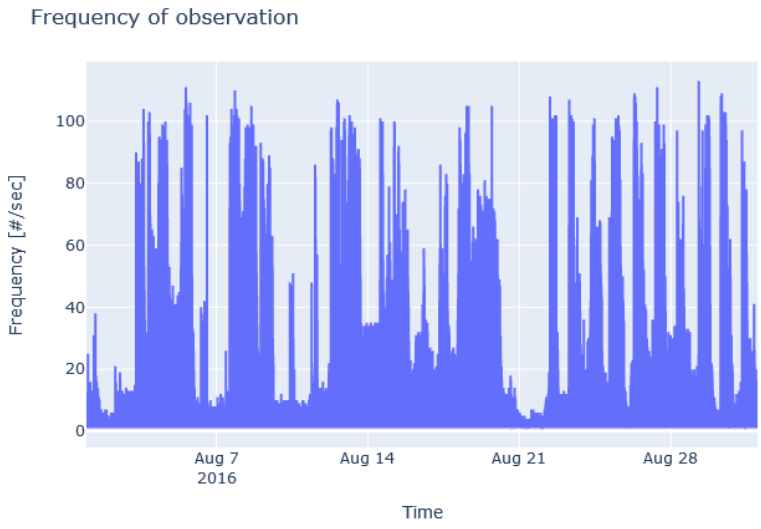
Pro přípravu dat jsou využity charakteristiky ze sekce 4.1. Jelikož data musí obsahovat nejen hodnoty ze senzorů, ale také změny pozice, v úvahu připadá pouze jezdící flotila vozidel.

Zdroj dat

Data mohou být uměle vytvořena (například náhodným generátorem) nebo použita z reálného prostředí. V tomto případě jsou k dispozici reálná data, která lze použít, a lze na nich ověřit, jak moc se shodují s teoretickými daty uvedené v sekci *Charakteristika dat*. Z dat jsou extrahovány vlastnosti, které slouží pro generování syntetických dat využité pro testování.

Zdrojem dat je testovací provoz 6 traktorů, ze kterých byly po dobu jednoho měsíce sbírána data o poloze a z nainstalovaných senzorů. Počet

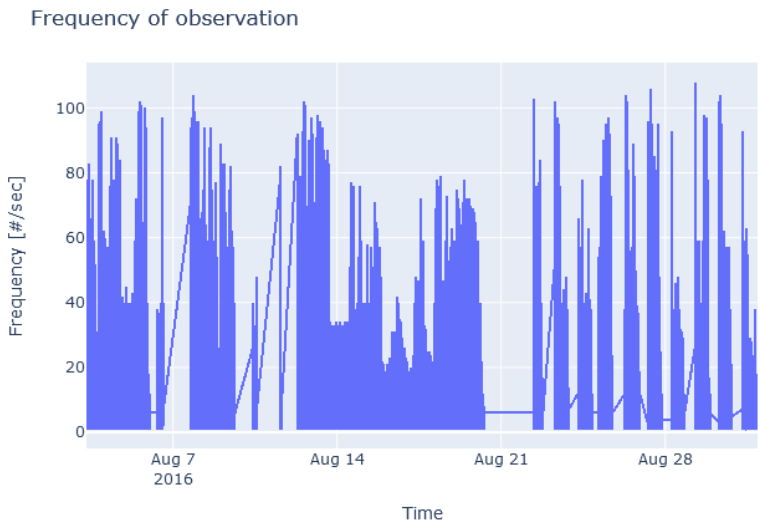
aktivních senzorů byl 21, 21, 20, 5, 5 a 7. Na obrázku 4.1 je znázorněna vizualizace přijatých observací ze všech traktorů. Celkově bylo přijato systémem přes 13.1 milionů záznamů.



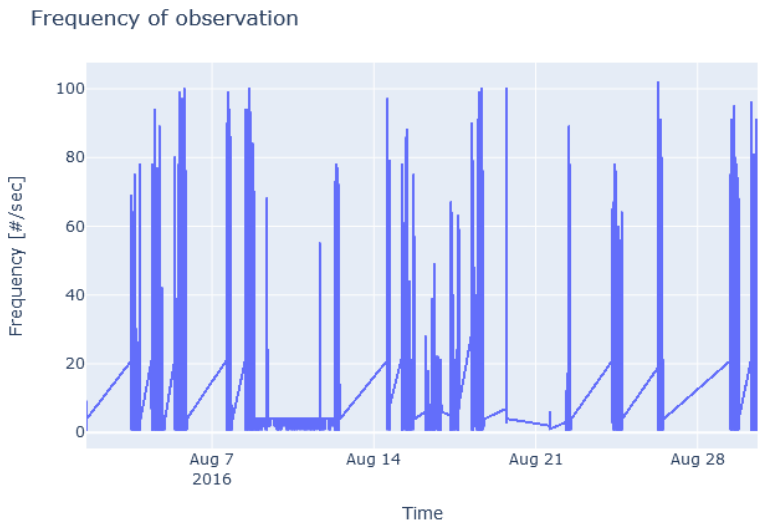
Obrázek 4.1: Počet přijatých záznamů za sekundu ze 6 traktorů.

Z grafu lze vyčíst, že přijaté observace nejsou konstantní v čase, ale v určitý časový okamžik je intenzita posílání zpráv větší. Tento nárůst je způsobený pracovní dobou strojů, kdy při pohybu je interval kratší, tedy za stejnou jednotku je odesláno více zpráv, než když se traktor nepohybuje.

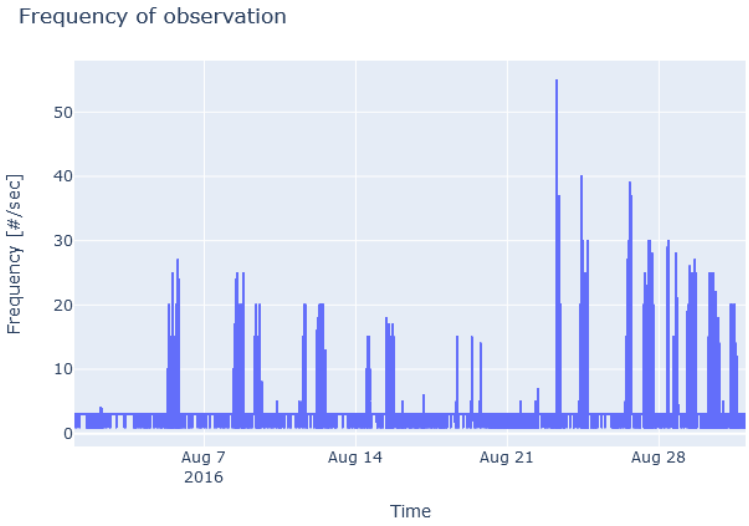
Na následujících grafech jsou zobrazeny frekvence observací u vybraných traktorů, které vykazují rozdílné chování. To je dané tím, že obsahují jiné počty senzorů s rozdílnou konfigurací. Zde stojí za povšivnutí, že různé traktory zobrazené na grafech 4.2 a 4.3 vykazují aktivitu pouze v pracovní době a vizualizace traktoru na grafu 4.4 odesílá data neustále. Taktéž se liší množství odeslaných dat za sekundu v pracovní době.



Obrázek 4.2: Vizualiace více využívaného traktoru s 21 senzory.



Obrázek 4.3: Vizualizace méně využívaného traktoru s 21 senzory.



Obrázek 4.4: Počet přijatých záznamů za sekundu z traktoru s 5 senzory.

Jednoduchým výpočtem (viz rovnice 4.1) lze získat průměrný počet senzorů instalovaných na traktoru. Porovnáním s teoretickou hodnotou z charakteristiky dat, která činí 15, se jedná o velice reálnou hodnotu.

$$\frac{21+21+20+7+7+5}{6} = \frac{27}{2} = 13.5 \tag{4.1}$$

Tuto hodnotu lze použít pro následný výpočet průměrné hodnoty počtu odeslaných observací za sekundu (req/s) z jednoho senzoru. Dle celkové zátěže ze všech traktorů se vrcholy přijatých observací pohybují nejčastěji v intervalu 80 - 100. Vydělením intervalu průměrným počtem senzorů vyjde 0.988 - 1.235 (rovnice 4.2 a 4.3). Použití této metodiky výpočtu lze vytvořit model, kdy aktivní senzor aktualizuje svůj stav každou sekundu, tj. 1 req/s.

$$\frac{100}{6} * \frac{2}{27} = 1.235 \tag{4.2}$$

$$\frac{80}{6} * \frac{2}{27} = 0.988 \tag{4.3}$$

Ze zdrojových dat lze vytvořit i druhý model, který je zaměřen na analýzu jednotlivých traktorů. Traktory obsahující 21 senzorů dosahují vrcholu 80-100 observací za sekundu. Použije-li se průměrná hodnota 90, vyjde frekvence

odesílání 4.3 req/s pro jeden senzor (rovnice 4.4). Stejným přepočtem pro traktor s 5 senzory (rovnice 4.5), vyjde 5 req/s. Po zaokrouhlení lze říci, že z aktivního senzoru bylo přijato 4.5 observací každou sekundu.

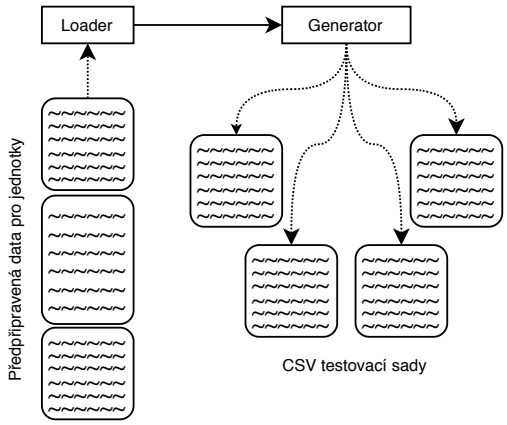
$$\frac{90}{21} = 4.29 \tag{4.4}$$

$$\frac{25}{5} = 5 \tag{4.5}$$

Druhý model ukazuje reálnější situaci konfigurace senzorů na traktoru, ale nereflektuje fakt, že k odesílání dat nedochází ve všech traktorech ve stejný čas. Při nečinosti se hodnota observací pohybovala v rozmezí 2 - 5 req/s.

Zpracování dat

Exportovaná data z databáze (tzv. SQL Dump), která obsahují data z tabulek observace a pozic jednotek, je potřeba nejdříve zpracovat pro následné použití v testovacím nástroji, konkrétně pro JMeter ve formátu CSV. Dalším úkolem je vygenerovat testovací sady pro určitý počet traktorů. Prvním krokem je získaná data oddělit od sebe a sloučit je podle identifikátorů jednotek. Druhým krokem, zobrazený na obrázku 4.5, je samotné generování testovacích dat. V levé části jsou předpřipravená data pro každou jednotku, která se postupně načítají, a generátor generuje testovací sady podle konfigurace. U každého záznamu je změnen identifikátor jednotky tak, aby odpovídal nové jednotce.



Obrázek 4.5: Generování testovacích sad ze zdrojových dat.

4.3 Sady testů

Z vypočítaných odeslaných observací z jednoho senzoru v sekci 4.2 vychází, že hodnota se pohybuje kolem 4.5 req/s. Použije-li se tato hodnota pro zpětné přepočítání zátěže pro 2 traktory s 21 senzory, vyjde 189 req/s. Za předpokladu, že všechny senzory mají nastavený stejný čas odesílání by se jednalo o reálnou hodnotu. Z testovacího provozu je ale patrné, že tento děj nenastal a pro konfigurace testů je vybrán model, kdy senzor odesílá jedno pozorování za sekundu.

Konfigurace virtuálního traktoru obsahuje maximální počet osazených senzorů, což znamená frekvenci odesílání 30 req/s, změna polohy se při jízdě zaznamenává po každém ujetém kilometru - traktor jedoucí 25 km/h odesílá 7 req/s, a v nečinnosti odesílá 3 req/s. Úspěšný dotaz na server je takový, který obsahuje návratovou hodnotu *true*, v ostatních případech se dotaz bere jako neúspěšný a připočítává se k chybovosti systému.

4.3.1 Zahřátí serveru

Pro každý test jsou nastaveny stejné počáteční podmínky - stejná konfigurace aplikačního serveru a prázdná databáze. Tento stav není úplně žádoucí, protože se v databázi nenacházejí identifkátory jednotek a senzorů a musí být generovány pokaždé znovu. Tyto operace přinášejí určitou režii a výsledky testů by tímto byly ovlivněny.

Z těchto důvodů je před každou sadou testů provedeno takzvaně zahřátí serveru. Tím dojde k inicializaci potřebných struktur a tabulek v databázi a také připraví aplikační server na zátěž. Jedná se o test, který běží po dobu 5 sekund a obsahuje minimálně jednu observaci a jednu pozici pro každou jednotku/traktor a senzor.

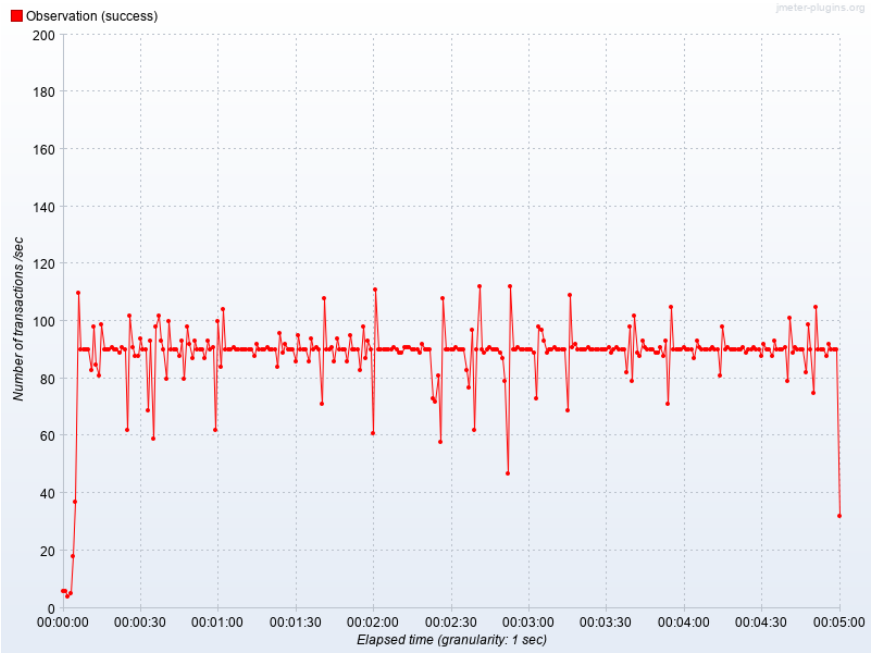
4.3.2 Nízká zátěž observacemi

Základní test simuluje situaci, kdy server má přijmout observace z polí nebo z nepohybujících se traktorů, které odesílají data mimo jejich pracovní dobu. Smysl testu je vizualizovat chování systému při nízké zátěži, sledovat vytížení procesoru, alokaci paměti během testu a následně správné uvolnění. Nebo-li ukázat, zda je systém schopen uvolnit veškeré alokované prostředky a kolik času je k tomu potřeba. Konfigurace testu je v tabulce 4.4.

Počet traktorů	30
Doba testu	5 minut
Celkem observací	90 req/s

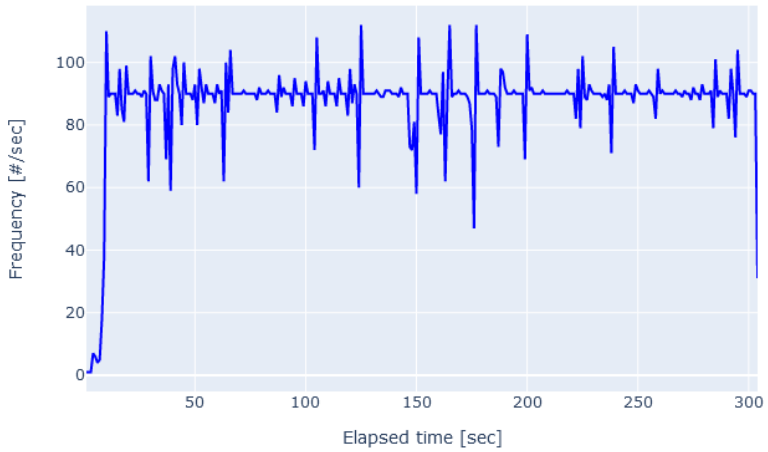
Tabulka 4.4: Konfigurace testu pro nízkou zátěž.

Na obrázku 4.6 je zobrazen průběh odesílání observací z testovacího stroje. Množství odeslaných zpráv osciluje kolem hodnoty 90, tuto zátěž tedy není problém přijmout a zpracovat.



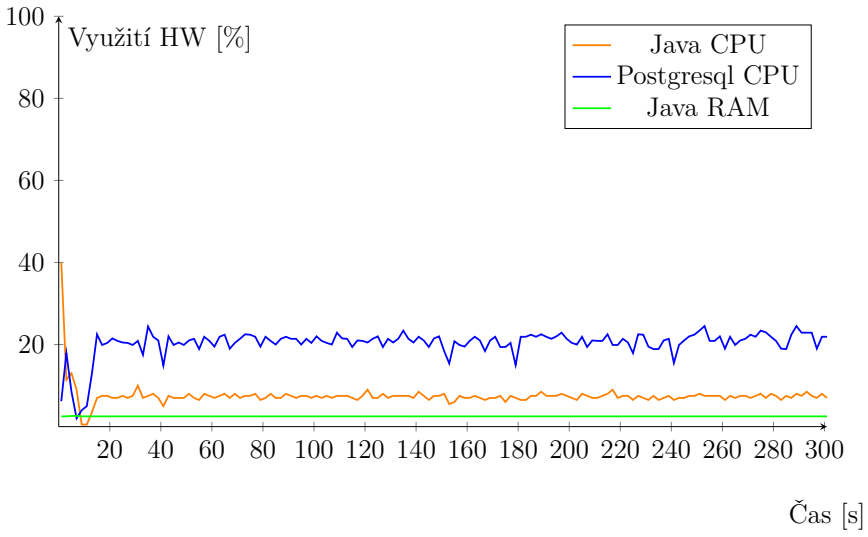
Obrázek 4.6: Odeslané observace při nízké zátěži.

Graf na obrázku 4.7 naopak ukazuje přijaté observace z pohledu databáze. Množství přijatých zpráv se skutečně pohybuje kolem 90 observací za sekundu a odpovídá množství jich odeslaných.



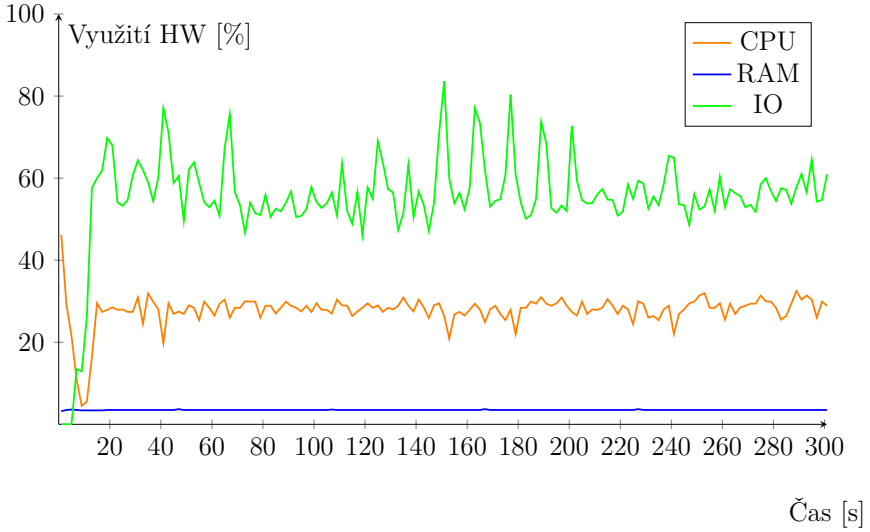
Obrázek 4.7: Přijaté observace při nízké zátěži.

Při pohledu na využití zátěže na grafu 2 je patrné, že pro server není problém zpracovat přijaté množství zpráv. Celkové využití stroje je menší než 40%, což dává velké rezervy pro větší zátěž.



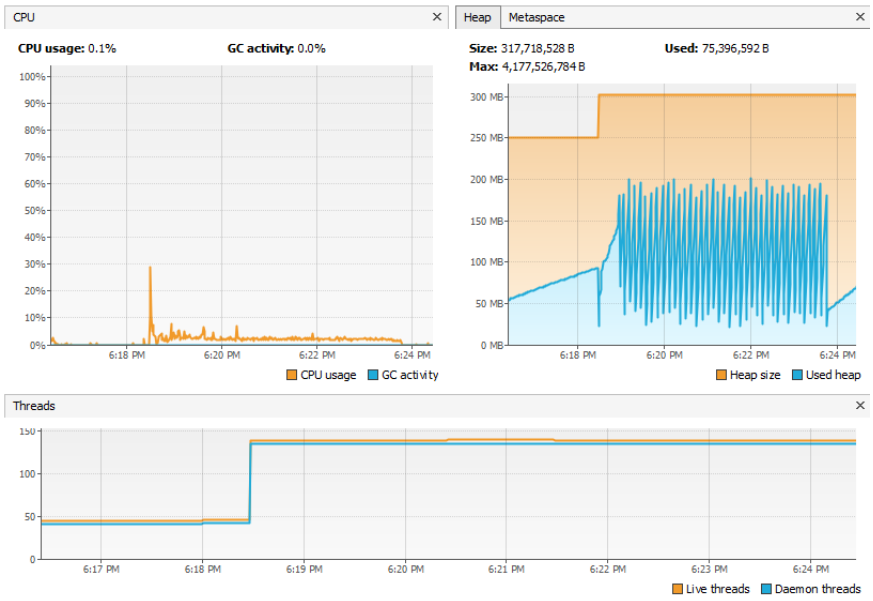
Graf 2: Využití CPU a RAM aplikace a databáze při nízké zátěži.

Jiná situace ovšem nastává při zápisu na disk. Na grafu 3 je zobrazený souhrn využití stroje a procentuální využití disku databází se blíží k vyšším číslům a mohl by představovat problém při větší zátěži. Překvapujícím zjištěním je využití paměti RAM, která se drží na konstantní hodnotě. Toto chování je způsobené tím, že aplikace si před spuštěním alokují paměť na haldě, a tu následně při svém běhu využívají.



Graf 3: Celkové využití serveru při nízké zátěži.

Profilováním aplikace se ukázalo, že neobsahuje chyby, které jsou popisované v kapitole *Únik paměti*. Na obrázku 4.8 je zobrazen (z leva do prava) průběh využití procesoru, využití paměti a počet využitých vláken. Důležitým ukazatelem z těhle grafů je práce s pamětí. Je zde vidět, že postupně během testu docházelo k uvolňování a po skončení testu se uvolnila všechna paměť. Zde je také vidět alokovaná paměť na haldě (oranžově označená, *Heap size*), která si drží konstatní velikost a pro reálné využití (modře označené, *Used heap*) se využívá již alokovaná paměť.

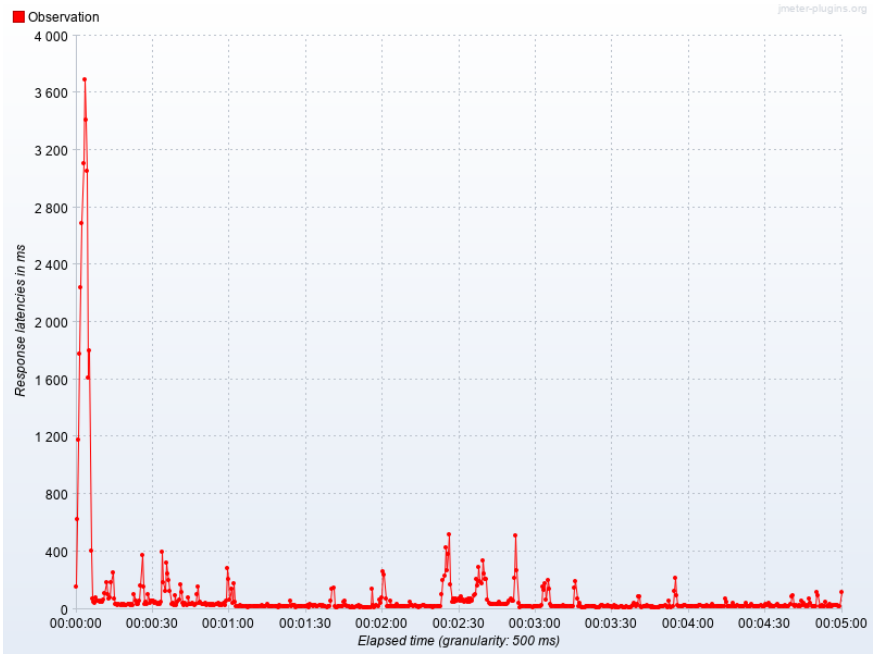


Obrázek 4.8: Profilování při nízké zátěži.

V tabulce 4.5 jsou uvedeny souhrnné výsledky testu. Hodnota propustnosti potvrzuje, že test byl splněn. Alarmující je maximální latence požadavku, zatímco hodnoty mediánu (Me) a průměru (symbol Ø) jsou nízké. Skok v latenci ovlivnil i směrodatnou odchylku (SD), která dosahuje hodnoty 3x vyšší než je průměrná hodnota. Vše vysvětluje graf na obrázku 4.9, kde jsou zobrazeny latence v průběhu testu. Při detailnějším zkoumání původu observací bylo zjištěno, že se jedná o první řádek každé z testovacích sad.

	min.	max.	Ø	Me	SD	propustnost
Observace	4 ms	5782 ms	52 ms	21 ms	163 ms	88 req/s

Tabulka 4.5: Souhrn výsledků pro test nízké zátěže.



Obrázek 4.9: Latence požadavků při nízké zátěži.

Na tomto základním testu bylo ověřeno, že se v aplikaci nenachází žádná chyba, která by ovlivlina provoz při nízké zátěži a dlouhodobém použití. Vytížení procesoru se pohybuje v nízkých procentech z celkového výkonu, což platí také pro využití paměti RAM. Jediné riziko zde představuje delší čekání na I/O operace, které by mohlo celkovou rychlost velice ovlivnit.

4.3.3 Průměrná zátěž observacemi

Průměrná zátěž simuluje běžný stav provozu traktorů. Test je složen z pohybujících se traktorů, které odesílají průměrný počet observací. V závislosti na čase se počet traktorů zvyšuje o 10 při každé následující minutě.. Rychlost všech jednotek zůstává neměnná. Cílem tohoto testu je zjistit, jaký je maximální počet traktorů, které mohou pracovat v jeden okamžik při dané konfiguraci serveru. V tabulce 4.6 je uvedena kompletní konfigurace testu.

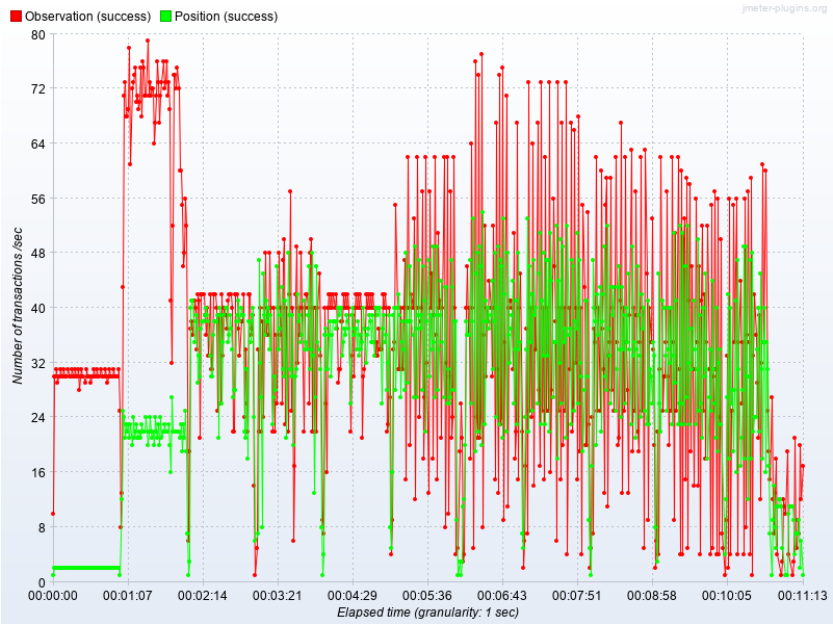
Počet traktorů	1 - 100
Rychlost 1 traktoru	2 m/s
Počet observací 1 traktoru	30 req/s
Doba testu	11 minut
Celkem observací	30(1) - 3 000(100) req/s
Celkem pozic	2(1) - 200(100) req/s

Tabulka 4.6: Konfigurace testu pro průměrnou zátěž.

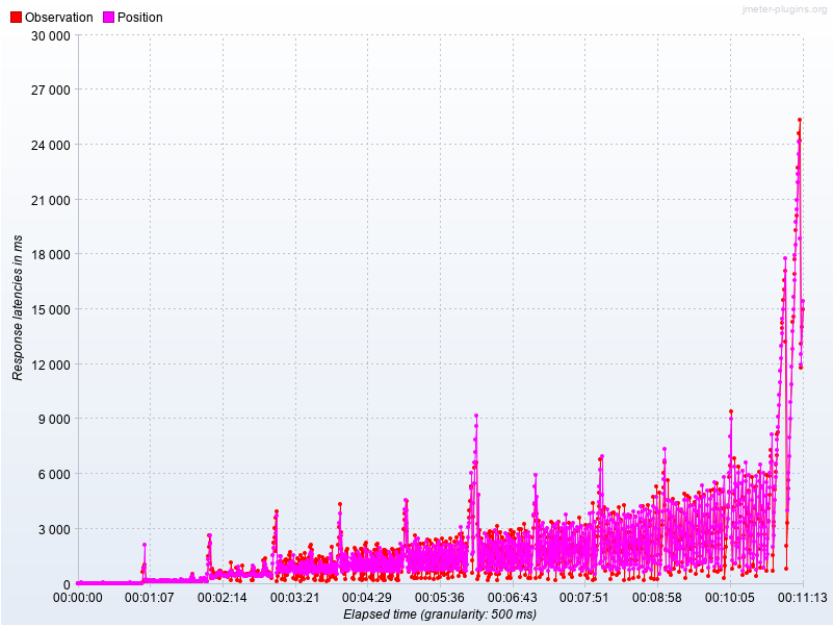
Průběh tohoto testu vykazuje velice nekonzistentní průběh. Již při pohledu na vrchol odeslaných zpráv na obrázku 4.10 je jasné, že server takové množství přijmout nedokáže. V grafu je průběh odeslaných observací označený červenou barvou a zelenou odeslané pozice. Po první minutě, kdy dojde ke zvyšení jednotek na 10, dochází k tomu, že server přijme změny pozic ze všech traktorů, ale již nedokáže zpracovat odeslané observace.

Zde stojí za zmínění, že nastavená zátěž je maximální, kterou se nástroj JMeter snaží udržet. Požadavky pro jednotlivé jednotky jsou odesílány sériově, což znamená, že bude-li na zpracování požadavku potřeba 1 sekunda, v dané sekundě bude přijata pouze jedna observace pro danou jednotku, i když propustnost je nastavena na vyšší hodnotu. Aby byl požadavek považován za chybný, muselo by dojít k odeslání na server a následně k odmítnutí.

Propady výkonu při příjmu změny pozice dochází při 30 traktorech. V tomto čase dochází i k dalšímu propadu přijímaných observací. Pro vysvětlení chování lze použít graf na obrázku 4.11, kde se odezva v daném čase pohybuje kolem 1 000 ms. Dále se latence při nárustu zátěže neustále zvětšuje a dochází k velké nestabilitě systému, kdy se v určitém bodě povede odeslat observace jen některým jednotkám.

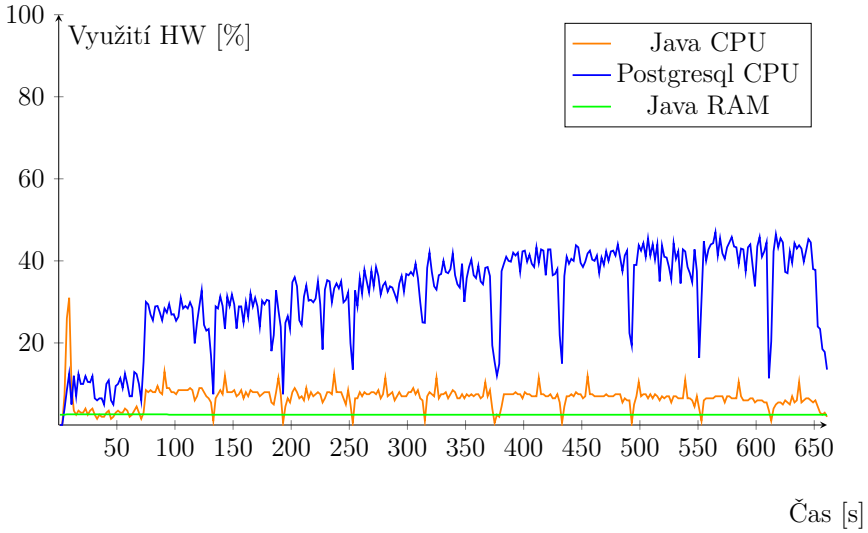


Obrázek 4.10: Odeslané observace při průměrné zátěži.

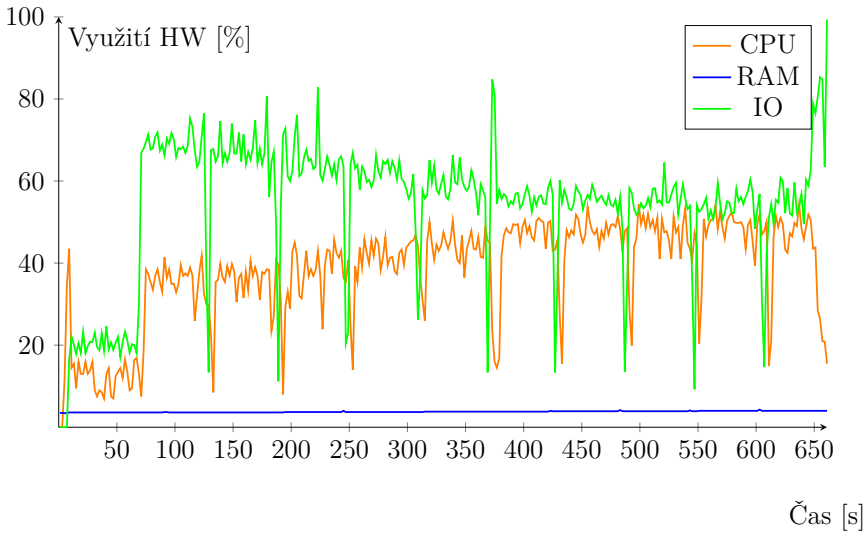


Obrázek 4.11: Latence požadavků při průměrné zátěži.

Zvyšující se odezva systému by mohla znamenat, přetížení serveru a využití všech jeho zdrojů (CPU a RAM). Graf 4 ovšem dokazuje, že tomu tak není. Dokonce ani využití disku nenasvědčuje, že by zde bylo úzké hrdlo (viz graf 5). Se zvyšující se zátěží se zvyšuje i využití procesorového času databáze, ale aplikace využívá stále stejné procento výkonu. Zde je potřeba připomenout, že aplikace neobsahuje žádné matematické výpočty, ke kterým by byl procesor potřebný. Největší zátěž představuje příjem HTTP požadavků, který je limitován počtem vláken dané konfigurací Apache Tomcat.



Graf 4: Využití CPU a RAM aplikace a databáze při průměrné zátěži.



Graf 5: Celkové využití serveru při průměrné zátěži.

Tabulka 4.7 rekapituluje naměřené výsledky, které jsou zobrazeny v předchozích grafech. Celková propustnost je pouze 35.3 observací za sekundu, která odpovídá pouze jednomu pohybujícím se traktoru. Tento test má celkovou propustnost menší než předchozí test *Nízká zátěž observacemi* i přesto, že vytížení serverové aplikace je podobné. Rozdílné chování lze pozorovat u vytížení databáze, především zvýšení CPU zátěže. Patrné je také rozdíl v přístupu na disk, kde dochází k častějším propadům. Jelikož je tento test rozšířen o změny pozice jednotky, dochází na straně databáze ke spouštění více funkcí, které v sobě obsahují další SQL dotazy pro načítání dat.

	min.	max.	Ø	Me	SD	propustnost
Observace	4 ms	26299 ms	1451 ms	391 ms	2109 ms	35.3 req/s
Pozice	13 ms	26415 ms	1757 ms	1284 ms	1808 ms	28.2 req/s

Tabulka 4.7: Souhrn výsledků pro test průměrné zátěže.

Tento test ukázal, že změna pozice jednotky zásadně ovlivní propustnost při zpracování observací. Při vzrůstajícím počtu jednotek ovšem nedochází k lineárném propadům propustnosti. Toto chování může způsobovat aplikační server nebo skrytá chyba v serverové aplikaci.

4.3.4 Vysoká zátěž observacemi

Na předchozím testu 4.3.3 je patrné, že se v systému může vyskytovat chyba v podobě konfigurace aplikačního serveru nebo v aplikaci. Sečtením počtu požadavků z předchozího testu dokázal systém obsloužit maximálně 100 požadavků za sekundu. Cílem tohoto testu je zjistit, zda konfigurace aplikačního serveru, konkrétně množství vláken obsluhující HTTP požadavky, mají vliv na celkovou propustnost.

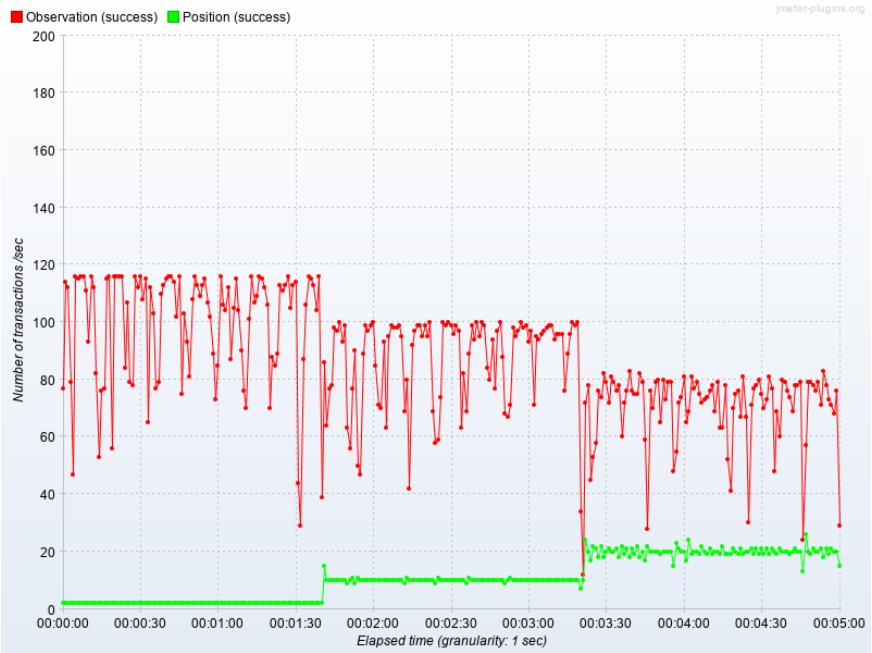
Konfigurace testu je nastavena tak, aby nebyl vyčerpán maximální počet obsluhujících vláken aplikačního serveru. Test začíná na 1 traktoru a postupně je počet navýšen každých 100 sekund až na 10 (1, 5, 10) a maximální propustnost je nastavena na 150 observací za sekundu pro jeden traktor (viz tabulka 4.8). Jedná se o větší zátěž z jednoho traktoru, než by teoreticky měl být server schopný zpracovat.

Očekávané chování je, že jedno vlákno bude obsluhovavat jeden traktor, což znamená, že při zvyšujícím se počtu traktorů by propustnost neměla klesnout a server by měl požadavky zpracovávat paralelně.

Počet traktorů	1 - 10
Rychlost 1 traktoru	2 m/s
Počet observací 1 traktoru	150 req/s
Doba testu	5 minut
Celkem observací	150(1) - 1 500(10) req/s
Celkem pozic	2(1) - 20(10) req/s

Tabulka 4.8: Konfigurace testu pro vysokou zátěž.

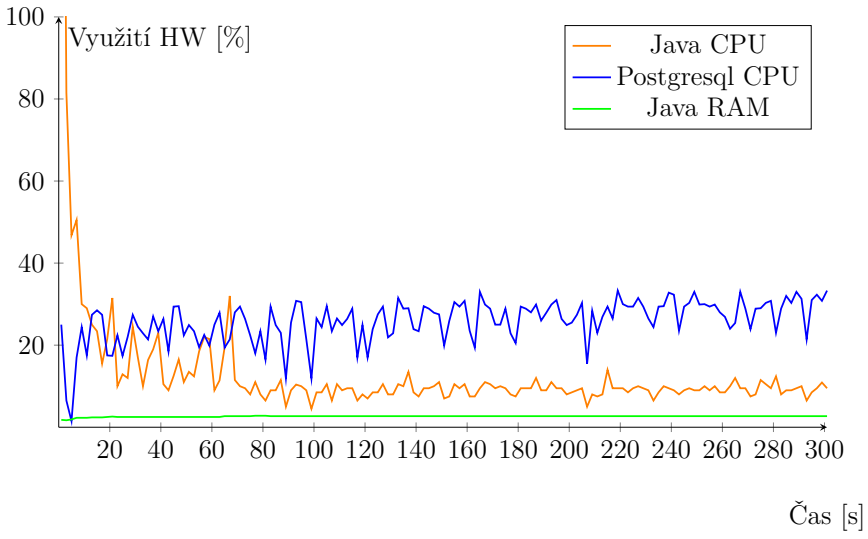
Předpoklad, že server nebude schopný obsloužit maximální zátěž jednoho traktoru, se potvrdil. Maximální zátěž nepřesáhne hranici 120 požadavků za sekundu a při zvýšení počtu traktorů na 5 dojde ke znížení propustnosti. Při následném zvýšení počtu dojde k dalšímu poklesu (viz obrázek 4.12).



Obrázek 4.12: Odeslané observace při vysoké zátěži.

Průběh přijímání požadavků také doprovází výrazné kolísání výkonu. Pohlédnutím na graf 7 dojde i k vyvrácení teze, že se tak děje z důvodu přetížení serveru. Průběh využití procesoru aplikací Java je při vytížení jedním traktorem vyšší, ovšem následně při zvýšení zátěže se využití CPU zníží a osciluje kolem stejné hodnoty. Podobná situace je i s databází, kdy dochází k propadům, ovšem přímá úměra se zvýšenou zátěží zde také vidět není.

Alarmující je čekání na disk na grafu 7. Porovnáním s předchozími testy dochází k velice podobnému průběhu, přestože konfigurace pro test je odlišná.



Graf 6: Využití CPU a RAM aplikace a databáze při vysoké zátěži.

Graf 7: Celkové využití serveru při vysoké zátěži.

	min.	max.	Ø	Me	SD	propustnost
Observace	4 ms	1620 ms	61 ms	41 ms	77 ms	86 req/s
Pozice	16 ms	1531 ms	128 ms	97 ms	120 ms	10.6 req/s

Tabulka 4.9: Souhrn výsledků pro test vysoké zátěže.

Výsledky testu odhalily, že úzké hrdlo se nenachází v konfiguraci aplikačního serveru, ale aplikaci samotné. Důkazem je snížení počtu přijatých požadavků při zvýšení zátěže, aniž by se takové chování projevilo na výkonu serveru. V tabulce 4.9 jsou shrnuté výsledky testu, které se v případě přijatých observací moc neliší s porovnáním se základním testem.

4.3.5 Vzrůstající rychlost traktorů

Odesílání aktuální polohy je důležité při autonomním pohybu traktorů. Ty se na poli pohybují různými rychlostmi a na základě jejich polohy je možné provádět nějaké operace. Konfigurace testu (tabulka 4.10) je sestavena z 10 zrychlujících traktorů, které odesílají průměrný počet observací a jejich rychlost se zvyšuje o 1 m/s každou minutu až do maximální rychlosti 10 m/s (36 km/h). Z předchozích testů je zřejmé, že celkovou maximální zátěž 300 observací nelze dosáhnout, tudíž lze říct, že celková zátěž observacemi bude maximální, kterou server dokáže zpracovat. Cílem testu je zjistit, zda zrychlující traktory mají vliv na propustnost systému observacemi a jak takový průběh bude vypadat.

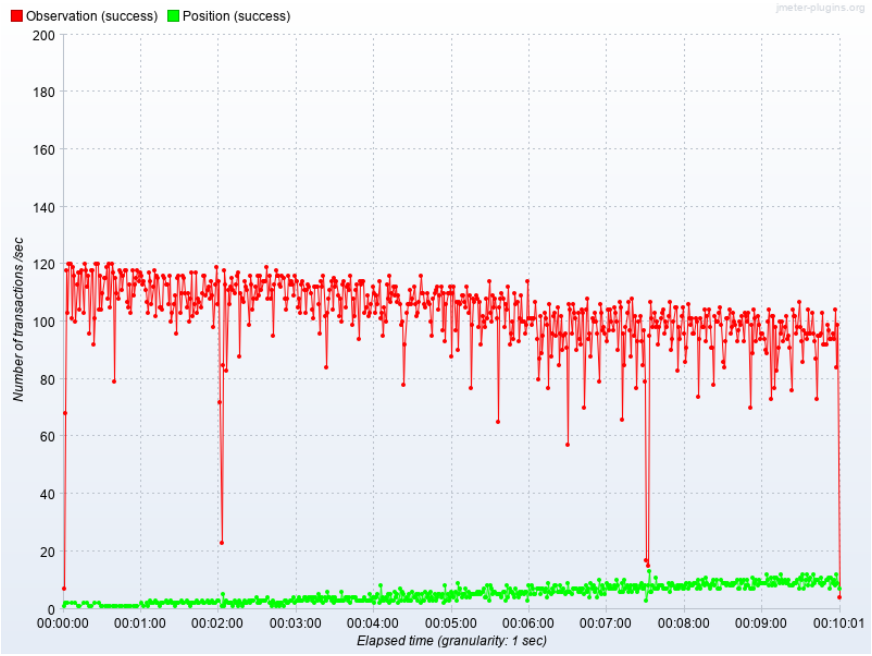
Počet traktorů	10
Rychlost 1 traktoru	1 - 10 m/s
Počet observací 1 traktoru	30 req/s
Doba testu	10 minut
Celkem observací	300 req/s
Celkem pozic	10(1) - 100(10) req/s

Tabulka 4.10: Konfigurace testu pro vzůstající rychlost traktorů.

Průběh tohoto testu vykazuje velice rozdílé výsledky. I při opakovaném spuštění se výsledek neustálil, jako v případě předchozích testů, ale vždy v polovině případů došlo ke stejnému průběhu jako na grafu 4.13 a v druhé polovině jako na grafu 4.14.

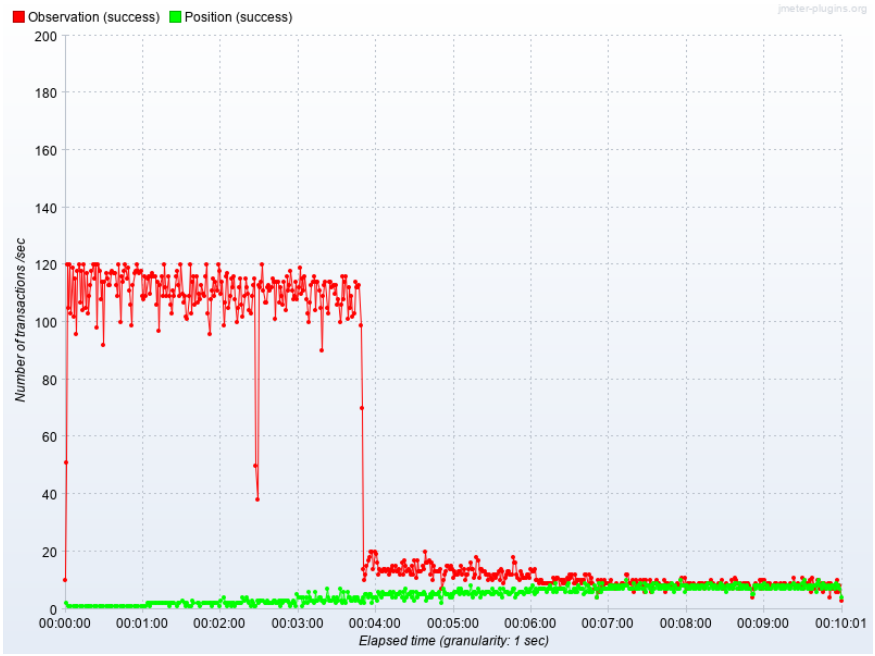
Nedojde-li k propadu, průběh testu je velice podobný jako v případě *Vysoká zátěž observacemi*. Zrychlující traktory ovlivňují, nebo-li snižují maximální propustnost pro observace. Stejná situace nastává i v případě využití

HW, kdy snížená propustnost není viditelná v grafu 8. Pokud by byl závěr složený pouze na základě tohoto testu, nijak by se nelišil od závěru z předchozího testu.



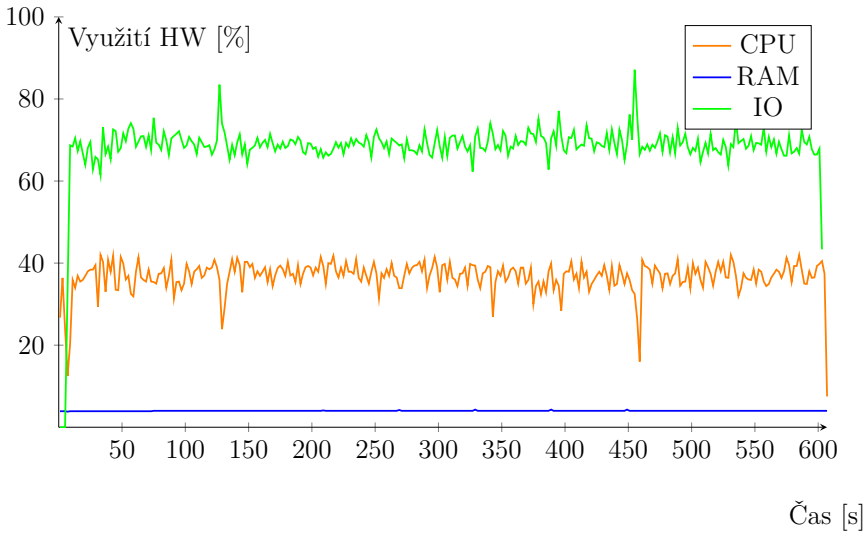
Obrázek 4.13: Odeslané observace se stabilním průběhem.

Druhý průběh na grafu 4.14 je charakterizován výkonnostním propadem. K takto velkému propadu docházelo ve všech případech, kdy se nejednalo o průchod bez propadu. Společné pro všechny testy je, že pokles nastal až po 3 minutě testu, což znamená, že se traktory pohybovaly rychleji než 3 m/s. To vysvětluje, proč se tento jev neobjevil v předchozích testech, kde maximální rychlost byla nastavena na 2 m/s.

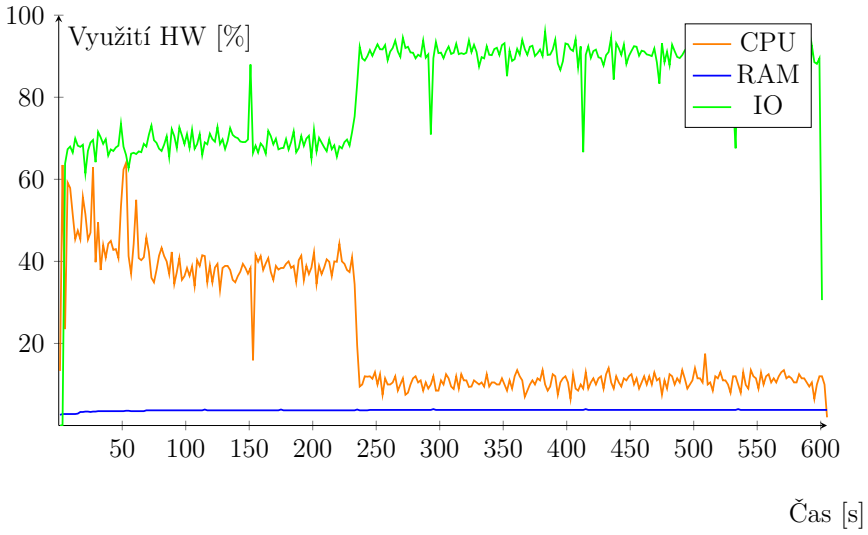


Obrázek 4.14: Odeslané observace s propadem propustnosti.

Při porovnání grafů 8 a 9 je využití zdrojů do 4. minuty testu velice podobné. Zlom nastane až po propadu, kdy databáze velice sníží svoji aktivitu, ale výrazně se zvýší vytížení I/O operacemi při práci s diskem.



Graf 8: Využití serveru při vzrůstající rychlosti se stabilním průběhem.



Graf 9: Využití serveru při vzrůstající rychlosti s propadem propustnosti.

	min.	max.	Ø	Me	SD	propustnost
Observace	16 ms	1588 ms	96 ms	91 ms	54 ms	103 req/s
Pozice	16 ms	1147 ms	114 ms	107 ms	70 ms	5 req/s

Tabulka 4.11: Souhrn výsledků pro test se stabilním průběhem.

Tento test odhalil nekonzistentní chování systému, je-li rychlost traktorů zvyšována a dochází tak k častějším změnám polohy. Takové chování je velice nebezpečné v případě, že by systém byl využívaný v autonomní oblasti. Problém ve zpracování požadavků je především kvůli nadměrnému vytěžování diskových operací. Toto chování nadvědčuje tomu, že obsluha pro změnu pozice vyžaduje více než jen jeden SQL příkaz.

4.4 Souhrn zjištění

Během testování bylo zjištěno, že systém v aktuální verzi dokáže zpracovat pouze pozorování ze senzorů umístěných na poli. Při zvýšené zátěži a využití služby pro změny polohy jednotky dochází k velice nestabilnímu chování nebo vede k dramatickému propadu propusnosti. Dle nastavené konfigurace virtuálního traktoru dokáže systém zpracovat data maximálně ze 3 jezdících jednotek.

Porovnáním se zdrojem dat, kdy server byl schopný přijmout minimálně data ze 6 traktorů, i když ne rovnoměrně vytížených, se může zdát, že na vině je slabá konfigurace serveru. Tato hypotéza může být potvrzena pouze částečně. K maximálnímu vytížení CPU a RAM nedošlo u žádného testu, a dokonce se u všech testů pohybuje na podobné úrovni. Porovnáním se základním testem *Nízká zátěž observacemi* se zněna vytížení HW projevila pouze u testu *Vzrůstající rychlost traktorů*, a to v podobě nadměrného vytěžování I/O operací. Disk je slabým článkem testovací konfigurace a jeho výměna by měla zaručit zvýšení propustnosti.

Mimo uvedené testovací prostředí byly některé testy spuštěny i v distribuovaném prostředí na virtuálním počítači umístěném v serverovně fakulty a výsledky byly až 4x lepší. Distribuované prostředí bylo velice nestabilní a závislé na zatížení univerzitní sítě, tudíž nebylo možné testy replikovat se stejnými výsledky a nemohly být použity v této práci.

V naměřených grafech lze pozorovat určité vzory, které se vyznačují poklesem propustnosti při zvýšeném počtu jezdících traktorů. Za předpokladu, že se požadavky na serveru zpracovávají paralelně, k tomuto poklesu by dojít nemělo.

5 Analýza příčin a úprava systému

Z naměřených výsledků je patrné, že systém obsahuje několik problémů, které systém zpomalují. V této kapitole jsou podrobně analyzovány jejich příčiny a provedeny úpravy pro jejich odstranění. Následující úpravy jsou spíše servisního charakteru, protože tato verze se již nadále nevyvíjí a nalezené příčiny budou využity jako antivzory, kterým se vyvarovat v implementaci SensLog 2.0.

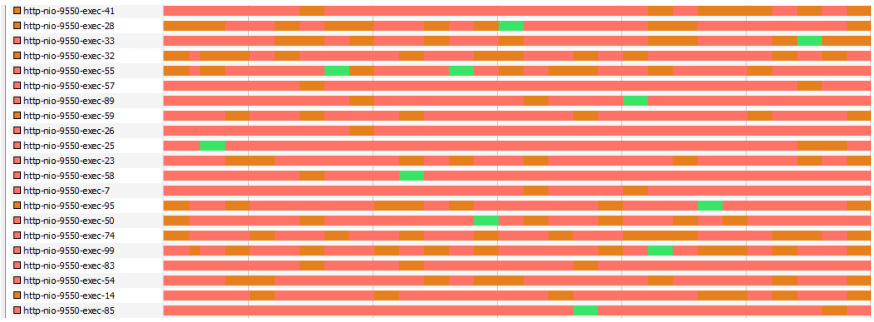
5.1 Řešení na stávající verzi SensLog

Jednotlivé návrhy popsané níže vycházejí z naměřených testů ze sekce 4.2. Každá příčina je detailně popsána a je navrhnuto řešení, které problém ve stávající verzi SensLogu odstraní.

5.1.1 Kritická sekce

Na naměřených grafech v testech *Průměrná zátěž observacemi* a *Vysoká zátěž observacemi* se nachází podivné chování při narůstajícím počtu jednotek. Skutečný problém, proč dochází ke snižování propustnosti systému, odhalí až profilování serverové části.

Obrázek 5.1 zobrazuje výstup z programu *VisualVM*, kde jsou zachyceny průběhy aktivity vláken zpracovávající HTTP požadavky. Vlákna se zde střídají ve třech stavech - pozastavené (oranžová barva), čekající na podmínkové proměnné (červená barva) a běžící (zelená barva). Produktivní čas je pouze v běžícím stavu. Z průběhu lze vyčíst, že se vlákna nacházejí v aktivním režimu po velice krátkou dobu a žádné z nich nemá aktivní činnost paralelně s jiným vláknem. Toto chování nasvědčuje tomu, že v kódu se nachází kritická sekce, nad kterou se jednotlivá vlákna sychronizují.



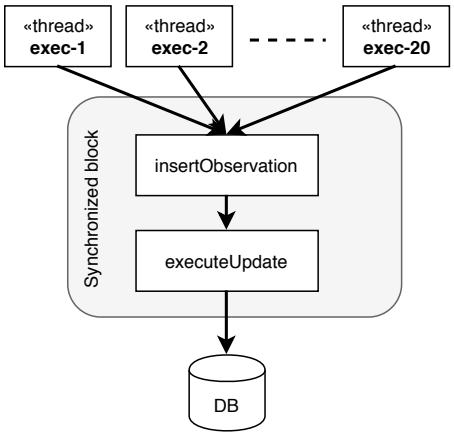
Obrázek 5.1: Aktivita vláken serveru při zátěži.

Synchronizovaný kód se v aplikaci vyskytuje na několika místech, ale nejpodstatnější je výskyt v databázové vrstvě. V ukázce 5.1 je výňatek z kódu, který vykonává spuštění připraveného SQL dotazu. Tato metoda je využívána pro všechny operace, které vkládají nebo upravují záznamy v databázi.

```
1  public static synchronized int executeUpdate(String sql) {
2      PooledConnection con = mycp.getPooledConnection();
3      try {
4          Statement st = con.createStatement();
5          int rs = st.executeUpdate(sql);
6          con.release();
7          return rs;
8      } catch (SQLException e) {
9          mycp.removeConnection(con);
10         throw new SQLException(e);
11     }
12 }
```

Listing 5.1: Synchronizovaná část kódu.

Při testování docházelo k tomu, že operaci pro vložení záznamu mohlo provést pouze jedno vlákno a ostatní vlákna přijímající HTTP požadavky musela čekat. Na diagramu 5.2 je toto chování znázorněné.

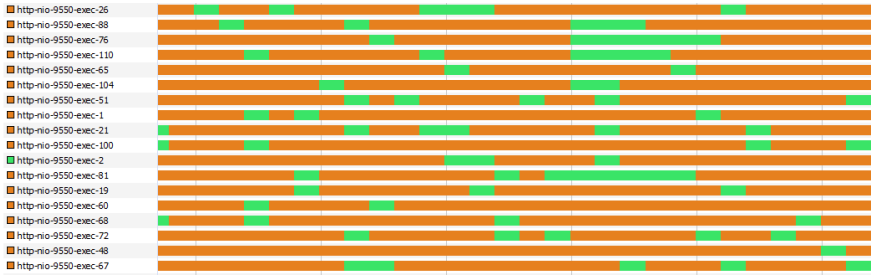


Obrázek 5.2: Diagram synchronizace vláken při ukládání do databáze.

PostgreSQL dodržuje ACID (atomicita, konzistentnost, izolovanost, trvalost) vlastnosti [27], kterých lze využít při paralelním zpracování. Každé vlákno může pracovat s vlastním připojením do databáze a databázový systém zařídí, aby byly ACID vlastnosti dodrženy. Při využití těchto vlastností není nutné provádět synchronizaci na úrovni vláken v aplikaci a posílat dotazy do databázi sériově.

Řešením problému je odstranění jazykové konstrukce *synchronized* v hlavičce metod, které pracují s databází. Po odstranění této konstrukce je nutné zajistit vláknově bezpečné připojení. Komunikace mezi aplikací a databází zajišťuje *PostgreSQL JDBC Driver*, který vláknově bezpečný není [28] a nelze jedno připojení sdílet mezi více vlákny. Je tedy nutné zajistit, aby bylo obsluhováno pouze jedním vláknem. Vytváření připojení znamená nějakou komunikační režii v aplikaci a je vhodné použít seznam uložených/-předpřipravených připojení pro znovupoužití, které jej budou efektivně přidělovat vláknům (viz sekce *Connection pool*).

Opravení této chyby přineslo delší doby běhu jednotlivým vláknům a paralelní zpracování (viz obrázek 5.3).



Obrázek 5.3: Aktivita vláken serveru po odstranění synchronized.

5.1.2 Connection pool

Pouze odstranění chyby se synchronizací vláken nepomohlo ke zvýšení propustnosti. Synchronizace přikryla další chybu, která se nachází taktéž v databázové vrstvě. Při vzorkování procesoru se mezi vytíženými vlákny objevilo nové s názvem *Thread-4* (viz obrázek 5.4). Při pohledu na třídu je jasné, že tato chyba je obsažena ve třídě spravující připojení do databáze a obsahuje metodu **Thread.sleep()**.

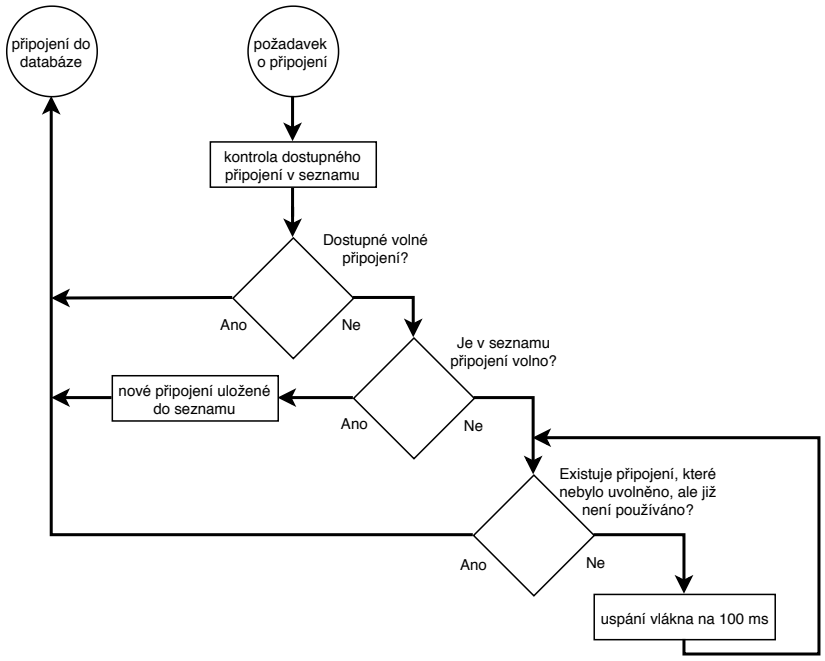
Name	Total Time	Total Time (CPU)
main	487,461 ms (100%)	0.0 ms (-%)
AsyncFileHandlerWriter-1531448569	487,461 ms (100%)	0.0 ms (-%)
http-nio-9550-BlockPoller	487,461 ms (100%)	487,461 ms (100%)
ajp-nio-8009-BlockPoller	487,461 ms (100%)	487,461 ms (100%)
Thread-4	487,461 ms (100%)	0.0 ms (-%)
cz.hsrs.db.pool.ConnectionPool$ConnectionReaper.**run** ()	487,461 ms (100%)	0.0 ms (-%)
java.lang.Thread.**sleep[native]** ()	487,461 ms (100%)	0.0 ms (-%)
Self time	0.0 ms (0%)	0.0 ms (-%)
Catalina-utility-1	487,461 ms (100%)	0.0 ms (-%)
Catalina-utility-2	487,461 ms (100%)	0.0 ms (-%)
http-nio-9550-exec-1	487,461 ms (100%)	3,796 ms (100%)
http-nio-9550-exec-2	487,461 ms (100%)	3,572 ms (100%)
http-nio-9550-exec-3	487,461 ms (100%)	3,923 ms (100%)
http-nio-9550-exec-4	487,461 ms (100%)	3,167 ms (100%)
http-nio-9550-exec-5	487,461 ms (100%)	4,105 ms (100%)
http-nio-9550-exec-8	487,461 ms (100%)	3,679 ms (100%)
http-nio-9550-exec-9	487,461 ms (100%)	3,291 ms (100%)
http-nio-9550-exec-10	487,461 ms (100%)	4,169 ms (100%)
http-nio-9550-ClientPoller	487,461 ms (100%)	487,461 ms (100%)

Obrázek 5.4: Využití vláken po odstranění jejich synchronizace.

Databázová vrstva obsahuje vlastní implementaci správy připojení do databáze se seznamem aktivních připojení (connection pool). Proces získání nového připojení je zobrazen na diagramu 5.5. Chybou v této implementaci je uspání vlákna na 100 ms. Jelikož výchozí počet připojení je nastaven na 4, velice rychle dojde k využití všech dostupných připojení a ostatní vlákna

zpracovávající požadavky musejí čekat. Výsledkem je, že všechna vlákna většinu svého času čekají na připojení. Výchozí počet připojení lze samozřejmě změnit, tím ale dojde pouze k oddálení problému a ne k efektivnímu odstranění.

Řešením problému je použití již hotové implementace pro správu připojení do databáze. Oblíbená a stále aktivně vyvíjená knihovna je HikariCP. Tato knihovna poskytuje rychlé, jednoduché, spolehlivé JDBC připojení s minimálními režijními náklady [14]. Použití této knihovny je tak populární, že je implementována ve frameworcích jako Spring nebo Hibernate [13].



Obrázek 5.5: Diagram získání nového připojení do databáze.

5.1.3 SQL & PL/SQL

Dle výsledků testu *Vzrůstající rychlost traktorů* se zdá být problém mezi aplikací a databází, kdy dochází k využití disku na 100%. Příčina problému může být způsobena použitým typem disku, ale také neefektivní manipulací s daty.

K manipulaci s daty dochází na dvou místech - voláním SQL příkazů ze serverové aplikace a v uložených PL/SQL funkcích databázového systému. Po detailnějším zkoumání příčiny bylo zjištěno, že se jedná SQL příkazy *SE-*

LECT, *INSERT* a *BIND*. Tyto příkazy se pravidelně střídaly a procentuální vytížení I/O dosahuje až ke 100%. Analýzou zdrojového kódu byl lokalizován problém při ukládání změny pozice. Využívány jsou dva SQL příkazy, kde první je typu *SELECT* a načítá další ID ze sekvence identifikátorů, a druhý příkaz je typu *INSERT* a provede uložení do tabulky. Při ukládání observace dochází k volání pouze jednoho příkazu typu *INSERT*.

Před každým uložením záznamu do tabulky jsou spuštěny PL/SQL funkce, jejíž funkcionalita je popsána v sekci 3.3.2, které provádějí další manipulaci s daty (viz např. oddílování). Spuštěním všech zmíněných operací pro každý záznam znamená vytížení disku, které vede k jeho přetížení.

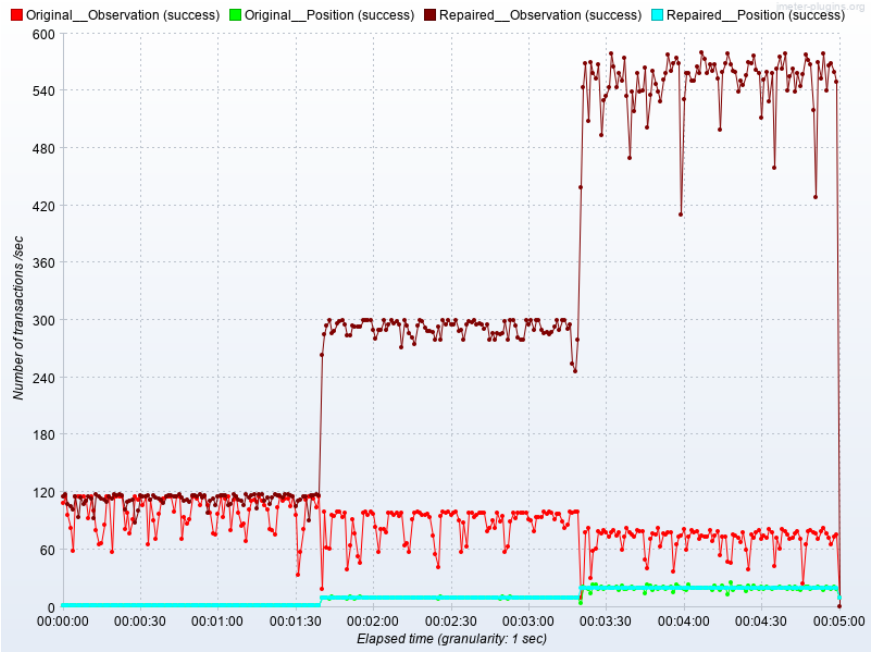
Problém by mohl být vyřešen optimalizací jednotlivých SQL příkazů a přesunutí funkcionality z PL/SQL do aplikace. Tím by se systém stal více nezávislý na typu databáze a mohla by být jednoduše nahrazena jiným typem. Zda by tyto úpravy vedly ke snížení nároků na I/O operace nelze posoudit v rámci této práce, protože úpravy by byly velice náročné a jejich oprava by znamenala reimplementaci celého systému. Aktuální implementace využívá rozšíření PostGIS, které lze použít pouze s PostgreSQL. Pouhá výměna databáze není v tomto případě možná. Součástí SensLog 2.0 je i nový datový model, kde již současné databázové funkce nebude možné použít. Za současné situace se nevyplatí investovat výrazné množství času do optimalizace, ale hledají se možnosti, jak využít výhod časových a NoSQL databází.

5.2 Souhrn problémů a porovnání

Opravena byla synchronizace vláken na všech místech ve zdrojovém kódu, který je spouštěn při testování, a byla přepracována databázová vrstva tak, aby využívala externí knihovnu *HikariCP* pro správu připojení do databáze. Opravená verze je přiložena na CD.

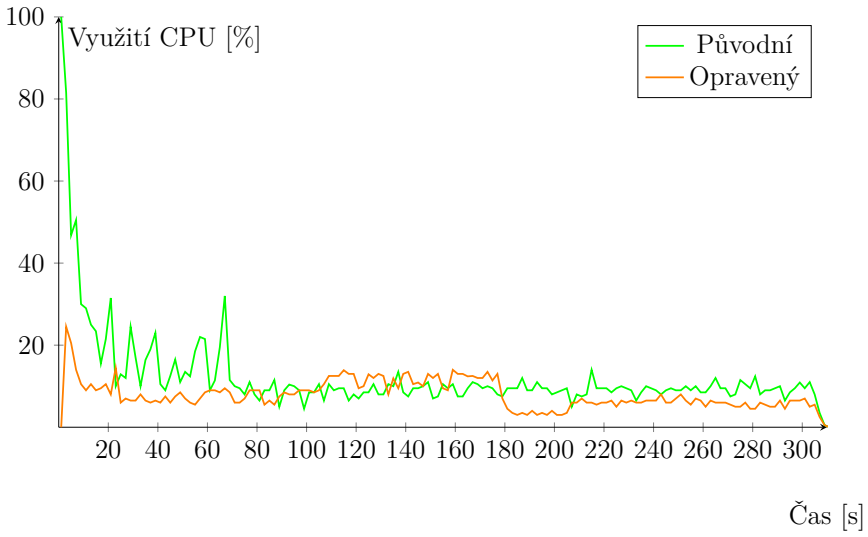
Pro porovnání výkonu systému s odstraněnými chybami a původního systému byl vybrán test *Vysoká zátěž observacemi*. Dle zobrazeného grafu na obrázku 5.6 došlo k dramatickému nárustu propustnosti. Rozdíl je vidět na přijatých observacích, kde původní verze je v grafu označena prefixem *Original_* a opravená verze *Repaired_*. Vrchol přijatých observací u opravené verze se pohybuje kolem hodnoty 550 požadavků za sekundu. Porovnáním této hodnoty s konfigurací testu, která činí 1 500, ani opravený systém nezvládne zpracovat takovou zátěž. Zde je potřeba zmínit, že celý průběh testu dosahuje jiných hodnot, než jaké jsou nakonfigurovány. Bohužel se nepodařilo zjistit, co takové chování způsobuje. Problém může být na straně generování

zátěže a množství dotazů, které lze přes jedno vlákno vygenerovat, ale tato myšlenka nebyla dostatečně ověřena.



Obrázek 5.6: Porovnání původního a opraveného systému.

Na grafu 10 je zobrazeno využití CPU obou verzí. Zde je jasně vidět, že i při x-násobném zlepšení propustnosti je využití procesorového času podobné. Je to dáno paralelním během vláken, ale také tím, že systém neobsahuje výpočetně náročné operace na procesoru a hlavním úkolem je příjem a uložení dotazů do databáze, které procesor značně nevytěžuje.



Graf 10: Porovnání využití CPU původního systému s upraveným.

	propustnost	Ø	90% zpráv
Původní (observace)	86 req/s	41 ms	167 ms
Opravený (observace)	317.3 req/s	16 ms	17 ms
Původní (pozice)	10.6 req/s	97 ms	249 ms
Opravený (pozice)	10.7 req/s	21 ms	38 ms

Tabulka 5.1: Porovnání výsledků původního systému s upraveným.

V tabulce 5.1 jsou zobrazeny souhrnné informace z testu. Průměrné množství přijatých dotazů na změnu pozice je stejné pro oba systémy. To je dáno tím, že oba systémy dokázaly přijmout všechny odeslané pozice. Markantní rozdíl je viditelný v latenci. To samé platí i pro přijaté observace. Latence se pohybovala v rozmezí 10 - 20 ms a server následně mohl obsloužit více dotazů. Uvedené hodnoty ovšem nejsou maximální, protože nedošlo k úplnému vytížení serveru. Pro tyto účely by bylo vhodnější přistoupit k distribuovanému testování, na které ovšem testovací platforma nebyla připravená.

5.3 Proof of concept systému

Vznikající SensLog 2.0 měl být využitý pro porovnání s opravenou aktuální verzí 1.4. Nová verze obsahuje jiný datový model a jedním z požadavků

na systém je, aby byl zpětně kompatibilní. Jiný datový model přináší i jinou funkční logiku v databázi, konkrétně v uložených PL/SQL procedurách. Vytvořené testovací prostředí a konfigurace testů měly ověřit kompatibilitu systému a zároveň potvrdit nebo vyvrátit hypotézu, že úzké hrdlo se nachází právě v uložených funkcích. Pokud by se tak nestalo, následující kroky práce by směřovaly do hledání příčin a možných optimalizací databáze pro zlepšení výkonu.

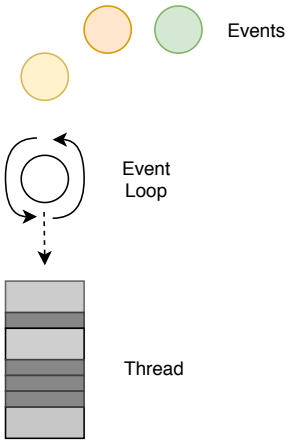
Bohužel v době testování nebylo stádium implementace nového systému na takové úrovni, aby mohl být použitý pro testování. V rámci této práce vznikla nová aplikace jako tzv. Proof of concept (PoC), která implementuje testovanou funkcionalitu a je možné ji použít pro porovnání. PoC přináší nové technologie a architektonická rozhodnutí, která by mohla být začleněna do verze 2.0 za předpokladu, že využité technologie přinesou nějaké výrazné vylepšení. Z časových důvodů se PoC nezaměřuje na optimalizaci databázového systému a funkcí zde uložených. Nalezený problém v podobě dlouhého čekání na I/O operace aplikace neřeší.

5.3.1 Architektura

Hlavním úkolem systému je rychlá práce s daty, především ukládání ale také načítání. Neobsahuje mnoho výpočetní logiky a práce s daty znamená především schopnost správného přiřazení do tabulky nebo uložení se správným identifikátorem. Po systému je požadováno, aby byl schopný obsloužit velké množství připojení a efektivně škáloval. Tyto vlastnosti skvěle splňuje architektura řízená událostmi (EDA). Pro implementaci PoC byla vybrána technologie Eclipse Vert.x, která je založena na EDA a umožňuje asynchronní zpracování zpráv a vysokou rozšiřitelnost díky zasílání událostí.

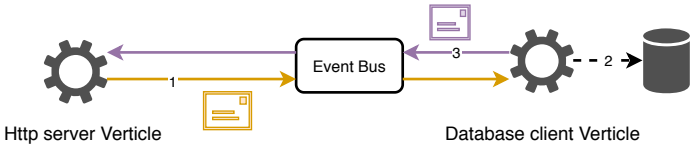
Eclipse Vert.x

Eclipse Vert.x je projekt od Eclipse Foundation s otevřeným zdrojovým kódem (opensource) založen v roce 2012. Nejedná se o framework, ale o sadu nástrojů: knihovny jádra definující základní API pro psaní asynchronních síťových aplikací a dalších modulů, které rozšiřují funkcionalitu na základě požadované funkčnosti (např. připojení do databáze, monitoring, logování a další). Vert.x je založen na projektu Netty a asynchronní síťové knihovně pro JVM zaměřenou na vysoký výkon [33].



Obrázek 5.7: Zpracování událostí ve Vert.x.

Mnoho síťových knihoven a frameworků se spoléhají na jednoduchou strategii vláken - každému klientovi je při připojení přiděleno vlákno do doby, než je spojení ukončeno. Tohle je případ aktuálního systému, který je založen na technologii Servlet a využívá kód z balíků *java.io* a *java.net* v Javě. I když tento model má tu výhodu, že je snadno pochopitelný, problém nastává při velkém počtu souběžných připojení, protože systémová vlákna obsahují určitou režii obsluhy a při velkém zatížení jádro operačního systému tráví značnou dobu jen správou těchto vláken. V takových případech je lepší použít asynchronní přístup. Zpracování příchozích událostí dochází přes smyčku událostí a zajišťuje jej tzv. *Verticle*. Událost může představovat cokoliv, jako je příjem síťových zpráv, práce s databází atd. Zpracování událostí je zobrazené na obrázku 5.7.



Obrázek 5.8: Asynchronní zpracování požadavků přes sběrnici událostí.

Verticle tvoří základní jednotku pro Vert.x. Sběrnice událostí je hlavním nástrojem pro komunikaci mezi nimi využívající asynchronního předávání zpráv. Na obrázku 5.8 je uveden proces zpracování HTTP požadavku a uložení do databáze. Sběrnice událostí umožňuje komunikaci mezi Verticle nejen v rámci stejného JVM, ale je možné je spojit do počítačového clusteru

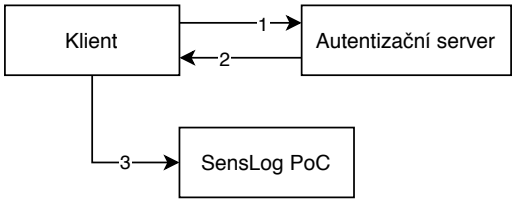
a zprávy mohou být doručeny do jiného uzlu, nebo povolit odběr událostí přes TCP protokol aplikacím třetích stran [33].

5.3.2 Zabezpečení

Jedním z kritických nedostatků systému SensLog je slabá úroveň zabezpečení. Jedná se o jeden z důležitých požadavků pro SensLog 2.0, který je ale stále ve stádiu návrhu, a proto byl tento požadavek implementován do PoC. Cílem je nejen zjistit, jak zabezpečení provést, ale také ověřit, jaké dopady na výkon to může přinášet. Zabezpečení aplikace je v tomto případě omezené pouze na autentizaci a autorizaci.

Množství aplikací v platformě se stále rozrůstá (viz obrázek 3.1) a některé z nich mohou být škálovatelné. Pro tyto případy nelze použít typ autentizace, kde stav bude držen na straně serveru (stateful authentication), protože udržení konzistentního stavu při škálování aplikace by znamenalo značnou komplexitu řešení. Vhodnější je využít bezestavovou autentizaci (stateless authentication), kde stav je držen na straně klienta, který se službou komunikuje. Na základě interních jednání bylo dohodnuto, že autentizace bude provedena centrálním způsobem a autorizace přístupu na jednotlivé služby bude provedena až v dané aplikaci, která služby poskytuje.

Centrální autentizační služba bude vydávat tokeny, které budou použity pro autorizaci. Jelikož v době vzniku této práce ještě není tato služba dostupná, pro účely otestování funkčnosti byla vytvořena aplikace, která generuje platné autorizační tokeny a je možné provést testování včetně potřebné řežije. Na diagramu 5.9 je znázorněno získání tokenu. Klient, který představuje jednotku odesílající data, zažádá o token (1) a na základě ověření identity je mu token vydán (2). Následně je token vložen do požadavku (3) a data jsou odeslána do aplikace. Pro účely testování zde klient představuje testovací nástroj JMeter a autorizační server je aplikace generující tokeny. Na straně PoC je pro autorizaci využita podpora přímo ve Vert.x, který je nakonfigurováný tak, aby tokeny z testovací aplikace mohly být použity.



Obrázek 5.9: Diagram získání tokenu.

Tím, že byl stav přesunut do klientské části, se server stává nezávislým, ale nemá potřebné údaje, podle kterých lze autorizaci provést. Stav je tedy nutné přikládat ke každému požadavku na server. Pro tyto účely existuje vrstva identity nad protokolem *OAuth 2.0* s názvem *OpenID Connect* (protokol OAuth 2.0 není v rámci práce implementován). Jedná se o standard, který umožňuje klientům ověřovat identitu uživatele nebo aplikace na základě autentizace prováděné autorizačním serverem a také získat základní informace o profilu uživatele [24].

Jednoduchou formu přesosu informací o uživateli nabízí token typu JWT (JSON Web Token). Jedná se o otevřený standart RFC7519 [16], který definuje kompaktním způsobem přenos informací mezi dvěmi službami v JSON formátu. Obsažené informace jsou považovány za důvěryhodné, protože jsou digitálně podepsány. JWT může být podepsán HMAC algoritmy nebo algoritmy založené na veřejném a soukromém klíči (např. RSA nebo ECDSA) [17].

Token je složen ze tří částí - hlavička, datový obsah a podpis. Informace obsažené v hlavičce a datový obsah jsou ve formátu JSON. Následně jsou první dvě části zakódovány *Base64* kódováním a spojeny tečkou. Vniklý řetězec je podepsán a spojen tečkou k zakódované části. V boxu níže je uvedena funkce, výsledný token a informace v něm obsažené

```
Funkce:
base64UrlEncode(header).
base64UrlEncoder(payload).
RS256(base64UrlEncoder(header).base64UrlEncoder(payload))

Token:
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ
zdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4
gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.SflKxwRJ
SMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c

Header (hlavička):
{
  "alg": "RS256",
  "typ": "JWT"
}

Payload (datový obsah):
{
  "sub": "1234567890",
  "name": "John Doe",
  "iat": 1516239022
}
```

- **Hlavička** je obvykle složena ze dvou částí - typ tokenu, který je JWT, a algoritmu použitý pro podpis (např. HMAC SHA256 nebo RSA).
- **Datový obsah** obsahuje informace o času vypršení platnosti tokenu, o uživateli a může být rozšířen o další uživatelsky definované atributy.
- **Podpis** je vytvořen zašifrováním hlavičky a datového obsahu a slouží pro ověření, zda zpráva nebyla změněna při přenosu.

Token je vkládán do hlavičky každého HTTP dotazu v následující formě.

```
Authorization: Bearer <token>
```

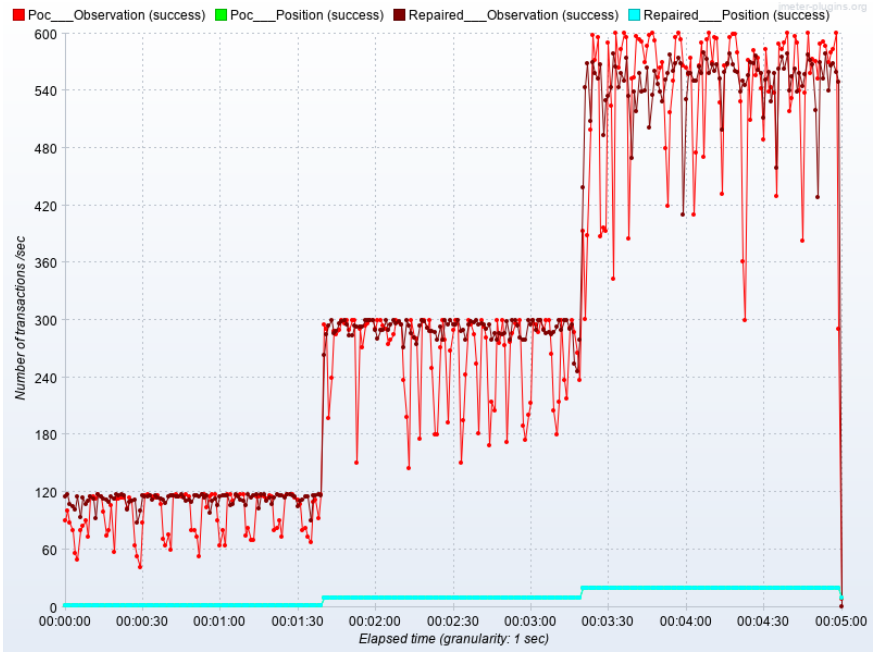
5.4 Opravený systém versus PoC

Do porovnání je zařazena i opravená verze s nově vytvořeným konceptem. Dle vývojářů platformy Vert.x by událostně řízená architektura měla přinést určité výhody oproti Java aplikačním serverům založeným na technologii Servlet. Mezi výhody patří především neblokující zpracování požadavků přes sběrnici událostí, která by měla vést ke snížení HW nákladů na provoz. Především se jedná o menší

počet potřebných vláken a s tím související menší režije při jejich řízení operačním systémem.

Průběh obou systémů dosahuje podobné maximální hodnoty, ovšem stejně jako v předchozím porovnání, se i zde projevilo omezení testovací platformy. Překvapením je zvyšená nestabilita PoC, kde dochází k propadům propustnosti díky nárazovému zvýšení latence. Na základě sledovaných parametrů z testování nelze přesně lokalizovat příčinu tohoto problému. Bohužel nedošlo ani ke snížení HW náročnosti a průběh využití CPU je podobný jako na grafu 10.

Celkový výsledek je překvapující, protože dle webu www.techempower.com[1], zaměrující se na testování webových frameworků, se Vert.x v kombinaci s PostgreSQL řadí mezi jedny z nejlepších kombinací na platformě Java.



Obrázek 5.10: Porovnání opraveného systému a PoC.

Byly vytvořeny další testy, se zaměřením na schopnost přijmout co nejvíce požadavků a velkým množstvím navázaných spojení. Konkrétně se jednalo o vzrůstající zátěž několika desítek observací za sekundu pro až 600 jednotek. Cílem bylo ověřit, zda se prokážou výhody událostně řízené architektury a zpracování přes sběrnici událostí. Požadavky měly stejnou podobu jako observace, ovšem na straně serveru nedocházelo k perzistenci a server odpovídal vždy úspěšně.

[1] https://www.techempower.com/benchmarks

Testování bohužel skončilo neúspěšně z důvodu síťových problémů na straně počítače generujícího zátěž. Výsledky pro obě verze byly stejné a neměly žádnou vypovídající hodnotu. Na základě naměřených výsledků se neprojevily žádné výhody PoC, které vedly k jeho vytvoření, přestože veřejné výsledky výkonnostních testů dokazují opak. Příčinou může být zvolený HW nebo konfigurace testů.

6 Závěr

Cílem práce bylo zjistit, zda je systém připravený na využití ve scénářích zahrnujících IoT, analyzovat případná slabá místa a navrhnout řešení vedoucí k jejich odstranění.

V rámci této práce došlo k seznámení se systémem SensLog a byla provedena analýza jeho funkčnosti. Byly prozkoumány různé metody výkonnostního testování a seznámeno se s vhodnými nástroji. Ze získaných poznatků byly vytvořeny testovací scénáře, které se zakládají na reálných požadavcích zákazníků, zaměřující se především na schopnost přijmout hodnoty pozorování fyzikálních veličin v čase a údaje o poloze.

Testy poukázaly na slabá místa v systému a využitím specializovaných nástrojů došlo k jejich lokalizování. Systém obsahuje dvě vážné chyby nacházející se v serverové aplikaci, konkrétně v databázové vrstvě. První z jich je sériové ukládání do databáze. Přestože k přijímání požadavků na úrovni technologie Servlet dochází paralelně, metoda zajišťující komunikaci s databází je uzavřena synchronizačním primitivem monitor a potencionál paralelizace je nevyužit. Druhá vážná chyba se nachází ve vlastní implementaci správy připojení do databáze, kde se nachází konstrukce pro uspání vlákna při využití všech dostupných připojení. Opravení těchto chyb vedlo k dramatickému nárustu propustnosti, která se zvýšila z maximálního vrcholu 120 observací za sekundu až na hodnotu pohybující se kolem 550 observací za sekundu pro upravený systém (nejedná se o reálnou maximální hodnotu systému, ale takovou kterou dovoluje testovací prostředí). Cloudové testování přes Blazemeter nebylo využito z důvodu nízké generované zátěže ve verzi Starter, která je zdarma.

Další kategorií je optimalizace systému. Systém byl vytvořen za účelem správné funkčnosti a v některých případech dochází k nadměrnému volání SQL příkazů pro dotazování se dalších informací. Při nadměrné komunikaci s databází se stane úzkým hrdlem disk, což následně vede ke zpomalení nebo zhoršené stabilitě systému. V případě použití stejně nakonfigurovaného testovacího prostředí může být chyba i na straně disku, který není vhodný pro takový typ úlohy.

Pro SensLog 2.0 byly v rámci práce vyzkoušeny nové technologie, které by mohly být použity. Jedná se o autorizaci a změnu architektury, která by měla přinést lepší škálování do šířky při nadměrné zátěži. Z důvodu limitace testovacího prostředí se tato hypotéza nepotvrdila.

Všechny body zadání této dipolomové práce byly splněny a získané poznatky a částí implementace budou použity při současném vývoji systému SensLog 2.0.

6.1 Návrhy pro rozšíření práce

Práce by mohla být rozšířena o testy z distribuovaného prostředí, kde by bylo umožněno vygenerovat větší zátěž na testovaný systém a minimalizovat tím chyby způsobené síťovou kartou.

Další oblast se týká konfigurace databáze, analýzy funkčnosti uložených procedur, jejich redukce, optimalizace a přesun funkční logiky do aplikace. PL/SQL funkce částečně implementují vlastní funkcionalitu časových databází, která by mohla být nahrazena databází zaměřenou přímo na tento typ dat. Jednou z možností je *TimescaleDB*, která využívá ekosystému PostgreSQL a lze tak využít rozšíření PostGIS. Seskupování observací a pozic jednotek, v práci nazvané jako oddílování, by mohlo být nahrazeno rozšířením *Citus Data*, které přidává funkcionalitu distribuovaných databází. Tato rozšíření by mohla vést k odstranění několika PL/SQL funkcí, které by mohly být méně optimalizované pro tuto činnost, a eliminovat tím možné výkonnostní chyby způsobené vlastní implementací. Všechny zmíněné návrhy podmiňuje detailní znalost funkčních požadavků systému, poskytované funkcionality jednotlivých rozšíření a ověření kompatibility s požadavkami.

Literatura

[1] *JMeter Elements: Thread Group, Samplers, Listeners, Configuration* [online]. www.guru99.com, 2019. [cit. 2019/1/10]. Dostupné z: `https://www.guru99.com/images/Jmeter.png`.

[2] *Lifecycle and States of a Thread in Java* [online]. www.geeksforgeeks.org, 2019. [cit. 2019/1/15]. Dostupné z: `https://www.geeksforgeeks.org/lifecycle-and-states-of-a-thread-in-java/`.

[3] *iotop(1) Linux User's Manual*, January 2020.

[4] *top(1) Linux User's Manual*, January 2020.

[5] Anley, C. Advanced SQL Injection In SQL Server Applications. An NGSSoftware Insight Security Research (NISR) Publication, 2002.

[6] *Apache JMeter Overview* [online]. Apache, 2019. [cit. 2019/09/27]. Apache JMeter. Dostupné z: `https://jmeter.apache.org`.

[7] *Apache JMeter Manual* [online]. Apache, 2019. [cit. 2019/09/27]. Apache JMeter. Dostupné z: `http://jmeter.apache.org/usermanual`.

[8] Barna, C. – Litoiu, M. – Ghanbari, H. Model-based Performance Testing (NIER Track). In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, s. 872–875, New York, NY, USA, 2011. ACM. doi: 10.1145/1985793.1985930. Dostupné z: `http://doi.acm.org/10.1145/1985793.1985930`. ISBN 978-1-4503-0445-0.

[9] Barse, E. L. – Kvarnstrom, H. – Jonsson, E. Synthesizing test data for fraud detection systems. In *19th Annual Computer Security Applications Conference, 2003. Proceedings.*, s. 384–394, 2003.

[10] Beizer, B. *Software System Testing and Quality Assurance*. Van Nostrand Reinhold Co., 1984. ISBN 0-442-21306-9.

[11] Charvat, K. et al. FOODIE — Open data for agriculture. In *2014 IST-Africa Conference Proceedings*, s. 1–9, 2014.

[12] Charvat, K. et al. SDI4Apps. In *2014 IST-Africa Conference Proceedings*, s. 1–10, 2014.

[13] *Hibernate* [online]. 2020. [cit. 2020/04/20]. Dostupné z: https://docs.jboss.org/hibernate/orm/5.4/userguide/html_single/Hibernate_User_Guide.

[14] *HikariCP* [online]. 2020. [cit. 2020/04/20]. Dostupné z: https://github.com/brettwooldridge/HikariCP.

[15] *Industry 4.0 in agriculture: Focus on IoT aspects* [online]. 2019. [cit. 2019/11/20]. Dostupné z: https://ec.europa.eu/growth/tools-databases/dem/monitor/content/industry-40-agriculture-focus-iot-aspects.

[16] Jones, M. – Bradley, J. – Sakimura, N. JSON Web Token (JWT). RFC 7519, RFC Editor, May 2015. Dostupné z: http://www.rfc-editor.org/rfc/rfc7519.txt. http://www.rfc-editor.org/rfc/rfc7519.txt.

[17] *Json Web Token* [online]. 2020. [cit. 2020/04/20]. Dostupné z: https://jwt.io/introduction/.

[18] Kepka, M. SensLog. https://github.com/mkepka/senslog, 2020.

[19] Kepka, M. et al. The SensLog Platform – A Solution for Sensors and Citizen Observatories. In Hřebíček, J. et al. (Ed.) *Environmental Software Systems. Computer Science for Environmental Protection*, s. 372–382, Cham, 2017. Springer International Publishing. ISBN 978-3-319-89935-0.

[20] Luo, L. Software Testing Techniques. Carnegie Mellon University, 2001. Institute for Software Research International.

[21] *Understanding Memory Leaks in Java* [online]. baeldung.com, 2019. [cit. 2019/11/25]. Dostupné z: https://www.baeldung.com/java-memory-leaks.

[22] Metz, E. – Lencevicius, R. Efficient Instrumentation for Performance Profiling. 08 2003.

[23] *OGC SensorThings API Doc* [online]. sensorup.com, 2019. [cit. 2019/12/28]. Dostupné z: https://developers.sensorup.com/docs/#introduction.

[24] *Welcome to OpenID Connect* [online]. www.openid.net, 2020. [cit. 2020/03/21]. Dostupné z: https://openid.net/connect/.

[25] *Points of interest* [online]. openstreetmap.org, 2019. [cit. 2019/12/02]. Dostupné z: https://wiki.openstreetmap.org/wiki/Points_of_interest.

[26] *PostGIS* [online]. 2019. [cit. 2019/11/29]. Dostupné z: https://postgis.net.

[27] *About PostgreSQL* [online]. www.postgresql.org, 2020. [cit. 2020/03/20]. Dostupné z: https://www.postgresql.org/about/.

[28] *Using the Driver in a Multithreaded or a Servlet Environment* [online]. www.jdbc.postgresql.org, 2020. [cit. 2020/03/20]. Dostupné z: https://jdbc.postgresql.org/documentation/head/thread.html.

[29] *Registr vozidel po propojení se základními databázemi selhává. Praha hlásí úplný kolaps* [online]. https://domaci.ihned.cz, 2012. [cit. 2019/11/19]. Dostupné z: https://domaci.ihned.cz/c1-58960650-centralni-registr-vozidel-kolabuje.

[30] *SensLog About* [online]. SensLog, 2019. [cit. 2019/11/15]. Dostupné z: http://www.senslog.org/about/.

[31] Sheryl Gay Stolberg, M. D. S. *Inside the Race to Rescue a Health Care Site, and Obama* [online]. www.nytimes.com, 2013. [cit. 2019/11/19]. Dostupné z: https://www.nytimes.com/2013/12/01/us/politics/inside-the-race-to-rescue-a-health-site-and-obama.html.

[32] Veeckman, C. et al. Geodata interoperability and harmonization in transport: a case study of open transport net. *Open Geospatial Data, Software and Standards*. 2017, 2, 1, s. 3. ISSN 2363-7501. doi: 10.1186/s40965-017-0015-6. Dostupné z: https://doi.org/10.1186/s40965-017-0015-6.

[33] *Eclipse Vert.x* [online]. 2020. [cit. 2020/04/22]. Dostupné z: https://vertx.io/docs/guide-for-java-devs/.

[34] *VisualVM* [online]. VisualVM, 2019. [cit. 2019/09/27]. VisualVM. Dostupné z: https://visualvm.github.io/documentation.html.

[35] *3 Reasons Web Performance Matters* [online]. obicreative, 2019. [cit. 2019/11/20]. OBICreative. Dostupné z: https://www.obicreative.com/web-performance-matters/.

[36] Weyuker, E. J. – Vokolos, F. I. Experience with Performance Testing of Software Systems: Issues, an Approach, and Case Study. *IEEE Trans. Softw. Eng.* December 2000, 26, 12, s. 1147–1156. ISSN 0098-5589. doi: 10.1109/32.888628. Dostupné z: https://doi.org/10.1109/32.888628.

[37] Zhen Ming Jiang, I. M. – Ahmed E. Hassan, I. M. *A Survey on Load Testing of Large-Scale Software Systems*. IEEE Transactions on software engineering, vol. 41, no. 11, November, 2015.

Seznam zkratek

API (Application Programming Interface) - rozhraní pro programování aplikací.
CAN bus (Controller Area Network bus) - sběrnice pro komunikaci senzorů a funkčních jednotek v automobilu.
CPU (Central processing unit) - elektronická součást počítače vykonávající strojové instrukce.
CSV (Comma-separated values) - souborový formát, kde hodnoty jsou oddělené čárkami.
DBMS (Database Management System) - softwarové vybavení zajišťující práci s databází.
ECDSA (Elliptic Curve Digital Signature Algorithm) - protokol digitálního podpisu s využitím eliptických křivek.
EDA (Event Driven Architecture) - typ architektury softwarových systémů, kde dochází k vytváření, detekci, zpracování reakcí na události.
GC (Garbage collection) - automatická správa paměti.
GNSS (Global Navigation Satellite System) - služba umožňující za pomoci družic autonomní prostorové určování polohy s celosvětovým pokrytím.
GPS (Global Positioning System) - typ GNSS provozovaný Ministerstvem obrany USA.
GUI (Graphic User Interface) - grafické uživatelské rozhraní umožňující ovládání aplikace využitím grafických ovládacích prvků.
HMAC (Keyed-hash Message Authentication Code) - typ autentizačního kódu zprávy počítané s použitím hašovací funkce v kombinaci s tajným klíčem.
HTTP (Hypertext Transfer Protocol) - internetový protokol určený pro komunikaci s webovými servery.
HW (Hardware) - označení pro veškeré fyzické vybavení počítače.
I/O (Input/output) - označení pro vstup a výstup zprostředkovávající hardwarové zařízení s okolím.
ID (Identifier) - stručná informace používaná k odlišení jednotlivých entit.
IP (Internet Protocol) - síťový protokol pracující na síťové vrstvě.
IoT (Internet of Things) - Internet věcí, označení pro síť fyzických zařízení, vozidel a dalších elektronických zařízení se síťovou konektivitou, které mohou mezi sebou komunikovat.
JAX-RS (Java API for RESTful Web Services) - Java API poskytující podporu pro vytváření webových aplikací.
JDBC (Java Database Connectivity) - Java API poskytující přístup k relačním databázím.
JDK (Java Development Kit) - soubor nástrojů pro vývoj aplikací na platformě Java.

JSON (JavaScript Object Notation) - způsob zápisu dat, který je čitelný člověkem.
JSP (JavaServer Pages) - Java technologie pro vývoj webových stránek.
JVM (Java Virtual Machine) - běhové prostředí pro programovací jazyk Java.
OGC (Open Geospatial Consortium) - mezinárodní standardizační organizace podporující vývoj a implementaci standardů pro geoprostorová data a služby.
PID (Process identifier) - jednoznačné číslo určující proces v jádře operačního systému.
PL/SQL (Procedural Language/Structured Query Language) - nástavba nad jazykem SQL umožňující procedurální programování.
POI (Point of interest) - specifický lokalizační bod, kde se nachází zajímavý objekt.
PoC (Proof of concept) - realizace určité metody nebo myšlenky za účelem prokázání její proveditelnosti.
RAM (Random Access Memory) - polovodičová paměť s přímým přístupem pro čtení a zápis.
REST (Representational State Transfer) - architektonický styl softwaru definující sadu omezení, která mají být použité při vytváření webových služeb.
SHA256 (Secure Hash Algorithm 256) - hašovací funkce vytvářející ze vstupních dat výstup délky 256 bitů.
SQL (Structured Query Language) - standardizovaný strukturovaný dotazovací jazyk využívaný při práci s daty v relačních databázích.
TCP (Transmission Control Protocol) - protokol transportní vrstvy využívaný v síti Internet.
VGI (Volunteered Geografic Information) - prostorové informace získané od dobrovolníků.
XML (Extensible Markup Language) - značkovací jazyk umožňující ukládání informací ve stromové struktuře.

Seznam obrázků

Seznam tabulek

Přílohy

Obsah přiloženého CD

Součástí práce jsou následující soubory, které jsou přiložené na CD. Struktura souborů je následující:

```
- DBService-original.war
- DBService-repaired.war
- poc.jar
- generate-token.jar
- jmeter-5.1.1/
- apache-tomcat-9.0.24/
- senslog1-empty.sql
- configs/
- dataSets/
- keys/
- thesis.pdf
```

Uživatelská příručka

Testovací prostředí je připravené pro operační systém Debian, následující příkazy jsou kompatibilní pouze s tímto systémem. Předpokladem je, že příkazy jsou spouštěny ze složky, kde se nachází všechny soubory z přiloženého CD.

Instalace databáze

```
$ sudo apt-get install postgresql postgresql-client postgis
```

```
$ sudo su - postgres
$ psql

root# \i senslog1-empty.sql;
```

Nasazení systému SensLog

Prvním krokem je přesun zkompilované aplikace do složky *webapp*, odkud aplikační server Tomcat načítá aplikace. Stejný postup nasazení aplikace platí i pro opravenou verzi *DBService-repaired.war*.

```
$ cp DBService-original.war <tomcat-home>/webapp/DBService.war
```

Je zapotřebí nastavit správný port, na kterém aplikace bude komunikovat. To se provede v souboru *server.xml* a v elementu *Connector* se nachází atribut *port*. Hodnotu změňte na číslo 9550.

```
Cesta k souboru:
    <apache-tomcat-home>/conf/server.xml

Obsah souboru:
    <Connector port="9550" protocol="HTTP/1.1"
        connectionTimeout="20000" redirectPort="8443" />
```

Spuštění aplikačního serveru se nachází ve složce *bin*, kde se nachází skript *start.sh*.

```
$ <tomcat-home>/bin/start.sh
```

Generování platného tokenu

Aplikace pro generování tokenu obsahuje všechny součásti v již připraveném balíku *jar*. Následujícím příkazem dojde ke spuštění a do konzole se vypíše token. Nakonfigurované testy již token obsahují, tento krok není nutné provádět.

```
$ java -jar generate-token.jar
```

Spuštění PoC

Archiv *poc.jar* obsahuje všechny potřebné knihovny pro svůj běh, spuštění je provedeno standardně (viz box).

```
$ java -jar poc.jar
```

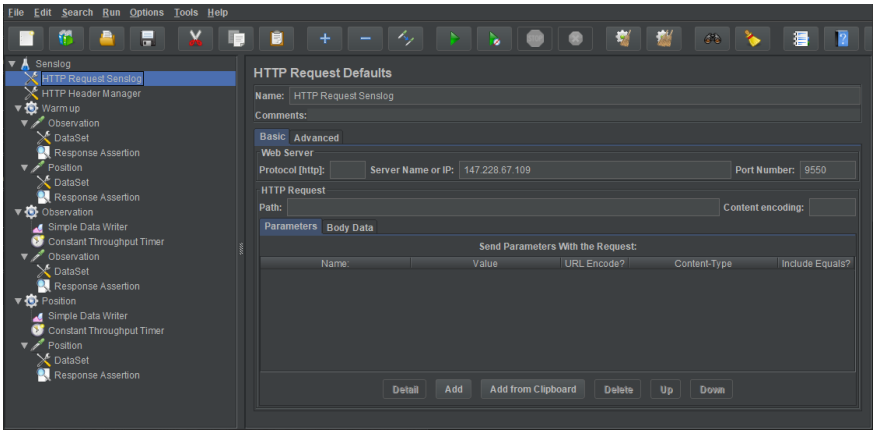
Spuštění testů

Generování zátěže provádí nástroj JMeter, který je spuštěn taktéž z příkazové řádky. Následujícím příkazem dojde ke spuštění, kde za parametrem *-t* je uvedena cesta ke konfiguraci testu ve formátu *jmx*.

```
$ <jmeter-home>/bin/jmeter.sh -n -t configs/test.jmx
```

Konfigurace testů

Testy jsou nakonfigurované tak, aby je šlo spustit bez změny konfigurace. Jediné, co je potřeba změnit je IP adresa serveru. Konfigurační element se jmenuje *HTTP Request Senslog* a atribut *Server Name or IP* (viz obr. 6.1).



Obrázek 6.1: Změna IP adresy v JMeter s GUI.

Instalace a spuštění měřících nástrojů

```
$ sudo apt-get install top iotop
```

V boxu jsou uvedeny příkazy, které měřící nástroje spustí a uloží příslušných souboru. U měření aplikace Java je nutné zvolit příslušný *PID* procesu.

```
$ top -bc -u postgres -d 2 >> hw_postgres.txt
$ top -bc -p <PID> -d 2 >> hw_java.txt
$ iotop -b -u postgres -d 2 >> io_postgres.txt
```

Konfigurace VisualVM

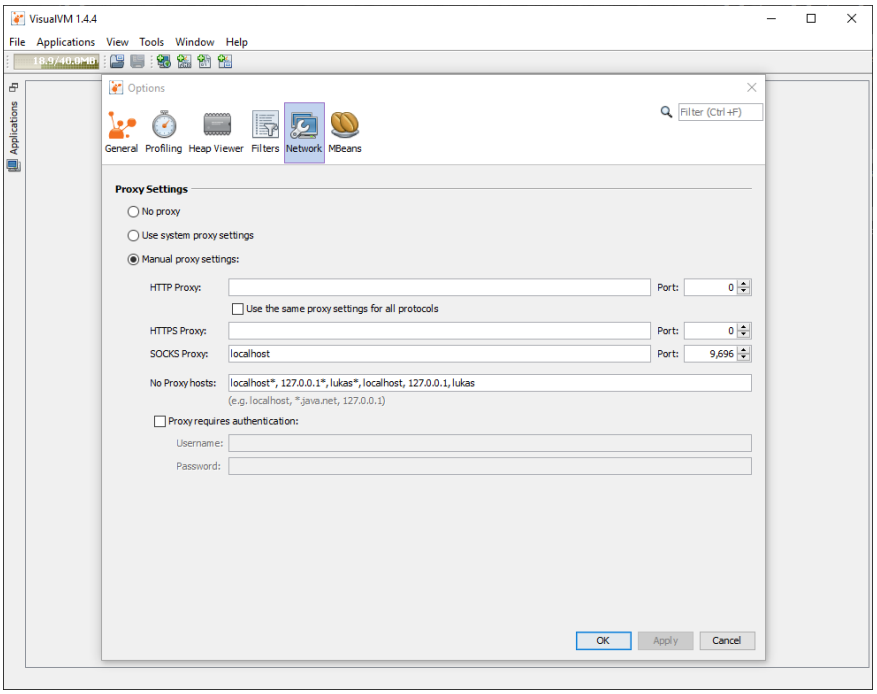
Bude-li monitorovací nástroj VisualVM spuštěný na jiném počítač, než běží serverová aplikace, je potřeba vytvořit SSH tunel mezi klientským počítačem a serverem. Příkaz níže vytváří z klientského počítače SSH tunel na server. Předpokladem pro správné připojení je vytvořený účet na serveru. Příkaz obsahuje port, který bude použit pro proxy ve VisualVM (například 9696).

```
$ ssh -v -D <port> -C -N <username>@<domain>
```

Dalším krokem je povolení odesílání statistických dat z JVM. Pro aplikační server Tomcat vložte do souboru *setenv.sh* parametry *CATALINA_OPTS* a zvolte port (například 1098), na kterém data budou dostupná.

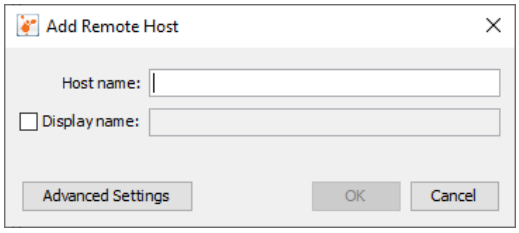
```
Cesta k souboru:
    <tomcat-home>/bin/setenv.sh

Obsah souboru:
    CATALINA_OPTS="
    -Dcom.sun.management.jmxremote
    -Dcom.sun.management.jmxremote.port=<port>
    -Dcom.sun.management.jmxremote.ssl=false
    -Dcom.sun.management.jmxremote.authenticate=false"
```

Ve VisualVM je potřeba vytvořit SOCK Proxy. V nabídce *Tools -> Options* se nachází okno 6.2 a pod záložkou *Network* je manuální konfigurace proxy. Pro atribut SOCK Proxy zvolte *localhost* a zvolený port (například 9696). Tím je nastavení proxy hotové.



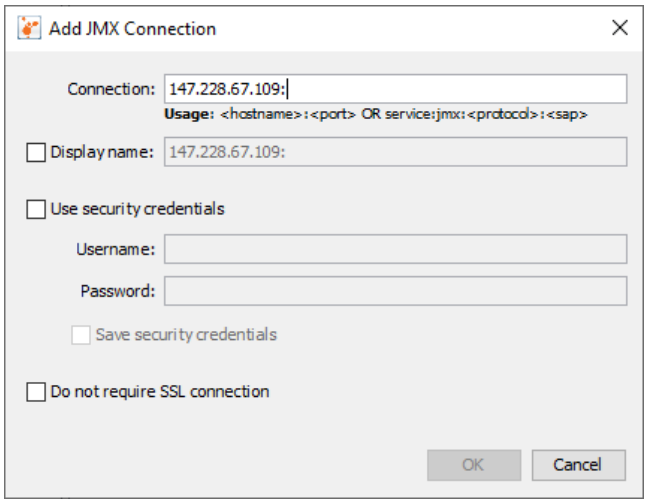
Obrázek 6.2: Nastavení SOCK Proxy ve VisualVM

Následně je nutné vytvořit připojení na vzdáleného hosta. Klikněte pravým tlačítkem myši na ikonu vzdálených hostů a vyberte *Add Remote Host...*. Objeví se okno 6.3, kde vyplňte IP adresu serveru.



Obrázek 6.3: Přidání vzdáleného hosta ve VisualVM

Po úspěšném přídání vzdáleného hosta vytvořte JMX spojení. To provedete stejným způsobem jako přidání vzdáleného hosta, ovšem kliknutím na nově přidaný server. Objeví se okno 6.4, kde vyplňte adresu serveru a za dvojtečkou zvolený port SSH tunelu (například 1098).



Obrázek 6.4: Přidání JMX spojení ve VisualVM

Po této konfigurace lze již spustit profilování aplikačního serveru dvojím poklepáním na nově přidané spojení v levém sloupci *Applications*.

Model databáze