INVESTICE DO ROZVOJE VZDĚLÁVÁNÍ

Platforma průmyslové spolupráce

CZ.1.07/2.4.00/17.0041

Název

Publish-subscribe založený na obsahu pro účely monitorování

Popis a využití

- pokročilá práce s Java SE 7
- výuka: pokročilá Java

Jazyk textu

- český

Autor (autoři)

- Svatopluk Novák

Oficiální stránka projektu:

- http://lasaris.fi.muni.cz/pps

Dostupnost výukových materiálů a nástrojů online:

- http://lasaris.fi.muni.cz/pps/study-materials-and-tools

Obsah

1 **Úvod** . . . 1
- 1.1 Publish-subscribe založený na obsahu . . . 1
- 1.2 Projekt Ngmon . . . 2
- 1.3 Cíle práce . . . 3

2 **Problém publish-subscribe** . . . 4
- 2.1 Formální definice . . . 6
- 2.2 Požadavky na algoritmus . . . 7
  - 2.2.1 Multidimenzionální vyhledávání . . . 7
- 2.3 Implementace . . . 8

3 **Algoritmus Matching Tree** . . . 9
- 3.1 Princip zjišťování konzumentů pro danou událost . . . 9
- 3.2 Princip přihlašování konzumentů . . . 14
- 3.3 Rychlost algoritmu . . . 17
- 3.4 Poznámky k algoritmu a implementaci . . . 17

4 **Counting algoritmus** . . . 20
- 4.1 Princip zjišťování konzumentů pro danou událost . . . 20
- 4.2 Princip přihlašování konzumentů . . . 22
- 4.3 Rychlost algoritmu . . . 22
- 4.4 Poznámky k algoritmu a implementaci . . . 22
  - 4.4.1 Prefix . . . 23
  - 4.4.2 Rozsah . . . 25
- 4.5 Optimalizace . . . 25
  - 4.5.1 Duplikáty . . . 25
  - 4.5.2 Ukládání čítačů . . . 25
  - 4.5.3 Ukládání čítačů v hašovací tabulce . . . 26
  - 4.5.4 Agregace hodnot z indexů . . . 27
- 4.6 Siena . . . 27

5 **Porovnání algoritmů a implementací** . . . 28
- 5.1 Výkonnostní testy . . . 28
  - 5.1.1 Testovací prostředí . . . 28
  - 5.1.2 Caliper . . . 29
  - 5.1.3 Popis benchmarku . . . 29
  - 5.1.4 Výsledky benchmarku . . . 30
  - 5.1.5 Vyhodnocení výsledků . . . 33
- 5.2 Celkové srovnání . . . 38

6 **Zasazení do systému Ngmon** . . . 40
- 6.1 Publish-subscribe . . . 41

6.2 Zpracování požadavků klienta na serveru 43
6.3 Klient 47
7 **Závěr** 50
A **Zdrojové kódy** 53
A.1 Algoritmy a benchmark 53
A.2 Systém Ngmon 55

1 Úvod

Složitost některých dnešních systémů s sebou také přináší nemalé nároky na jejich monitorování a s tím spojenou infrastrukturu. Je nutné sledovat chování jak softwaru, kterým může být například webový či databázový server, tak i hardwaru, typicky zatížení procesorů či využití operační paměti nebo sítě. Z ekonomických důvodů se v současnosti navíc často využívá virtualizace, obvykle jako provoz více virtuálních strojů pod jedním skutečným. V takovém případě ale značně stoupá objem informací, které je nutné monitorovat, jelikož každý virtuální stroj má vyhrazené výpočetní zdroje a na každém mohou být provozovány různé aplikace. Toto všechno vede k požadavku na automatizované a distribuované zpracování těchto dat.

1.1 Publish-subscribe založený na obsahu

Kvůli flexibilitě při vyhodnocování informací získaných monitorováním je také výhodné mít možnost zpracovávat různá data různými aplikacemi, které mohou dokonce běžet na různých strojích a komunikovat s monitorovacím systémem pomocí počítačové sítě. Zde se nabízí využití modelu (vzoru) publish-subscribe, ve kterém figurují dvě strany – producent, který data vytváří, a konzument, jenž je získává a dále zpracovává. V našem případě je producentem typicky nějaká aplikace generující monitorovací události (například server nebo program monitorující systémové prostředky či hardware). Konzumentem je pak klient, který data přijímá a nějak dále zpracovává.

Není nutné, dokonce ani preferované (z výkonnostních a bezpečnostních důvodů), aby každý konzument získával všechny dostupné informace od všech producentů a teprve dodatečně je případně na své straně filtroval. Řešením je, že konzument musí nejprve specifikovat, o jaká data má zájem (proces dále označovaný pojmem *subscribe*). Teprve od této chvíle může začít získávat informace od producentů. Procesu, kdy aplikace (producent) předá data monitorovacímu systému, budu dále říkat pouze *publish*. Zmíněná monitorovací data jsou v dalším textu označována jako *události*, případně *zprávy*.

Výše zmíněný proces subscribe může být realizován různě. Jeden z nejjednodušších způsobů je mít několik skupin (ať už statických či dynamických). Každá událost od producenta pak patří do jedné skupiny (nebo i do více skupin) a naopak každý konzument se přihlašuje k odběru událostí

z jedné či více skupin. Asi je zřejmé, že takovýto přístup je sice velice jednoduchý a obecně nenáročný na výpočetní zdroje (alespoň z hlediska rozhodnutí, jaké události komu poslat), ale není moc flexibilní. Může totiž nutit konzumenty přijímat velké množství událostí, o které vůbec zájem nemají a tyto pak zahazovat. V případě, že máme takovýchto zpráv a konzumentů mnoho, je také možné, že toto řešení bude vyžadovat příliš mnoho systémových prostředků, například času procesoru či propustnosti sítě.

Mnohem pružnější je dát konzumentovi možnost specifikovat požadované události formou podmínek kladených přímo na obsah zpráv. Můžeme pak říct, že každý konzument specifikuje funkci či *predikát*, který v závislosti na obsahu každé jednotlivé zprávy je buď pravdivý (splněný), v kterémžto případě je tato zpráva pro konzumenta určena, či nepravdivý (nesplněný), což naopak znamená, že konzument nemá o danou zprávu zájem. V uvažovaném modelu není možné specifikovat, že události splňující predikát naopak dostávat nechci, což však nepředstavuje v kontextu uvažovaného použití pro konzumenta žádné praktické omezení.

Takovýto přístup ke vzoru publish-subscribe už ovšem vyžaduje složitější algoritmy. Jejich analýze, implementaci a porovnání se věnuje první část práce.

1.2 Projekt Ngmon

Předpokladem pro fungování výše uvedeného je, aby obsah zprávy měl vhodný formát. Ve skutečnosti ale různé typy údajů či událostí, které chceme monitorovat, bývají často reprezentovány pomocí naprosto odlišných formátů. Nezřídka jsou tyto informace navíc naprosto nestrukturované, jako například takzvané logy typicky používané u aplikačních serverů, které nabízejí informace o jejich stavu a probíhajících operacích. Tyto sice mohou být snadno pochopitelné pro člověka, ale pro automatické zpracování se ukazují jako velmi nepraktické. Pak je totiž nutné z těchto dat získat strojově zpracovatelné informace pomocí syntaktické analýzy (jinak také nazývané parsování), což vede k větší náročnosti při vývoji monitorovacího systému a způsobuje navíc jeho pomalejší běh. Kromě toho je možné, že se formát těchto dat v budoucí verzi programu změní, což si vyžádá další úpravy.

Ngmon [1] je název systému pro sběr, zpracování a distribuci událostí, který řeší mj. výše uvedený problém. Jeho záběr sahá od prvotní agregace dat, přes jejich předzpracování a uložení, až po distribuci pomocí sítě podle

zmíněného modelu publish-subscribe. Systém využívá platformy a programovacího jazyka Java.

Tato práce se zabývá pouze částí systému. Konkrétně subsystémem či komponentou publish-subscribe a také komunikací systému s konzumenty.

1.3 Cíle práce

Hlavní cíle práce jsou tři:

- Implementace dvou vybraných algoritmů pro publish-subscribe a srovnání těchto dvou implementací a jedné již existující z hlediska jejich rychlosti s využitím výkonnostních testů (benchmarků),
- výběr vhodného algoritmu na základě výsledků výkonnostních testů a dalších kritérií,
- využití vybraného algoritmu v systému Ngmon a implementace komunikace mezi tímto systémem a konzumenty událostí.

2 Problém publish-subscribe

Problémem, který má algoritmus řešit, je rozhodnutí, kterým konzumentům poslat zprávu vygenerovanou (jakýmkoliv) producentem na základě jejich předchozí žádosti. Tento problém ilustruje obrázek 2.1. Zde nejprve konzument požádá monitorovací systém o přeposílání určitého druhu informací (operace subscribe) a od té doby jsou mu tyto informace pocházející od producentů přeposílány (generování těchto událostí producenty označujeme názvem publish). Operace publish probíhají neustále a pouze od okamžiku zpracování žádosti reprezentované zprávou typu subscribe jsou vyhovující události zasílány příslušným konzumentům. Tedy producent ani konzument na sebe vzájemně nijak nečekají.

Konzument má možnost specifikovat požadované informace pomocí predikátu, který tyto informace musí splňovat. Mějme například jednoduchou zprávu ve formátu JSON[1] uvedenou ve výpisu 2.1.

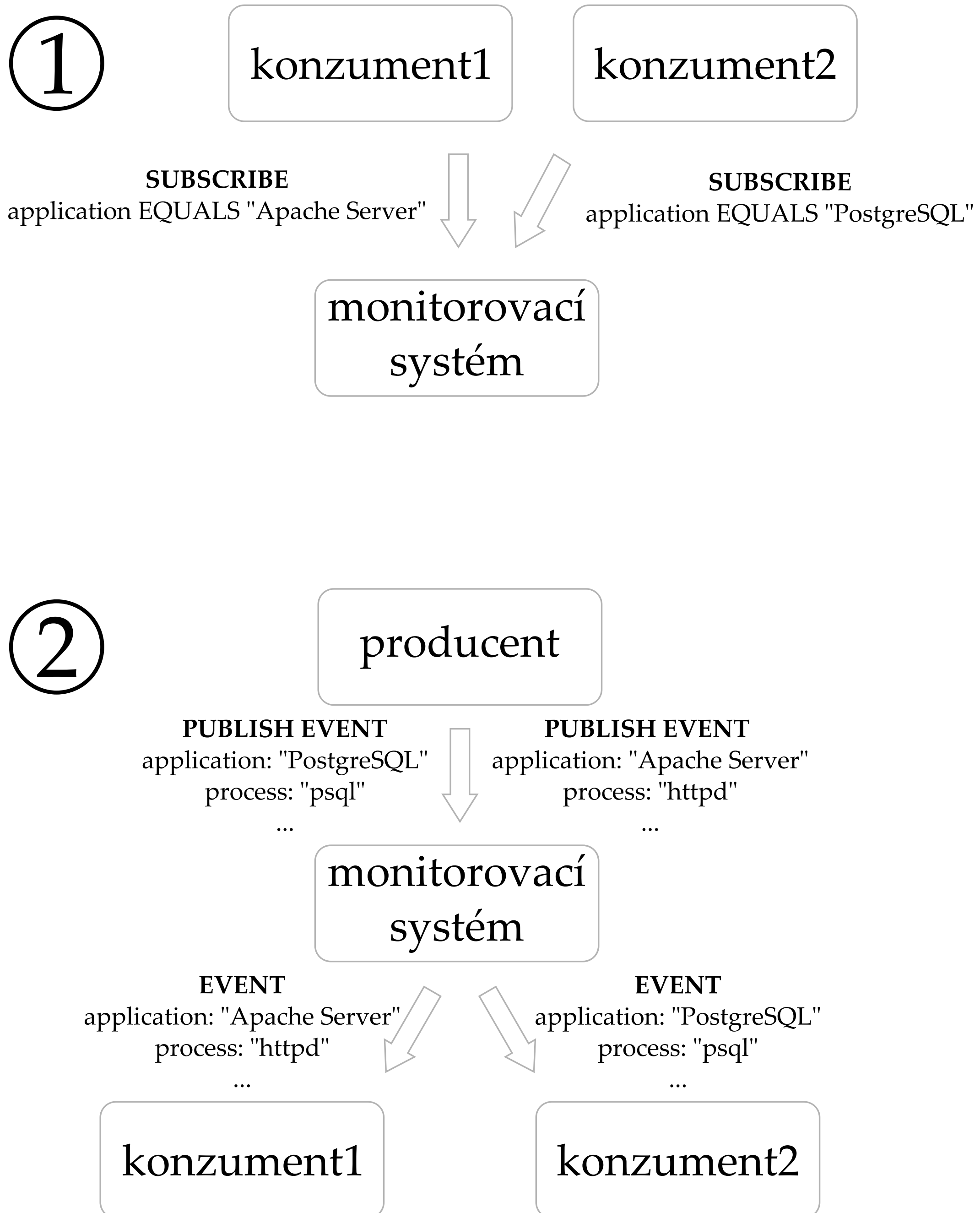
Ze zprávy je asi zřejmé, že obsahuje informace o přijetí HTTP požadavku typu GET webovým serverem Apache HTTP Server. Zpráva obsahuje jak obecné informace, které se týkají všech procesů (např. ID procesu, čas výskytu události či úroveň její závažnosti), tak i doplňující informace specifické pro tuto aplikaci.

Konzument může například specifikovat, že má zájem o události od procesu, jehož přesný název je „Apache Server“, o události, jejichž úroveň závažnosti je větší než 3 nebo třeba o události, které se staly 11. 4. 2012 mezi 8:25 a 8:26. Tato elementární pravidla budu nazývat omezení (anglicky constraints). V případě výše naznačeného formátu zpráv se jednotlivá omezení mohou skládat ze tří částí:

1. Název atributu – například *application*
2. Hodnota atributu – např. *Apache Server*
3. Operátor – např. *je rovno nebo je menší než*

Tato omezení je možné kombinovat pomocí operátoru logické konjunkce do tzv. filtrů. Tedy aby zpráva splňovala daný filtr, musí splňovat všechna elementární omezení, ze kterých se daný filtr skládá. Tak lze umožnit konzumentovi specifikovat například to, že chce dostávat všechny informace o událostech od aplikace Apache Server vyvolané požadavkem typu HTTP GET. Může být výhodné použít také operátor logické disjunkce. V případě, kdy takto spojíme více filtrů, budu hovořit o tzv. predikátu.

1. JavaScript Object Notation, `http://www.json.org/`



Obrázek 2.1: Příklad efektu operací subscribe a publish

```
{"Event":{
"id":16051986,
"occurrenceTime":"2012-04-11T08:25:13.129Z",
"hostname":"lykomedes.fi.muni.cz",
"type":"org.apache.httpd.request.GET",
"application":"Apache Server",
"process":"httpd",
"processId":4219,
"severity":1,
"http://httpd.apache.org/v2.4/events.jsch":{
 "resource":"/apache_pb.gif",
 "protocol":"HTTP/1.0",
 "response":200
 }
}}
```


Výpis 2.1: Příklad monitorovací události ve formátu JSON

2.1 Formální definice

Predikát je konečná a neprázdná množina filtrů, filtr je konečná a neprázdná množina omezení. Omezení je uspořádaná trojice (N, V, O). N je název atributu, V hodnota atributu a O operátor.

Událost je konečná a neprázdná množina uspořádaných dvojic, kde prvním prvkem každé dvojice je název a druhým hodnota atributu. Pro každou událost E také platí:

$\forall x \in E,\, x = (A, B) \,:\, \forall y \in E,\, y = (C, D) \,:\, x \neq y \Rightarrow A \neq C$

Neboli není povolena existence více dvojic se stejným názvem atributu v jedné události.

Dále následují definice funkcí, které formalizují výrazy *splňovat* a *nesplňovat*. Všechny tyto funkce mají jako druhý parametr událost a jako první parametr postupně predikát, filtr a omezení.

Mějme konkrétní predikát P a událost E.

$f_P(P, E) = \exists F \in P \,:\, f_F(F, E) = true$

$f_F(F, E) = \forall C \in F \,:\, f_C(C, E) = true$

C = (N, V, O).

$f_C(C, E) = \exists t \in E,\, t = (A, B) \,:\, N = A \wedge f_E(O, V, B) = true$

Funkce f_P má hodnotu *true* právě tehdy, když je daný predikát splněn pro konkrétní událost. To nastane tehdy a jen tehdy, když tento predikát obsahuje alespoň jeden filtr, který je pro konkrétní událost splněn. Funkce f_F má hodnotu *true* právě tehdy, když je daný filtr splněn pro konkrétní

událost. To nastane tehdy a jen tehdy, když všechna omezení, která tento filtr obsahuje, jsou pro konkrétní událost splněna. Funkce f_C má hodnotu *true* právě tehdy, když je dané omezení splněno pro konkrétní událost. To nastane tehdy a jen tehdy, pokud existuje v události atribut s názvem odpovídajícím názvu atributu v omezení a je splněna funkce f_E.

Funkce f_E má tři parametry – operátor, „testovací" hodnotu a testovanou hodnotu. Uvedu zde pouze dva příklady pro konkrétní hodnoty.

f_E(„<", 5, 2) = true. Dva je totiž menší než pět.

f_E(„=", „httpd", „postgresql") = false. Hodnoty (druhý a třetí parametr) se nerovnají.

2.2 Požadavky na algoritmus

Abychom mohli správně doručovat požadované zprávy, potřebujeme algoritmus, který pro každou zprávu zjistí, kterým konzumentům má být, na základě jejich předchozí žádosti, zpráva doručena. Nejdůležitějšími službami, které má hledaný algoritmus poskytovat, jsou následující:

1. Přihlášení k odběru specifikovaných informací neboli *subscribe*. Z hlediska uživatele algoritmu spočívá v zadání predikátu a identifikace uživatele (konzumenta). Na straně algoritmu dochází k uložení těchto informací do speciální datové struktury. V případě uvedených dvou algoritmů budu tuto strukturu označovat podle algoritmu – tedy Matching Tree, resp. Counting.

2. Zjištění, na základě vložené události, kteří konzumenti se přihlásili k odběru této zprávy pomocí služby *subscribe*. Tuto službu budu nazývat *match*. Parametrem je událost, výstupem seznam konzumentů.

Dále můžeme chtít snadné odhlášení konzumenta neboli *unsubscribe*. Ačkoliv toto se dá obejít opětovným vybudováním příslušné datové struktury, je zřejmé, že taková operace bude typicky velmi časově náročná.

Jak se ukazuje, takovéto algoritmy rozhodně nelze považovat z hlediska jejich složitosti a výpočetní náročnosti za triviální. Tato práce se věnuje dvěma algoritmům, které řeší uvedený problém. Konkrétně se jedná o algoritmy Matching Tree a Counting.

2.2.1 Multidimenzionální vyhledávání

Stojí za zmínku, že naznačený problém se podobá problému multidimenzionálního vyhledávání (multidimensional search), ve kterém jde o hledání

objektu podle více parametrů jeho obsahu. Příkladem může být vyhledávání a indexace geografických souřadnic s využitím datové struktury nazvané R-tree [2]. V našem případě je možné atributy v událostech, resp. predikátech, považovat za dimenze, predikáty pak vkládat například do multidimenzionálního stromu a při příchodu události v něm pak vyhledávat.

Kdybychom se ale rozhodli využít zmiňovaný R-tree, zjistili bychom, že při vyhledávání musíme provádět větvení nejen na základě hodnoty atributu, ale také podle použitého operátoru, což by bez dalších opatření mohlo znamenat větší hloubku stromu, ve kterém vyhledáváme, a tedy snížení rychlosti. Také například test na shodu s konkrétními hodnotami bychom museli provádět klasickým porovnáváním a vyhledáváním ve stromě místo využití často rychlejšího hašování. Algoritmus popisovaný v kapitole 3 má ovšem se zmiňovaným R-tree mnoho společného.

2.3 Implementace

Samotný kód obou algoritmů je v programovacím jazyce Java 7. K samotným algoritmům patří také testy využívající knihovny JUnit a dále benchmarky, kde je použit framework Caliper. Kromě toho je použit nástroj Maven a systém pro správu verzí Git.

Kompletní kód je volně k dispozici díky službě GitHub [3, 4].

3 Algoritmus Matching Tree

Prvním zkoumaným algoritmem je Matching Tree pojmenovaný podle datové struktury, kterou využívá. Existuje ve dvou variantách. První řeší jednodušší problém, kdy se omezíme na operátor rovnosti. Tedy konzument může pouze specifikovat, že chce dostávat události, kde je nějaký atribut roven zadané hodnotě, případně neuvede atribut vůbec, čímž dá najevo, že nechce omezit přijímané události v závislosti na hodnotě tohoto atributu. Přestože je jednodušší, důkaz složitosti algoritmu rozhodně zřejmý není [5]. Této se dále věnovat nebudu. Ve druhé variantě nejsme omezeni a můžeme tak využívat i další operátory jako například „větší než". Obě varianty jsou k dispozici prostřednictvím služby GitHub [3].

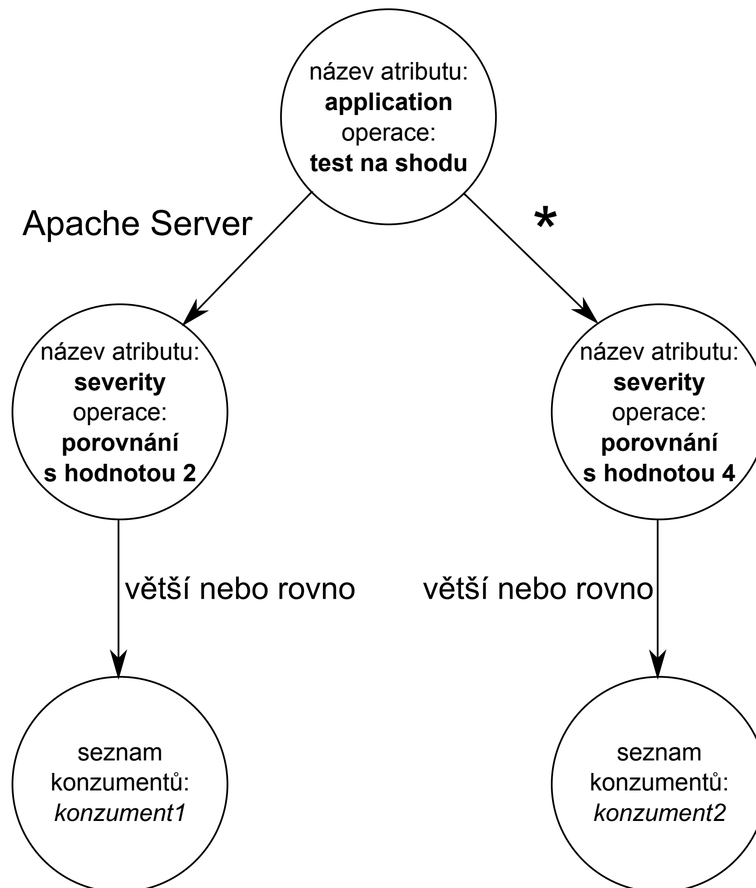
3.1 Princip zjišťování konzumentů pro danou událost

Algoritmus (přesněji operace *match*) je založen na prohledávání datové struktury n-nární strom od kořene k listům. Každý uzel představuje test na hodnotě atributu, podle jehož výsledku se pokračuje v prohledávání. V listech stromu jsou pak uloženy identifikace konzumentů. Predikát specifikovaný konzumentem je tedy jednoznačně určen konkrétní cestou od kořene k listu. Množina konzumentů ze všech listů, ke kterým při prohledávání dojdeme, pak je tvořena přesně těmi, kteří na základě pravidel (predikátů) mají o tu konkrétní událost zájem.

Předpokládejme nyní jednoduché schéma pro události, které obsahuje pouze dva atributy – název aplikace jako řetězec a závažnost události, což bude celé číslo od 1 do 5. Dále mějme dva konzumenty, dále označované jako *konzument1* a *konzument2*. Konzument1 chce dostávat všechny události, jejichž původcem je aplikace „Apache Server" a zároveň závažnost je větší nebo rovna 2. Konzument2 se zajímá o události od všech aplikací, jejichž závažnost je minimálně 4. Příslušný strom pak může vypadat tak jako ten na obrázku 3.1.

Proces zjišťování konzumentů pak probíhá takto:

1. Začínáme v kořenovém uzlu. Pokud se hodnota atributu *application* rovná *Apache Server*, prohledávání pokračuje v obou potomcích. Pokud je hodnota atributu jiná, pokračuje se pouze v pravém uzlu. Za zmínku stojí speciální přechod (dále označovaný jako *-přechod), jehož hodnota je symbolicky označena hvězdičkou. Ten vyjadřuje, že konzumenti v listech pod uzlem, do kterého takto označený přechod



Obrázek 3.1: Ukázka stromu pro dva konkrétní predikáty

```
procedure match(Tree, event)
    visit(Tree, root, event)

procedure visit(Tree, v, event)
    if v is a leaf node of Tree then output(v)
    else
        perform test prescribed by v on event
        if v has an edge e with the result of test
        then visit(Tree, (child of v at the endpoint
            of e in Tree), event)
        if v has a *-edge e
        then visit(Tree, (child of v at the endpoint
            of * in Tree), event)
```

Výpis 3.1: Princip zjišťování konzumentů pro danou událost (operace match) [5]

ukazuje, nechtějí na základě hodnoty tohoto atributu nijak omezovat množství událostí, které dostanou.

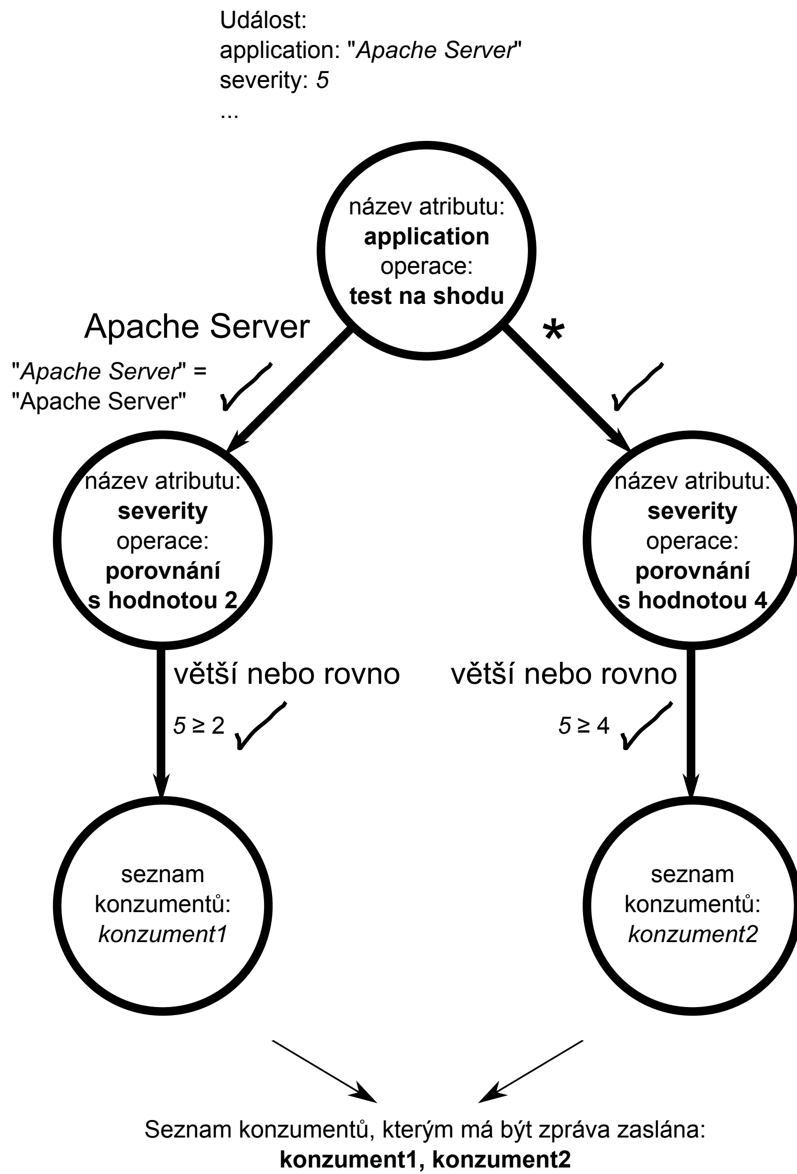
2. Pokračujeme na druhé úrovni buď tedy pravým nebo oběma uzly. Každopádně porovnáme hodnotu atributu *severity* s hodnotou 2, resp. 4. Podle výsledku pak buď pokračujeme nebo nepokračujeme v prohledávání.

3. Dojdeme k listům a vezmeme všechny konzumenty ze všech těchto listů.

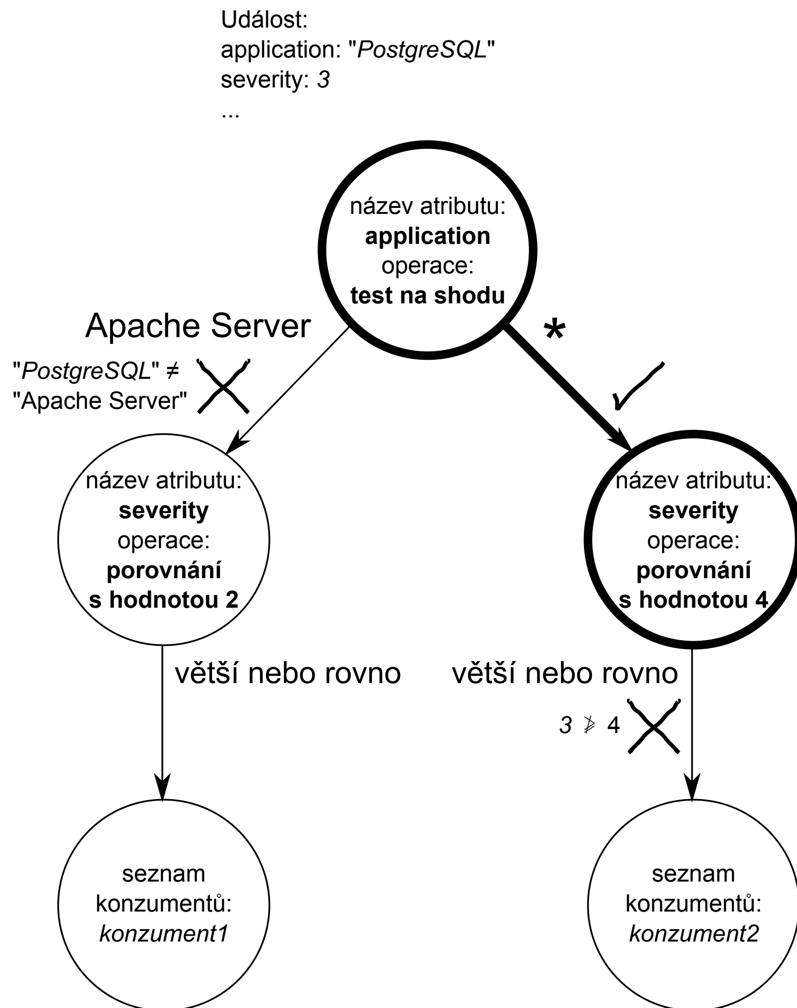
Pro konkrétní strom a danou událost může algoritmus postupovat způsobem uvedeným na obrázku 3.2, kde výsledek představuje seznam, ve kterém se nachází oba konzumenti, nebo na obrázku 3.3, kde je naopak výsledkem prázdný seznam, tedy událost není určena ani pro jednoho z konzumentů.

Obecně je tento proces popsán pomocí pseudokódu uvedeném ve výpisu 3.1.

Zmíněný kód tedy prohledává strom do hloubky, přičemž vždy pokračuje větví představující výsledek testu (pokud existuje) a větví označenou hvězdičkou (opět samozřejmě pouze pokud taková větev existuje) a příkazem *output* vypisuje nalezené konzumenty nacházející se v listech stromu.

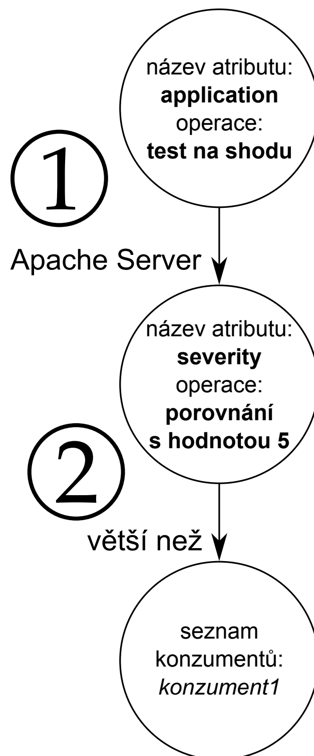


Obrázek 3.2: Ukázka průchodu algoritmem



Obrázek 3.3: Ukázka průchodu algoritmem

Predikát (konzument1)
1. application JE ROVNO "Apache Server"
2. severity JE VĚTŠÍ NEŽ 5



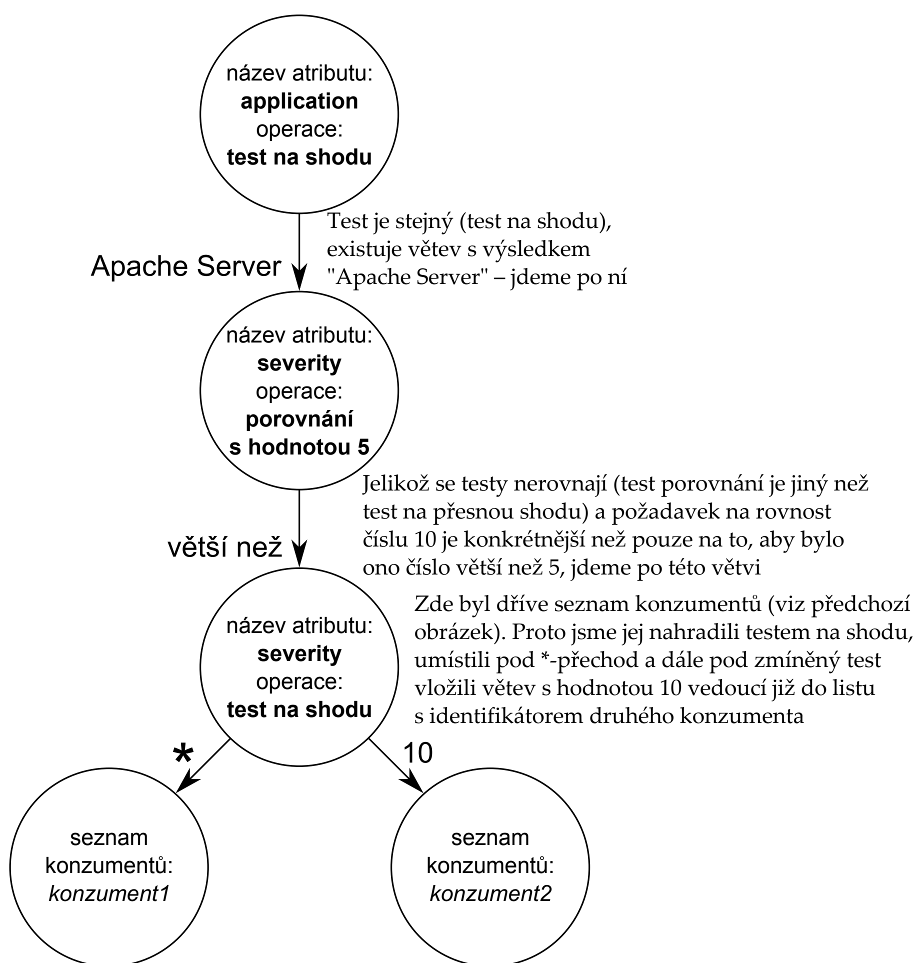
Obrázek 3.4: Ukázka konstrukce datové struktury Matching Tree

3.2 Princip přihlašování konzumentů

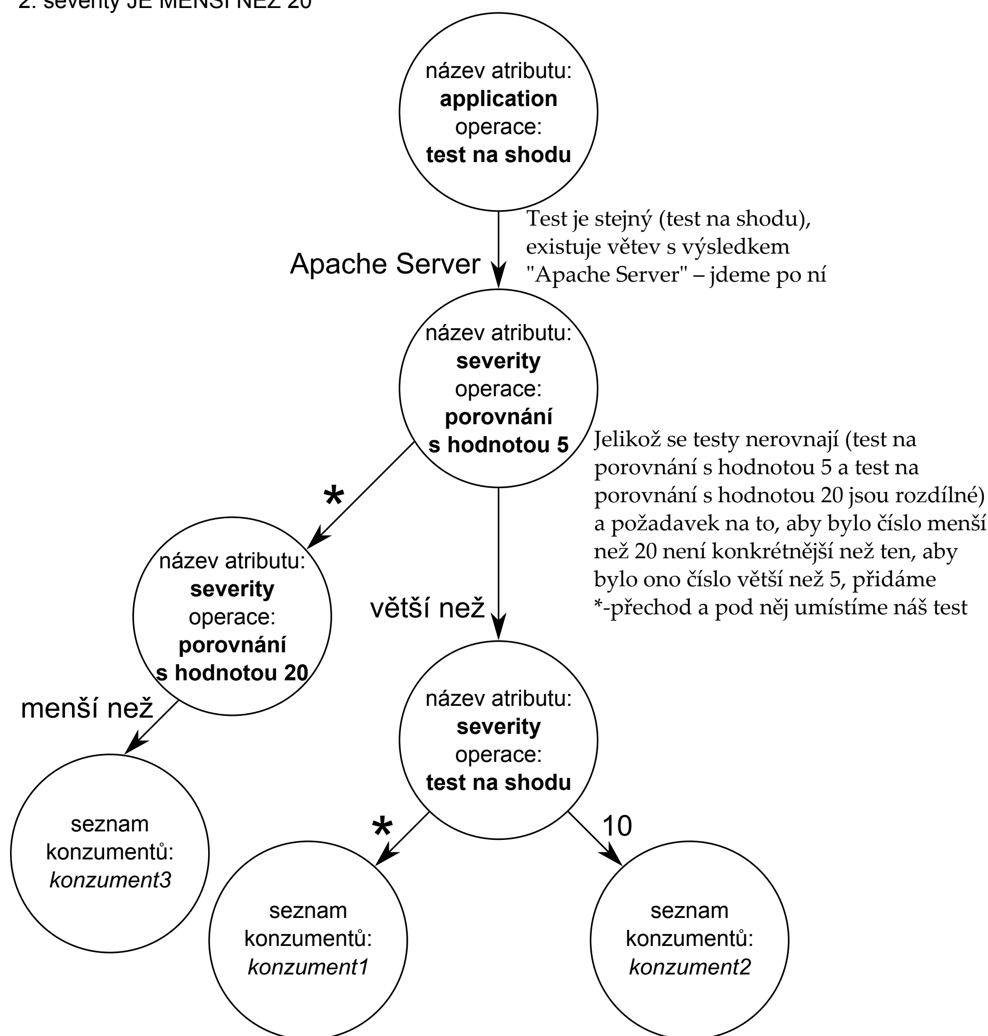
Přidávání konzumentů a jejich predikátů do této datové struktury je o něco složitější. Je totiž mj. nutné spolu „porovnávat" různé operace a podle toho strom upravovat.

Na následujících obrázcích (3.4, 3.5 a 3.6) je ukázána konstrukce stromu pro tři konkrétní konzumenty a jejich predikáty.

Příslušný pseudokód je pak k dispozici v odkazované literatuře [5]. Je patrné, že je potřeba řešit několik různých situací. Tento postup také přináší nutnost mít nad atributy (resp. jejich názvy) definované uspořádání.



Obrázek 3.5: Ukázka konstrukce datové struktury Matching Tree



Obrázek 3.6: Ukázka konstrukce datové struktury Matching Tree

3.3 Rychlost algoritmu

Rychlost algoritmu v nejhorším případě je lineárně závislá na celkovém počtu uzlů ve stromě. Dále je zřejmé, že operace test na shodu nebudou mít na rychlost prohledávání stromu příliš velký vliv. Například pomocí hašování rychle zjistíme, kterou větví máme pokračovat a dále pokračujeme maximálně dvěma větvemi (větví s konkrétní hodnotou a s libovolnou hodnotou označenou hvězdičkou). Naopak pokud máme složitější operaci, například porovnání, a konzumenti toto často využívají na konkrétním atributu, může se snadno stát, že získáme „řetěz“ uzlů naznačený na obrázku 3.7. V takovém případě může velmi výrazně narůst celková hloubka stromu, na které závisí doba prohledávání grafu (v nejhorším případě).

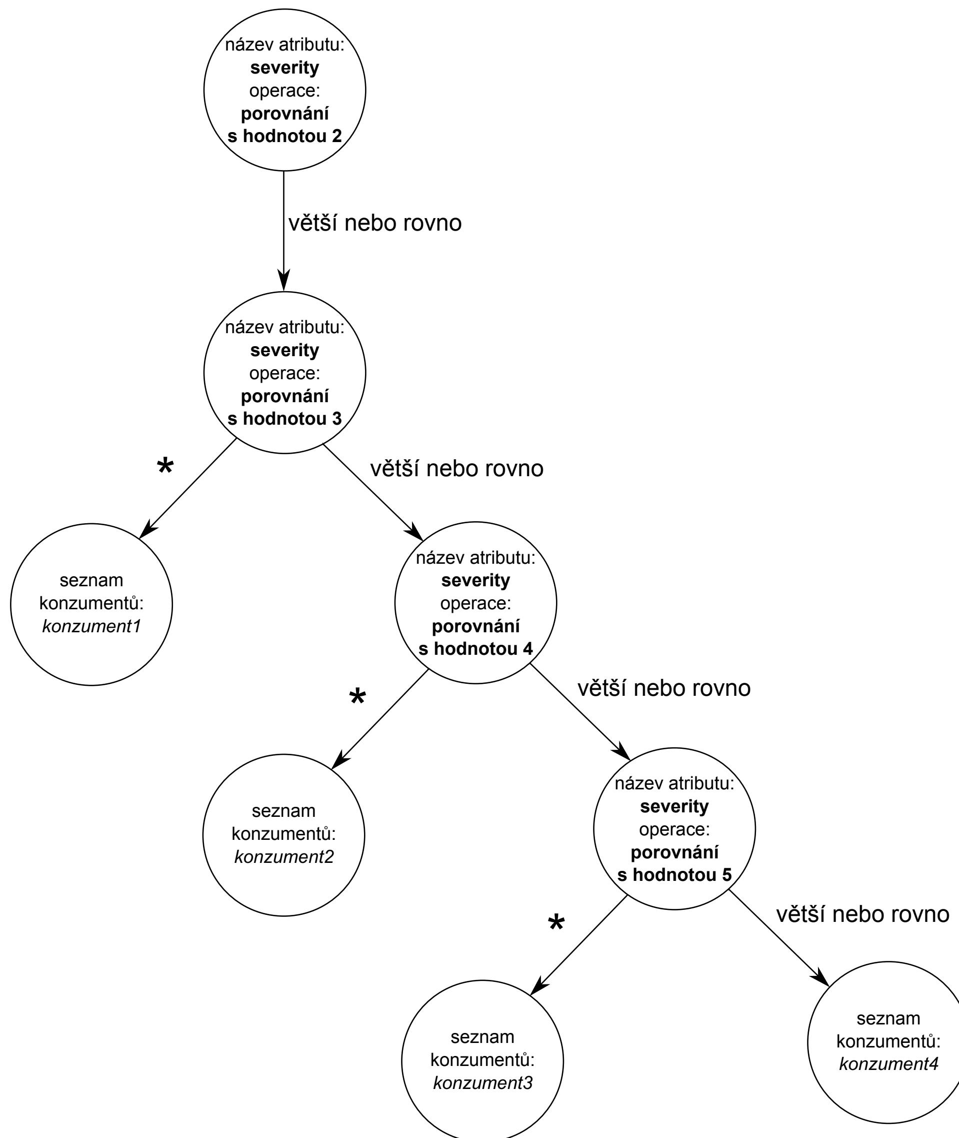
Jak je asi patrné, není možné příliš přesně odhadnout velikost a strukturu stromu (a tím i průměrnou dobu běhu operace match) bez znalosti obsažených predikátů.

Základní algoritmus je dále možné zrychlit pomocí několika optimalizací [5].

3.4 Poznámky k algoritmu a implementaci

Součástí práce je i jednoduchá implementace tohoto algoritmu včetně testů. Operace unsubscribe zde není podporována. Je možné použít již zmíněné testy: test na přesnou shodu a porovnání s konkrétní hodnotou. Vzhledem k tomu, že test na shodu využívá hašování (tedy standardní Java metodu hashCode()) a test porovnání funkce definované rozhraním Comparable, je možné využít prakticky jakýkoliv datový typ. Přidání dalších testů by také nemělo být příliš obtížné, každý další test s sebou ale přináší požadavek ve formě nutnosti doplnění algoritmu o možnost porovnání jednotlivých testů mezi sebou (formou každý s každým). Algoritmus pro přidávání predikátů do stromu (subscribe) totiž musí být pro korektní přidání schopen rozhodnout, v jakém jsou vztahu test, který je součástí přidávaného predikátu a test, který je již ve stromě obsažen. Konkrétně je nutné zjistit, zda jeden z testů je „obecnější“ než ten druhý. Například požadavek, aby hodnota atributu severity byla větší než 2 je obecnější než pouze aby byla větší než 3. Naopak pokud konzument přijímá pouze zprávy, kde má tento atribut hodnotu přesně 4, pak je konkrétnější než oba výše zmíněné požadavky. Obecnější požadavek pak bude v uzlu „nad“ požadavkem konkrétnějším, tedy příslušný test bude vyhodnocen dříve.

Počet všech případů, které musíme v algoritmu zohlednit, pak s přibývajícím počtem testů velmi rychle roste, což nemusí být příliš praktické.



Obrázek 3.7: Ukázka rychle rostoucí hloubky stromu

Vzhledem k tomu, že v uvedené implementaci jsou pouze 2 testy, to ale samozřejmě nepředstavovalo žádný problém.

Vzhledem k povaze algoritmu (konkrétně testů v jednotlivých uzlech) nelze plně využít statické typové kontroly překladačem, ale je nutné spoléhat na dynamickou (běhovou) typovou kontrolu.

4 Counting algoritmus

Tento algoritmus je založen na počítání, kolik omezení (constraints) je splněno pro příslušnou událost. Pokud jsou v konkrétním filtru splněna všechna omezení, znamená to, že predikát obsahující tento filtr je splněn, a tedy tato událost by měla být poslána odpovídajícímu konzumentovi (tomu, který si ten predikát dříve zaregistroval) [6].

Naše implementace je opět k dispozici díky službě GitHub [4].

4.1 Princip zjišťování konzumentů pro danou událost

Tento proces probíhá ve dvou krocích:

1. Jako první identifikujeme všechna dříve přidaná individuální omezení (constraints), která jsou pro příslušnou událost splněna. Například pro událost uvedenou ve výpisu 2.1 to mohou být omezení jako *atribut severity je menší než 3* nebo *atribut application se rovná „Apache Server“*. Zde je kvůli rychlosti nutné využít indexy, díky kterým můžeme tuto zdánlivě výpočetně náročnou operaci provádět relativně rychle.

2. V druhém kroku nalezená omezení procházíme a v příslušném filtru inkrementujeme hodnotu *counter* (odtud název algoritmu), dále jako *čítač*. Tato nám udává počet omezení, která jsou pro daný filtr (a konkrétní událost) splněna. Například pokud se filtr skládá ze tří omezení, může čítač nabývat hodnot od nuly do tří. Jakmile při inkrementaci čítače zjistíme, že hodnota čítače je rovna počtu omezení, znamená to, že filtr je pro danou událost splněn, a tedy je splněn i predikát, který tento filtr obsahuje. Do výstupního seznamu uživatelů, kterým má být událost zaslána, pak přidáme toho, jenž při procesu *subscribe* uvedl onen predikát.

Pro názornost je uveden příklad. Následuje několik filtrů a omezení, která obsahují. Z důvodu jednoduchosti předpokládám, že každý filtr je obsažen v predikátu se stejným číslem (např. Filtr 2 je obsažen v predikátu s názvem Predikát 2) a každý predikát obsahuje právě jeden filtr.

Filtr 1		
název atributu	operátor	hodnota atributu
type	=	org.apache.httpd.request.GET

Filtr 2		
název atributu	operátor	hodnota atributu
type	=	org.apache.httpd.request.POST

Filtr 3		
název atributu	operátor	hodnota atributu
type	=	org.apache.httpd.request.GET
hostname	=	example.com

Filtr 4		
název atributu	operátor	hodnota atributu
process	=	httpd
severity	$\leq$	2

Filtr 5		
název atributu	operátor	hodnota atributu
process	=	postgresql
severity	>	3

Nyní přijde událost uvedená ve výpisu 2.1. Nejprve je zjištěno, která omezení jsou pro tuto událost splněna.

název atributu	operátor	hodnota atributu	splněno?	názvy příslušných filtrů
type	=	org.apache.httpd.request.GET	ano	Filtr 1, Filtr 3
type	=	org.apache.httpd.request.POST	ne	Filtr 2
hostname	=	example.com	ne	Filtr 3
process	=	httpd	ano	Filtr 4
severity	$\leq$	2	ano	Filtr 4
process	=	postgresql	ne	Filtr 5
severity	>	3	ne	Filtr 5

Pro každé splněné omezení je inkrementován čítač u příslušného filtru (či více filtrů). Výsledné hodnoty čítačů jsou uvedeny v následující tabulce:

název filtru	hodnota čítače	celkový počet omezení u filtru	splněno
Filtr 1	1	1	ano
Filtr 2	0	1	ne
Filtr 3	1	2	ne
Filtr 4	2	2	ano
Filtr 5	0	2	ne

Splněné filtry jsou právě ty, které mají hodnotu čítače rovnu celkovému počtu omezení u filtru. To také znamená, že všechna omezení, která konkrétní filtr obsahuje, byla splněna.

Tedy byly splněny filtry s čísly 1 a 4 a také predikáty tyto filtry obsahující.

4.2 Princip přihlašování konzumentů

Přihlašování konzumentů spočívá především v uložení jednotlivých omezení do příslušných indexů. Tyto indexy jsou rozděleny podle jednotlivých atributů v události (predikátu), resp. jejich názvů. Toto je výhodné pro proces *match*.

4.3 Rychlost algoritmu

I zde závisí rychlost na mnoha faktorech, ovšem tím dominujícím je jednoznačně celkový počet jednotlivých omezení, která jsou pro danou událost splněna. Všechna tato omezení totiž v procesu *match* procházíme, jelikož potřebujeme provést inkrementaci příslušného čítače. Zde je nutné si uvědomit, že pokud se neomezíme na predikáty s jedním filtrem a filtry s jedním omezením, není možné z počtu splněných predikátů předpokládat mnoho o počtu splněných omezení.

Předpokládejme, že pro danou událost jsou splněny všechny predikáty. V tom případě je možné, že i všechna omezení (ve všech filtrech) jsou splněna. Stejně tak je ale možné, že byl splněn vždy pouze jeden filtr (tedy i všechna jeho omezení) z mnoha, a tedy většina jednotlivých omezení splněna nebyla. Na druhou stranu můžeme také předpokládat, že nebyl splněn ani jeden predikát. Mohlo se samozřejmě stát, že nebylo splněno jediné omezení. Také to ale mohlo být způsobeno tím, že splnění filtru vždy „zabránilo“ pouze jedno omezení z mnoha, a tedy většina omezení tentokrát splněna byla. Tedy z pouhého počtu splněných predikátů toho o počtu splněných omezení (a naopak) předpokládat příliš nemůžeme.

4.4 Poznámky k algoritmu a implementaci

Zdrojový kód (včetně testů) je samozřejmě součástí práce [4]. Podporovány jsou všechny hlavní operace, tedy jak *subscribe* a *match*, tak i *unsubscribe*. Přes relativní jednoduchost algoritmu se ukázalo jako důležité vhodně navrhnout třídy umožňující rychlé získání splněných omezení (tedy in-

dexy). Kromě samotného rozhraní představovaného třídou *Counting* stojí za zmínku především třídy *FilterMatcher*, která ukládá filtry a také používá třídu *AttributeIndex*, jež představuje jednoduché rozhraní pro přidávání jednotlivých omezení do indexů.

Samotná implementace indexů není nijak zvláštní. Pro operátory typu *je menší než* nebo *je větší či rovno* je využito třídy *NavigableMap*, naopak pro operátor *je rovno* klasická třída *HashMap*. Klíčem je samotná hodnota uložená v příslušném omezení, zatímco hledanou hodnotou celý objekt typu *Constraint*.

Diagram zobrazující hlavní třídy se nachází na obrázku 4.1.

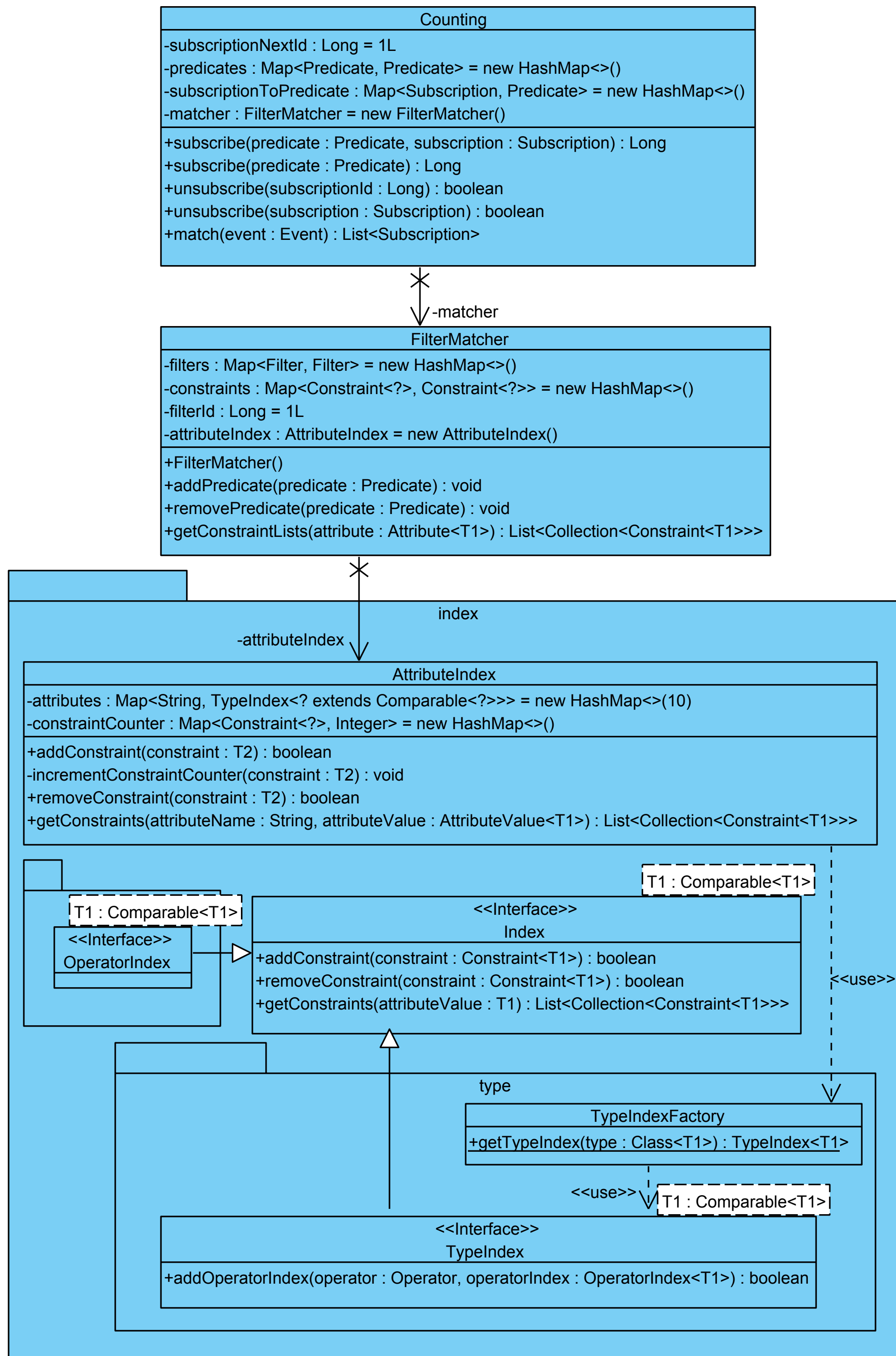
Zmíněná třída *AttributeIndex* si podle typu atributu v omezení (Constraint) pomocí třídy *TypeIndexFactory* zpřístupní objekt typu *TypeIndex* představující úložiště pro hodnoty tohoto typu. Je zde ještě abstraktní třída *AbstractTypeIndex* implementující rozhraní *TypeIndex*. Od této třídy pak již dědí konkrétní implementace, například *LongIndex* či *StringIndex*. Tyto ještě neukládají hodnoty přímo, ale využívají různé indexy pro různé operátory, přičemž každý tento index je reprezentován rozhraním *OperatorIndex*. Toto rozhraní pak implementují třídy, které již přímo ukládají hodnoty do indexů, například *EqualsIndex* využívající hašovací tabulku (resp. třídu *HashMap*), *StringPrefixIndex* ukládající hodnoty v datové struktuře *TernarySearchTree*, *RangeIndex* používající strukturu *RangeTree* a nakonec také „příbuzné“ třídy *LessThanIndex*, *LessThanOrEqualToIndex*, *GreaterThanIndex*, *GreaterThanOrEqualToIndex*, kterým k ukládání hodnot slouží standardní třída *NavigableMap*.

4.4.1 Prefix

Kromě již zmíněných operátorů byla přidána podpora operátoru pro zjišťování, zda hodnota typu řetězec začíná zvolenými znaky. To nám může usnadnit práci s řetězci s hierarchickou strukturou, jako například URL nebo názvy balíků používané v jazyce Java.

Při implementaci tohoto operátoru byla využita datová struktura ternární vyhledávací strom. Tato umožňuje mj. efektivní ukládání a následné vyhledávání řetězců. Jako základ slouží implementace z projektu eBus[1], která byla doplněna o metodu *getAllPrefixes*.

1. `http://sourceforge.net/projects/ebus/`



Obrázek 4.1: Hlavní třídy naší implementace algoritmu Counting

4.4.2 Rozsah

Náš kód zahrnuje také implementaci operátoru pro rozsah, resp. interval hodnot. Ta je založena na speciální stromové struktuře, nazývané Range Tree či Interval Tree [7].

4.5 Optimalizace

Při implementaci se objevilo několik věcí, které mohou znatelně ovlivnit rychlost operace *match*.

4.5.1 Duplikáty

Pokud konzument požaduje vložení predikátu, filtru či omezení, které již v datové struktuře existuje, není toto uloženo znovu, zaznamenány jsou pouze nové relace. Například pokus o přidání již existujícího filtru způsobí pouze uložení příslušného predikátu do již existujícího filtru. Ani existující omezení nejsou duplikována, pouze je udržován počet referencí (inkrementován při přidávání, dekrementován při odebírání), aby bylo možné omezení korektně smazat. To nám umožní především snížení paměťové náročnosti a tím také mírné zvýšení rychlosti.

Z tohoto také vyplývá, že mezi uloženými predikáty a filtry, resp. filtry a omezeními není relace typu 1 : N, ale M : N.

4.5.2 Ukládání čítačů

Protože inkrementace čítače je mnohokrát prováděná operace, má smysl se zamyslet nad její optimalizací. Přímočarým řešením pro ukládání čítačů splněných omezení pro jednotlivé filtry je využití hašovací tabulky, resp. třídy *HashMap*, jejíž objekt je lokální proměnnou v metodě *match*. Máme ale také možnost je ukládat přímo u samotných filtrů, jako jeden z jejich atributů. Experimentálně bylo zjištěno, že tato úprava zrychlí operaci *match* asi o 50 procent (přesněji na provedení stejných operací potřebuje přibližně $\frac{2}{3}$ času). A to i přes to, že musíme na konci (případně na začátku) operace všechny tyto čítače vynulovat.

Toto řešení ale přináší přinejmenším dva problémy. Za prvé není příliš čisté z hlediska objektově orientovaného programování. Větším a praktickým problémem je ale to, že takto upravená operace *match* nebude fungovat korektně, pokud bude prováděna více souběžně běžícími vlákny. Čítače již totiž nejsou drženy v lokálních proměnných, ale stávají se částí používané datové struktury, která je sdílena mezi vlákny. Tedy i tyto čítače jsou

sdíleny. Vzhledem k efektu nazývanému falešné sdílení (false sharing [8]) se nepodařilo nalézt způsob, jak bez značného zvýšení paměťové náročnosti upravit operaci tak, aby i na více vláknech fungovala tak jak má.

Vzhledem ke zmíněným komplikacím není uvedená modifikace součástí hlavní větve v repozitáři, nýbrž se nachází pouze ve větvi vedlejší.

4.5.3 Ukládání čítačů v hašovací tabulce

Ukázalo se, že vzhledem k častému využívání hašovací tabulky pro ukládání čítačů je velmi důležité, jaké klíče jsou použity. V původní verzi se používal haš objektu Filter, který byl počítán klasickým způsobem z jeho obsahu. Když se místo tohoto haše použil unikátní identifikátor, tedy ID, rychlost celé operace *match* se znatelně zvýšila. A to i v případě, kdy byla hodnota haše spočítána předem. Tento efekt byl zřejmě způsobem vhodnější distribucí hodnot klíčů – především menším (až nulovým) počtem kolizí (ID je unikátní).

Zde se nabízí otázka, proč nebylo pro čítače využito klasické pole. Jedním, spíše teoretickým, problémem je to, že ani klasické pole ani třída *ArrayList* v Javě neumožňují uložení více než 2^{32} prvků, zatímco identifikátor filtru je 64bitové číslo (typ Long). Důležitějším je ovšem následující fakt. Nejen, že nepotřebujeme mít vždy v paměti ke každému filtru čítač, protože mohou existovat filtry, které neobsahují ani jedno splněné omezení pro danou událost (pokud pak čítač v paměti není, předpokládáme, že má hodnotu nula), hlavním důvodem je existence operace *unsubscribe*, která již nepotřebné filtry odstraňuje. Konkrétně se pak může stát, že ačkoliv máme v datové struktuře například 1000 filtrů, nejsou mezi nimi filtry s identifikátory 10, 42 a 312 a naopak naposledy přidaný filtr má ID 1003. V extrémním případě a častém používání operací *subscribe* a *unsubscribe* se pak může stát, že ačkoliv máme celkem pouze několik málo filtrů, je mezi nimi jeden či více s identifikátory o velmi vysokých hodnotách. Pak bychom museli použít zbytečně velké pole, jehož správa by mohla způsobit velké zpomalení oproti prostému použití hašovací tabulky.

Do budoucna je možné uvažovat nad tím, zda by nebylo lepším, i když složitějším, řešením udržovat již nevyužité identifikátory a používat je pro nově přidané filtry nebo dokonce už existujícím filtrům měnit ID tak, aby byla využita všechna celá čísla od minimálního po nejvyšší přidělené. To by mohlo být užitečné i při využívání hašovací tabulky (méně kolizí).

4.5.4 Agregace hodnot z indexů

Základem moha tříd a rozhraní v implementaci algoritmu je rozhraní *Index* s operacemi přidání a odebrání konkrétního omezení a získání kolekce omezení na základě hodnoty atributu. Bylo ale zjištěno, že agregace mnoha hodnot z indexů (prováděna v metodě *getConstraints* třídy *AbstractTypeIndex*, která představuje základní implementaci indexu pro konkrétní typ jako Long či String a všechny operátory) do jedné kolekce má negativní vliv na výkon. Proto byla provedena úprava, takže už nejsou všechny hodnoty vrácené indexy (tedy omezení splňující konkrétní událost) dávány do jedné kolekce, ale místo toho jsou samotné kolekce hodnot přidávány do „obalující" kolekce. Proto operace získání kolekce omezení na základě hodnoty atributu (*getConstraints*) nevrací kolekci omezení, ale kolekci kolekcí omezení.

4.6 Siena

Siena [9] je notifikační systém založený na modelu publish-subscribe. Je určen pro přímé nasazení na sítích s mnoha uzly. Používá výše zmíněný Counting algoritmus, který je zde označen jako „fast forwarding algorithm". Chybí zde operace *unsubscribe* a rozhraní se od naší implementace mírně liší. Siena je psána primárně v programovacím jazyce Java, nevyužívá ovšem některé jeho modernější prvky.

5 Porovnání algoritmů a implementací

Obsahem této kapitoly je porovnání algoritmů Matching Tree a Counting. Co se týče algoritmu Counting, je zde kromě mé implementace uvedena navíc i ta použitá v systému Siena. Kromě porovnání algoritmů a implementací z hlediska výkonu jsou srovnány také další jejich aspekty, především jednoduchost a kvalita kódu.

5.1 Výkonnostní testy

Jelikož zmíněné algoritmy rozhodně nepatří mezi triviální a také proto, abychom získali lepší představu o výkonu v reálných podmínkách (například správa paměti může mít na celkovou rychlost docela velký vliv), bylo rozhodnuto provést měření výkonu pomocí výkonnostního testu. Ačkoliv pro porovnávání publish-subscribe systémů existuje benchmark *jms2009-PS*, je určen pro použití s klasickými systémy založenými na tématech (topics), nikoliv na obsahu [10]. Proto byl v rámci této práce napsán i jednoduchý benchmark. Vzhledem k předpokladu, že nejčastěji prováděnou operací bude *match*, jsme se zaměřili právě na ni.

Pro účely testů rychlosti byl vyvinut jednoduchý adaptér, díky kterému je možné psát samotné benchmarky univerzálně, přičemž pro změnu použité implementace stačí upravit konstantu ve třídě *AdapterFactory*. Adaptér byl psán tak, aby co možná nejméně ovlivňoval rychlost běhu.

Implementaci benchmarku je možné si prohlédnout díky službě GitHub [11].

5.1.1 Testovací prostředí

Testy jsem prováděl na necelé tři roky starém desktopovém PC, které se dá dnes již považovat za mírně výkonově podprůměrné. Procesorem je Intel Core i5-760 (běžící standardně na frekvenci 2.8 GHz), operační paměti je dostatek – 4 GB. Relevantní softwarové vybavení tvoří operační systém Ubuntu 13.04 (v 64bitové variantě), Oracle JDK verze 1.7.0_21 (také v 64bitové variantě). Unixový systém byl vybrán vzhledem k předpokladu, že na podobném bude software běžet v produkčním prostředí.

Benchmark byl spouštěn s následujícími parametry: `-Xss2m` (velikost zásobníku v megabajtech), `-Xms8m` (minimální velikost haldy v megabajtech) a `-Xmx256m` (maximální velikost haldy v megabajtech).

5.1.2 Caliper

Při psaní výkonnostních testů byl použil framework Caliper. Ten poskytuje podporu pro psaní tzv. mikrobenchmarků, které testují dobu provádění menších kusů kódu. Umožňuje vyhnout se ručnímu měření (nemusíme tedy přímo volat metody jako *System.nanotime*) a výpisu výsledků. Především ale eliminuje nutnost řešit opakování benchmarku, které je nutné provádět především v případě, že tento trvá velmi krátkou dobu (řádově jednotky až desítky milisekund a méně). V opačném případě by získané výsledky totiž nemusely být příliš přesné. Caliper toto řeší tak, že vždy nejprve provede měření doby trvání příslušného kusu kódu a na základě této informace pak rozhodne o vhodném počtu opakování. Platí zde nepřímá úměrnost: čím rychleji je zmíněný kód vykonán, tím vyšší počet opakování je nutný k zachování rozumné míry přesnosti. Například pokud vykonávání trvá řádově desítky milisekund, můžeme očekávat, že počet opakování bude v řádu stovek. Pokud naopak je kód proveden za několik sekund, počet opakování bude mnohem nižší, například tři. Z technického hlediska tento počet opakování jednoduše dostaneme jako parametr metody představující jeden elementární benchmark a náš testovaný kus kódu umístíme do cyklu for.

5.1.3 Popis benchmarku

Dále zmiňovaný benchmark (resp. benchmarky) je reprezentován třídou *TwelveLongAttributesLessThan*. Cílem tohoto testu bylo přiblížit se parametrům, které očekáváme v ostrém provozu.

V uvedeném benchmarku má každý predikát právě jeden filtr a každý filtr obsahuje 12 omezení, každé pro jiný atribut. Všechny atributy jsou typu Long a ve všech omezeních je použit jediný operátor – menší než. Všechna omezení se vzájemně liší, jinak řečeno v průběhu naplňování datové struktury do ní není nikdy přidáváno omezení, kde se zde již nachází. Počet predikátů je nastavitelný, pro účely práce byly použity hodnoty 1000, 2000 a 4000. Počet událostí je také možné upravit, ale je zřejmé, že x-násobným zvýšením počtu událostí dosáhneme téměř přesně x-násobné délky trvání.

V benchmarku jsou varianty metod pro různý poměr mezi predikáty splněnými pro danou událost a všemi existujícími predikáty. Tento poměr je uveden přímo v názvu příslušné metody jako jedno z následujících čísel: 25, 50, 75 a 100. Dále jsou tu testy, kde jsou predikáty přidávány v náhodném pořadí. Příslušné metody mají na konci názvu *random*. Je vhodné uvést, že tato náhodná data jsou vygenerována pouze jednou (a uložena do

souboru *random.txt*) a použita pro všechny následné běhy benchmarku. Samozřejmě není vhodné porovnávat mezi sebou výsledky, které vznikly za použití různých náhodných dat (tedy různých souborů *random.txt*). No a konečně je také rozdíl mezi metodami s názvem začínajícím *timeMatch_* a těmi, které začínají *timeMatch2_*. U těch prvních se používá taková událost, že pro všechny predikáty, které ji nesplní, platí, že není splněno ani žádné omezení, které tento predikát obsahuje. Naproti tomu u druhé skupiny metod je použita událost způsobující, že pro všechny predikáty, které ji nesplní, platí, že je splněna většina (konkrétně 11 ze 12) omezení, které tento predikát obsahuje. Tedy kdyby platilo i to dvanácté, celý predikát by byl pro danou událost splněn. Takže v prvním případě je poměr splněných omezení a všech omezení stejný jako poměr splněných predikátů a všech predikátů, zatímco v případě druhém jsou vždy splněna téměř všechna omezení (konkrétně více než $\frac{11}{12}$). Výše uvedený popis je ilustrován obrázkem 5.1.

Příklad pojmenování metod se pak nachází na obrázku 5.2.

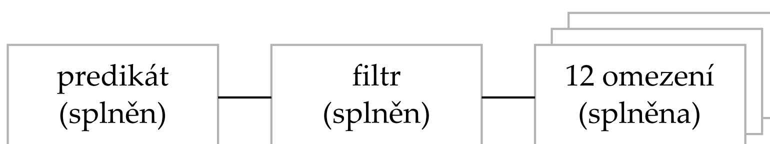
5.1.4 Výsledky benchmarku

Vzhledem k tomu, že elementárních výsledků výkonnostních testů je relativně mnoho (přes 100), a proto není praktické je přímo zde v textu uvést všechny, rozhodl jsem se pro agregaci výsledků dále popsaným způsobem. Pro konkrétní test zjistím minimální časový výsledek přes všechny algoritmy a implementace. Výsledek pro konkrétní implementaci pak vydělím tímto minimem, čímž získám číslo z intervalu $< 1; \infty)$. Výsledné hodnoty pro konkrétní implementaci pak zprůměruji přes všechny čtyři kombinace počtu predikátů a událostí, čímž opět získám číslo ve stejném intervalu.

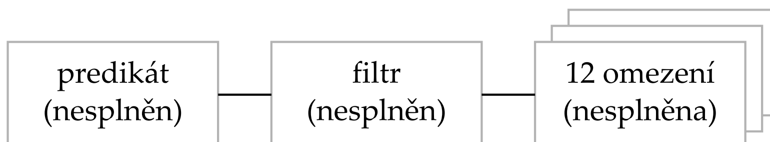
Tabulka 5.1 ilustruje popsaný postup. Měřenou metodou je *Match_25_-random*. P znamená predikát, U událost, MT Matching Tree, CAM Counting algoritmus ve verzi „multicore“, CAS ten stejný algoritmus ve verzi „singlecore“. Multicore verze znamená pouze to, že by měla být schopna bez problému běžet na více procesorech (po drobných úpravách), každopádně všechny algoritmy byly spouštěny pouze na jednom vlákně.

Například číslo 1.077 v druhém řádku ve třetím sloupci znamená, že doba běhu metody s využitím algoritmu Matching Tree byla 1.077krát delší než pokud byla použita nejrychlejší implementace, kterou je konkrétně na tomto řádku naše implementace algoritmu Counting ve verzi singlecore. Tedy čím je tato hodnota nižší (a čím víc se blíží k jedné), tím je výsledek lepší.

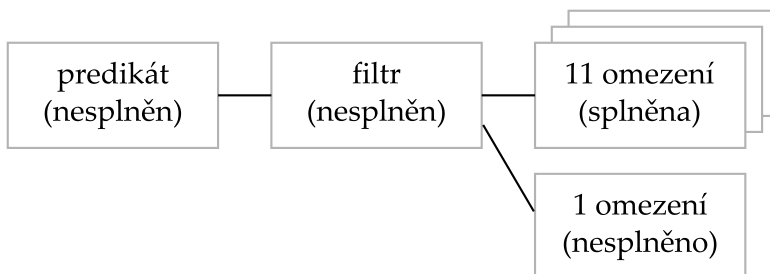
Splněný predikát vypadá vždy následovně:



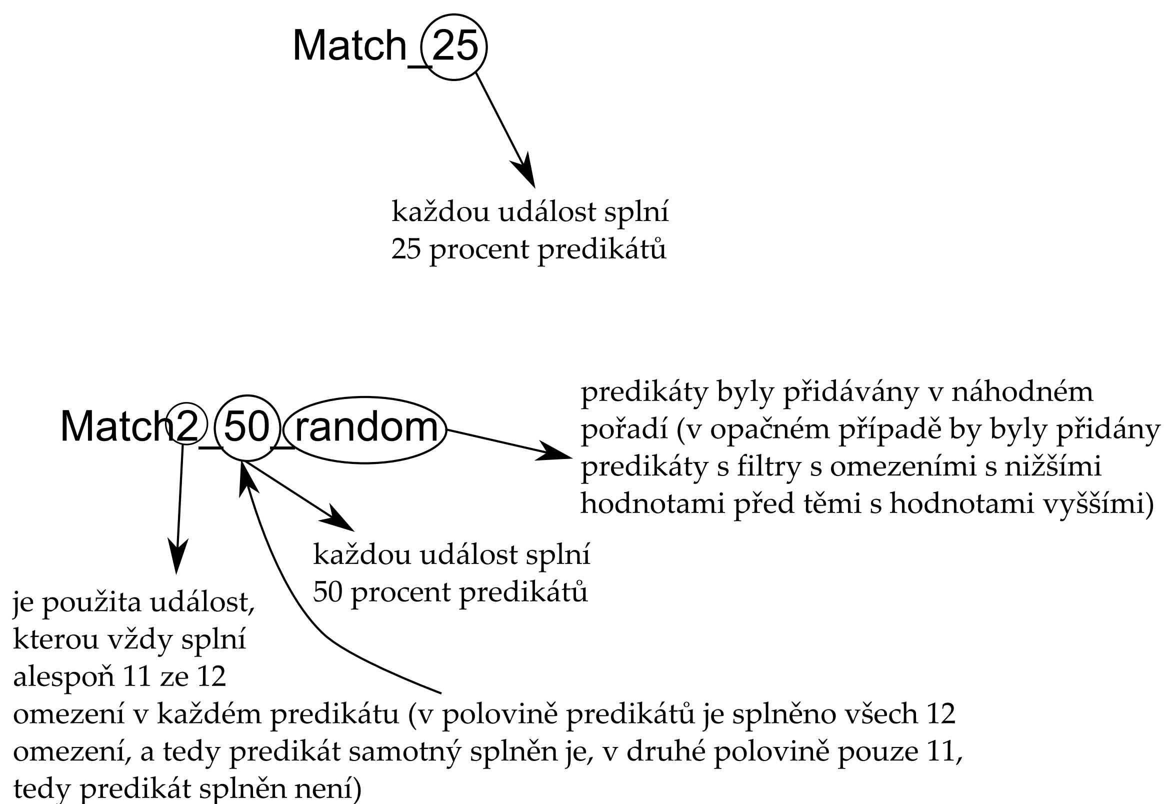
Nesplněný predikát vypadá v případě metod
začínajících timeMatch_ takto:



V případě metod začínajících timeMatch_ ovšem
vypadá takto:



Obrázek 5.1: Rozdíl mezi predikáty použitými v „timeMatch_“ metodách a „timeMatch2_“ metodách



Obrázek 5.2: Pojmenovávání metod v benchmarku

Match_25_random	minimum	MT	CAM	CAS	Siena
1000 P, 1000 U	220 ms	1.077	1.564	1.000	1.332
1000 P, 100 U	21.8 ms	1.046	1.564	1.000	1.326
2000 P, 100 U	45.8 ms	1.000	1.644	1.028	1.264
4000 P, 100 U	93.7 ms	1.026	1.750	1.000	1.292
průměr		1.037	1.631	1.007	1.304
počet „vítězství“		1	0	3	0

Tabulka 5.1: Příklad agregace elementárních výsledků benchmarku

Místo všech šestnácti dostupných výsledků vezmu vždy nakonec pouze čtyři průměry uvedené v předposledním řádku. Kromě toho uvádím také tzv. *počet vítězství*, což je číslo, které udává, kolikrát byla daná implementace nejrychlejší ze všech.

Průměry pro všechny metody jsou pak uvedeny v tabulce 5.2.

Poslední tabulkou je 5.3, kde jsou uvedeny všechny výsledky pro naši implementaci Counting algoritmu ve verzi multicore.

Je také možné srovnat různé algoritmy a implementace z hlediska toho, jaká je jejich rychlost při různých počtech predikátů. Takové srovnání ukazuje obrázek 5.3. Z celkových třinácti metod jsem kvůli úspoře místa vybral pouze osm. Údaje na svislé ose ukazují délku trvání testu, tedy menší hodnota je lepší.

Poslední grafy na obrázku 5.4 ilustrují rozdíly v době trvání operace match v závislosti na poměru splněných predikátů.

Je z nich také patrné, že rychlost závisí především na počtu splněných omezení, obzvláště u algoritmu Counting (který používá i Siena). Doba provádění metod začínajících „Match2“ (tedy těch používajících událost, kterou vždy splní téměř všechna omezení) je totiž téměř nezávislá na počtu splněných predikátů.

5.1.5 Vyhodnocení výsledků

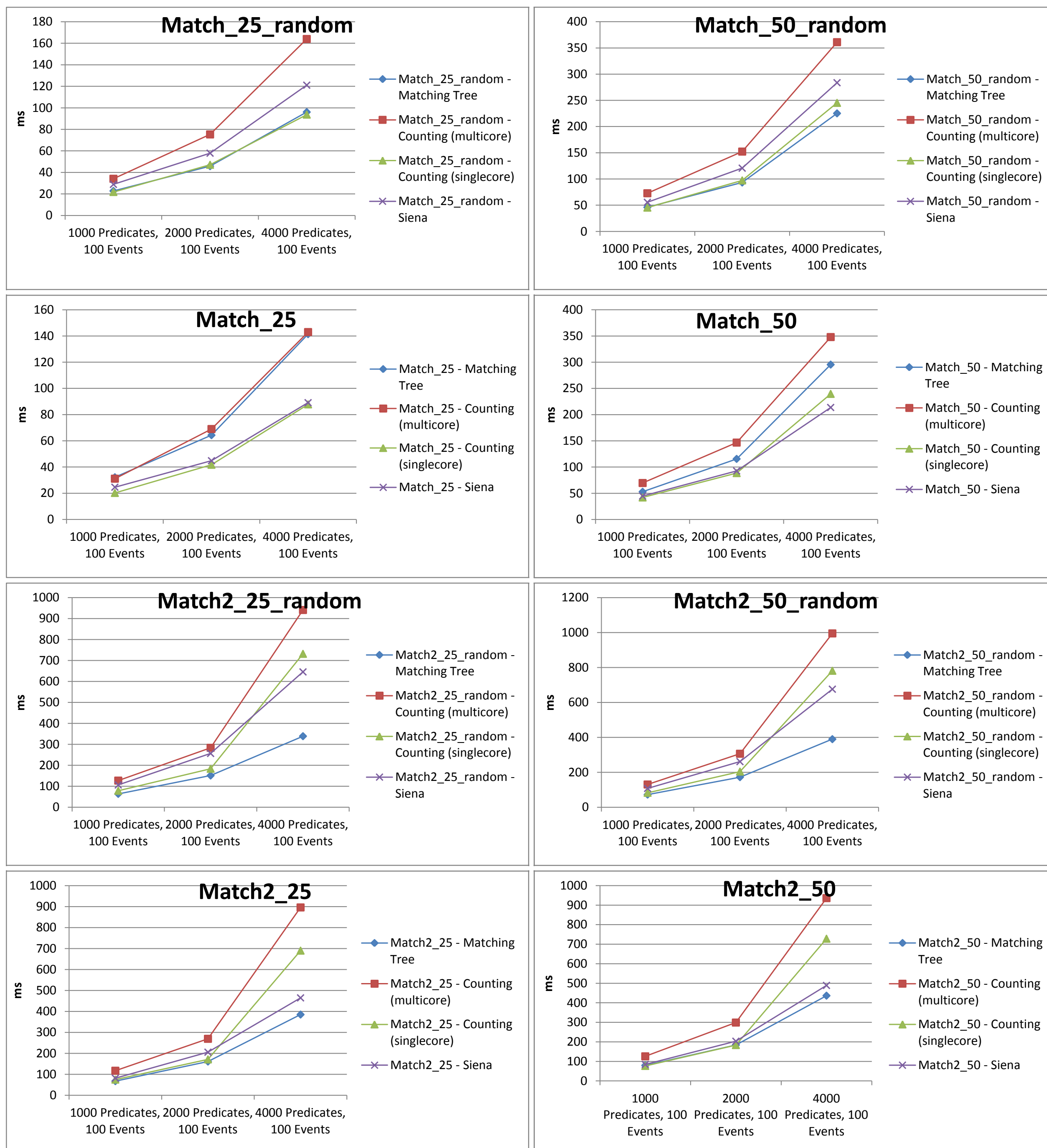
Algoritmus Counting (ve verzi multicore) byl tedy jasně nejpomalejší, naopak jako nejrychlejší se ukázala implementace algoritmu Matching Tree. Překvapivý je rozdíl mezi naší implementací algoritmu Counting a implementací obsaženou v systému Siena, jedná se totiž o různé implementace téhož algoritmu. Důvodem může být fakt, že Siena používá jednodušší a úspornější datové struktury, díky čemuž dochází k efektivnějšímu využití vyrovnávací paměti. Siena také obsahuje navíc některé optimalizace. Jejich

	MT	CAM	CAS	Siena
Match_25_random	1.037	1.631	1.007	1.304
Match_50_random	1.003	1.608	1.034	1.246
Match_75_random	1.004	1.716	1.170	1.337
Match2_25_random	1.000	2.145	1.463	1.727
Match2_50_random	1.000	1.958	1.360	1.542
Match2_75_random	1.000	1.874	1.295	1.420
Match_25	1.583	1.579	1.000	1.131
Match_50	1.302	1.656	1.030	1.047
Match_75	1.172	1.734	1.136	1.096
Match2_25	1.000	1.878	1.266	1.236
Match2_50	1.009	1.756	1.169	1.100
Match2_75	1.038	1.739	1.149	1.032
Match_100	1.109	1.741	1.180	1.010
průměr	1.097	1.770	1.174	1.248
počet „vítězství“	25	0	21	6

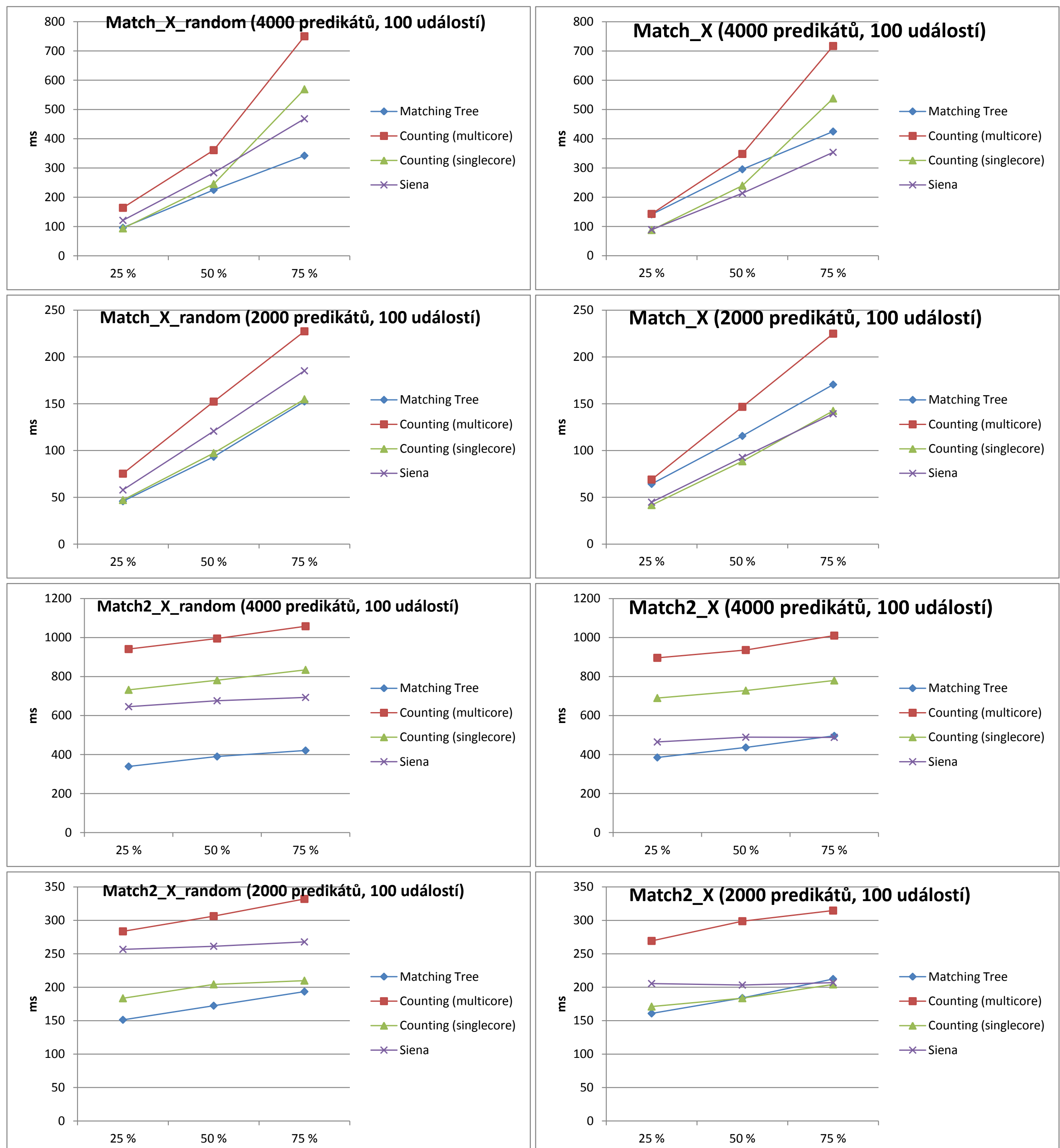
Tabulka 5.2: Stručné výsledky výkonnostních testů

	1000 P, 1000 U	1000 P, 100 U	2000 P, 100 U	4000 P, 100 U
Match_25_random	344	34.1	75.3	164
Match_50_random	729	72.8	152.3	361
Match_75_random	1104	111.2	227.4	750
Match2_25_random	1245	127.1	283.6	941
Match2_50_random	1317	130.4	306.4	995
Match2_75_random	1372	138.3	332.1	1058
Match_25	300	31.1	69	143
Match_50	705	69.6	146.7	348
Match_75	1053	110.2	224.9	717
Match2_25	1181	117.1	269.3	896
Match2_50	1267	125.9	298.8	936
Match2_75	1417	133.1	314.7	1010
Match_100	1394	139.5	336.2	1058

Tabulka 5.3: Kompletní výsledky pro Counting algoritmus, verze multicore (hodnoty jsou v milisekundách)



Obrázek 5.3: Srovnání při různých počtech predikátů



Obrázek 5.4: Srovnání při různých poměrech splněných predikátů

vlivem pak podle typu dat může dojít buď ke zrychlení nebo naopak k mírnému zpomalení výpočtu.

Jak již bylo uvedeno, u algoritmu Counting je doba provádění operace *match* závislá především na počtu splněných omezení pro danou událost, což potvrdily i naměřené hodnoty.

Při čtyřech tisících uložených predikátech a celkem 48 000 vzájemně různých omezeních tak stihneme za jednu sekundu zpracovat něco mezi stem a tisícem událostí. Navíc měření probíhalo na starším stroji; s využitím modernějšího procesoru bychom tak mohli dosáhnout zrychlení v řádech desítek procent. Použité algoritmy jsou také poměrně triviálně paralelizovatelné, každou událost totiž můžeme zpracovávat jiným procesorem.

Benchmark byl spuštěn také na starším stroji (notebook s procesorem Core 2 Duo T5750) s odlišným operačním systémem (Windows). Zde se výsledky značně lišily – v relativním srovnání byl algoritmus Matching Tree o mnoho pomalejší, naopak naše implementace algoritmu Counting si vedla lépe.

5.2 Celkové srovnání

Co se týče rychlosti, nejlépe dopadla naše implementace algoritmu Matching Tree, za ní se pak umístila naše implementace algoritmu Counting ve verzi „singlecore" a na třetím místě pak skončila implementace stejného algoritmu používaná systémem Siena. Pokud ale porovnám námi vytvořené implementace Matching Tree a Counting algoritmu z programátorského hlediska, tak ta druhá je z mého pohledu rozhodně praktičtější. Jednak proto, že u Matching Tree je z podstaty testů nutné provádět mnoho explicitních přetypování a také kvůli potřebě porovnávání testů různých typů mezi sebou (jak jsem popsal v sekci 3.4). I samotný algoritmus považuji za jednodušší na pochopení. Tato skutečnost nakonec převážila nad rychlostí a proto byl pro naše účely vybrán právě tento.

Mezi naší implementací Counting algoritmu a implementací ze systému Siena (která je z uživatelského hlediska představovaná především třídou *SFFTable*) je několik rozdílů. Za prvé operace *match* v Sieně nevrací seznam konzumentů, ale volá metodu z dodaného objektu. Také zde často nejsou využity moderní prvky jazyka Java. V systému Siena jsou například všechny možné typy hodnot atributu (Long, String, Double atd.) uvedeny přímo ve třídě *AttributeValue* místo použití generických typů. Dále se tu často v jednom souboru vyskytuje mnoho tříd a celkově není kód příliš přehledný a pochopitelný. A snad nejvýznamnějším rozdílem je naprostá

absence operace *unsubscribe*. Na druhou stranu jsou zde některé zajímavé optimalizace, které ovšem mohou být ve většině případů do naší implementace doplněny také.

Mohlo by se zdát, že implementace Counting algoritmu ve verzi single-core, zvlášť v dnešní době široce rozšířených víceprocesorových systému, příliš užitečná není. Pokud ale máme k dispozici dostatek paměti, je zde možnost udržovat pro každý procesor (nebo dokonce úplně samostatný systém) vlastní datovou strukturu. Operace *subscribe* a *unsubscribe* pak sice musíme provádět zároveň na všech datových strukturách, ale *match* můžeme spouštět nad kteroukoliv z nich a vždy bychom měli získat stejný výsledek.

6 Zasazení do systému Ngmon

Závěrečnou částí práce je využití algoritmu Counting v monitorovacím systému Ngmon [1]. Hlavním úkolem tohoto systému je sběr dat od tzv. senzorů poskytujících monitorovací data ve formě událostí, předzpracování těchto událostí a jejich následné přeposílání zainteresovaným klientům. Z hlediska modelu publish-subscribe představují senzory producenty a klienti konzumenty.

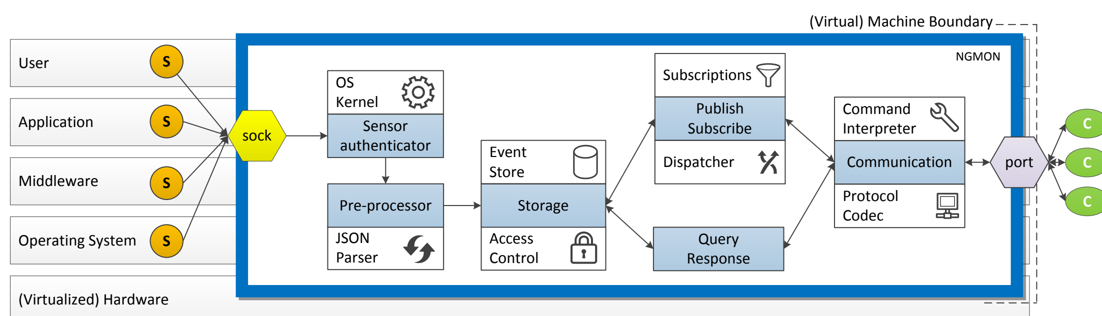
Komunikace se systémem Ngmon probíhá skrze počítačovou síť. Vzhledem k požadavku na výkon a rychlost odezvy se zde používá neblokující (asynchronní) komunikace, a to především pro samotné přeposílání událostí klientům pomocí serveru. Pro tento účel byl využit framework Netty [12].

Jednotlivé činnosti prováděné systémem Ngmon při zpracování události pocházející od senzoru budu ilustrovat za pomoci schématu na obrázku 6.1.

Na začátku procesu jsou tedy výše zmíněné senzory (v obrázku na levé straně) posílající monitorovací události systému Ngmon. Tyto mohou fungovat na úrovni uživatelské, aplikační, middleware nebo operačního systému. Zpracování těchto událostí pak probíhá v následujících krocích:

1. Autentizace senzoru (*Sensor authenticator*). Zde je ověřeno, že daný senzor patří mezi důvěryhodné, a tedy událost, která z něj přichází, nebyla podvržena nějakou třetí stranou.

2. Předzpracování události (*Pre-processor*). V tomto kroku se provádí především převod příchozího proudu bytů, v našem případě řetězce



Obrázek 6.1: Schéma systému Ngmon [13]

ve formátu JSON, na objekt typu *Event*. Může zde být také přidání doplňujících informací, například aktuálního času.

3. Vložení události do perzistentního úložiště (*Storage*). Typicky se jedná o klasickou relační databázi. Tento krok je nutný mj. kvůli tomu, abychom byli schopni později přeposlat událost klientovi pokud mu nemohla být doručena například z důvodu jeho nedostupnosti.

4. Zjištění, kterým klientům máme událost přeposlat (neboli kteří klienti se přihlásili k odběru příslušného typu událostí) (*Publish Subscribe*). Součástí tohoto kroku je také příprava samotného procesu odeslání. Zde je právě potřeba řešit dříve zmíněný problém publish-subscribe, což představuje hlavní náplň této práce.

5. Odeslání události klientovi (*Communication*).

Pro implementaci jednotlivých kroků byl využit návrhový vzor *pipes and filters* [14]. Zpracování událostí od senzorů probíhá uvnitř tzv. pipeline („roury") vytvořené třídou *DefaultPipelineFactory*. V rámci této pipeline je událost postupně v určitém pořadí zpracovávána pomocí tzv. handlerů (handlers). Konkrétně pro ukládání do databáze je použit handler s názvem *Store*, pro publish-subscribe je zde handler *DetermineRecipient* a pro samotné odeslání *SubmitToDispatcher*.

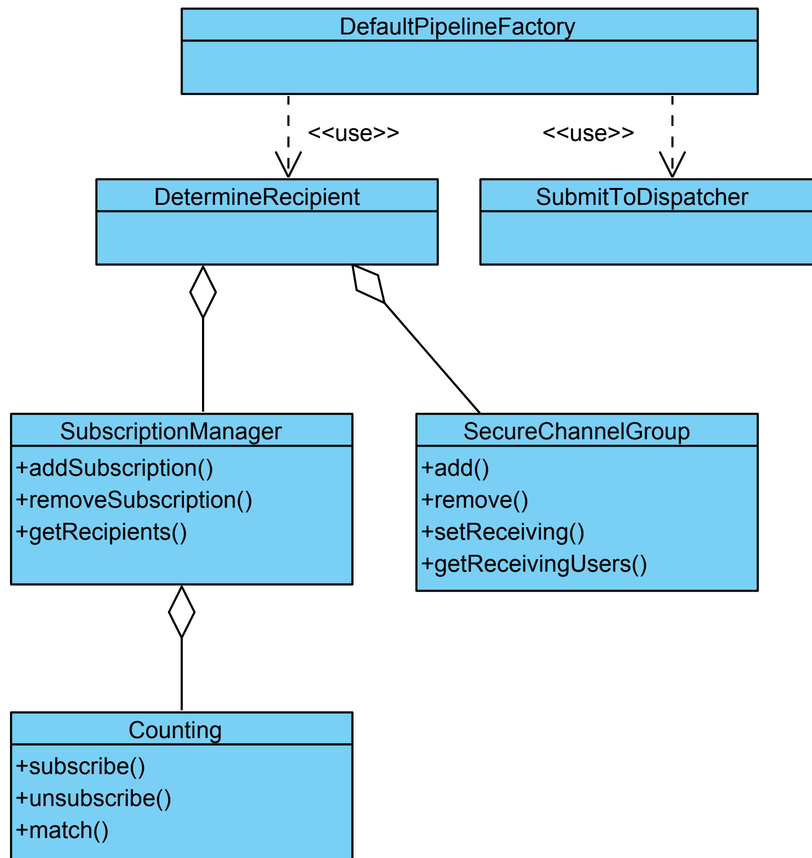
Můj úkol měl tři části. První bylo doplnění implementace předposlední fáze zpracování, tedy *Publish Subscribe*. V kódu jde o handler *DetermineRecipient*. Druhou částí byla implementace příjmu a zpracování požadavků od klienta na straně serveru. Nakonec jsem také napsal jednoduchého klienta pro přímé využití v aplikacích.

6.1 Publish-subscribe

Zde bylo potřeba doplnit implementaci handleru *DetermineRecipient* zajišťujícího korektní přeposílání událostí od senzorů ke klientům. Na obrázku 6.2 se nachází diagram hlavních tříd zabezpečujících požadovanou funkcionalitu.

Průběh činnosti metody *handle* zpracovávající událost lze rozdělit do tří částí:

1. Zjištění, kteří klienti se dříve pomocí operace subscribe přihlásili k odběru tohoto typu zpráv. K tomu je použita třída *SubscriptionManager*



Obrázek 6.2: Třídy zajišťující přeposílání událostí od senzorů příslušným klientům

spravující asociace mezi typy zpráv (které jsou představovány predikáty) a příslušnými klienty. Tato ke své činnosti už přímo využívá třídy algoritmu Counting.

2. Nyní je zjištěno, kteří z těchto klientů jsou aktuálně připojeni. Pouze jim bude událost přeposlána. Kolekce právě připojených klientů je udržována třídou *SecureChannelHandler*.

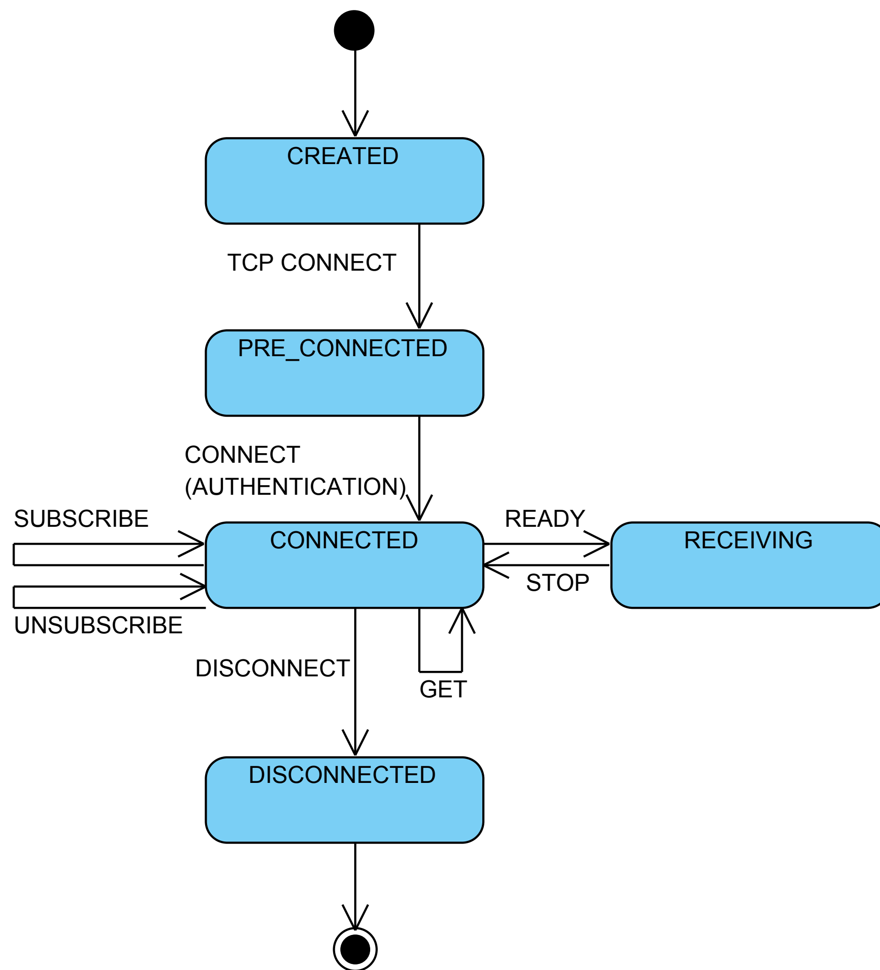
3. Událost je doplněna o seznam klientů, kterým má být událost přeposlána a jsou zároveň připojeni a předána následujícímu handleru, *SubmitToDispatcher*, k samotnému odeslání.

6.2 Zpracování požadavků klienta na serveru

Vzhledem k faktu, že klient komunikuje se serverem (systémem Ngmon) pomocí počítačové sítě, bylo nutné dohodnout se na konkrétním protokolu, pomocí něhož bude moci klient posílat požadavky na přihlášení k odběru událostí, na začátek a konec příjmu událostí a podobně. Náš protokol je inspirován textovým protokolem s názvem STOMP [15]. Na obrázku 6.3 je zobrazen stavový diagram klienta ilustrující průběh komunikace mezi klientem a serverem.

Na začátku se klient nachází ve stavu *created*. Následuje připojení k serveru Ngmon, v diagramu představované přechodem s názvem *TCP connect*, čímž klient přejde do stavu *pre_connected*. V tomto stavu může dále poslat zprávu typu *connect* s připojenými autentizačními údaji – přihlašovacím jménem a heslem. Po ověření a potvrzení serverem přechází klient do stavu *connected*. Nyní je možné beze změny stavu provádět operace *subscribe* (přihlášení k odběru událostí), *unsubscribe* (zrušení konkrétního odběru událostí) a *get* (požadavek na zaslání nepřeposlaných událostí, typicky z důvodu předchozí nedostupnosti klienta). Poté, co se klient přihlásí k odběru událostí pomocí subscribe, zavolá *ready* a od tohoto okamžiku může začít přijímat příslušné události produkované senzory. Jakmile se rozhodne jejich příjem zastavit, použije zprávu *stop* a tím se dostane opět do stavu connected. Nakonec pošle serveru zprávu *disconnect* a po potvrzení provede samotné odpojení.

Všechny uvedené příkazy jsou serverem explicitně potvrzovány. Z tohoto důvodu klient ve skutečnosti pracuje navíc s „mezistavy“, ve kterých se nachází od okamžiku odeslání požadavku na server do doby přijetí odpovídajícího potvrzení.



Obrázek 6.3: Stavový diagram klienta

typ zprávy (direktiva)	význam zprávy	dodatečná data	odpověď serveru
CONNECT	autentizace klienta vůči serveru	uživatelské jméno a heslo	CONNECTED (a connection ID) nebo ERROR
SUBSCRIBE	přihlášení klienta k odběru událostí (produkovaných senzory) splňovaných zadaným predikátem	predikát	ACK (a subscription ID) nebo ERROR
UNSUBSCRIBE	zrušení odebíraní událostí (zrušení efektu některé z předchozích zpráv SUBSCRIBE)	ID přihlášení (subscription ID)	ACK nebo ERROR
READY	zahájení příjmu událostí	–	
STOP	ukončení (přerušení) příjmu událostí	–	
GET	požadavek na zaslání událostí, které nebyly z nějakého důvodu (např. nedostupnosti klienta) přeposlány dříve	–	
DISCONNECT	odpojení klienta od serveru	–	

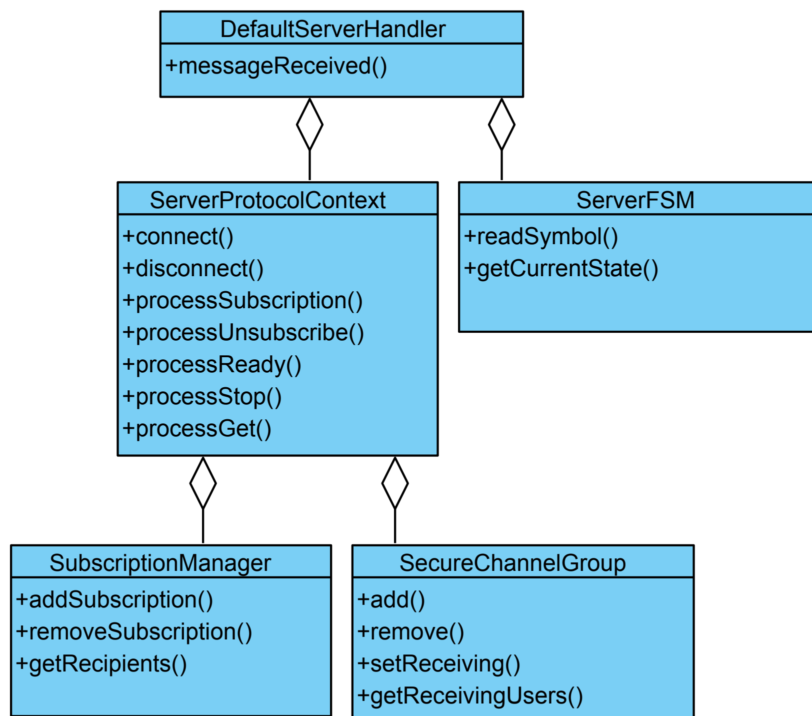
Tabulka 6.1: Popis jednotlivých typů zpráv využívaných klientem pro komunikaci se serverem

Formát samotných zpráv je velice jednoduchý. Prvním údajem je délka celé zprávy (kvůli rekonstrukci zprávy u příjemce). Dále následuje tzv. direktiva neboli typ zprávy. Zde je použit výčtový typ a samotné hodnoty jsou například výše uvedené *connect*, *subscribe*, *unsubscribe*, *ready*, *stop*, *get* nebo *disconnect*. Server dále používá pro potvrzování těchto zpráv typy jako *connected*, *error* či *ack*. Poslední a také hlavní částí zprávy je její tělo, což může být obecně jakýkoliv seznam bytů, v našem případě jde o řetězec ve formátu JSON. Pomocí tohoto pole předáváme dodatečné informace jako například autentizační údaje u zprávy typu *connect*, predikát u zpráv typu *subscribe* nebo *subscription ID* posílané serverem klientovi v potvrzující zprávě mající typ *ack*.

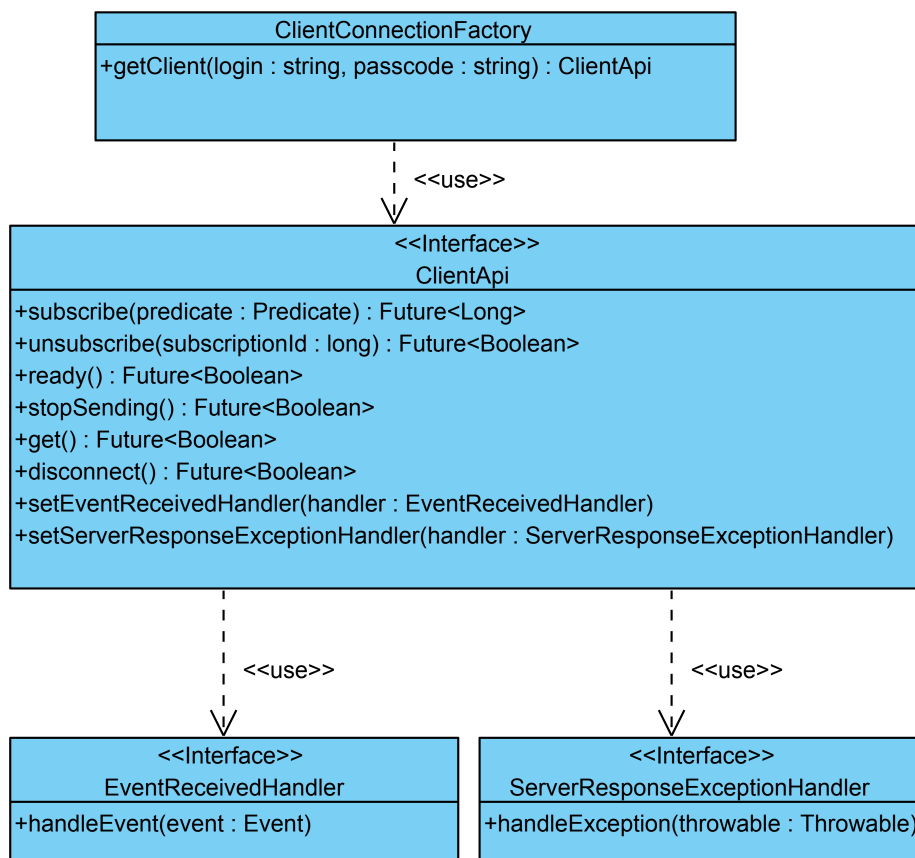
Všechny aktuálně používané zprávy jsou uvedeny v tabulce 6.1.

Zpracování těchto zpráv na serveru mají na starosti třídy na obrázku 6.4.

Pomocí třídy *DefaultServerHandler* provádíme příjem zpráv od klienta. Stav serveru je udržován objektem třídy *ServerFSM* představujícím sta-



Obrázek 6.4: Třídy zpracovávající na serveru požadavky klienta



Obrázek 6.5: Rozhraní klienta serveru Ngmon

vový automat. Ostatní reakce má na starosti třída *ServerProtocolContext* využívající další dvě třídy: *SubscriptionManager* (pro zpracování zpráv typu subscribe a unsubscribe) a *SecureChannelGroup* (pro udržování seznamu přihlášených klientů a ukládání informací o tom, kteří z nich právě přijímají zprávy, tedy se nachází ve stavu *sending*).

6.3 Klient

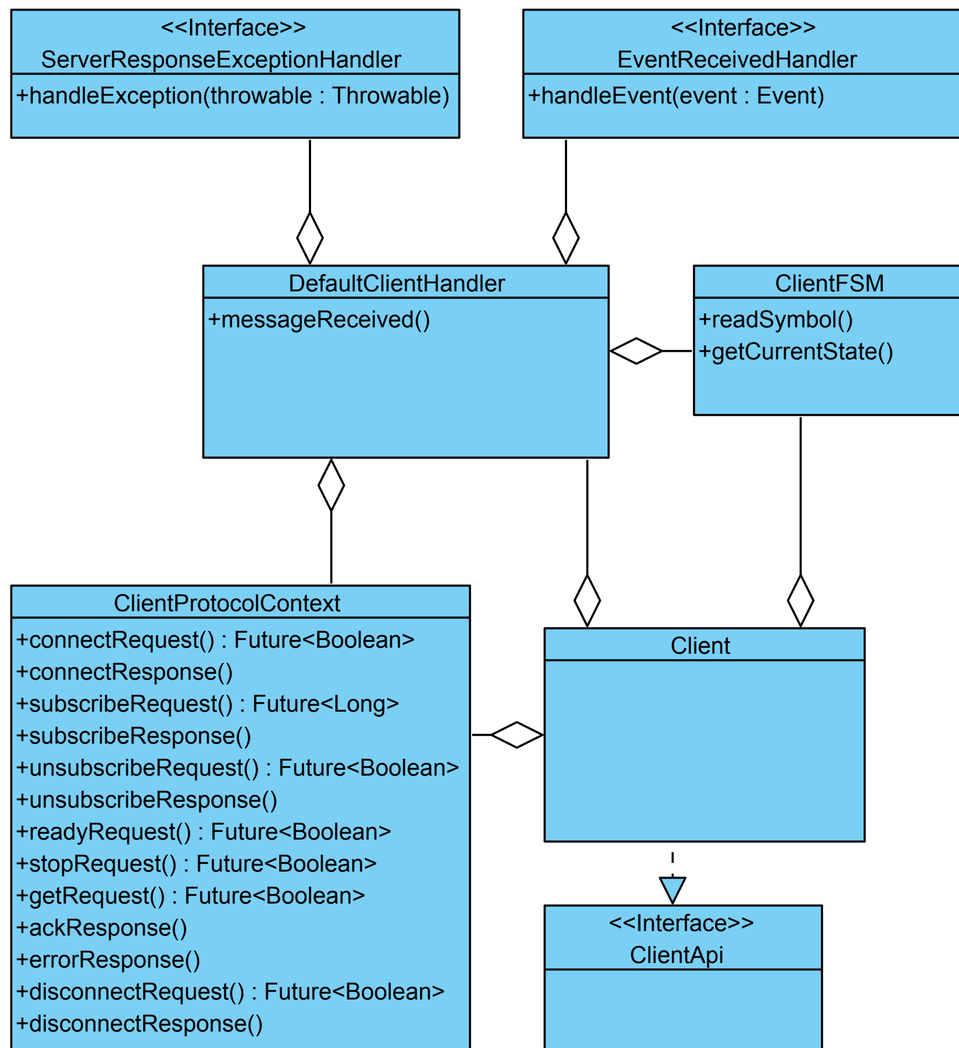
Pro snadnější využívání služeb serveru byl v rámci práce také napsán jednoduchý klient. Nejedná se o plnohodnotnou aplikaci, ale o sadu tříd či knihovnu pro následné využití v uživatelských aplikacích. Rozhraní klienta je znázorněno na obrázku 6.5.

Uživatel tedy pomocí třídy *ClientConnectionFactory* získá objekt třídy implementující rozhraní *ClientApi*, na kterém pak jednoduše volá požadované operace. Před použitím metody *ready* ještě nastaví pomocí *setEventReceivedHandler* objekt zpracovávající příchozí zprávy.

Všechny vhodné metody se chovají asynchronně a vracejí objekt typu Future (příslušná třída je vnitřně implementována pomocí synchronizační konstrukce CountDownLatch), díky čemuž je možné počkat na kompletní dokončení operace a získat její výsledek. To nastane ve chvíli, kdy klient obdrží od serveru potvrzující zprávu a zpracuje ji.

Implementace odesílání požadavků na server a zpracování přicházejících odpovědí je z hlediska návrhu tříd poměrně jednoduchá, jak je naznačeno na obrázku 6.6.

Hlavními třídami zde jsou *Client* implementující zmíněné rozhraní *ClientApi* a *DefaultClientHandler* přijímající zprávy od serveru. Tyto používají stavový automat reprezentovaný třídou *ClientFSM* a „výkonnou" třídu *ClientProtocolContext*, jejímž úkolem je reagovat na příkazy od uživatele třídy Client a na odpovědi od serveru. Třída Client využívá metody typu „request", které provádějí zasílání příslušných požadavků na server a vrací jejich výsledek „obalený" pomocí rozhraní Future. DefaultClientHandler pak volá metody typu „response" reagující na odpovědi od serveru, které následně tuto odpověď předají uživateli klienta pomocí objektu typu Future dříve vráceného odpovídající metodou typu „request".



Obrázek 6.6: Diagram tříd klienta z hlediska implementace

7 Závěr

Hlavní téma práce bylo řešení problému matchingu založeného na obsahu zpráv v modelu publish-subscribe s předpokládaným primárním využitím pro monitorovací události. Byly představeny a implementovány dva existující algoritmy – Maching Tree a Counting – a tyto spolu s již existujícím řešením obsaženém v systému Siena analyzovány a porovnány, přičemž hlavní důraz byl kladen na rychlost zpracování a přeposílání přicházejících zpráv. Pro tento účel byl napsán výkonnostní test (benchmark).

Ačkoliv algoritmus Matching Tree byl obecně rychlejší, jako výhodnější z implementačního hlediska se ukázala naše implementace algoritmu Counting, a to i oproti té obsažené v systému Siena. Také je nutné mít na paměti, že se jedná o syntetický test a v reálném provozu na jiných strojích a systémech se může rychlost jednotlivých implementací značně lišit.

Naše implementace algoritmu Counting pak byla použita v monitorovacím systému Ngmon pro přeposílání událostí od tzv. senzorů (producentů) ke klientům (konzumentům). Dále byl implementován protokol pro komunikaci mezi klientem a serverem a také napsán jednoduchý klient – knihovna pro použití na straně konzumenta. Tato část systému je tedy plně funkční, jak dokazují i obsažené jednotkové testy. Veškeré zdrojové kódy jsou k dispozici v on-line repozitáři služby GitHub [1].

Literatura

[1] Tovarňák D. et al. Ngmon - new generation monitoring daemon. URL: http://github.com/ngmon.

[2] V. S. Subrahmanian. Multidimensional data structures. URL: coitweb.uncc.edu/~ras/MDB/ch4-VS.pdf.

[3] Novák S. Matching tree. URL: http://github.com/ngmon/publish-subscribe-matching-tree.

[4] Novák S., Tovarňák D. a Vašeková A. Counting algorithm. URL: http://github.com/ngmon/ngmon-pub-sub.

[5] Marcos K. Aguilera, Robert E. Strom, Daniel C. Sturman, Mark Astley a Tushar D. Chandra. Matching events in a content-based subscription system. In *Proceedings of the eighteenth annual ACM symposium on Principles of distributed computing*, PODC '99, s. 53–61, New York, NY, USA, 1999. ACM. URL: http://doi.acm.org/10.1145/301308.301326, {doi:10.1145/301308.301326}.

[6] Antonio Carzaniga a Alexander L. Wolf. Forwarding in a content-based network. In *Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, SIGCOMM '03, s. 163–174, New York, NY, USA, 2003. ACM. URL: http://doi.acm.org/10.1145/863955.863975, {doi:10.1145/863955.863975}.

[7] Kevin Dolan. Interval tree java implementation. URL: http://thekevindolan.com/2010/02/interval-tree/index.html.

[8] William J. Bolosky a Michael L. Scott. False sharing and its effect on shared memory performance. In *USENIX Systems on USENIX Experiences with Distributed and Multiprocessor Systems - Volume 4*, Sedms'93, s. 3–3, Berkeley, CA, USA, 1993. USENIX Association. URL: http://dl.acm.org/citation.cfm?id=1295480.1295483.

[9] Siena: A wide-area event notification service. URL: http://www.inf.usi.ch/carzaniga/siena/.

[10] Kai Sachs, Stefan Appel, Samuel Kounev a Alejandro Buchmann. Benchmarking publish/subscribe-based messaging systems. In *Proceedings of the 15th international conference on Database systems for*

advanced applications, DASFAA'10, s. 203–214, Berlin, Heidelberg, 2010. Springer-Verlag. URL: http://dl.acm.org/citation.cfm?id=1880853.1880878.

[11] Novák S. Publish-subscribe benchmark. URL: https://github.com/ngmon/ngmon-pub-sub-benchmark.

[12] Netty. URL: http://netty.io/.

[13] Daniel Tovarňák a Tomáš Pitner. Towards multi-tenant and interoperable monitoring of virtual machines in cloud. In *Proceedings of 14th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, s. 436–442, Los Alamitos (CA), 2012. URL: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=6481063.

[14] Gregor Hohpe a Bobby Woolf. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[15] Stomp - the simple text oriented messaging protocol. URL: http://stomp.github.io/.

A Zdrojové kódy

A.1 Algoritmy a benchmark

Zdrojové kódy algoritmu Matching Tree jsou na adrese `https://github.com/ngmon/publish-subscribe-matching-tree`, algoritmu Counting na `https://github.com/ngmon/ngmon-pub-sub`, benchmark pak na `https://github.com/ngmon/ngmon-pub-sub-benchmark`. Stažení z repozitáře je možné pomocí programu Git[1] příkazem `git clone`, tedy například `git clone https://github.com/ngmon/publish-subscribe-matching-tree`, případně `git clone https://github.com/ngmon/publish-subscribe-matching-tree.git`. Zdrojové kódy jsou strukturovány pro použití s nástrojem Maven[2], tedy pro zkompilování je možné jej použít pomocí příkazu `mvn package`, lépe ale rovnou `mvn install`. Ovšem co se algoritmů týče, nejsou nijak spustitelné (jedná se pouze o knihovny), tedy přímo je možné spustit pouze testy (`mvn test`). Ty se ovšem při použití `mvn package`, resp. `mvn install`, spouští automaticky, takže to trochu postrádá smysl.

Pro úspěšné zkompilování benchmarku je potřeba nejprve přidat oba algoritmy do lokálního repozitáře příkazem `mvn install`.

Dále je také nutné přidat do lokálního repozitáře nástroje Maven systém Siena. To můžeme provést následovně.

1. Nejprve z Git repozitáře na adrese `http://www.inf.usi.ch/carzaniga/siena/software/siena.git` (opět pomocí příkazu `git clone`) software stáhneme.

2. Celý adresář `src/siena` přesuneme do adresáře `src/main/java` (neexistující adresáře vytvoříme). Cesta ke zdrojovému kódu (nyní `src/main/java/siena`) tak bude odpovídat konvencím nástroje Maven.

3. V „hlavním“ adresáři (tedy v tom, kde se nachází podadresář `src`) vytvoříme soubor pom.xml s následujícím obsahem (tento kód je k nalezení i uvnitř souboru `README.md` u benchmarku):

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
      http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
```

1. `http://git-scm.com/`
2. `http://maven.apache.org/`

```
    <groupId>siena</groupId>
    <artifactId>siena</artifactId>
    <version>2.0.3</version>
    <name>Siena</name>

    <build>
      <plugins>
        <plugin>
          <groupId>org.apache.maven.plugins</groupId>
          <artifactId>maven-compiler-plugin</artifactId>
          <version>2.5.1</version>
          <configuration>
            <source>1.7</source>
            <target>1.7</target>
          </configuration>
        </plugin>

        <plugin>
          <groupId>org.apache.maven.plugins</groupId>
          <artifactId>maven-resources-plugin</artifactId>
          <version>2.6</version>
          <configuration>
            <encoding>UTF-8</encoding>
          </configuration>
        </plugin>
      </plugins>
    </build>
</project>
```

4. Nyní již můžeme pomocí příkazu `mvn install` Sienu zkompilovat a přidat do lokálního repozitáře.

Následně vytvoříme balík typu jar příkazem `mvn package` uvnitř adresáře s benchmarkem. Tímto by se v podadresáři target měly vytvořit dva soubory – jeden menší a druhý, jehož název by měl končit *with-dependencies*. Právě ten použijeme. Ukázkové spuštění benchmarku pak může vypadat následujícím způsobem (tento příkaz je nutné zadávat v adresáři target):

```
java -cp benchmark-0.0.1-SNAPSHOT-jar-with-dependencies.jar
com.google.caliper.Runner
cz.muni.fi.ngmon.pubsub.benchmark.
   TwelveLongAttributesLessThan
-DPREDICATE_COUNT=4000 -DEVENT_COUNT=100
```

Třída *com.google.caliper.Runner* spouští benchmark, za ní následuje plně kvalifikovaný název třídy obsahující benchmark a poté případné

parametry závisející na konkrétním benchmarku. V tomto případě počet predikátů a počet událostí.

Můžeme také přidat další parametry, pomocí kterých se dá specifikovat například minimální a maximální velikost haldy (při používání příkazu `java` přímo to odpovídá parametrům `Xms` a `Xmx`) či velikost zásobníku (odpovídá parametru `Xss`). Příklad:

```
java -cp benchmark-0.0.1-SNAPSHOT-jar-with-dependencies.jar
com.google.caliper.Runner
cz.muni.fi.ngmon.pubsub.benchmark.
   TwelveLongAttributesLessThan
-DPREDICATE_COUNT=4000 -DEVENT_COUNT=100
-JmemoryMax=-Xmx256m -JmemoryMin=-Xms8m -Jstack=-Xss2m
```

Kromě toho je možné si připravit kód pro spuštění několika benchmarků za sebou. Příklad je uveden ve třídě *Main*, kterou spustíme tímto příkazem:

```
java -cp benchmark-0.0.1-SNAPSHOT-jar-with-dependencies.jar
cz.muni.fi.ngmon.pubsub.benchmark.Main
```

Pro změnu používané datové struktury stačí upravit atribut *dataStructure* ve třídě *AdapterFactory* a poté projekt znovu zkompilovat a sestavit (např. pomocí příkazu `mvn clean package`). Pro použití naší implementace algoritmu Counting ve verzi singlecore je nutné nejprve stáhnout příslušnou větev a kód znovu zkompilovat a umístit do lokálního repozitáře (příkazy `git checkout -b singlecore origin/singlecore` a následným `mvn clean install` v adresáři s touto implementací).

A.2 Systém Ngmon

Zdrojové kódy tohoto systému se nachází v repozitáři na adrese `https://github.com/ngmon/ngmon` a stáhneme je opět snadno příkazem `git clone https://github.com/ngmon/ngmon`. Je možné, že aktuální změny (vzhledem k datu psaní této práce) se zatím nenacházejí v hlavní větvi, ale pouze v té s názvem *development*. Proto je potřeba právě tuto větev mít v pracovním adresáři, čehož dosáhneme příkazem `git checkout -b development origin/development`. Dále je nutné se ujistit, že v podadresáři *core* existuje adresář *database* a pod ním další adresář, *events*. Pokud ne, tak je vytvoříme.

Kompilaci a umístění do repozitáře opět provedeme příkazem `mvn install`. To může tentokrát trvat vzhledem k mnoha testům, které obsa-

hují čekání, relativně dlouho – zhruba dvě až tři minuty. Pokud se tomuto chceme vyhnout, je možné použít příkaz `mvn install -DskipTests`.

Jakmile máme vše zkompilováno, nic nám nebrání server spustit. Jelikož se nachází v modulu *core*, stačí se přepnout do příslušného adresáře a zadat `mvn exec:java`. Nyní už můžeme server používat, tedy například připojit se k němu klientem (pro toto lze využít knihovnu v modulu *client*), přihlásit se k odběru zpráv pomocí příkazu subscribe a posílat na server události. Příklady použití lze nalézt v přiložených jednotkových testech. Aktuálně je možné server vypnout pouze „natvrdo“, tedy např. použitím klávesové zkratky Ctrl-C.