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 a návrh

Zde bude zakomponovan i prepis filipova zadani

Aktuálně největším zdrojem dat je jednoznačně síťová komunikace. 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.

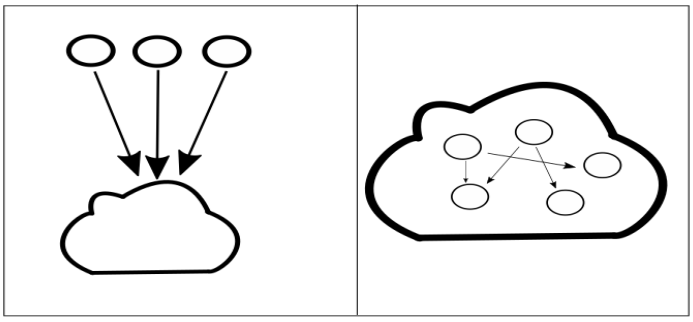
Problém nastane, když je množství produkovaných dat větší než je kapacita analyzujícího (analyzujících) počítačů. V takovém případě můžeme přidat výpočetní sílu nebo zvolit úplně jiný přístup k datové analýze.

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 aplikace znázorňuje obrázek 4.1

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.



Obrázek 4.1: Srovnání

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 Vytvoření clusteru v rámci sítě

Kapitola je jádrem této práce. Popisuje implementaci a fungování DEM.

5.1 Motivace

5.2 Technologie

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

5.2.1 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 5.2.4. 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 5.2.3 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.

5.2.2 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.

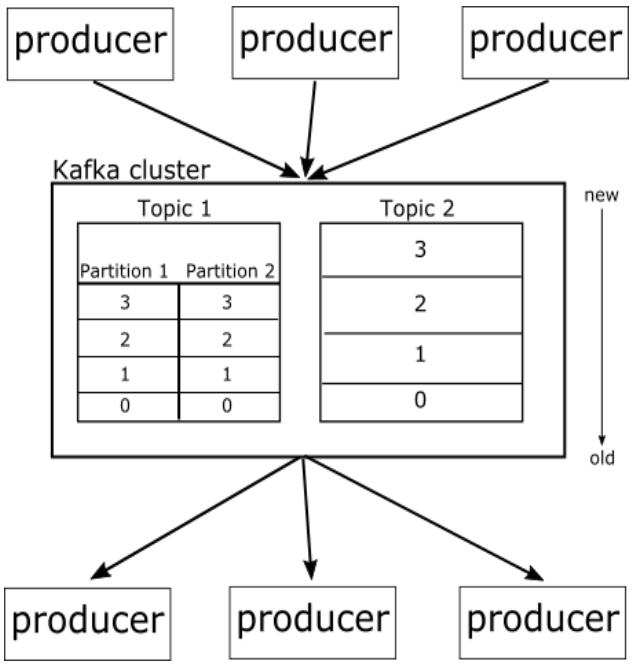
Konzument a producent

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

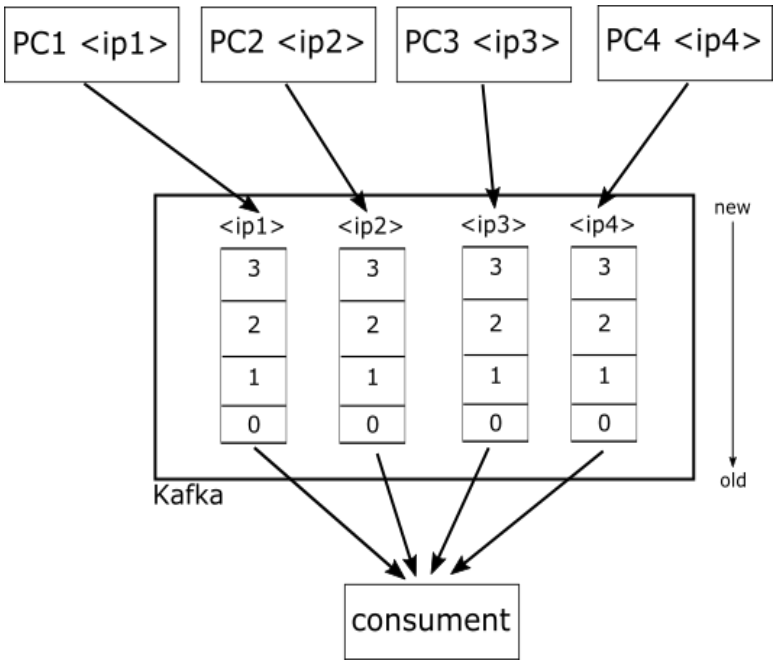
Garance

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 vytvořen jeden konzument a počet producentů je roven počtu počítačů. Viz obrázek 5.2



Obrázek 5.2: Struktura Kafky při iniciálním spuštění

5.2.3 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.

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");
```

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

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.

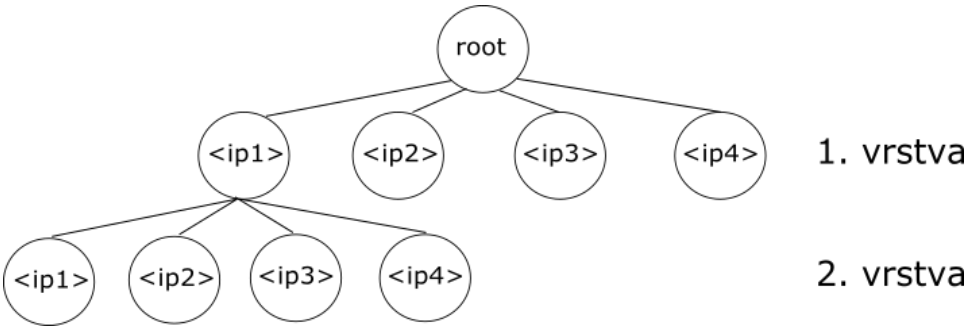
Obrázek 5.3 dále znázorňuje logickou strukturu DEM. První vrstva reprezentuje konzumenty a druhá vrstva producenty. Jinak řečeno "kdo komu posílá data". Podstatné je, uvědomit si, že strom zobrazuje pouze virtuální stav. Aplikace jako taková běží na daném stroji vždy jen jednou a strom pak reprezentuje 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 jasné. 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.

5.2.4 Esper

5.3 Události v systému

Kapitola popisující hlavní myšlenku povahy dat, která bude algoritmus vyhodnocovat. Také zde budou ukázky používaných dat.



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

5.3.1 Hrubozrnné

5.3.2 Jemnozrnné

5.4 Konfigurace

5.5 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.

Architektura aplikace

Je popsána v kapitole Maven. Hodilo by se sem ale ještě něco dopsat. Zeptat se jak je např řešen idependent deploy modulů u Paříze.

5.5.1 Iniciální spuštění

5.5.2 Přechody mezi stavy

Jedna z nejdůležitějších kapitol, která ukazuje co způsobí, že aplikace začne vyhodnocovat jemnozrnné události. Počínaje tím, že Esper vyhodnotí proud událostí a emituje ep-událost. Dále přes zpracování ep-

události, nastavení příslušných dat jednotlivým zk-uzlům v zk-stromě, či modifikaci zk-stromu. Až po ukončení zpracovávání jemnozrnný událostí a návrat k iniciálnímu stavu.

Bude zde také zmíněno dynamické nasazování nových ep pravidel. To by se dalo shrnout pod nadpis "ovládání clusteru z venčí"- řešeno přes nastavování dat jednotlivým uzlům v zookeeperu.

5.5.3 Esper pravidla

Mini kapitola, kterou bych věnoval použitým esper pravidlům.

5.6 Demo

Nějaké print screeny. Jednoduše ukázka běhu programu.

5.7 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?

6 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`