Masarykova univerzita
Fakulta informatiky

Distribuované Komplexní Zpracování Událostí

Diplomová práce

Bc. Tomáš Skopal

Brno, jaro 2016

Místo tohoto listu vložte kopie oficiálního podepsaného zadání práce a prohlášení autora školního díla.

Prohlášení

Prohlašuji, že tato diplomová práce je mým původním autorským dílem, které jsem vypracoval samostatně. Všechny zdroje, prameny a literaturu, které jsem při vypracování používal nebo z nich čerpal, v práci řádně cituji s uvedením úplného odkazu na příslušný zdroj.

Bc. Tomáš Skopal

Vedoucí práce: RNDr. Filip Nguyen

Shrnutí

Goal of this thesis is to develop a algorithm for distributed Event Pattern matching.. The application should be able to run any number of processing nodes. For the needs of this thesis, example of 4 nodes will be sufficient.

Klíčová slova

CEP, Kafka, ZooKeeper, analýza, distribuovaný

Obsah

1 Úvod

Obecně zadání práce říká, že má být vytvořen middleware pro distribuované zpracování vzorů událostí (angl. middleware solution for distributed event pattern matching). Z anglického popisu vychází zkratka MSFDEPM. Zjednodušením dostaneme "distributed event matching", neboli DEM. Pro účely této práce a snadnější orientaci budu popisované řešení identifikovat zkratkou *DEM*.

TODO - dopsat

2 Zpracování událostí

Se zvyšujícím se počtem zařízení, která jsou schopna produkovat data, se zvyšuje potřeba tato data analyzovat. Běžně rozšířeným způsobem je zpracování dat dávkově. Tedy, data se uloží a ve vhodnou dobu, typicky v noci, se analyzují.

Pokud však uvažujeme reálný provoz na síti, který se dnes v centrálních uzlech pohybuje okolo $1Tb/s$ [1], je dávkové zpracování nereálné. Potřebujeme data analyzovat za běhu (angl. real time).

Jednotkou zpracování dat je událost (angl. event). Událost je základním pojmem používaným v oblasti zpracování událostí. Je definována jako objekt, který reprezentuje záznam o aktivitě v daném systému. Událost může mít vlastnosti. Typickým příkladem takové vlastnosti je čas vzniku události, příčina jejího vzniku nebo její typ [2]. Jednoduchým příkladem události může být paket. Je to datová schránka, která obsahuje informace, které můžeme analyzovat. Samostatný paket nemá téměř žádnou vypovídající hodnotu, kdežto proud paketů je základem Internetu.

Takový proud událostí skrývá množství dat, která je možné získat až při komplexní analýze, která zohledňuje více událostí v řadě. To nazýváme *komplexní zpracování dat (angl. complex event processing neboli CEP)*

2.1 CEP

CEP je dynamický vědní obor, který se v poslední době hodně rozvíjí[1]. David Luckham ve své knize *The Power Of Events: An Introduction To Complex Event Processing In Distributed Enterprise Systems* [2] definuje CEP jako soubor technik a nástrojů, které pomáhají k pochopení a kontrole událostmi řízených systémů.

Jak už bylo řečeno, množství událostí v systémech je enormní. Při jejich zpracování se setkáváme s pojmem *komplexní událost*. Taková událost se může vyskytnout pouze jako reakce na sled jiných, dílčích,

1. Důkazem je aktivní komunita, která diskutuje problematiku na konferencích. Například konference v Oslu v roce 2015 (`http://dblp.uni-trier.de/db/conf/debs/debs2015.html`)

událostí. Dílčí události mohou spolu souviset mnoha různými způsoby, nejčastěji je však spojujeme na základě vlastností (čas vzniku, příčina vzniku, typ, atd).

Příkladem komplexní události může být akce nakoupení produktu v internetovém obchodě. Je to v dnešní době elektronického marketingu velké téma. Běžnému uživateli Internetu je v mnoha kanálech (Facebook ads, Google ads, mailing) zobrazována reklama. Některý uživatel nakoupí produkt při prvním zobrazení určité reklamy. Jiný uživatel potřebuje reklamu vidět alespoň pětkrát, než nakoupí. Způsob jak tuto "cestu"měřit se jmenuje atribuční model [7]. Atribuční model je tak defacto soubor událostí, které vyvolaly konečnou, komplexní, událost. Tedy nákup produktu v obchodě. S roustoucím počtem zařízení, roste počet reklam a také se komplikují atribuční modely. Vhodnými technikami zpracování dílčích událostí (zobrazení reklamy, kliknutí na reklamu, nainstalování aplikace) lze například predikovat chování uživatele.

CEP nabízí techniky pro definici a využití vztahů mezi událostmi. Může být využíván pro analýzu libovolného typu událostí v aplikaci, počítačové síti nebo v informačním systému. Jednou z těchto technik je i definování vlastních událostí, jakožto pravidla. Jinak řečeno, můžeme vytvořit vlastní reakci na soubor určitých událostí v našem systému. Touto cestou můžeme pochopit co se v našem systému odehrává.

To zvyšuje míru flexibility. Uživatel může za pomocí CEP specifikovat taková pravidla, která jej aktuálně zajímají a jsou pro něj přínosem. Může analyzovat jak nízko-úrovňové, tak vysoko-úrovňové procesy. Různé druhy událostí mohou být v CEP monitorovány současně. Velkou výhodou je, že pravidla mohou být měněna, odebírána a přidávána za běhu, tedy bez výpadku systému.

Zpracování proudu událostí a vyhodnocení, jestli se pravidlo vyskytlo, stojí samozřejmě určitý výpočetní výkon. V závislosti na typu a množství dat. Pokud je dat hodně (například analýza síťového provozu), musíme výpočet distribuovat.

2.2 Distribuované CEP

Úvodem kapitoly je potřeba jasně vymezit co v tomto kontextu chápeme pod pojmem "distribuovaný". Obecně se používají dva výklady:

- Distribuované zpracování událostí jako zpracování událostí z více heterogenních zdrojů (distribuovaný systém). [3] Takový výpočet může běžet i na jenom stroji a samotná analýza většinou nebývá paralelní. Takto pojem používá i *David Luckham* v knize *The Power of Events* [2] v popisu obrázku *1.1*.

- Distribuované zpracování událostí jako výpočet rozdělený na více menších, méně náročných úloh za účelem rychlejšího zpracování s využitím paralelismu[2]. Dále v práci budeme pojem chápat právě takto.

Požadavky na distribuované zpracování událostí (DCEP) jsou v mnoha ohledech jiné než na zpracování centralizované. Dvě hlavní vlastnosti, které vyžadujeme jsou vysoká míra dostupnosti (angl. availability) a nízké zpoždění (angl. latency). Cílem je pak maximalizovat dostupnost a minimalizovat zpoždění. Bohužel u většiny návrhů platí, že tyto vlastnosti jsou závislé a zlepšení jedné, zhoršuje druhou. Zlepšení dostupnosti, zvýší zpoždění, protože potřebujeme více času na synchronizaci všech řídících informací. [4]

Zmíněné dvě vlastnosti nejsou jediné. Mezi vlastnosti distribuovaného zpracování události můžeme dále zařadit:

- rovnoměrné rozdělení dat mezi výpočetní uzly (angl. data partitioning)

- automatické škálování výpočtu

- tolerance chyb

- správa datového úložiště

2. Takto zpracovává data například Twitter [12]. A to jejich distribucí pomocí nástroje *Apache Storm*

3 Nástroje pro distribuované zpracování událostí

Ve třetí kapitole představím dva existující nástroje a jeden nový přístup, jejichž použitím můžeme distribuovaně zpracovávat události. Cílem této práce je vytvoření nástroje pro detekci určitých vzorů v analyzovaných událostech v distribuovaném prostředí. Představení existujících nástrojů, které řeší podobnou problematiku, nám pomůže pochopit souvislosti. Také je možné v těchto technologiích najít inspiraci, jak řešit některá úskalí při práci v distribuovaném prostředí. Například odolnost systému proti výpadkům výpočetních uzlů.

3.1 Apache Samsa

První ze dvou technologií je Apache Samsa. Je to aplikační rámec sloužící k distribuovanému zpracování proudu dat. *Samsa* splňuje všechny důležité vlastnosti pro operace v distribuovaném prostředí jako jsou vysoká míra tolerance chyb, rozšiřitelnost a odolnost proti výpadku. Projekt byl původně vyvíjený společností LinkedIn a později byl vydán pod hlavičkou *Apache Software Foundation*. Tato kapitola popisuje základní strukturu a fungování aplikačního rámce Samsa. Zdrojem informací je oficiální dokumentace [11].

Samsa je postavena na dvou technologiích: *Apache Kafka* a *Apache Hadoop Yarn*. Aplikační rámec *Kafka* se stará o posílání zpráv (angl. messaging) a *Yarn* je zodpovědný za odolnost proti výpadkům.

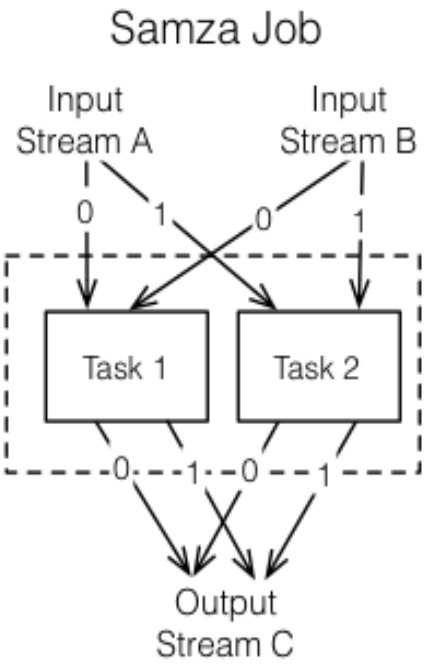
Princip zpracování dat je postaven na dvou pilířích. Těmi jsou datové proudy (angl. data stream) a úkoly (angl. job). Proud je složen z neměnitelných zpráv, které jsou podobného typu nebo jsou ze stejné kategorie. Příkladem datového proudu můžou být například všechna kliknutí na daný web nebo všechny log zprávy produkované nějakou aplikací. Abstrakce datového proudu koresponduje s *tématy* v aplik. rámci *Kafka*[1]. Každý proud může být zpracováván více konzumenty, proto mohou zprávy obsahovat unikátní klíče, které slouží k rozdělení do oddílů (angl. partitioning). Stejně jako se Kafka témata mohou dělit do oddílů.

1. Detailnější popis, co to jsou *témata* je v kapitole [5.1.2]

Jak bylo řečeno, druhým pilířem jsou úkoly. Úkol je jednotka práce, která zajišťuje transformaci množiny vstupních proudů na množinu výstupních proudů. Každý úkol se může dělit na menší výkonné jednotky zvané *Task*. Protože *task* i *job* se do češtiny překládají shodně jako "úkol", budu dál v textu, kvůli přesnosti vyjádření, používat hlavně anglické ekvivalenty. Každý *task* zpracovává data z jednoho nebo více oddílů z každého vstupního proudu.

Dělení *job* na *task* a *proud* na *oddíl* je kvůli zvýšení možností paralelního zpracování. *Task* zpracovává zprávy z každého oddílu sekvenčně. Neexistuje žádné definované řazení. To umožňuje nezávislý běh pro každý *task*. *Yarn* přiřazuje každý *task* určitému stroji, proto *job* jako celek může být spuštěn simultánně na více zařízeních.

Každý *task* logicky používá stejný kód jako kterýkoli jiný *task* v rámci stejného úkolu (*job*). Počet podúkolů (*task*) je omezený počtem oddílů, které vstupují do daného úkolu (*job*). Výše popsaný princip demonstruje obrázek 3.1, na kterém jsou zpracovávány dva vstupní proudy se dvěma oddíly.



Obrázek 3.1: Ukázka *Samsa* úkolu (*job*) [11]

.

Po představení základního konceptu aplikačního rámce *Samsa*, přejdu k příkladu použití. Příklad dobře demonstruje nasazení správné technologie na distribuované zpracování dat. Příklad je přepisem případové studie [13] společnosti *LinkedIn*.

Každý web si lze představit jako kompozici velkého množství jednoduchých akcí. U větších webů, jako je ten společnosti *LinkedIn* to platí dvojnásob. Jsou například akce: "načtení nových notifikací", "načtení seznamu navrhovaných přátel", atd. Každá z těchto akcí vyvolává sérii aplikačních volání. Počínaje dotazem na nějaké rozhraní (typicky *REST*) a konče databázovým dotazem. Každý dotaz je opatřen unikátním číslem, které se předává a je tak možné sestavit celý strom. Díky tomu je možné zjistit, které části kódu se volají zbytečně. Architektura společnosti je distribuovaná a aby byla data vypovídající, je potřeba sestavený strom neustále aktualizovat. Také *LinkedIn* zpracovává velké množství dat, proto je vhodné výpočet distribuovat.

Informace o probíhajících dotazech se spolu s unikátním identifikátorem posílá do aplik. rámce *Kafka*. *Samsa* je pak odtud čte. Následné zpracování se sestává ze dvou *Samsa* úkolů (*job*). První *job* připravuje události, které jsou přijímány z *Kafka* témat. Původní unikátní čísla u zpráv transformuje do jiného tvaru. Zatímco druhý *job* následně shromažďuje zprávy se stejným identifikátorem a vytváří výsledný strom.

3.2 Apache Storm

Druhou technologií pro distribuované zpracování dat, kterou přestavím, je *Apache Storm*. Je to otevřený software určený k zpracování proudu událostí v reálném čase. Byl vyvinut společností Twitter a nyní je dostupný pod licencí *Apache Software Foundation*.

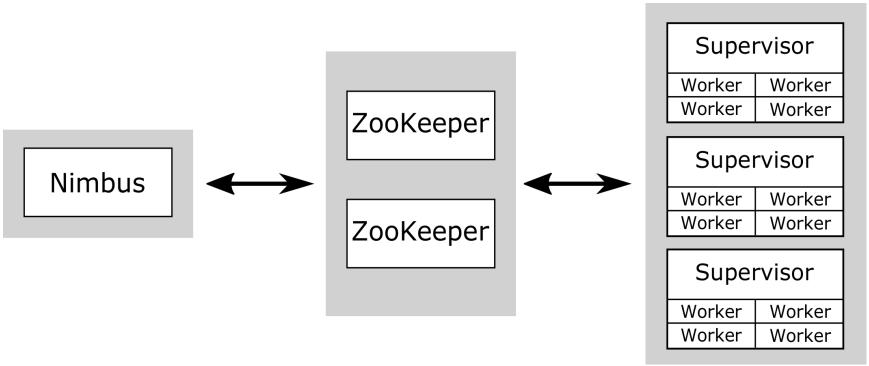
Hlavní předností u *Apache Storm* je vysoká míra tolerance chyb. Když jakákoli komponenta selže je garantováno, že systém bude dál plynule běžet. Úkoly z chybné komponenty jsou automaticky přeřazeny ostatním komponentám. *Storm* má také vysokou míru škálovatelnosti. To znamená, že s přidáním dalších počítačů do platformy, jsou některé úkoly z vytížených komponent automaticky přeřazeny na nový stroj. Poslední důležitou vlastností, je spolehlivost. Je zaručeno, že každá zpráva bude zpracována, i když dojde k výpadkům v komunikaci. Garance doručení zpráv funguje na podobném principu jako protokol *TCP*. Je možné tuto vlastnost vypnout a tím zvýšit rychlost zpracování [19].

Architektura *Apache Storm* se sestává ze dvou druhů uzlů. Mohou běžet simultánně na jednom počítači nebo na více. Jsou to:

- Nimbus – Ve *Storm* klastru běží právě jeden *Nimbus*. Je to hlavní síťový uzel, který je zodpovědný za distribuci kódu, přiděluje úlohy jednotlivým pracovním uzlům a monitoruje chyby. Jinak řešeno *Nimbus* se stará aby celý klastr fungoval a byl rovnoměrně vytížený.

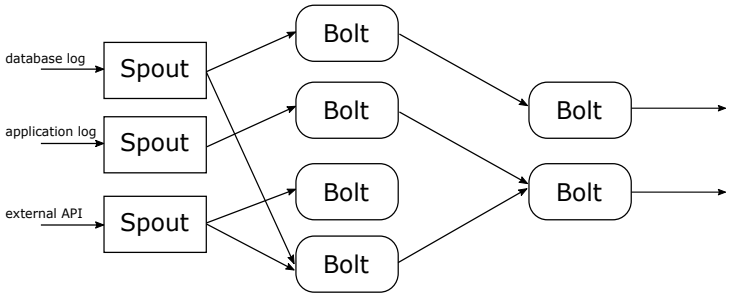
- Supervisor – Je to uzel, který se řídí instrukcemi, které posílá *Nimbus*. V tomto uzlu běží vlákna, zvaná *workers*, která provádí úkoly přidělené řídícím uzlem.

Storm využívá aplikační rámec *ZooKeeper* ke koordinaci mezi hlavním uzlem (*Nimbus*) a pracovními uzly (*Supervisor*). *Nimbus* je bezstavový, proto *ZooKeeper* musí monitorovat stav pracovních uzlů a udržovat stav celého klastru. ZooKeeper může běžet v jedné nebo ve více instancích.



Obrázek 3.2: Architektura aplikačního rámce *Storm*

Topologie *Storm* clusteru je určena jako orientovaný acyklický graf, jehož hrany jsou datové toky a uzly jsou komponenty. Komponenta je vždy typu *Spout* nebo typu *Bolt*. *Spout* čte data z externího (logy, rozhraní aplikace, apod.) zdroje a posílá je dál do topologie. Všechna "business"logika je soustředěna do komponent typu *bolt*. Jak je vidět z obrázku 3.3, každý *Bolt* může na svém výstupu generovat nový datový proud.



Obrázek 3.3: Ukázka topologie ve *Storm* klastru. Obrázek znázorňuje tři různé komponenty typu *Spout*, které čtou data ze tří různých zdrojů. Dále několik komponent typu *Bold*, které datové proudy zpracovávají [18].

Hlavní datová struktura v aplik. rámci *Storm* je tzv *tuple*. Je to seznam hodnot, kde každá hodnota může být jiného typu. Typy nemusí být předem definované, protože *tuple* je dynamicky typovaný. Sekvence objektů *tuple* tvoří datový proud (angl. *stream*).

Apache Storm má mnoho využití. Jedním z nich je například nepřetržité zpracování dat (angl. *continuous computation*). Díky tomu, je možné zpracovaná data zobrazovat téměř ve stejném čase kdy byla emitována. Druhým příkladem využití je analýza velkého množství dat. Díky distribuovanému výpočtu je možné analyzovat velké množství dat s nízkým zpožděním. [20]

3.3 CAVE

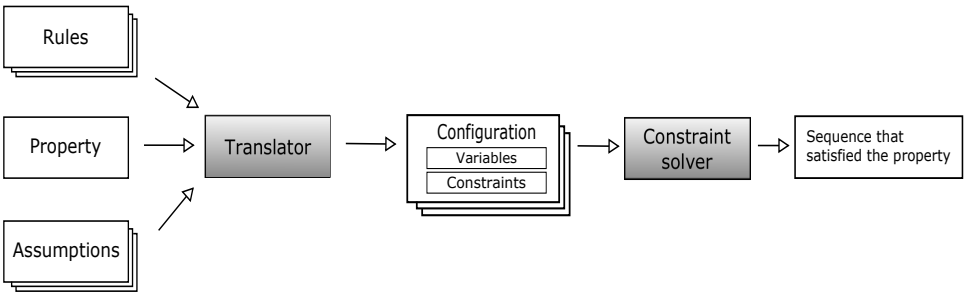
CAVE (Constraint-based Analysis of eVEnts) [15], je nová metodologie a nástroj pro vývojáře představený v roce 2015 na konferenci v Norském Oslu [14]. *CAVE* je určen na analýzu chování aplikací, které zpracovávají události. *CAVE* si klade za cíl identifikovat potenciální problémy pomocí detekce splnitelnosti vlastností, které vycházejí z charakteristik analyzovaného prostředí a množiny pravidel definovaných vývojářem. Navíc *CAVE* automaticky generuje sekvence událostí, které daným vlastnostem nastavují pravdivostní hodnoty. Díky tomu může vývojář jednoduše vyzkoušet vyhodnocení pravidel, které vytvořil.

Předpokladem je, že každá událost je charakterizována svým typem a množinou vlastností. Typ události definuje počet, pořadí a jména vlastností, které tvoří danou událost. Na příkladu 3.4 je pravidlo s typem události *Host* obsahující vlastnosti *cpu_load*, *gpu_load*.

```
Host(cpu_load > 80 and gpu_load > 80 or cpu_load = 100)
```

Obrázek 3.4: Příklad pravidla [15].

Na obrázku 3.5 je znázorněn postup zpracování vstupních informací. Na vstupu je vlastnost (property), která má být kontrolována, množina *CEP* pravidel (rules) a předpoklady dané prostředím (assumptions). Vlastnost je zde žádoucí nebo nežádoucí cíl, který má být detekován. *CAVE* vypočítá kdy může být vlastnost splněna a kdy ne na základě vstupních pravidel a proměnných prostředí.



Obrázek 3.5: Přehled postupu zpracování v *CAVE* [15]

.

CAVE pracuje ve dvou krocích. V prvním přeloží danou vlastnost na konečný počet konfigurací. Každá konfigurace reprezentuje jedno řešení problému. Konfigurace je složena z proměnných a podmínek nad těmito proměnnými. Ve druhém kroku výpočtu, *CAVE* využívá tzv "Constraint solver"k nalezení řešení pro alespoň jednu konfiguraci. Řešení spočívá v identifikaci takové sekvence událostí, která splní hledanou vlastnost (cíl). Pokud není žádná taková konfigurace nalezena, nemůže být samozřejmě nikdy hledaná vlastnost splněna.

Pro lepší pochopení uvedu příklad. Nechť je vstupní množina *CEP* pravidel pro jednoduchost složena z jednoho pravidla *R*:

```
R:
A() := [X then Y(y > 0)]
```

Obrázek 3.6: Příklad pravidla [15].

Dále uvažujme jednoduchou vlastnost *P1* která má za cíl zjistit jestli je možné detekovat alespoň jeden výskyt události typu *A*.

Podle výše uvedeného pravidla *CAVE* zjistí, že událost *A* může být generována pouze z pravidla *R*. Tedy, vlastnost *P1* může být splněna pouze pokud se pravidlo *R* vyvolalo alespoň jednou. To přivádí *CAVE* k vytvoření konfigurace, která zahrnuje událost typu *X* a událost typu *Y*. Konfigurace definuje tři proměnné: *X.timestamp*, *Y.timestamp*[2], and *Y.y*. Pravidlo *R* pak definuje podmínky nad těmito proměnnými:

2. *timestamp* je v překladu "časová známka". Předpokládá se, že každá událost má iniciálně nastavený čas. Typicky jde o čas vzniku dané události.

```
Y.y > 0 and X.timestamp < Y.timestamp
```

Obrázek 3.7: Resolvované podmínky [15].

Jak je znázorněno na obrázku 3.5, *CAVE* přikládá ke každé konfiguraci také podmínky definované aplikačním prostředím. V tomto případě prostředí limituje pouze vzdálenost mezi dvěma událostmi typu *Y*.

Zde končí fáze překladu. V následující fázi "constraint solver"vyhodnotí vygenerovanou konfiguraci jako splnitelnou a každé proměnné přiřadí konkrétní hodnotu se kterou bude klauzule splněna. V našem případě například: *X.timestamp = 1*, *Y.timestamp = 2*, *Y.y = 1*.

Problém detekování výskytu určité vlastnosti byl převeden na ekvivalentní problém jehož obsahem je zjišťování že alespoň jedna konfigurace vyhovuje zadání. Neboli problém splnitelnosti konečné množiny podmínek.

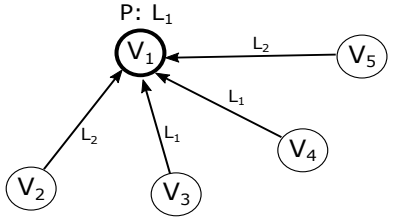
4 Analýza

Tato kapitola popisuje požadavky na systém a zasazuje systém do prostředí ve kterém bude nasazen. Počátek kapitoly parafrázuje a rozvíjí zadání práce. Dále je diagram užití, který shrnuje funkční požadavky. Na konci kapitoly jsou na modelovém příkladě demonstrovány vstupy a výstupy systému.

Cílem této práce je vytvoření software pro distribuované zpracování událostí, který bude schopen detekovat komplexní události na základě pravidel. Aplikace by měla být schopna zvládnout libovolný počet připojených uzlů. Síť propojených uzlů je možné reprezentovat grafem G(V, E), kde V jsou uzly grafu, neboli konkrétní počítače na kterých probíhá zpracování. E jsou pak orientované hrany grafu, které reprezentují informaci o tom který počítač posílá data kterému. Každá hrana je označena příznakem L_i, který označuje typ události, která je po dané hraně posílána. Konkrétním příkladem tak může být událost L_1, představující zprávu typu *chyba*, která není tak častá, ale její závažnost je vyšší. A událost L_2, představující zprávu typu *upozornění*, která je častá, ale méně závažná.

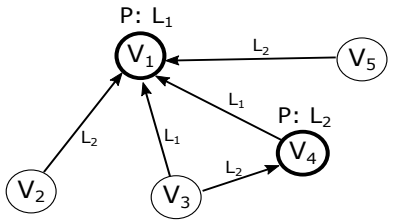
Pravidla pro detekci komplexních událostí, budou mít časovou platnost. Po uplynutí daného času se pravidlo automaticky stane neaktivní. Pravidla bude také možné nasadit pouze na určitou podmnožinu uzlů V grafu G. Více o pravidlech v kapitole 4.2.

Na obrázku 4.1 je vidět příklad grafu, kde je na uzlu V_1 nasazeno pravidlo analyzující události L_1.



Obrázek 4.1: Před vyhodnocením pravidla *P:* L_1 existuje jeden uzel zpracovávající události

Po určitém počtu výskytů události L_1 je vytvořena nová skupina z uzlů V_3 a V_4 kde je V_4 zvolen hlavním uzlem. Viz obr. 4.2



Obrázek 4.2: Po vyhodnocení pravidla *P:* L_1 existují dva uzly zpracovávající události

Typickým místem pro nasazení distribuovaného zpracování událostí je analýza síťové komunikace. To je také místo, kde bude DEM nasazen.

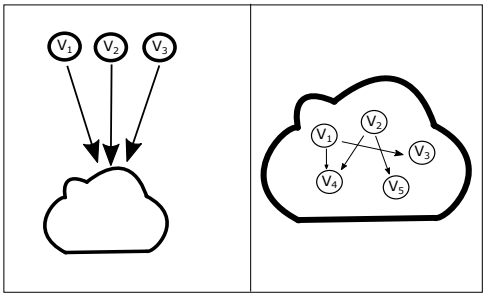
Nejčastější případ, je ten, že přes jeden hlavní uzel probíhá většina komunikace. Podobně je tomu i v softwaru. Aplikační server zpracovává všechny klientské požadavky. Takový uzel je zdrojem dat, která chceme analyzovat. Vytváří sekvenční proud dat, který můžeme vhodně distribuovat do clusteru. Ten jej zpracovává.

Problém nastane, když je množství produkovaných dat větší než je kapacita analyzujícího (analyzujících) počítačů. V takovém případě můžeme přidat výpočetní sílu nebo zvolit úplně jiný přístup k datové analýze.

4.1 Vstupy a výstupy

Uvažujme situaci, kdy necháme na síťovém uzlu aby analyzoval běžně známe hrozby. Taková zařízení jsou na trhu běžně dostupná. A zbytek analýzy bude probíhat až na uzlech v rámci sítě. V modelové situaci útočník obejde firewall a na koncových zařízeních v síti začnou vznikat anomálie. Za normální situace by se takový útok neodhalil, protože na koncových zařízeních neběží žádná detekce. Často to ani není prakticky možné, protože kdyby počítače odesílaly všechen provoz do clusteru na analýzu, byla by to ještě větší zátěž než, kdyby to dělal hlavní síťový prvek. Cílem této práce je tak vytvořit řešení, které bude možno nasadit přímo na koncová zařízení a provádět analýzu tam. Obecnou představu znázorňuje obrázek 4.3. Na obrázku je vidět srovnání, kde jsou v levé části data posílána do nějakého cloudu (černé

skřínky), kde jsou analyzována. Naopak v pravé části je výpočetní cluster pro analýzu vytvořen přímo z uzlů, které data produkují.



Obrázek 4.3: Srovnání přístupu zasílání dat do cloudu (vlevo) a vytvoření cloudu přímo na koncových zařízeních (vpravo)

Vstupem jsou data zaznamenána na každém stroji připojeném do *DEM*. Formát, množství a povaha dat je pak určena konkrétní situací, která je potřeba sledovat. Pro účely této práce jsou data naivně generována aby vytvořila simulaci síťového útoku. Detailní popis je v kapitole 5.3

Výstupem by mělo být upozornění na nestandardní situaci v části nebo celém systému. Formát výstupu DEM je opět individuální vzhledem k situaci. Zjištěné výsledky mohou být ukládány ve formě logů (tak je to vyřešeno v této práci), odesílána do centrálního dohledu nebo ukládána do databáze.

Do DEM bude možno nasadit pravidla na detekci vzorů událostí. Tato pravidla mohou být omezena časovou platností a nasazena na analýzu pouze určitého množství uzlů.

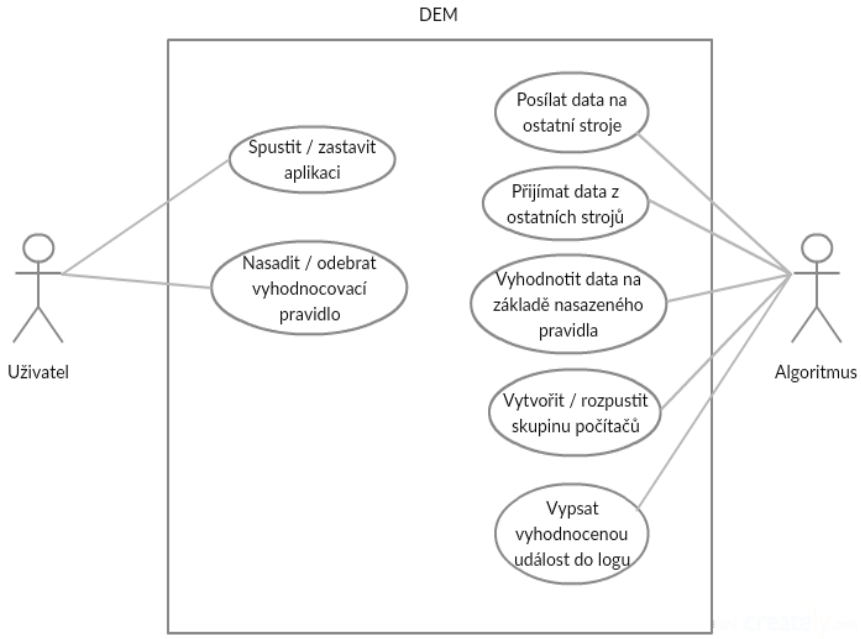
4.2 Diagram užití

Cílem této kapitoly je jasně formulovat seznam požadavků, které má systém splňovat. Soubor požadavků je znázorněn graficky pomocí diagramu užití. Dále jsou všechny požadavky rozepsány. U některých je popis doplněn o pseudokód[1].

1. Pseudokód je psán v angličtině, protože angličtina je obecně uznávaný jazyk pro psaní jakéhokoli kódu.

Požadavky jsou kladeny hlavně na variabilitu, možnost nasazení pravidel s časovou platností. Menší důraz je na výkon a proces nasazování.

Na obrázku 4.4 je znázorněn diagram užití, který přehledně shrnuje požadavky na systém. Vystupují v něm dvě role. Uživatel, je zde člověk, který spouští analýzu. Druhou rolí je Algoritmus, který reprezentuje samotný software. Je zde uveden proto, aby bylo zřejmé jaké operace se od systému vyžadují.



Obrázek 4.4: Diagram užití

Dále následuje popis jednotlivých případů užití. Kvůli přehlednosti jsou nadpisy některých pravidel zkráceny.

Spustit / zastavit aplikaci

Jak je z názvu patrné uživatel může spustit a zastavit aplikaci a s tím samozřejmě i analýzu. Je zde vhodné zdůraznit, že spuštění i zastavení by mělo být centralizované a jednoduché. To znamená kontrola pomocí

jednoho skriptu. Tím se ušetří uživateli čas a předejde se některým chybám.

Nasadit / odebrat pravidlo

Uzly, které budou přijímat data, je budou zároveň i analyzovat. Tím, že data mají charakter potenciálně nekonečného proudu, bude pro analýzu potřeba zvolit vhodný nástroj. Nástroj bude muset umět na základě definovaného pravidla vyhodnotit která z příchozích dat pravidlu vyhovují. Jak již bylo zmíněno dat může být opravdu hodně, proto vyhodnocování musí probíhat pouze v omezeném časovém intervalu. Demonstrace pravidla za použití pseudokódu na obrázku 4.6 vyhodnocuje počet událostí typu L_1 v časovém okně pěti sekund.

```
SELECT count(*) FROM Event(type=L1) in window (5 sec)
```

Obrázek 4.5: Příklad pravidla

Poslat / přijmout data

Data, která jsou určena k analýze je potřeba odeslat na správný stroj. Každý uzel v systému je producentem dat, ale pouze některé jsou i konzumenty. Pro demonstraci popíšu pomocí pseudokodu posílání a přijímání dat na uzlu V_4 z obrázku 4.2:

```
send data: {type = L1} to node V1;
read data: {type = L2} from queue;
```

Obrázek 4.6: Přijímání a posílání dat

Vyhodnotit data na základě pravidla

Přijatá data jsou analyzována a průběžně se vyhodnocuje jestli některá sekvence událostí odpovídá aktuálně nasazenému pravidlu. Pokud ano, je vytvořena nová, komplexní, událost, obsahující potřebná data pro adekvátní reakci. Adekvátní reakcí je myšleno vytvoření nové skupiny nebo vypsání události do logu.

Zpracování komplexní události záleží na typu analyzovaných dat. Pokud jsou to hrubozrné události (nízká frekvence výskytu, ale vysoká důležitost) bude vytvořena nová skupina počítačů, které budou mezi sebou analyzovat jemnozrné události (vysoká frekvence výskytu, ale nízká důležitost). Naopak, pokud jsou to jemnozrné události, bude komplexní událost pouze zaznamenána do logu. Detailní popis jemnozrných a hrubozrných událostí je v kapitole 5.3.

Vytvořit / rozpustit skupinu počítačů

Důvod vzniku nové skupiny je popsán v předchozím případě užití. Rozpuštění je vyvoláno uplynutím časového intervalu. Skupiny počítačů analyzující jemnozrné události jsou vytvářeny s časovou platností. Daný interval by měl stačit k analýze. Pokud dojde k detekci sledu událostí, který odpovídá aktuálně nasazenému pravidlu pro jemnozrnou analýzu, vzniklá komplexní událost je zapsána do logu.

5 Návrh

Tato kapitola postupně popisuje jak budou reálně splněny požadavky na systém uvedené v analýze. Stěžejní částí je volba technologií. U každé zvolené technologie je její popis a jak bude použita ve výsledném řešení.

5.1 Technologie

Výběr technologií nebyl náhodný. Byly mi doporučeny vedoucím práce. Každá z nich by se dala nahradit, ale myslím si, že právě tato kombinace (*Java + Maven, Kafka, Esper, ZooKeepr*) je ideální, protože jsou si navzájem přizpůsobené. Namátkou zmíním některé vazby, které usnadňují vývoj: *Esper, Kafka i ZooKeeper* mají rozhraní pro Javu, proto je *Maven* ideální. *Kafka* používá *ZooKeeper* pro distribuci nastavení, apod.

5.1.1 Apache Maven

Apache Maven je nástroj pro správu, řízení a automatizaci sestavování aplikací (angl. build). Maven sám nemá žádné uživatelské rozhraní a běží pouze na příkazové řádce. Jeho účelem je usnadnit práci vývojáři tím, že definuje jednotný proces sestavení. 5 Také definuje strukturu aplikace, protože jednotlivé typy souborů hledá v určitých balíčcích. Například spustitelné Java soubory by měly být v adresáři *src/maven/java*.

Konfigurační soubor Mavenu je *pom.xml*, ve kterém jsou uvedeny zásuvné moduly (pluginy), podle kterých Maven pozná, co má dělat. Také je zde seznam závislostí na externí knihovny, které Maven dokáže stáhnout. Při použití Mavenu je sestavení programu otázkou jen jednoho příkazu (*mvn clean install*).

Použití

Kromě definování závislostí je Maven v projektu použit na vytvoření modulů. Moduly jsou celkem čtyři a jsou navrženy tak, aby názvem i logikou odpovídaly účelu a nepřesahovaly své určení. Jsou to:

- Producer – slouží pro generování dat a jejich posílání do aplikačního rámce Kafka. Data jsou generována v určitých časových intervalech a jejich struktura je přesně daná. Syntax a sémantika vzorových dat je popsána v kapitole 5.3. Při reálném použití by byl tento modul přepsán nebo nahrazen za takový, který bude reálná data číst z nějakého systémového nebo aplikačního logu.

- Consumer – tento modul čte data z aplikačního rámce Kafka a analyzuje je za pomocí nástroje Esper 5.1.4. Události, které Esper vyhodnotí slouží v DEM jako řídící zprávy pro přechod do dalšího stavu. Případně jsou události pouze zapsány do logu.

- MainApp – modul obsahuje spustitelnou třídu a je tak vstupním bodem programu. Také se zde zpracovávají všechny řídící události z aplikačního rámce ZooKeeper 5.1.3 a rozhoduje se jaká vlákna (konzument nebo producent) se mají vytvořit nebo zastavit.

- AppData – modul obsahuje několik výčtových typů, které jsou společné v celé aplikaci a jednu singleton [1] třídu, která drží některá řídící data. Například ip adresu počítače. Tato data mají k dispozici všechny moduly.

Protože je veškerá komunikace řízena událostmi a zasíláním zpráv, moduly jsou na sobě relativně nezávislé a v případě potřeby je možné je zaměnit za jiné.

5.1.2 Apache Kafka

Apache Kafka je systém pro zasílání zpráv. [6] Kafka klastr může být rozdělený mezi několik počítačů, každý nazýváme *broker*. Základem aplikačního rámce Kafka je fronta zpráv. Ta je reprezentována tématem (angl. topic) respektive přepážkou (angl. partition). Při vytváření tématu udáváme kromě jejího jména, také kolikrát se má replikovat mezi *brokery* a počet přepážek. Přepážka je menší jednotka než téma. Každé téma může být replikováno mezi brokery.

1. Singleton neboli jedináček je návrhový vzor, který je využívaný, když je potřeba mít pouze jednu instanci dané třídy.

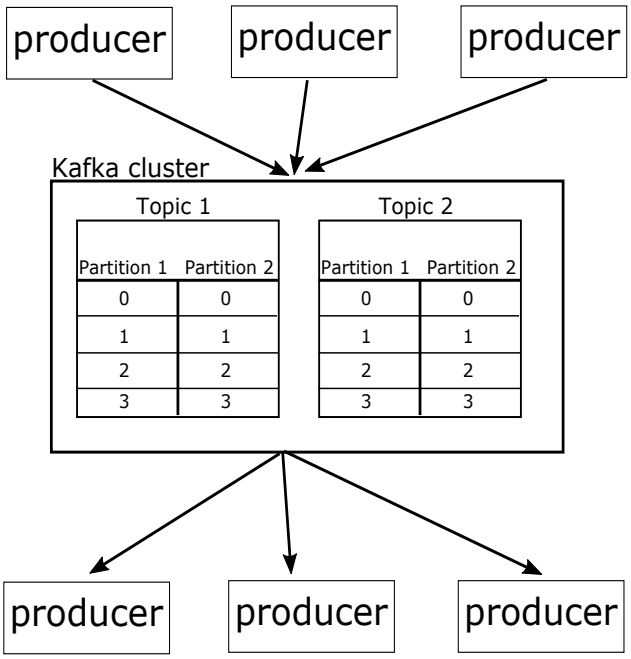
Konzument a producent

Do Kafka klastru data zapisují *producenti* a na druhé straně z ní data čtou *konzumenti* (angl. producer and consumer). Zapisování dat producentem je přímočaré. Producent rozhoduje do kterého tématu či přepážky se má zpráva zapsat. Zprávy jsou ukládány do fronty, tedy způsobem FIFO [2]. Čtení zpráv závisí na aktuálním stavu. Konzumenty je možné seskupovat do skupin a podle toho se pak rozlišuje způsob čtení zpráv na "queuing"a "publish-subscribe". V modelu "queue"je zpráva poslána vždy jednomu z konzumentů. Naopak v modelu "publish-subscribe"je každá zpráva poslána všem konzumentům. V Kafka dokumentaci se říká [6]:

- Pokud jsou všichni konzumenti ve stejné skupině, pak Kafka funguje v modelu "queue".

- Pokud je každý konzument v jiné skupině, pak je Kafka v modelu "publish-subscribe". Všechny zprávy jsou distribuovány všem konzumentům [3].

2. FIFO – první zapsán, první přečten (angl. first in first out)
3. Broadcast



Obrázek 5.1: Znázornění základního schématu aplikačního rámce Kafka

Garance

Kafka poskytuje následující seznam garancí:

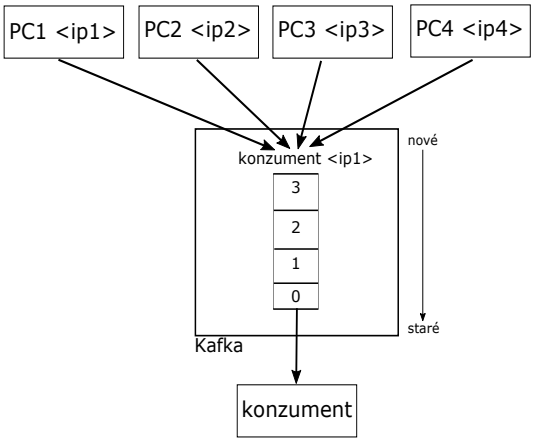
- Zprávy poslané do určitého tématu budou seřazeny v pořadí v jakém byly odeslány. Například pokud je zpráva *M1* odeslána producentem dříve než zpráva *M2*, pak Kafka zaručuje, že bude mít zpráva *M1* menší offset a bude v logu zobrazena dříve než *M2*.

- Konzument vidí zprávy v takovém pořadí, v jakém byly uloženy do logu.

- Pro téma s faktorem replikování *N* (bude replikováno mezi *N* brokerů), Kafka zaručuje zachování dat až pro *N-1* vypadlých serverů.

Použití

Jak už bylo řečeno dříve analýza událostí bude probíhat na počítačích, které data produkují. V navrhovaném řešení jsem nezaznamenal potřebu dělit témata na příčky (partition). Také Kafka není distribuovaná do několika brokerů a je spuštěna pouze na jenom stroji. Distribuci bych nakonfiguroval až v případě produkčního nasazení pro prevenci výpadků. Na každém počítači běží nejméně jeden producent a nejvýše jeden konzument. Jednotlivá témata jsou pojmenována podle ip adres počítačů. Konkrétně při iniciálním spuštění je jeden ze strojů určen jako konzument a počet producentů je roven počtu strojů. Protože existuje pouze jeden konzument, data jsou posílána pouze do jednoho odpovídajícího tématu. Viz obrázek 5.2



Obrázek 5.2: Struktura aplik. rámce Kafka při iniciálním spuštění

5.1.3 Apache ZooKeeper

ZooKeeper (dále také jako "ZK") je další z rodiny "open-source"Apache technologií. Je to centralizovaná služba pro správu konfiguračních dat, pojmenování a poskytování synchronizace distribuovaných uzlů. [8]

ZK je navržen, tak aby jej bylo jednoduché použít. Je napsán v Javě a poskytuje konektory pro Javu a pro C. Struktura datového modelu je podobná stromovému uspořádání souborového systému. Datový model ZK je sestaven z uzlů zvaných *znodes*. Na rozdíl od klasického souborového systému, který je navržen pro dlouhodobé ukládání dat, ZK data drží v paměti, a tak může dosahovat nízkého zpoždění a vysoké propustnosti. Pro ilustraci, uvažujme jednoduchý příklad: Každý znode může obsahovat data (až do limitu 1MB). V takovém případě znode reprezentuje soubor (řekněme textový) v souborovém systému. V ZK může mít každý uzel nula až N potomků. Pak znode vystupuje obdobně jako složka v souborovém systému. Každý znode může zároveň obsahovat data a mít potomky.

Garance

Obdobně jako Kafka a naprostá většina distribuovaných systémů, musí i ZK poskytovat určité garance běhu v distribuovaném prostředí.

- Sekvenční konzistence – změny na uzlu jsou aplikovány v takovém pořadí v jakém byly odeslány (modifikace dat, odebrání nebo přidání potomka, apod).
- Atomicita – změny jsou úspěšně aplikovány úplně nebo neprovedeny vůbec. Neexistuje žádný mezistav.
- Spolehlivost – Jakmile je změna aplikována, je stav uzlu garantován. Nemění se až do doby další změny.
- Časová konzistence – Stav systému je po uplynutí definovaného času vždy aktuální.

API

ZK nabízí uživateli (programátorovi) velice jednoduché rozhraní, pro manipulaci s jednotlivými uzly. Sestává se z těchto operací:

- Vytvoření (create)
- Smazání (delete)
- Kontrola existence uzlu (exists)
- Získání dat (get data)
- Zapsání dat (set data)
- Získání všech potomků daného uzlu (get children)
- Ukazatel dokončení synchronizace dat (sync)

Kromě operace *sync* jsou v této práci použity všechny.

Pozorovatelé

Pozorovatelé [4] jsou nedílnou součástí téměř každého událostmi řízeného systému. Stejně tak, je tomu u ZK. Na znode je možné zaregistrovat dva druhy pozorovatelů:

- Datový pozorovatel – změní stav, když jsou změněna data v uzlu. Tzn někdo zavolal operaci "zapsání dat". Změna stavu obsahuje nově nastavená data
- Pozorovatel potomků – změní stav, když nastane změna v některém z potomků daného uzlu. Událost obsahuje, mimo jiné, informaci o typu změny (smazání, vytvoření, zápis dat, ztráta konektivity) a identifikaci potomka.

Apache Curator

Aplikační rámec *Curator* je nástroj vytvořený společností Neflix pro jednodušší práci se ZK. Rozhraní ZK se potýká s několika problémy, jako je například nedostatečné ošetření výpadků konektivity nebo selhání operace. Proto byl navržen *Curator*, který se osvědčil a později přešel pod licenci *Apache*. Poskytuje následující benefity:

4. Z anglického slova *Watches*. Jde o konkrétní implementaci návrhového vzoru "observable". [9]

- Komplexnější rozhraní pro jednodušší práci se ZK. 5.1
- Automatické připojení na instanci ZK a automatické opravy výpadků.
- Kompletní a dobře otestovaná implementace některých složitějších operací.

```
curatorFramework.delete()
   .guaranteed()
   .deletingChildrenIfNeeded()
   .forPath("/zkNode1");
```

Ukázka kódu 5.1: Demonstrace jednoduchého použití aplikačního rámce Curator, na smazání uzlu *zkNode1* a všech jeho potomků.

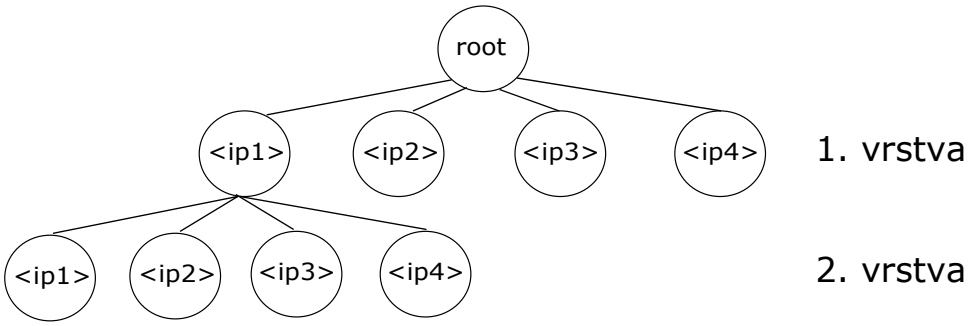
Použití

ZK je použit pro řízení běhu celého procesu DEM. Jeho hlavní úlohou je distribuce řídících dat mezi jednotlivými výpočetními jednotkami. Druhou funkcí, která logicky vychází ze stromové struktury datového modelu, je reflexe aktuálního stavu aplikace.

Je důležité zmínit, že cesta ve stromové struktuře je určena jednoznačným textovým identifikátorem. Jednotlivé uzly jsou pak odděleny lomítkem. Na obrázku 5.3 můžeme vidět strukturu uzlů, při iniciálním spuštění DEM. Příklad identifikátoru prvního uzlu z druhé vrstvy je *"/root/ip1/ip1"*, druhého uzlu z první vrstvy *"/root/ip2"*, atd.

Obrázek 5.3 dále znázorňuje logickou strukturu DEM. První vrstva reprezentuje konzumenty a druhá vrstva producenty. Jinak řečeno "kdo komu posílá data". Podstatné je, uvědomit si, že strom zobrazuje pouze virtuální stav. Aplikace jako taková běží na daném stroji vždy jen jednou a strom pak reprezentuje spíše množství běžících vláken a jejich úlohu. Jak je vidět z obrázku 5.3, v DEM existují 4 producenti (jeden na každém fyzickém stroji) a jeden konzument (fyzický stroj s *ip1*). To odpovídá stavu aplik. rámce Kafka z obrázku 5.2.

Výhody stromové struktury jsou zřejmé. U každého uzlu se dá zjistit jaké má potomky (které stroje posílají data a kam). Změna cíle, kam má stroj posílat data, je na úrovni rozhraní triviální. Stačí přesunout uzel v druhé vrstvě pod jiného rodiče.



Obrázek 5.3: Stromová struktura uzlů v ZK při iniciálním spuštění.

5.1.4 Esper

Esper je "open source"nástroj pro komplexní zpracování událostí a zpracování proudu událostí[5] dostupný pod GNU GPL licencí[6]. Jedná se o velice užitečný nástroj, jehož největší předností je efektivita zpracování proudu událostí. V této kapitole popíšu základní princip jak Esper funguje a ukážu názorné ukázky použití. Základním zdrojem pro tuto kapitolu je oficiální dokumentace [16]. Esper je napsaný v Javě, ale existuje i verze pro *.NET*, zvaná *NEsper*. Esper je primárně navržen na efektivní analýzu velkého množství událostí v reálném čase. S daty pracuje pouze v paměti a drží si minimální stav.

Princip aplikačního rámce Esper je odvozen od *SQL* standardu a nabízí možnosti agregace dat, detekci vzorů událostí nebo analýzu v plovoucím časovém okně. Základem je dotazovací jazyk zvaný "Event Processing Language (EPL)". *EPL* je deklarativní[7] jazyk určený na analýzu časově seřazených událostí a detekci definovaných vzorů v těchto událostech. *EPL* nabízí kaluzule: *SELECT, FROM, WHERE, GROUP BY, HAVING* a *ORDER BY*, které jsou běžně známé z *SQL*.

Rychlost výpočtu dobře demonstruje příklad z dokumentace [17]. Esper by měl zvládnout analyzovat na dvou jádrovém procesoru s taktem *2GHz* okolo 500 000 zpráv za sekundu. Samozřejmě rychlost záleží na konkrétním použití. Esper umožňuje datový proud i modifi-

5. Zpracování proudu událostí, neboli anglicky *event stream processing (ESP)*.
6. GNU GPL General Public Licence) je všeobecná veřejná licence - `https://www.gnu.org/licenses/gpl.html`
7. Deklarativní jazyky specifikují cíl a algoritmizace je ponechána programu (interpretu) daného jazyka.

kovat, nejen číst. Při takové operaci se logicky analýza značně zpomalí. Esper neumožňuje distribuci výpočtu, protože to by mělo dramatický dopad na zpoždění výpočtu. Přesto je možné Esper kombinovat s distribuovanými nástroji jako jsou *Samsa* nebo *Storm*. Nicméně oficiální doporučení jak zvýšit výkon je přidání výpočetní kapacity (CPU, RAM) a zvýšení počtu vláken (max 32).

Použití

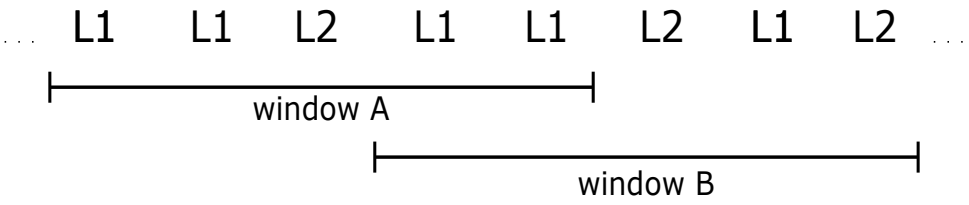
Protože je pro vývojáře *EPL* ústředním prvkem celého nástroje Esper, udělám podrobný rozbor jednoho vzorového pravidla (ukázka 5.4), na kterém přiblížím možnosti nástroje. Esper také bude nedílnou součástí aplikace DEM. Proto je v závěru kapitoly diagram aktivit, který demonstruje tok dat mezi aplikací a výpočetním jádrem (Esper).

```
SELECT * FROM
IncomingEvent(level='LEVEL1').win:time_batch(5 sec) WHERE
flag = 'SYN' GROUP BY source HAVING count(*) > 3
```

Obrázek 5.4: Vzorové pravidlo nástorje Esper napsané v jazyce *EPL*.

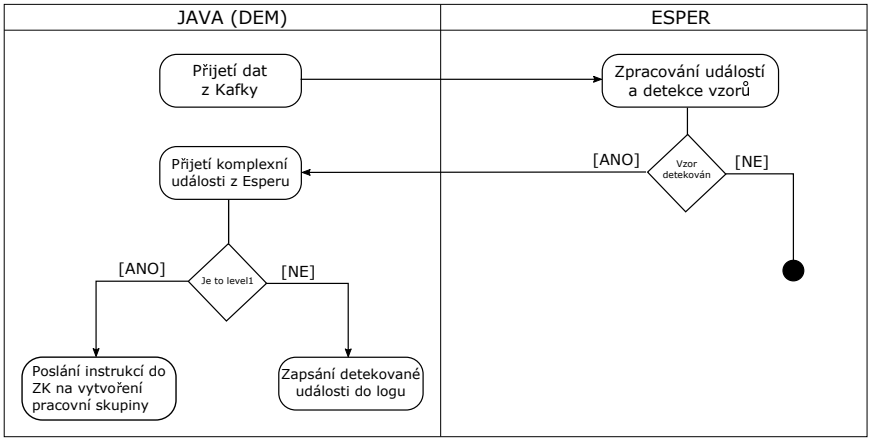
Příklad 5.4 ilustruje dotaz napsaný v jazyce *EPL*, který vrátí výsledek (jeden řádek s daty) jakmile je v rámci plovoucího časového okna pěti sekund splněna sada podmínek. První podmínka je, že událost *IncomingEvent* musí mít úroveň jedna (LEVEL1). Dále musí obsahovat atribut *flag*, který nabývá hodnoty *SYN*. Poslední částí výrazu jsou klauzule *GROUP BY a HAVING*, které plní stejný účel jako v klasickém *SQL*. Tedy v tomto případě seskupují události podle zdroje (source) a musí být detekovány více než tři události se stejným zdrojem.

Pro jednoduchost uvažme proud událostí úrovně jedna a dvě, které mají stejný zdroj i příznak (flag) a události jsou generované s rozestupem jedné sekundy. Obrázek 5.5. Pokud bychom v takovém proudu hledali výskyt pravidla 5.4, pak by vyhovovalo pouze pěti sekundové časové okno *A*, jak je vidět na obrázku.



Obrázek 5.5: Ukázka detekce *EPL* pravidla na proudu událostí

Závěrem kapitoly uvádím aktivity diagram 5.6, který reflektuje tok dat mezi Java aplikací a výpočetním jádrem aplik. rámce Esper. Proces v této fázi začíná čtením dat z aplik. rámce Kafka. Každá jednotlivá událost je přeposílána do nástroje Esper, který proud dat analyzuje. V tomto diagramu se předpokládá, že už je nasazeno nějaké pravidlo. Pokud Esper v proudu událostí detekuje hledaný vzor notifikuje posluchače v aplikaci (vznik nové komplexní události). Pro úplnost je ještě v diagramu obsažena i reakce na vzniklou komplexní událost. Ta se odvíjí od úrovně, kterou měly původní akce (*LEVEL1 vs LEVEL2*).



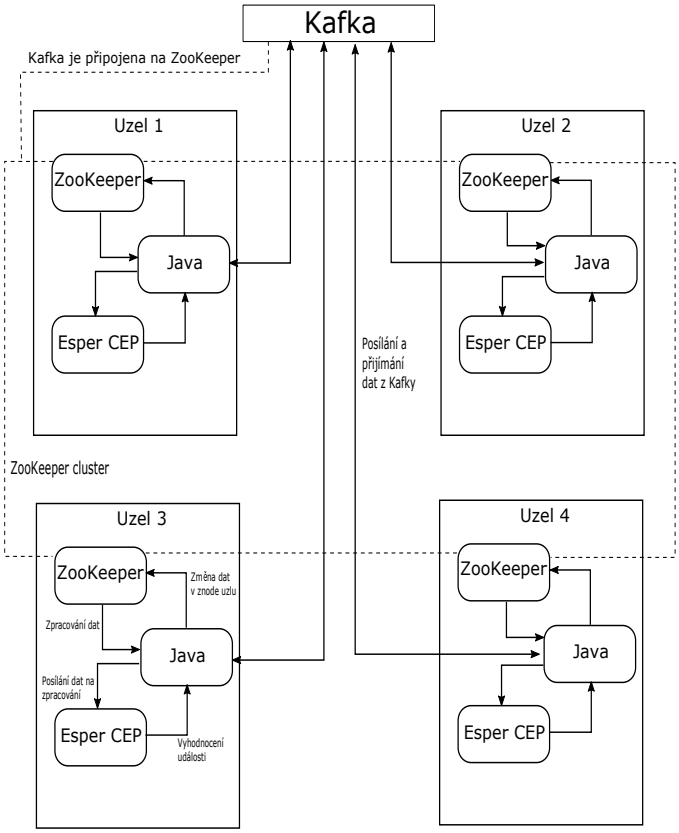
Obrázek 5.6: Aktivity diagram znázorňující tok dat mezi aplikací a Esper výpočetním jádrem.

5.2 Shrnutí

Kapitola obsahuje přehlednou tabulku, která konkrétně ukazuje která technologie je použita při implementaci daného požadavku z analýzy. A návrh schématu. Schéma je znázorněno na čtyřech strojích.

Požadavek	Technologie
Posílání zpráv mezi stroji	Kafka
Libovolný počet připojených uzlů	ZK + Kafka
Nasazení nového pravidla	ZK
Definování chování jednotlivých strojů	ZK
Vytvořit nebo rozpustit skupinu strojů	ZK
Nalezení určitého vzoru (pattern) v datech	Esper
Propojit technologie do funkčního celku	Java + Maven

Tabulka 5.1: Splnění požadavků na systém



Obrázek 5.7: Schéma aplikace, které znázorňuje které technologie jsou nasazeny na každém stroji a jaké mají mezi sebou vazby.

5.3 Události v systému

Povaha a struktura vyhodnocovaných dat je neopomenutelnou součástí každého systému, tedy i tohoto. Pro potřeby analýzy zavedeme dva druhy událostí, *hrubozrné* a *jemnozrné*. Hrubozrné jsou události, které nenastávají příliš často, ale mají v systému důležitý význam. Jemnozrné naopak.

Jak je nastíněno v kapitole , budeme se zabývat analýzou síťové komunikace. Základem je zde samozřejmě proud paketů. Příchozí pakety mohou mimo jiné znamenat pro daný stoj bezpečnostní hrozbu. Například při útoku *SYN flood* je restart (kolaps) stroje způsoben zahlcením pakety s příznakem *SYN*. Zde můžeme za hrubozrnou událost považovat signalizaci restartu stroje. Jemnozrné události jsou pak jednotlivé pakety.

Principem analýzy je identifikace strojů s výskytem stejných hrubozrných událostí a pustit na nich analýzu jemnozrných událostí, která má za cíl odhalit příčinu.

Hrubozrnné

Hrubozrné události mají komplexní povahu. Jsou tedy výsledkem nějakého složitějšího procesu. Ve výše uvedeném příkladě šlo o restart počítače způsobený cíleným útokem. Jiným příkladem může být chybová zpráva v aplikaci (angl. error log). Pokud je aplikace zasažena *DoS* útokem, pak je hrubozrnou událostí její nedostupnost. V DEM jsou hrubozrné události reprezentovány štítkem **LEVEL1**.

Jemnozrnné

Tento druh událostí se vyskytuje běžně a sama o sobě nemá událost prakticky žádný význam. Například jeden paket s s příznakem *SYN* se v síťové komunikaci objevuje běžně. V DEM jsou hrubozrné události reprezentovány štítkem **LEVEL2**.

6 Nasazení

Předchozí kapitoly Analýza a Návrh shrnují požadavky na systém a technologie, za pomoci kterých budou požadavky splněny. V Návrhu je také popsáno jak bude konkrétně daná technologie použita. Tato kapitola navazuje samotnou implementací a testováním. Je zde popsán běh programu od úplného začátku, tedy správné konfigurace, až po ukázky výstupu.

6.1 Konfigurace

Jak Kafka, tak ZK nabízejí velkou variabilitu v konfiguraci. Oficiální dokumentace obou technologií obsahují dobrý popis konfigurace. Proto je tato kapitola věnována pouze konkrétním ukázkám konfigurací v DEM.

Pro spuštění aplik. rámce Kafka je důležitý soubor *server.properties*. Obsahuje množství atributů, které se dají modifikovat. Většinu z nich není potřeba upravovat. Měnil jsem pouze *broker id*, *název stroje (host name)* a *seznam ZK instancí*. Výňatek z konfiguračního souboru s upravenými hodnotami je na obrázku 6.1.

```
broker.id=0
host.name=147.251.43.181
zookeeper.connect=147.251.43.181:2181,147.251.43.130:2181
```

Obrázek 6.1: Výňatek z konfiguračního souboru aplik. rámce Kafka

U ZK je potřeba upravit dva konfigurační soubory. První je *zoo.cfg*. V něm se nastavují ip adresy a porty ostatních ZK instancí. Také se nastaví složka do které se ukládají dočasné soubory a je zde i druhý z konfiguračních souborů *myid*. Soubor *myid* obsahuje pouze jedno číslo, které jednoznačně identifikuje instanci a musí být unikátní.

```
# the port at which the clients will connect
clientPort=2181

dataDir=~/data/zookeeper

# specify all zookeeper servers
# The fist port is used by followers to connect to the leader
# The second one is used for leader election
server.1=147.251.43.181:2888:3888
server.2=147.251.43.130:2888:3888
server.3=147.251.43.150:2888:3888
server.4=147.251.43.138:2888:3888
```

Obrázek 6.2: Ukázka konfiguračního souboru aplikačního rámce ZooKeeper

Na konec pouze krátce uvedu hardwarovou konfiguraci virtuálních strojů na kterých probíhalo testování:

```
OS: Ubuntu 12.04 x86_64
RAM: 1GB
Procesor: 1 core x86_64
HDD: 20GB
```

Obrázek 6.3: Konfigurace virtuálních strojů

6.2 Implementace

Pro implementaci byla zvolena Java (konkrétně ve verzi 1.8), protože všechny použité technologie mají dobrá API pro Javu a většina příkladů je právě v Javě. Dalším důvodem je také to, že Java se dobře hodí pro běh aplikací tohoto druhu, protože má dobrou práci s vlákny. Posledním, méně důležitým, důvodem je popularita Javy a snadná čitelnost kódu.

Aplikace je z pohledu počtu řádků poměrně malá, ale je kompletně řízena událostmi. Posílání zpráv v aplikačním rámci Kafa je

asynchronní a ZooKeeper i Esper jsou ze své podstaty událostmi řízené aplikační rámce (angl. frameworky). Proto je architektura pro přehlednost členěna do logických modulů. Jednotlivé moduly jsou podrobně popsány v kapitole 5.1.1.

Implementace probíhala iterativním vývojem. Nejdříve bylo potřeba nastudovat technologie a zvolit vhodnou konfiguraci. Dále jsem vyzkoušel jednoduchou implementaci základního Kafka API a posílání zpráv na jenom stroji. Dalším logickým krokem bylo vyzkoušet komunikaci mezi dvěma různými počítači v síti. V tomto bodě jsem mohl přidat vyhodnocení vzorů událostí pomocí aplikačního rámce Esper.

Aplikace byla tedy nachystána na přidání více počítačů a testování distribuovaného chování. Napsal jsem několik skriptů pro automatizaci vývoje. Skripty jsou součástí práce jako elektronická příloha. Když aplikace zvládala posílání zpráv mezi čtyřmi testovacími stroji mohl jsem začít pracovat na poslední a nejsložitější iteraci. Tou byla integrace aplikačního rámce ZooKeeper. Na základě vyhodnoceného vzoru událostí ZK automaticky sdružuje počítače do pracovních skupin. Skupiny také ruší a umožňuje nasazení nových pravidel pro analýzu.

ZK je jádrem algoritmu. Na každém uzlu je navěšen posluchač, který zaznamenává změny dat. Aplikace je řízena tím, že je do příslušného uzlu vložen nový datový objekt, který obsahuje akci a příslušná data. Například akce *CREATE* znamená vytvoření nového vlákna, které bude reprezentovat konzumenta dat nebo datového producenta. Ukázka datového objektu je na obrázku 6.4.

```
{
   action: "CREATE",
   mode: "producer",
   level: "LEVEL1",
   parent: "<ip>",
   path: "<zk-path>"
}
```

Obrázek 6.4: Ukázka datového objektu pro akci *CREATE* při vytváření producenta

Seznam hlavních řídících akcí pro ZooKeeper:

- CREATE – vytváří nové vlákno s producentem nebo konzumentem
- CREATE_CHILDREN_PRODUCER – pro daný uzel vytvoří úplně nového potomka a spustí v něm vlákno s producentem
- STOP_PRODUCER – zastaví producenta (stopne a zabije vlákno)
- STOP_CONSUMER – zastaví konzumenta (stopne a zabije vlákno)
- SET_EP_RULE – nastaví v uzlu nové pravidlo pro analýzu
- DELETEd_SELF – smaže sama sebe (ZK uzel) včetně zabití všech konzumentů a producentů, které v rámci daného uzlu běží

6.2.1 Spuštění

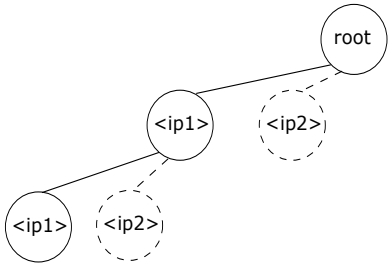
Před spuštěním algoritmu, tedy analýzy dat, je potřeba nastartovat aplik. rámec Kafka. Záleží na počtu replikací. V našem případě je Kafka pouze jako jedna instance. Kafka běží přímo na jednom ze strojů. Dále na každém stroji spustíme ZK. ZK potřebuje ke správnému fungování, aby bylo online více než n/2 uzlů, kde n je celkový počet uzlů, které jsou v clusteru. V této práci probíhá testování na čtyřech strojích, tedy pro ZK musí být online nejméně tři. Jakmile jsou spuštěny tři stroje, ZK zvolí řídící instanci.

Při spuštění samotné Java aplikace je potřeba zadat několik vstupních parametrů:

- **ip** – ip adresa stroje. Slouží také jako jednoznačný identifikátor v ZK
- **zklist** – seznam adres, kde běží jednotlivé instance ZK. Prvky v seznamu jsou odděleny čárkou (bez mezery). List potřebuje aplikační rámec Curator [10] pro připojení k ZK.
- **m** – označuje mód ve kterém je aplikace spuštěna. Jsou možné dva módy:
  * **producer** – aplikace je spuštěna jako producent. To znamená, že se zapojí do ZK stromu, zaregistruje příslušné posluchače a začne posílat data do aplik. rámce Kafka. Do tématu, pojmenovaného podle ip adresy.

* **combined** – aplikace je spuštěna jako producent i konzument. Kromě producenta je vytvořen i konzument, který přijímá data z aplik. rámce Kafka z tématu podle ip adresy. Data předává na analýzu do aplikačního rámce Esper.

V aplikaci je nastaveno výchozí pravidlo, podle kterého se mají data analyzovat. Jakmile je aplikace úspěšně spuštěna, začne produkovat, případně i konzumovat a analyzovat data. Spuštění každé jedné aplikace znamená, že se v ZK vytvoří dva nové uzly, tak jak je vidět na obrázku 6.5. Postupně se buduje strom až do finálního stavu, který je vidět na obrázku 5.3.



Obrázek 6.5: Postupné přidávání uzlů do ZK stromu

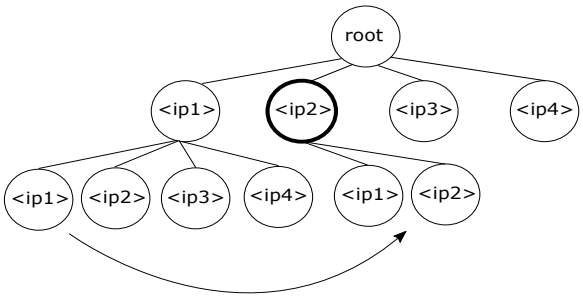
6.2.2 Přechody mezi stavy

V předchozí kapitole jsem rozvedl jak probíhá spuštění aplikace do jakého stavu se po spuštění dostane. Tato kapitola se bude podrobněji věnovat tomu, v jakých stavech se může DEM nacházet. Schválně zde zmiňuji DEM, protože předchozí kapitola se věnovala hlavně spuštění z pohledu jedné instance. Zde jsou naproti tomu popsány stavy více z globálního pohledu.

Jakmile je spuštěn konzument, automaticky všechny události předává do aplik. rámce Esper, který je vyhodnocuje. Uvažujme pravidlo, které detekuje příchozích pěti událostí během tří sekund. Každý z počítačů svou aktivitou toto kritérium splní. Jakmile Esper detekuje splnění pravidla, vytvoří událost. Událost je v aplikaci zpracována. Pokud je aktivní jen jeden počítač, nemá smysl spouštět detailní analýzu. Naopak v případě, že vyhodnocení pravidla způsobilo více počítačů, je žádoucí na této podmnožině spustit detailní analýzu.

Přechod do nového stavu začíná volbou nového konzumenta. Tedy počítače, který bude analyzovat proud jemnozrných událostí. Pro volbu jsem zvolil postup, kdy je nalezen první takový počítač, který ještě neprovádí žádnou analýzu. Díky zvolené ZooKeeper struktuře je to jednoduché. Stačí projít první úroveň a hledat uzel, který ještě nemá žádné potomky. Takovému uzlu je nejdříve poslána akce *CREATE*[1], aby spustil vlákno s konzumentem. Pak tolik akcí *CREATE_CHILDREN_PRODUCER*, kolik strojů je určených k jemnozrné analýze. Samozřejmě součástí každé akce jsou i potřebná data (ip adresa počítače apod.).

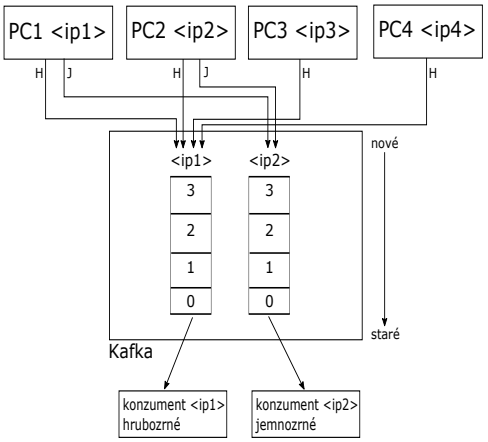
Na obrázku 6.6 je vidět příklad, kdy byly pro jemnozrnou analýzu vybrány stroje *ip1* a *ip2*. Jak je vidět hrubozrná analýza stále probíhá na stroji *ip1*, proto musí být jako konzument jemnozrných událostí zvolen stroj *ip2*.



Obrázek 6.6: Ukázka ZK stromu při současné analýze jemnozrných a hrubozrných událostí

Jak už bylo řečeno dříve, ZooKeeper reprezentuje logickou strukturu. Na obrázku 6.7 je znázorněn odpovídající stav v aplikačním rámci Kafa. Protože už existují dva konzumenti, jsou v aplik. rámci Kafka zaregistrována dvě témata.

1. Řídící akce ze seznamu z kapitoly 6.2



Obrázek 6.7: Ukázka stavu aplik. rámce Kafka při současné analýze jemnozrných a hrubozrných událostí. Zkratky *J* a *H* na obrázku znamenají jemnozrné a hrubozrné události.

V aktuálním stavu se tedy zároveň analyzují hrubozrné i jmenozrné události. Cílem další hrubozrné analýzy je odhalit jestli danému pravidlu neodpovídají ještě další počítače. Samozřejmě, pokud bude v průběhu jemnozrné analýzy vyhodnocen stejný vzor (stejná sada počítačů), pak se znovu jemnozrná analýza nespouští. Cílem jemnozrné analýzy je odhalit příčinu vzniku nestandardní situace. Logicky jsou tedy na uzlech *ip1* a *ip2* nasazena jiná pravidla detekující vzory událostí.

Jak jemnozrná, tak hrubozrná pravidla jsou omezena časovou platností. Po uplynutí této doby je hrubozrná analýza zastavena a DEM čeká na nasazení jiného (nebo stejného) pravidla. V případě jemnozrné analýzy je celá práce ukončena a je potřeba "uklidit". To znamená zastavit vlákna s producenty i konzumentem a smazat příslušné uzly v "druhé vrstvě"ZK stromu. Tak aby se DEM dostal do stavu z obrázku 5.3.

Do uzlu, který zpracovává hrubozrné události je možné kdykoli nasadit nové pravidlo pro detekci vzorů událostí.

6.3 Testování

Kapitola Testování se věnuje ukázkám datových objektů, které byly použity při testování funkčnosti algoritmu. Také jsou zde popsána pravidla, která byla použita pro analýzu. Data neodpovídají reálným, ale jsou vymyšlena aby simulovala určité situace, které jsou postaveny na reálných základech.

Protože bylo z počátku složité uchopit celou myšlenku, začal jsem testovat algoritmus po jednotlivých akcích[2]. Pravidlo pro detekci bylo jednoduché a rozlišovalo pouze události úrovně jedna a dvě. Data byla generována na každém stroji v náhodných intervalech *<0.5s - 1s>*. Ukázka datového objektu je na obrázku 6.8.

```
{
   level: "LEVEL1 / LEVEL2",
   source: <zk-path>
}
```

Obrázek 6.8: Ukázka datového objektu z první fáze testování

V této fázi bylo hlavním cílem zjistit jestli Esper správně detekuje vzor událostí. S velkými problémy jsem se setkal při vytváření uzlů v ZK. Někdy se uzly vytvářely duplicitně, někdy se vytvořily pod špatným rodičem. Nějakou dobu také trvalo zvolit správný postup vytváření a rušení vláken konzumentů a producentů aby byla správně uvolňována paměť. Jeden příklad za všechny: Když je na uzel v ZK navěšen nějaký posluchač a uzel je smazán, neznamená to, že se posluchač stane neaktivním a zruší se. Je ho potřeba explicitně vyjmout z kolekce a až potom smazat uzel. Jinak vznikají duplicity v posluchačích a akce jsou vyhodnoceny vícekrát.

Jakmile byly vyřešeny problémy při jednotlivých akcích, bylo potřeba vymyslet reálnější datový model, který bude ukazovat myšlenku hrubozrné a jemnozrné analýzy. Proto jsem datový objekt rozšířil o další vlastnosti. Viz obrázek 6.9. Myšlenka analýzy je tedy taková, že v akcích úrovně jedna se hledá vzor, který by nasvědčoval útoku. Pokud

2. Řídící akce z kapitoly 6.2

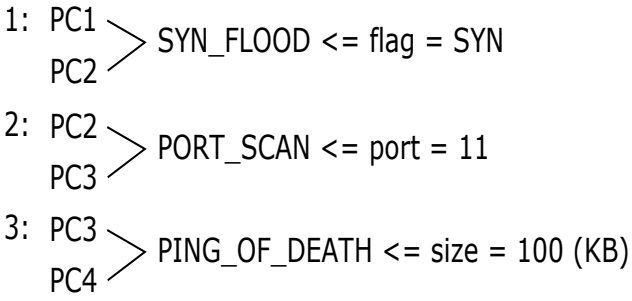
je takový vzor nalezen, je nad podezřelými počítači spuštěna detailní (jemnozrná analýza). Ta spočívá v počítání všech událostí úrovně dva. Pokud jejich počet v určitém časovém okně přesáhne daný limit, je "útok"prohlášen za reálný a zpráva je zapsána do logu.

```
{
   level: "LEVEL1 / LEVEL2",
   source: <zk-path>,
   msg: "info textual message",
   flag: "SYN",
   size: 10,
   port: 80
}
```

Obrázek 6.9: Ukázka datového objektu z první fáze testování, kde flag reprezentuje příznak v datagramu (SYN a ACK), size velikost paketu a port je cílový port v komunikaci.

Esper pravidla

Z datového objektu na obrázku 6.9 jsou pro nás nyní důležité atributy flag, size a port. Na obrázku 6.10 je vidět jak počítače generují testovací data. Neboli distribuce generovaných dat, které reprezentují potenciální hrozbu, mezi počítači.



Obrázek 6.10: Znázornění které počítače v testovacím prostředí produkují kritické typy datových objektů. Zbylé dva atributy jsou u každého počítače nastaveny na "neutrální"hodnotu.

Data u jednotlivých útoků jsou založena na skutečné podstatě, i když jde spíše o princip. Ve skutečnosti bývají síťové útoky velmi komplikované. Například určitě neplatí, že série dotazů na port číslo 11, automaticky znamená útok skenováním portů. Data jsou primárně zvolena pro demonstraci funkčnosti algoritmu.

Na jednotlivé typy útoků jsou nachystaná pravidla pro hrubozrnou analýzu, která mají potenciální hrozbu detekovat:

```
SYN_FLOOD: "select source, count(*) as cnt, level from
IncomingEvent(level='LEVEL1').win:time_batch(5 sec) WHERE
flag = 'SYN' group by source, level having count(*) > 3"
```

Obrázek 6.11: Pravidlo pro detekci útoku 'syn flood' (záplava pakety SYN)

```
PORT_SCAN: "select source, count(*) as cnt, level from
IncomingEvent(level='LEVEL1').win:time_batch(5 sec) WHERE
port = '11' group by source, level having count(*) > 3"
```

Obrázek 6.12: Pravidlo pro detekci útoku 'port scan' (skenování portů)

```
PING_OF_DEATH: "select source, count(*) as cnt, level from
IncomingEvent(level='LEVEL1').win:time_batch(5 sec) WHERE
size = '100' group by source, level having count(*) > 3"
```

Obrázek 6.13: Pravidlo pro detekci útoku 'ping of death' (ping smrti)

Posledním pravidlem, je pravidlo pro analýzu jemnozrných událostí. Detekuje jestli je počet přijatých událostí úrovně dvě během 3 sekund větší než sedm. Pokud ano, jemnozrná analýza ukázala že podezřelé počítače jsou opravdu pod útokem.

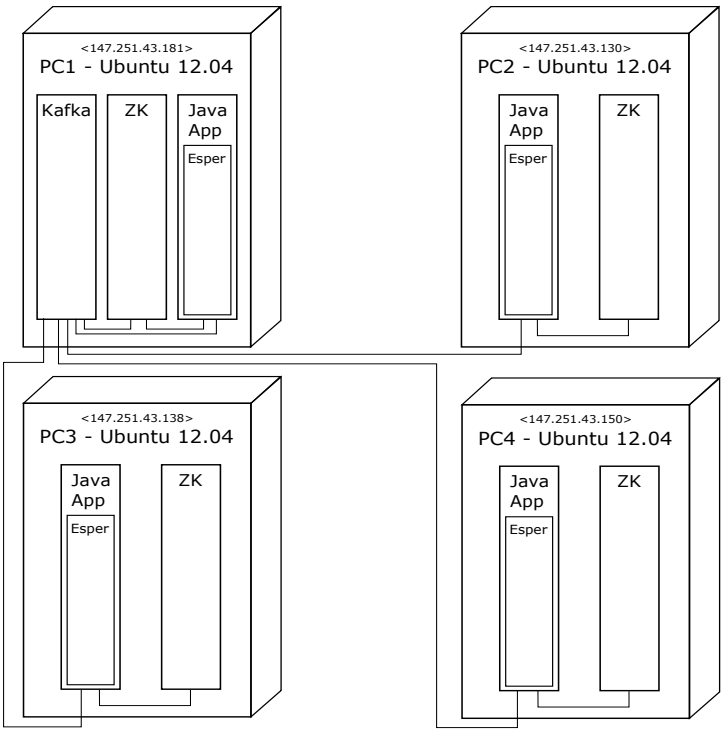
```
"select source, count(*) as cnt, level from
IncomingEvent(level='LEVEL2').win:time_batch(5 sec) group
by source, level having count(*) > 7"
```

Obrázek 6.14: Pravidlo pro analýzu jemnozrných událostí

6.4 Demo

Tato kapitola má za cíl demonstrovat reálný běh aplikace na testovacích strojích. Všechny výstupy aplikace jsou logovány do souboru. Myslím si, že vhodným způsobem ukázky běhu programu jsou konkrétní výňatky z *log* souborů. Každá ukázka je popsána, aby byly výstupy snadno pochopitelné.

Před samotnou demonstrací běhu programu ještě uvádím diagram nasazení 6.15. Ten dává jasnou představu o tom, které technologie jsou v jednotlivých uzlech nasazeny a v jakém jsou vztahu.



Obrázek 6.15: Diagram nasazení

V diagramu nasazení jsou nad jeho rámec uvedeny i konkrétní ip adresy strojů. Ip adresy jsou součástí logů a jsou tedy i součástí demonstrace.

Postupně budu na třech ukázkách demonstrovat běh aplikace v následujícím scénáři:

1. Nasazení pravidla pro hrubozrnou analýzu.
2. Detekce hrubozrných událostí na dvou počítačích.
3. Spuštění jemnozrné analýzy pro tyto dva počítače.
4. Potvrzení potenciálního útoku při jemnozrné analýze.
5. Návrat aplikace do iniciálního stavu.

První ukázka 6.16 obsahuje výpis ze dvou počítačů <*ip1*> a <*ip2*> při nasazování pravidla pro hrubozrnou analýzu. Postupně je uveden

výpis událostí na obou počítačích. Podrobný popis každého ze čtyř bodů je pod ukázkou.

```
1. zk-cli console:
[zk: 147.251.43.181:2181] set /root/147.251.43.181
{"action":"SET_EP_RULE","rule":"SYN_FLOOD"}

2. PC<ip1>:
INFO - Event received.
INFO - Bean: /root/147.251.43.181/147.251.43.181
INFO - Bean: /root/147.251.43.181/147.251.43.130
INFO - New Parent: 147.251.43.130
INFO - Create new consumer at: /root/147.251.43.130
INFO - Set data at node: /root/147.251.43.181
INFO - Set data at node: /root/147.251.43.130

3. PC<ip1> & PC<ip2>:
INFO - Evaluated action will be: CREATE_CHILDREN_PRODUCER.
   Data: {"parent":"147.251.43.130", "level": "LEVEL2"}

4. PC<ip1>:
INFO - Event received.
INFO - New Parent: null
```

Obrázek 6.16: Nasazení pravidla na hrubozrnou analýzu

Popis jednotlivých bodů z ukázky 6.16:

1. Nasazení pravidla v ZK konzoli. Pravidlo[3] má za cíl detekovat útok *syn flood*.

2. Aplikace spuštěná na stroji <*ip1*> analyzuje hrubozrné události. Druhý bod ukazuje, že pravidlu vyhověly dva počítače. Byl zvolen vhodný kandidát pro jemnozrnou analýzu (<*ip2*>). Dále jsou poslány řídící instrukce s daty oběma počítačům. <*Ip2*> se stane konzumentem i producentem, <*ip1*> se stane pouze producentem.

3. Detail pravidla je na ukázce 6.11

3. Třetí bod ukazuje přijetí řídící akce na obou počítačích. V objektu jsou i nezbytná data. V ukázce je vidět, pro zjednodušení, pouze cíl (*parent*) pro posílání dat a typ dat, která se mají posílat (*level*).

4. V tomto bodě je vidět, že hrubozrná analýza na stroji <*ip1*> stále pokračuje a vyhodnotila znovu stejný vzor událostí. Ale skupina pro jemnozrnou analýzu už byla vytvořena, takže se znovu nevytváří.

Druhá ukázka 6.17 znázorňuje detekci vzoru při jemnozrné analýze na PC <*ip2*>. Pravidlo[4] vyhodnocuje události v plovoucím časovém okně pět vteřin. Během těchto pěti vteřin, ve kterých Esper analyzoval příchozí událostí bylo detekováno osm a jedenáct výskytů. To je více než minimální hranice sedm událostí, kterou specifikuje pravidlo. Jednoduše řečeno, útok *syn flood* je potvrzen.

```
PC<ip2>:
INFO - Event received - level 2. Printing info. PC count: 2
INFO - Event data. Source:
   /root/147.251.43.130/147.251.43.130, count: 8
INFO - Event data. Source:
   /root/147.251.43.130/147.251.43.181, count: 11
```

Obrázek 6.17: Detekce vzoru při jemnozrné analýze

Na posledním snímku 6.18 je vidět postupné zastavování jemnozrné analýzy na počítači <*ip2*>. Tím se DEM vrací do výchozího stavu z obrázku 5.2. Odshora jsou vypsány události:

1. zastavení vlákna producenta (je také provedena na počítači <*ip1*>)

2. smazání uzlu v ZK stromu

3. zastavení vlákna konzumenta

4. Detail pravidla je na ukázce 6.14

```
PC<ip2>:
INFO - Evaluated action will be: STOP_PRODUCER
INFO - Deleting node: /root/147.251.43.130/147.251.43.130
INFO - Evaluated action will be: STOP_CONSUMER
```

Obrázek 6.18: Postupné zastavování jemnozrné analýzy

Data, použitá v ukázkách jsem zvýraznil také na snímcích obrazovky, které jsou součástí přílohy A této práce.

6.5 Známá omezení

Pomocí navrhovaného řešení je možné analyzovat širokou škálu událostí. Nejen dopady síťové komunikace, ale i síťový provoz jako takový. Dále je možné řešení nasadit například na analýzu aplikačních logů.

Během testování jsem narazil na některá omezení, nedostatky nebo doporučení, která v této kapitole rozvedu. K některým situacím uvedu i konkrétní řešení. U některých je řešení předmětem dalšího rozvoje algoritmu nebo jeho podrobení výkonnostním testům.

Nejdříve uvedu kompletní seznam. Každému bodu je následně věnován odstavec s popisem.

- Momentální nemožnost spustit více konzumentů na jednom stroji (Více konzumentů).
- Monitoring a spouštění jednotlivých uzlů (Monitoring).
- Dynamické přidávání nových uzlů do hierarchie (Přidávání uzlů).
- Výpadky některých uzlů. Zookeeper je na to připraven, Kafka také, ale není jasné do jakého stavu se dostane stromová hierarchie ZK (výpadky).

Více konzumentů

Tím, že jsou témata v aplikačním rámci Kafka členěna podle ip adres počítačů, není možné mít na jenom stroji více než jednoho konzumenta.

Kdyby to tak bylo, četli by oba ze stejného tématu a "kradli si zprávy". Teoreticky by bylo možné aby se konzumenti dívali na obsah zpráv a podle toho je posílali do některé z instancí aplik. rámce Esper, ale takové řešení je příliš náchylné k chybám a těžko se škáluje.

Vhodným řešením by bylo přidat ke každému tématu i přepážky, které by rozlišovaly druh zpráv. V případě této práce by v každém tématu byly dvě přepážky. Jedna pro jemnozrné (*LEVEL2*) a druhá pro hrubozrné události (*LEVEL1*). Přepážky je možné dynamicky přidávat a tím mít možnost na jednom stroji analyzovat události množství různých typů.

Monitoring

Pokud by algoritmus nasazen do produkčního prostředí, byl by zároveň potřeba zavést důsledný monitoring. Nejen to, jestli všechny instance pracují správně, ale také jaká pravidla jsou aktuálně nasazena a na jakých strojích. Dále vytížení aplikačního rámce Kafka. Objemy zpracovávaných dat. Atd.

Přidávání uzlů

Uvažoval jsem nad tím, jak by se dalo do systému dynamicky přidávat další stroje. Celá architektura je na to připravená. Jediný problém je u ZK. Jeho konfigurace je statická a zároveň se seznam adres předává i jako vstupní parametr při startu aplikace. Před verzí *3.5.0* bylo jedinou možností napsat skript, který bude postupně provádět rekonfiguraci. Od verze *3.5.0* má ZooKeeper v dokumentaci [8] popsán mechanizmus pro dynamickou rekonfiguraci[5].

Výpadky

V rámci testování jsem zkoušel jak se aplikace vzpamatuje z výpadků některých uzlů. Chování nebylo příliš predikovatelné. Záleželo na tom v jaké fázi analýzy došlo k výpadku. Jestli byla spuštěna jemnozrná analýza nebo nikoliv. Pravidelně se opakovaly dva problémy. V ZK zůstal zaregistrovaný některý z posluchačů a při obnově se stal znovu

5. Konkrétní část dokumentace:
https://zookeeper.apache.org/doc/trunk/zookeeperReconfig.html

aktivním a tím docházelo k duplicitnímu zpracování řídících akcí. Druhým problémem byly nesmazané uzly ze ZooKeeper stromu, který se používá jako reprezentace stavu DEM.

7 Závěr

TODO - dopsat

Bibliografie

[1] Celeda, Pavel. Network Traffic Analysis for Cyber Security. Brno, 2013. Masaryk Univerzity. [online]. Dostupné z: https://is.muni.cz/do/rect/habilitace/1433/44368572/44368651/HP_kor.-verejna.pdf

[2] LUCKHAM, David. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems.* Addison-Wesley Professional, 2002. ISBN 978-0-201-72789-0.

[3] COULOURIS, George F. *Distributed systems: concepts and design*. 5th ed. Boston: Addison-Wesley, c2012. ISBN 01-321-4301-1.

[4] KAMBURUGAMUVE, Supun; FOX, Geoffrey; LEAKE, David and QIU, Judy. *Survey of Distributed Stream Processing for Large Stream Sources*. Technical report, 2013. [online]. Dostupné z: http://grids.ucs.indiana.edu/ptliupages/publications/survey_stream_processing.pdf

[5] Apache Maven [online]. Dostupné z: http://maven.apache.org

[6] Apache Kafka [online]. Dostupné z: http://kafka.apache.org

[7] Attribution modeling overview. Official Google documentation [online]. Dostupné z: https://support.google.com/analytics/answer/1662518?hl=en&ref_topic=3205717

[8] Apache ZooKeeper [online]. Dostupné z: https://zookeeper.apache.org/

[9] GAMMA, Erich, Richard HELM, Ralph JOHNSON a John VLISSIDES. GANG OF FOUR. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994. ISBN 0-201-63361-2

[10] Apache Curator [online]. Dostupné z: http://curator.apache.org/index.html

[11] Apache Samsa [online]. Dostupné z: https://samza.apache.org

[12] Process real-time big data with Twitter Storm [online]. Dostupné z: http://www.ibm.com/developerworks/library/os-twitterstorm/

[13] Real time insights into LinkedIn's performance using Apache Samsa [online]. 2014. Dostupné z: https://engineering.linkedin.com/samza/real-time-insights-linkedins-performance-using-apache-samza

[14] DEBS 2015 [online]. Dostupné z: http://dblp.uni-trier.de/db/conf/debs/debs2015.html

[15] CUGOLA, Gianpaolo, Alessandro MARGARA, Mauro PEZZE a Matteo PRADELLA. Efficient analysis of event processing applications [online]. New York, 2015. Dostupné z: http://dl.acm.org/citation.cfm?doid=2675743.277183

[16] Esper [online]. Dostupné z: http://www.espertech.com/esper/

[17] Esper perforamnce [online]. Dostupné z: http://www.espertech.com/esper/performance.php

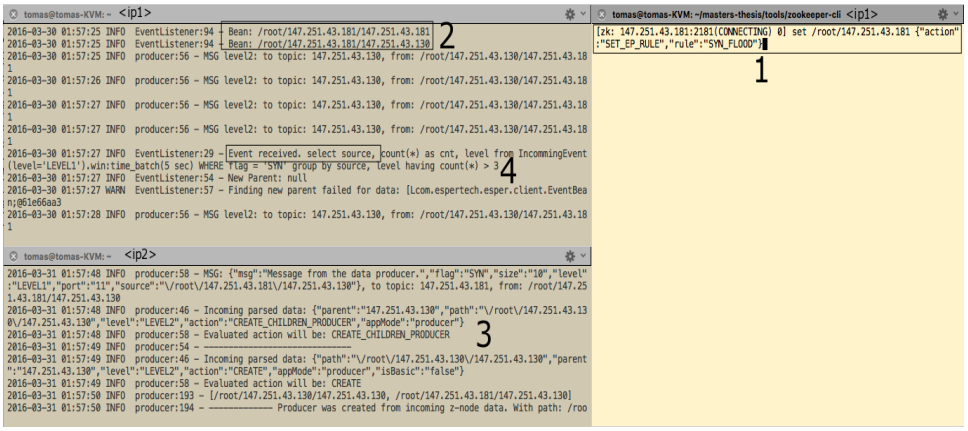
[18] Apache Storm [online]. Dostupné z: http://storm.apache.org

[19] LEIBIUSKY, Jonathan, Gabriel EISBRUCH a Dario SIMONASSI. Getting Started with Storm. O'Reilly Media, 2012. ISBN 978-144-9324-018.

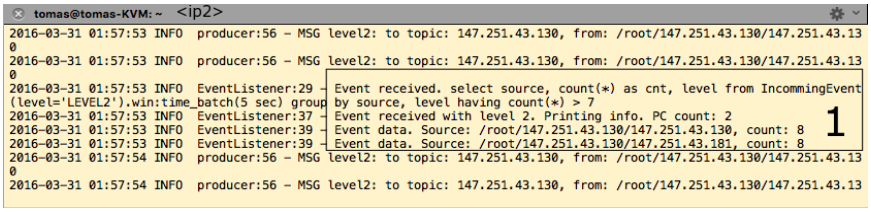
[20] Companies Using Apache Storm [online]. Dostupné z: http://storm.apache.org/documentation/Powered-By.html

A Příloha

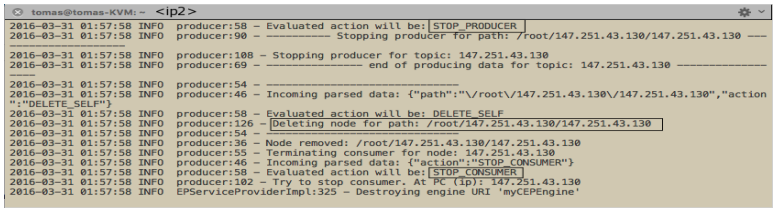
V kapitole 6.4 jsem vytvořil ukázky za pomocí dat z *log* souborů. V této příloze jsou snímky obrazovky na kterých jsou zvýrazněny záznamy, které jsem v ukázkách použil. Snímky po řadě odpovídají jednotlivým ukázkám.



Obrázek A.1: Nasazení pravidla na hrubozrnou analýzu



Obrázek A.2: Detekce vzoru při jemnozrné analýze



Obrázek A.3: Postupné zastavování jemnozrné analýzy