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 Peer to Peer 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

keyword1, keyword2, …

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. Velkým zjednodušením dostaneme "distributed event matching", neboli DEM. Pro účely této práce a snadnější orientaci budu popisované řešení identifikovat zkratkou *DEM*.

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 $1Tb/s$, 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. [1] 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

Je těžké shrnout celý vědní obor pod jednu všeobjímající definici. David Luckham ve své knize THE POWER OF EVENTS: AN INTRODUCTION TO COMPLEX EVENT PROCESSING IN DISTRIBUTED ENTERPRISE SYSTEMS [1] říká, že CEP je 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, 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 [6]. 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). [2] 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* [1] v popisu obrázku *1.1*.

- Distribuované zpracování událostí jako výpočet rozdělený na více menších, méně náročných úloh za účelem rychlejšího zpracování s využitím paralelismu. 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. [3]

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

3 Nástroje pro distribuované zpracování událostí

3.1 Apache Samsa

3.2 Apache Storm

3.3 CAVE

http://dl.acm.org/citation.cfm?doid=2675743.2771834

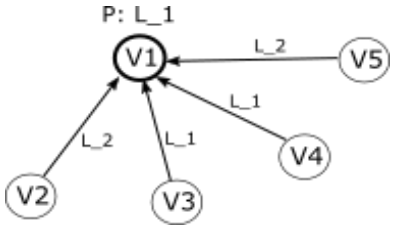
4 Analýza

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.

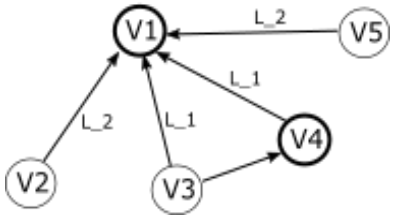
Pravidla pro detekci komplexních událostí, budou mít časovou platnost. Po uplynutí daného času se pravidlo automaticky stane neaktivní. Pravidla bude také možné nasadit pouze na určitou podmnožinu uzlů V grafu G.

Na obrázku 4.1 je vidět příklad grafu, kde je na uzlu *V1* nasazeno pravidlo analyzující události *L_1*.



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

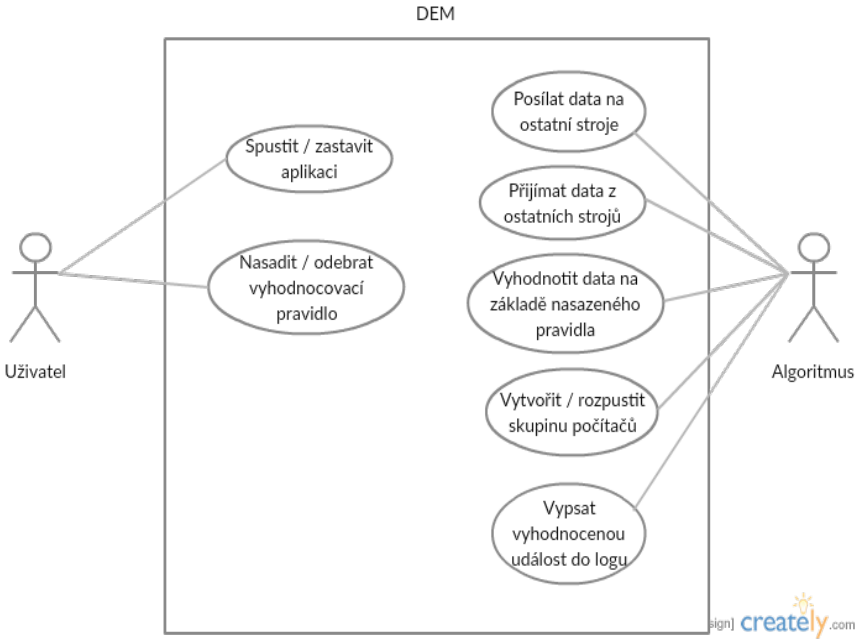
Po určitém počtu výskytů události *L_1* je vytvořena nová skupina z uzlů V3 a V4 kde je V4 zvolen hlavním uzlem. Viz obr. 4.2



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

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

Na obrázku 4.3 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 4.3: Diagram užití

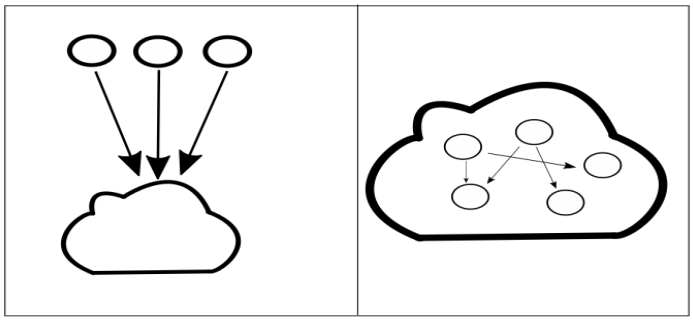
Typickým místem pro nasazení distribuovaného zpracování událostí je analýza síťové komunikace. To je také místo, kde bude DEM nasazen.

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. Hrubou představu znázorňuje obrázek 4.4



Obrázek 4.4: Srovnání přístupu zasílání dat do cloudu (v levo) a vytvoření cloudu přímo na koncových zařízeních (v pravo)

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.

5 Návrh

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

5.1 Technologie

Tato kapitola podkapitola jednotlivé technologie, které jsou použity při implementaci algoritmu. Na konci každé subsekce je popis toho jak konkrétně je technologie použita v mém řešení.

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. 4 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 Kafky. 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 Kafky 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 ze ZooKeeperu 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. [5] Klastr Kafky může být rozdělený mezi několik počítačů, každý nazýváme *broker*. Základem Kafky 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.

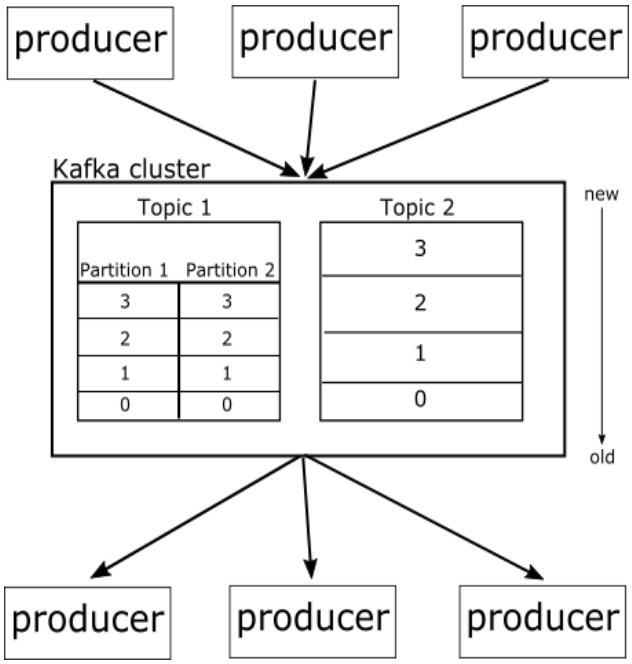
Konzument a producent

Do kafky 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

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

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 dokumentaci Kafky se říká [5]:

- 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].



Obrázek 5.1: Znázornění základního schématu Kafky

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

Garance

Kafka poskytuje následující seznam garancí:

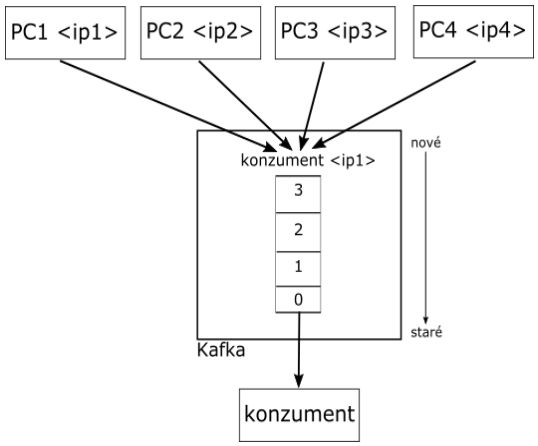
- Zprávy poslané do určitého tématu budou seřazeny v pořadí v jakém byly odeslány. Například pokud je zpráva *M1* odeslána producentem dříve než zpráva *M2*, pak Kafka zaručuje, že bude mít zpráva *M1* menší offset a bude v logu zobrazena dříve než *M2*.

- 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 Kafky při iniciálním spuštění

5.1.3 Apache ZooKeeper

ZooKeeper je další z rodiny "open-source"Apache technologií. Je to centralizovaná služba pro správu konfiguračních dat, pojmenování a poskytování synchronizace distribuovaných uzlů. [7]

ZooKeeper 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 ZooKeeperu je sestaven z uzlů zvaných *znodes*. Na rozdíl od klasického souborového systému, který je navržen pro dlouhodobé ukládání dat, ZooKeeper 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 ZooKeeperu 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 ZooKeeper 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

ZooKeeper 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 ZooKeeperu. 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.

4. Z anglického slova *Watches*. Jde o konkrétní implementaci návrhového vzoru "observable". [8]

Apache Curator

Aplikační rámec *Curator* je nástroj vytvořený společností Neflix pro jednodušší práci se ZooKeeperem. Rozhraní ZooKeeperu 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:

- Komplexnější rozhraní pro jednodušší práci se ZooKeeperem. 5.1
- Automatické připojení na instanci ZooKeeperu 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

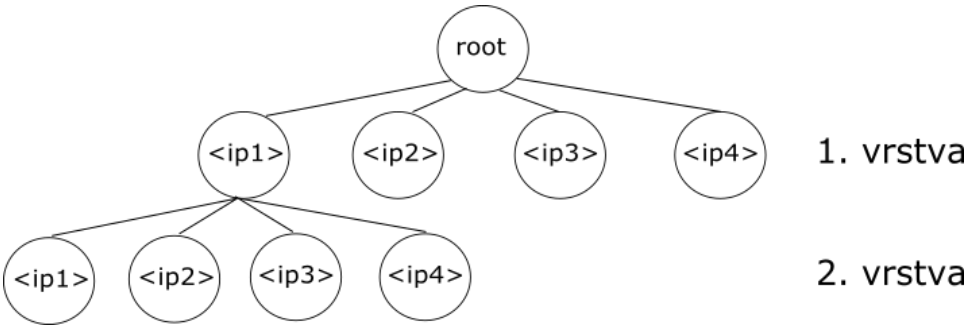
ZooKeeper je použit pro řízení běhu celého procesu DEM. Jeho hlavní úlohou je distribuce řídících dat mezi jednotlivými výpočetními jednotkami. Druhou funkcí, která logicky vychází ze stromové struktury datového modelu, je reflexe aktuálního stavu aplikace.

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 Kafky 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 Zookeeperu při iniciálním spuštění.

5.1.4 Esper

TODO - dopsat

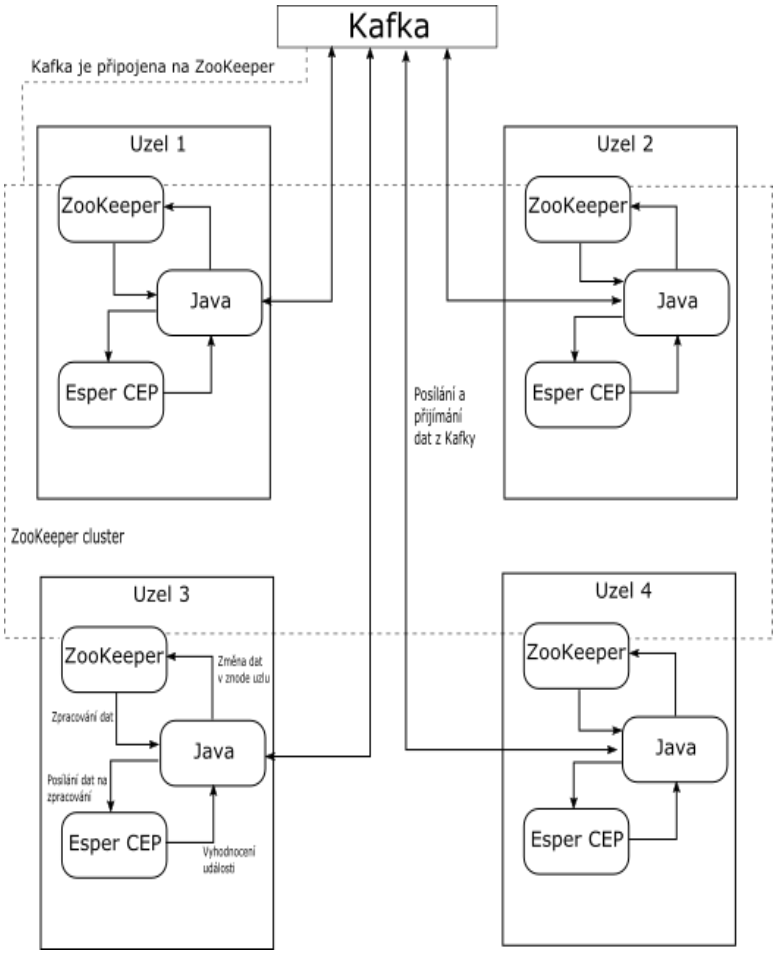
Použití

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	ZooKeeper + Kafka
Nasazení nového pravidla	ZooKeeper
Definování chování jednotlivých strojů	ZooKeeper
Vytvořit nebo rozpustit skupinu strojů	ZooKeeper
Nalezení určitého vzoru (pattern) v datech	Esper
Propojit technologie do funkčního celku	Java + Maven

Tabulka 5.1: Splnění požadavků na systém



Obrázek 5.4: 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 pa-

kety 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 ma 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

Zahrnout i popis prostředí OS a hw konfiguraci testovaných virtuálek.

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 Kafce 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 API Kafky 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 Esperu.

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 ZooKeeperu. Na základě vyhodnoceného vzoru událostí ZooKeeper automaticky sdružuje počítače do pracovních skupin. Skupiny také ruší a umožňuje nasazení nových pravidel pro analýzu.

ZooKeeper 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 6.1.

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

Obrázek 6.1: 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 (ZooKeeper uzel) včetně zabití všech konzumentů a producentů, které v rámci daného uzlu běží

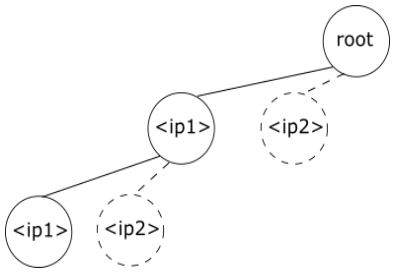
6.2.1 Spuštění

Před spuštěním algoritmu, tedy analýzy dat, je potřeba nastartovat Kafku. Záleží na počtu replikací Kafky. V našem případě je Kafka pouze jako jedna instance. Kafka běží přímo na jednom ze strojů. Dále na každém stroji spustíme ZooKeeper. ZooKeeper 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 ZooKeeper musí být online nejméně tři. Jakmile jsou spuštěny tři stroje, ZooKeeper zvolí řídící instanci.

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

- **ip** – ip adresa stroje. Slouží také jako jednoznačný identifikátor v ZooKeeperu
- **zklist** – seznam adres, kde běží jednotlivé instance ZooKeeperu. Prvky v seznamu jsou odděleny čárkou (bez mezery). List potřebuje aplikační rámec Curator [9] pro připojení k ZooKeeperu.
- **m** – označuje mód ve kterém je aplikace spuštěna. Jsou možné dva módy:
  * **producer** – aplikace je spuštěna jako producent. To znamená, že se zapojí do ZooKeeper stromu, zaregistruje příslušné posluchače a začne posílat data do Kafky. Do tématu, pojmenovaného podle ip adresy.
  * **combined** – aplikace je spuštěna jako producent i konzument. Kromě producenta je vytvořen i konzument, který přijímá data z Kafky z tématu podle ip adresy. Data předává na analýzu Esperu.

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 ZooKeeperu vytvoří dva nové uzly, tak jak je vidět na obrázku 6.2. Postupně se buduje strom až do finálního stavu, který je vidět na obrázku 5.3.



Obrázek 6.2: Postupné přidávání uzlů do ZooKeeper stromu

6.2.2 Přechody mezi stavy

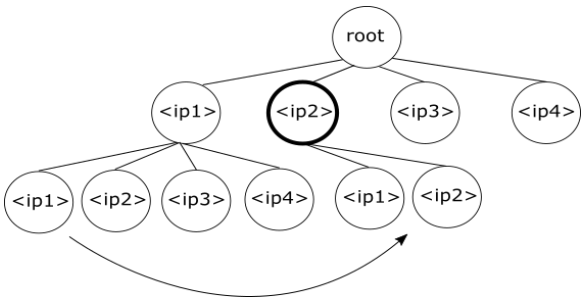
V předchozí kapitole jsem rozvedl jak probíhá spuštění aplikace do jakého stavu se po spuštění dostane. Tato kapitola se bude podrobněji věnovat tomu, v jakých stavech se může DEM nacházet. Schválně zde zmiňuji DEM, protože předchozí kapitola se věnovala hlavně spuštění z pohledu jedné instance. Zde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

Jakmile je spuštěn konzument, automaticky všechny události předává do Esperu, 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.).

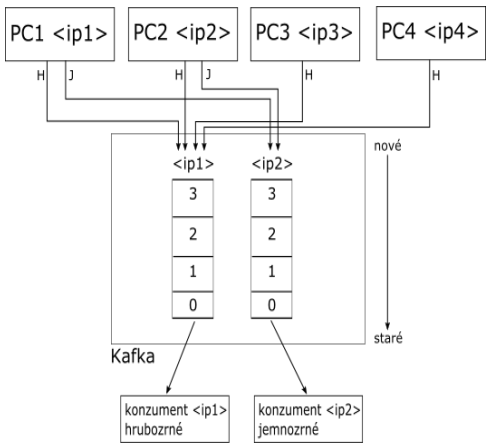
1. Řídící akce ze seznamu z kapitoly 6.2

Na obrázku 6.3 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 6.3: Ukázka ZooKeeper stromu při současné analýze jemnozrných a hrubozrných událostí

Jak už bylo řečeno dříve, ZooKeeper reprezentuje logickou strukturu. Na obrázku 6.4 je znázorněn odpovídající stav v Kafce. Protože už existují dva konzumenti, jsou v Kafce zaregistrována dvě témata.



Obrázek 6.4: Ukázka stavu Kafky 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"ZooKeeper 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 6.5.

2. Řídící akce z kapitoly 6.2

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

Obrázek 6.5: 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 ZooKeeperu. 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 nic nezůstalo "viset v paměti". Jeden příklad za všechny: Když je na uzel v ZooKeeperu 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.

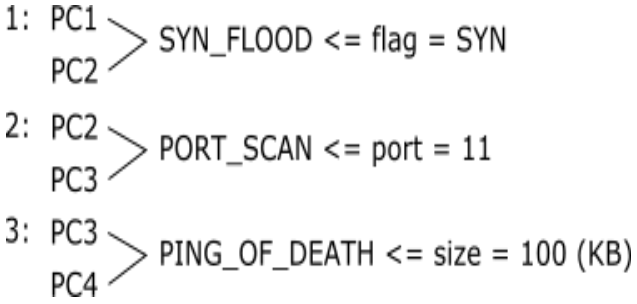
Jakmile byly vyřešeny problémy při jednotlivých akcích, bylo potřeba vymyslet reálnější datový model, který bude ukazovat myšlenku hrubozrné a jemnozrné analýzy. Proto jsem datový objekt rozšířil o další vlastnosti. Viz obrázek 6.6. Myšlenka analýzy je tedy taková, že v akcích úrovně jedna se hledá vzor, který by nasvědčoval útoku. Pokud 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 6.6: 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 6.6 jsou pro nás nyní důležité atributy `flag`, `size` a `port`. Na obrázku 6.7 je vidět jak počítače generují testovací data. Neboli distribuce generovaných dat, které reprezentují potenciální hrozbu, mezi počítači.



Obrázek 6.7: 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
IncommingEvent(level='LEVEL1').win:time_batch(5 sec) WHERE
flag = 'SYN' group by source, level having count(*) > 3"
```

Obrázek 6.8: Pravidlo pro detekci útoku 'syn flood' (záplava pakety SYN)

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

Obrázek 6.9: Pravidlo pro detekci útoku 'port scan' (skenování portů)

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

Obrázek 6.10: 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
IncommingEvent(level='LEVEL2').win:time_batch(5 sec) group
by source, level having count(*) > 7"
```

Obrázek 6.11: Pravidlo pro analýzu jemnozrných událostí

6.4 Demo

Nějaké print screeny. Jednoduše ukázka běhu programu.

6.5 Známá omezení

Diskuse nedostatků nebo možných vylepšení výše navrženého řešení.

- Momentální nemožnost spustit více consumerů na jednom stroji.
- Velmi náročný monitoring a spouštění jednotlivých uzlů.
- Zatím nevím jak je to s dynamickým přidáváním nových uzlů do hierarchie.
- Nejsou vůbec otestovány výpadky některých uzlů. Zookeeper to zvládne, kafka také, ale co se stane s virtuálním zk-tree v aplikaci?

7 Závěr

Závěr bude v tomto případě obsahovat obšírnější zhodnocení toho jak se povedlo splnit zadání. Že výsledkem práce je navržené řešení za použití kafky, zk, esperu, Javy.

Bibliografie

[1] 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.

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

[3] 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`

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

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

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

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

[8] 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

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