VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAVINTELIGENTNÍCHSYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENTOFINTELLIGENTSYSTEMS

ZBERNICA PRE PLATFORMU TESTOS

BUS FOR TESTOS FRAMEWORK

BAKALÁŘSKÁ PRÁCE

BACHELOR'S THESIS

AUTOR PRÁCE Tomáš Brada
AUTHOR

BRNO 2019

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav inteligentních systémů (UITS) Akademický rok 2018/2019

Zadání bakalářské práce

21394

Student: **Brada Tomáš**
Program: Informační technologie
Název: **Sběrnice pro platformu Testos**
Bus for Testos Framework
Kategorie: Analýza a testování softwaru
Zadání:

1. Nastudujte architekturu platformy Testos. Nastudujte rozhraní RestAPI a protokol OAuth.
2. Analyzujte požadavky pro komunikaci jednotlivých komponent platformy Testos. Uvažujte ztrátovost zpráv, dočasnou i permanentní nedosažitelnost cílových komponent. Navrhněte způsob komunikace jednotlivých komponent a architekturu komunikační komponenty pro platformu Testos.
3. Implementujte datovou sběrnici pro platformu Testos. Implementujte prototyp skladového úložiště pro dočasná data.
4. Ověřte funkcionalitu datové sběrnice na sadě jednoduchých umělých příkladů použití. Rozhraní komunikačního protokolu zdokumentujte s důrazem na použitelnost a udržitelnost.

Literatura:

1. Nástroj Swagger. [online] 2017. Dostupné z http://swagger.io/

Podrobné závazné pokyny pro vypracování práce viz http://www.fit.vutbr.cz/info/szz/

Vedoucí práce: **Smrčka Aleš, Ing., Ph.D.**
Vedoucí ústavu: Hanáček Petr, doc. Dr. Ing.
Datum zadání: 1. listopadu 2018
Datum odevzdání: 15. května 2019
Datum schválení: 1. listopadu 2018

Abstrakt

Testos je sada nástrojov určených na automatizované testovanie softvéru, ktoré by postupne chceli medzi sebou komunikovať. Oblasť výskumu nieje nová, už existuje niekoľko úspešných systémov na posielanie správ. V tejto práci boli zhodnotenie existujúce systémy na posielanie správ z pohľadu potrieb projektu Testos a bol navrhnutý nový systém na komunikáciu medzi komponentami projektu.

Abstract

Testos is a toolset for automatic software testing. Programs from this toolset might want to communcate with each other. The area of research of messaging systems is not new, in fact multiple successful messaging systems already exists. In this work, select few such systems were analyzed with requirements of Testos project in mind and a new system was proposed.

Klíčová slova

Sem budou zapsána jednotlivá klíčová slova v českém jazyce, oddělená čárkami.

Keywords

messaging system, message bus, broker, network, software testing.

Citace

BRADA Tomáš : Zbernica pre platformu Testos. Brno, 2019. Bakalárska práca. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Aleš Smrčka, Ph.D.

Zbernica pre platformu Testos

Prehlásenie

Prehlasujem, že som túto bakalársku prácu vypracoval samostatne pod vedením pána Ing. Aleša Smrčku, Ph.D. Uviedl som všetky literárne pramene a publikácie, z ktorých somčerpal.

……
Tomáš Brada
. mája 2018

Poďakovanie

Týmto by som sa chcel poďakovať vedúcemu práce Ing. Alešovi Smrčkovi, Ph.D. za jeho odbornú pomoc a nasmerovanie na správne riešenie.

Obsah

Obsah....1
1 Úvod....2
1.1 Message Queue....3
1.2 Publish-Subscribe....3
1.3 Požiadavka-odpoveď....4
1.4 Event Bus....4
1.5 Architektúra Serverov....6
2 Porovnanie súčasných riešení....8
2.1 Knižnica ZeroMQ....8
2.2 Platforma Apache Kafka....9
2.3 Message broker RabbitMQ....12
2.4 Zbernica správ D-Bus....14
2.5 Zhrnutie....16
3 Návrh Testos Bus....17
3.1 Publish-Subscribe (TBus)....17
3.2 Návrh smerovacej tabuľky....18
3.3 Tok Správ, Fronty....21
3.4 Analýza sieťovej architektúry....21
3.4.1 Úplné prepojenie uzlov v segmente....22
3.4.2 Service discovery server....23
3.5 Sieťový protokol....24
3.5.1 Komunikácia Klient-Klient....32
3.6 Aplikačné rozhranie....33
3.7 Príklady použitia....37
4 Záver....38
Literatúra....39
Obsah CD....40

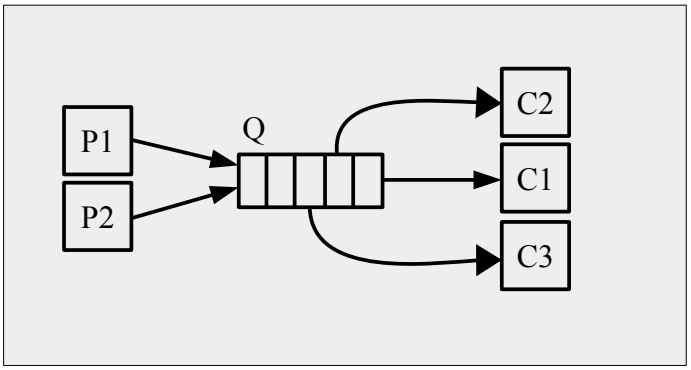
1 Úvod

Hlavný cieľ projektu Testos (TESTOS, 2019) je pripraviť sadu nástrojov na automatizované testovanie softvéru. Sada sa skladá z niekoľkých programov, ktoré už teraz pracujú nad výstupom iných programov z nástrojovej sady. Vznikla teda potreba zabezpečiť komunikáciu a výmenu dát medzi jednotlivými nástrojmi. Táto práca skúma možnosti a požiadavky na komunikačný systém pre projekt Testos.

V úvode práce sa pozriem na bežné komunikačné vzory, obecný návrh programov poskytujúcich služby a vybrané problémy s tým spojené. V druhej časti sa venujem zhodnoteniu existujúcich komunikačných systémov z pohľadu potrieb Testos-u. A v tretej časti sa pokúsim o návrh dátovej zbernice špeciálne pre projekt Testos.

1.1 Message Queue

Každá sprava poslaná v zmysle návrhového vzoru Message Queue sa doručí práve jednému odberateľovi. Často slúži ako rozšírenie thread pool-u na distribuovaný systém. *Thread Pool* (fond vlákien) je návrhový vzor, ktorý určuje ako rozdeliť prúd úloh pre spracovanie na vacerích procesoroch. Efektívne rozdelenie úloh dovoľuje vykonávať niekoľko úloh súčasne a tým zvýšiť priepustnosť (*Pecinovský, 2017)*. Dôležitá časť thread poolu je fronta úloh, ktorá najčastejšie býva konkurentná, veľkostne obmedzená a vlastná procesu. Ak je však fronta zdieľaná po sieti, je možné thread pool rozšíriť a zátaž distribuovať na množinu samostatných počítačov. Ak frontu spravuje samostatná služba, takáto služba sa nazýva message broker. Message Queue systémy, ktoré nevyužívajú správcu fronty sa označujú brokerless.



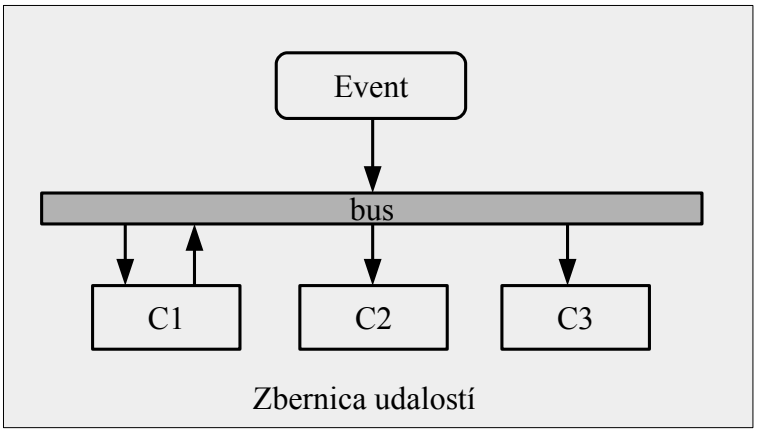
1.2 Publish-Subscribe

Na druhej strane, správy odoslané v zmysle vzoru Publish-Subscribe sa doručia všetkým konzumentom, ktorí sú na odber prihlásení. Publish-Subscribe systém, skrátene Pub-Sub, je tvorený producentmi, konzumentmi a témami. Kanál, tiež nazývaný téma (topic), je zdieľané médium, na ktorom počúva niekoľko, alebo aj žiadny konzument. Takýto systém bez perzistencie sa dá prirovnať k rádiovému vysielaniu. Systém sa dá rozšíriť o perzistenciu. Perzistencia zabezpečí doručenie správy, aj keď jej príjemca nebol v čase odoslania dostupný. Podobne ako pracovné fronty, perzistentný pub-sub systém musí udržiavať veľkosť úložiska pod kontrolou. Oproti message queue rozhodnúť, či je správa spracovaná, a teda ju je možné z úložiska odstrániť, nieje obecne jednoznačné. Problém je hlavne v tom, že odberateľov je viac, nový môžu pribudnúť a starí sa môžu natrvalo odpojiť. Jednotlivé implementácie riešia tento problém individuálne.

1.3 Požiadavka-odpoveď

Tretí vzor, ktorému sa budem venovať je dotaz-odpoveď (request–reply pattern). Tento vzor je veľmi jednoduchý, a tak populárny, že tvorí základ väčšiny klasických sieťových protokolov. Na rozdiel od predchádzajúcich dvoch vzorov, na jednu správu typu požiadavka sa očakáva práve jedna správa typu odpoveď. Na prenos požiadavky a odpovede je možné použiť point-to-point spojenie, napríklad IPC poskytované operačným systémom. Aplikácia buď vie, kde presne má požiadavky poslať, alebo to musí určitým spôsobom zistiť. Hľadanie adresáta schopného odpovedať na požiadavku sa nazýva Service discovery. Alternatívne službu nebude hľadať aplikácia, ale pošle správu sprostredkovateľovi, ktorý požadovanú službu nájde a požiadavku prepošle. Táto stredná služba plní úlohu smerovača, no nie na sieťovej, ale na aplikačnej vrstve.

1.4 Event Bus



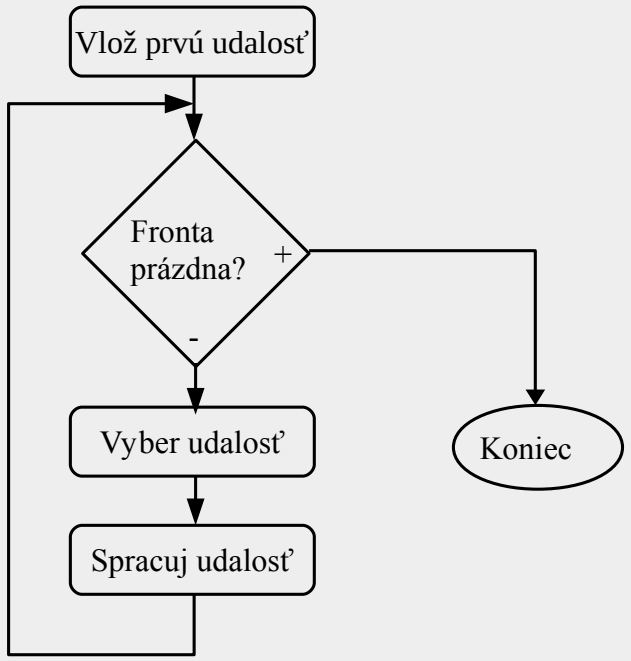
Bolo niekoľko krát navrhnuté, aby sa určitý problém rozložil na časti, ktoré sa potom spoja cez Event Bus (zbernica udalostí). Event bus je v podstate publish-subscribe. Jednotlivé moduly sa registrujú na udalosti (správy), ktoré vedia spracovať. Proces sa začne odoslaním prvej udalosti, ktorú začnú spracovávať niektoré moduly. Pri spracovaní jednej udalosti, môže modul poslať ďalšie udalosti, ktoré sa tiež budú spracovávať. Proces sa skončí, keď budú všetky udalosti spracované. Výsledkom procesu sú odoslané udalosti (správy), či také, ktoré vyhovujú filtru, alebo všetky.

Jedna z výhod takéhoto návrhu je oddelenie zodpovednosti a voľné previazanie jednotlivých modulov. Zodpovednosť každého modulu je iba správne spracovanie udalosti. Takýto návrh umožňuje implementovať moduly bez znalosti implementačných detailov ostatných modulov (Hohpe 2012). Ďalšia výhoda je možnosť pomerne ľahko škálovať výpočet na viac vlákien, prípadne na viac počítačov.

Problémy môžu nastať, ak nemáme dostatočnú kontrolu nad prítomnosťou modulov. Ak potrebný modul nieje na zbernici pri spustení procesu, výsledky procesu nebudú kompletné. Tiež, keď počas behu procesu dôležitý modul odpadne zo zbernice, výsledky tiež nebudú kompletné. Takýto problém nemá ako nastať, keď si moduly spúšťa zadávateľ úlohy v jednom procese. Ale ak je systém rozdelený na viacero procesov spojených externou pub-sub zbernicou, môže veľmi ľahko nastať, že modul, ktorý je potrebný na správne spracovanie úlohy, nieje dostupný, alebo jednoducho nieje nastavený, aby sa pripojil.

Problém je o to horší, že je tichý. (silent failure) Udalosť, ktorú mal nedostupný modul spracovať sa jednoducho odignoruje, bežiace udalosti dobehnú a proces sa prehlási za dokončený, aj keď jeho výsledky sú vlastne nekompletné.

Tiež nastáva problém ako detegovať dokončenie procesu. Predpokladám, že bežná aplikácia bude čakať na dokončenie procesu, už len preto aby mohla kompletné výsledky predať operátorovi, alebo nadradenému programu. V prípade jednovláknového spracovávania s frontou v jednom procese, je to jednoduché. Stačí počkať, dokým sa fronta nevyprázdni. Pri viacvláknovom spracovávaní, môžeme zaviesť počítadlo nespracovaných správ, a čakať kým toto počítadlo klesne na nulu. Ale ak používame zdielanú zbernicu udalostí, je to zložité. RabbitMQ, Kafka ani DBus (viď kapitolu 2) neposkytuje notifikáciu, že sú všetky správy spracované. ZeroMQ je flexibilný, takže by sa to asi dalo naprogramovať. Problém by sa dal obísť, tak, že po odoslaní prvej správy by program čakal na výsledky iba určitý čas. Je jasné, že takéto pseudo-riešenie by bolo nespoľahlivé a v Testose je spoľahlivosť a správnosť výsledkov dôležitá.



Vývojový diagram spracovania udalostí na jednom vlákne.

Pre aplikácie, ktoré nechcú čakať na kompletné výsledky je naopak event bus vhodný. Napríklad netransakčná akcia na udalosť vygenerovanú iným modulom.

Ak by viac klientov chcelo využiť pub-sub zbernicu s prítomnými modulmi, je potrebné oddeliť správy jedného klienta od druhého. Toto by sa dalo jednoducho zabezpečiť hodnotou v každej správe, ktorá identifikuje klienta. No problém nastane ak jeden klient potrebuje na spracovanie svojho zadania moduly 1, 2 a 3, a druhý klient moduly 3, 4 a 5. Jedno riešenie je mať na zdielanej zbernici spustené všetky moduly. Klient tak dostane aj výsledky, ktoré nepotrebuje, potenciálne plytvajúc výpočtovými prostriedkami.

Druhé riešenie je pre každého klienta vytvoriť vlastnú (pod-)zbernicu a pripojiť module k ňej, podľa potrieb klienta. Keďže modly inštanciuje klient, môže aj dostať chybovú správu ak niektorý z nich zlyhá alebo nieje dostupný. Oddelenie klientov je tiež ošetrené tým, že komunikácia beží na pod-zbernici. No celí koncept pod-zberníc sa zdá byť zbytočne komplikovaný a prieči sa prvotnou myšlienkou mať zdielanú zbernicu.

Event Bus má svoje výhody. Ale pri pokuse o distribuovaní Event Bus systému nastávajú problémy, ktoré sa ťažko riešia. Pri testovaní softvéru sú tiché chyby a neúplnosť výsledkov neprípustné. Ak je cieľom dostať na správu odpoveď, je vhodné použiť radšej vzor požiadavka-odpoveď. Ak potrebujeme odložiť správy na neskoršie spracovanie, pripadá do úvahy fronta správ (mq). Ak Event Bus nekomplikujeme distribúciou po sieti, vstavaný Event Bus je spoľahlivý a umožňuje lepšie oddeliť implementáciu modulov aplikácie.

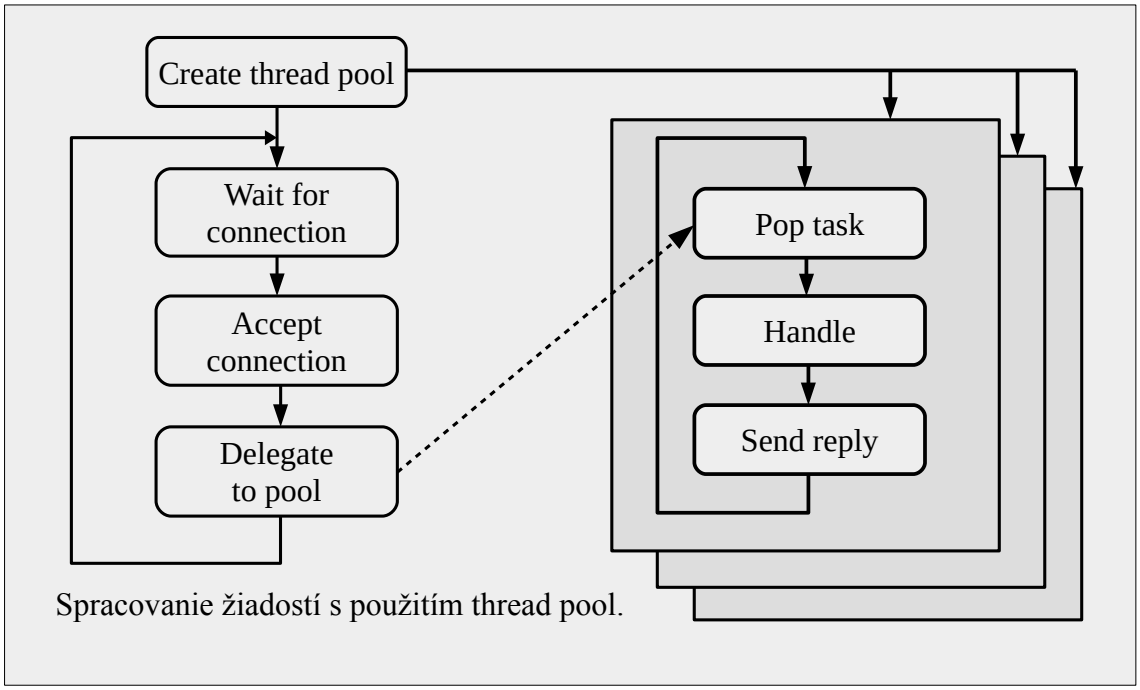
1.5 Architektúra Serverov

Programy, ktoré majú obsluhovať požiadavky, či už z TCP/IP soketu alebo z komunikačnej knižnice, sa dajú navrhnúť rôznymi spôsobmi. V tejto kapitole sa venujem analýze vlastností niekoľkých štandardných návrhov, vrátane ich relevantnosti k *Testos Bus*. Obdobne analyzujem aj ich správanie v medzných situáciách. Iteratívny a Konkurentný server sú dva základné návrhy (Matoušek 2014), a okrem nich sú aj iné.

Iteratívny server je najjednoduchšia implementácia poskytovateľa služieb. Program sa skladá z jedného vlákna, ktoré v hlavnej slučke postupne obsluhuje žiadosti. Iteratívny server obsluhuje v jednu chvíľu len jednu žiadosť, takže nemôže využiť multiprocesorové schopnosti počítača ani prekladať diskové, sieťové alebo výpočtové operácie. Na druhej strane takýto program potrebuje relatívne málo pamäti, iba toľko, koľko je potrebné na obslúženie jednej žiadosti. Server nikdy neprijme viac požiadaviek ako dokáže spracovať, ostatné žiadosti čakajú, dokiaľ neprídu na rad. V niektorých prípadoch je tento návrh vyhovujúci, napríklad keď vybavenie žiadosti zahŕňa interakciu s nekonkurentným komponentom ako je hardvér alebo transakčná báza dát. Testos BUS by mal tento návrh podporovať, už len preto, že iteratívny poskytovateľ služby je intuitívny a jednoduchý na implementáciu.

Kokurentný server môže spracovávať viac žiadostí súčasne, a to tak, že pre každú žiadosť vytvorí nové vlákno (či proces). Výhodou je, že obsluha žiadosti beží na samostatnom vlákne a nemá dopad na prijímanie a spracovanie ďalších žiadostí. Server vytvorený týmto spôsobom môže prekladať operácie a využívať viac procesorov. Intuitívnosť zostáva zachovaná, pretože obsluha jednej žiadosti zostáva sekvenčná. Na druhej strane je niekoľko nevýhod. Napríklad, ak obsluha žiadosti zahŕňa prístup k zdieľanému stavu, musí byť tento prístup správne ošetrený, aby nedošlo k synchronizačným chybám. Nevhodná synchronizácia prístupov k zdieľanému stavu môže v konečnom dôsledku zdegradovať výkon natoľko, že je horší ako výkon jednoduchšieho iteratívneho servera. S vytváraním a rušením vlákien je spojená nezanedbateľná réžia. Každé vlákno potrebuje k svojej činnosti určitú pamäť a systémové prostriedky. Aplikácia by teda mala mať implementovaný vhodný mechanizmus na obmedzenie počtu súbežných vlákien. V opačnom prípade hrozí, že sa pri záťaži tieto fyzické prostriedky vyčerpajú. Vzhľadom na uvedené nedostatky, by Testos BUS nemal tento návrh podporovať.

Návrh konkurentného serveru je ale možné optimalizovať, použitím návrhového vzoru **Thread Pool** (fond vlákien), a tým ho zbaviť oboch spomenutých nevýhod. Stručne povedané, server si pri štarte vytvorí do zásoby niekoľko vlákien, čím amortizuje čas potrebný na ich vytvorenie a zrušenie. Počas behu sú požiadavky obsluhované práve týmito vláknami z rezervy vlákien. Na distribúciu požiadaviek medzi vláknami sa môže použiť konkurentná blokujúca fronta. Keďže vlákna sa vytvárajú iba pri štarte, a ďalej sa už nevytvárajú, vyčerpanie prostriedkov nehrozí. Testos BUS tento návrh podporuje cez triedu CQueue.



Spracovanie žiadostí s použitím thread pool.

Ak pri spracovaní požiadavky konkurentný server potrebuje komunikovať s iným modulom, celé vlákno nečinne čaká na odpoveď druhého modulu. Ak takáto situácia nastáva bežne, je vhodné navrhnúť server tak, aby pracoval asynchrónne.

Asynchrónny server nečaká na odpovede od iných modulov, ale využije čas na obsluhu ďalších požiadaviek. Tento návrh ale vyžaduje zásadnú zmenu v štruktúre programu. Namiesto sekvenčných rutín, musí byť asynchrónny server rozdelený na rutiny obsluhujúce udalosti. Existujú však programovacie jazyky, ktoré boli vytvorené špeciálne na písanie takýchto reakčných programov, napríklad NodeJS. Veľkou výhodou asynchrónneho servera je skutočnosť, že dokáže spracovať oveľa viac žiadostí ako konkurentný server. Ich počet je obmedzený len veľkosťou pamäte potrebnej na spracovanie žiadosti prípadne inými zdrojmi systému.

Záverom zdôrazním, že samotný Testos BUS server a celá jeho klientská knižnica je vo svojej podstate programom reakčným, nie sekvenčným.

2 Porovnanie súčasných riešení

Pred návrhom nového komunikačného systému pre Testos je vhodné analyzovať existujúce riešenia, porovnať ich vlastnosti a zhodnotiť ich z hľadiska potrieb v Testos-e. Pri každom z vybraných systémov najprv stručne uvádzam opis činnosti a zaujímavé vlastnosti. Potom skúmam podporu komunikačného vzoru dotaz-odpoveď, schopnosť systému smerovať správy podľa zhody s filtrovacim vzorom a náročnosť nasadenia. Keďže Testos je

Pri dávkovom spracovávaní úloh sa stáva, že bude v jeden moment vygenerovaných viac správ ako je možné spracovať. Narozdiel od interaktívneho systému, pri dávkovom spracovaní uprednostňujeme eventuálne spracovanie všetkých úloh nad okamžitou dostupnosťou , aj keď to bude trvať dlhšie. Z tohto dôvodu je potrebné tiež zhodnotiť správanie systému v situácií keď je preplnený správami. Dôležité kritérium je aj spoľahlivosť systému. Keďže Testos je platforma na testovanie softvéru, je dôležité aby boli správy doručené na správne miesto a to bez strát.

2.1 Knižnica ZeroMQ

ZeroMQ (Hintjens 2013) je nadstavba nad BSD soketami zameraná na nízku réžiu. Na rozdiel od ostatných riešení, nepoužíva brokera na *distribúciu* správ. Základný objekt reprezentujúci akékoľvek spojenie je ZeroMQ soket, ktorých je niekoľko typov. ZMQ knižnica je prenositeľná a zakrýva rozdieli medzi platformami. Okrem soketov zapúzdruje aj platformovo závislé poll/epoll/select API pod vlastné poll rozhranie. Ďalej knižnica zaobaľuje sieťové adresy do prehľadného textového formátu. Aplikácie nieje oslobodená od sieťových problémov, stále musí vedieť adresu, kam sa chce pripojiť.Ak je adresát nedostupný, ZMQ sa bude pokúšať pripojiť do nekonečna. Tento nedostatok sa dá vyriešiť pomocou poll rozhrania, ktoré umožňuje nastaviť maximálny čas operácie.

Keďže knižnica je tak blízka rozhraniu operačného systému, je veľmi nenáročná na zdroje. Priepustnosť a latencia nieje ďaleko od čistých BSD soketov.

Podpora vzoru žiadosť-odpoveď

Návrhový vzor požiadavka-odpoveď je podporovaný pomocou zmq soketov typu REQ a REP. Server vlastní jeden REP soket, ktorý musí zviazať s určitou sieťovou adresou pred tým než môže začať obsluhovať požiadavky. Nad zmq soketom nieje potrebné volať listen ani accept, čo mierne zjednodušuje implementáciu servera. Potom už server len číta požiadavky z soketu a odpovedá na ne. ZMQ vynucuje dodržiavanie komunikačného vzoru: na každú požiadavku musí server vrátiť práve jednu odpoveď. Na druhej strane, klient si vytvorí REQ soket, nadviaže spojenie so serverom, pošle požiadavku a prečíta odpoveď, keď príde.

Filtrovanie podľa vzoru

Vzor Publish-Subscribe je tiež podporovaný knižnicou ZeroMQ a to pomocou soketov PUB a SUB. PUB soket slúži na vysielanie signálov. Na tému správy nieje v ZeroMQ vyhradené žiadne dodatočné miesto. Správy sa filtrujú pre odberateľov podľa prefixu tela správy. Teda ak aplikácia chce vysielať správy pod viacerými témami, musí hodnotu témy buď umiestniť na začiatok správy, alebo použiť oddelené sokety. Oddelenie témy od obsahu správy musí riešiť aplikácia. Téma však môže byť prázdny reťazec. Príjemcovia správ si musia vytvoriť SUB soket, pripojiť sa na všetkých producentov od ktorých chcú odoberať a požiadať soket o odber určitej témy. Potom je možné správy jednoducho čítať zo soketu. ZeroMQ pri svojom minimalistickom designe neskrýva sieťové detaily. Odberatelia musia vedeť sieťovú adresu všetkých producentov, od ktorých chcú odberať správy.

Spoľahlivosť

ZMQ je spoľahlivé v tom zmysle, že požiadavku aj odpoveď doručí aj pri dočasnom výpadku siete, ale nedokáže dobre spracovať trvalú nedostupnosť. Keďže pri komunikácií sa nepoužíva broker, správy sa pri páde aplikácia stratia. Prostriedky sú však dostupné na implementáciu brokera.

Riadenie toku

Ak niektorý odberateľ nestíha spracovávať správy dostatočne rýchlo, ďalšie správy sa zahodia, dokým odberateľ správy spracuje. Ostatní odberatelia nie sú dotknutí takýmto stavom. Toto správanie je závislé od typu soketu: niektoré sokety správy zahadzujú, iné blokujú odosielateľa. Blokujúci soket je napríklad PUSH, ktorý je vhodný na implementáciu pracovnej fronty.

Nasadenie a podpora

ZeroMQ sa inštaluje a používa ako aplikačná knižnica. Knižnica pre jazyk C je dostupná v repozitároch všetkých hlavných distribúcií Linuxu a tiež dostupná pre Windows. Kompatibilné rozhranie je tiež dostupné pre viac ako 40 programovacích jazykov vrátane Python, Java, C++. Rozhranie knižnice je predovšetkým blokujúce, ale podporuje polling.

2.2 Platforma Apache Kafka

Je to *open-source* softvérová platforma na spracovanie dátových tokov (*Stream processing*) napísaná v jazyku Scala a Java. Pre túto platformu je charakteristická najmä vysoká priepustnosť a nízka latencia.

Všetky správy, ktoré producenti posielajú tečú konzumentom cez broker. Všetky správy sa ukladajú do žurnálu (Narkhede, 2017). Kafka môže byť prevádzkovaná na klastri len jedného servera, alebo môže byť distribuovaná fyzicky na viaceré počítače. Toto distribuované riešenie architektúry umožňuje dodávať aj masívne toky správ spôsobom, ktorý je odolný voči výpadkom.

Základná filozofia ukladania správ je nasledovná: Kafka ukladá správy od ľubovoľného počtu procesov - producentov do oblastí, ktoré sú ďalej rozčlenené na jednotlivé fronty tzv. témy - *topic*. Správy sú striktne usporiadané, indexované a uložené aj s časovou pečiatkou a tie sú následne prístupné pre ďalšie procesy - konzumentov.

Žurnál je vlastne fronta správ usporiadaná v poradí, tak ako postupne tieto správy prichádzali. Za tým účelom je každej správe pridelené poradové číslo – *offset*. Konzumenti môžu čítať správy buď v poradí, alebo môžu preskakovať vpred a vzad na ľubovoľnú správu.

Každú tému je možné rozdeliť na viacero partícií tak, aby každá partícia mala svoj vlastný žurnál. Každá odoslaná správa sa uloží práve do jedného žurnálu. Odosielateľ buď určí do ktorého žurnálu sa správa doručí, alebo môže použiť jedno z predvolených pravidiel. Každý žurnál môže byť uložený na inom serveri, čím je možné systém horizontálne škálovať. Odolnosť voči výpadkom je možné zvýšiť replikovaním žurnálov na ďalšie serveri.

Implicitne dostáva každý konzument všetky správy zo svojej príslušnej témy, ale v prípade, že sa niekoľkí konzumenti združia do skupiny, správu dostane len jeden z nich. Rozdeľovanie správ v skupine sa realizuje už na úrovni partícií, takže na serveri musí byť aspoň toľko partícií, aby mohol systém priradiť každému konzumentovi aspoň jednu. Pomocou skupín a partícií sa dá jednoducho zabezpečiť rovnomerné rozloženie toku správ.

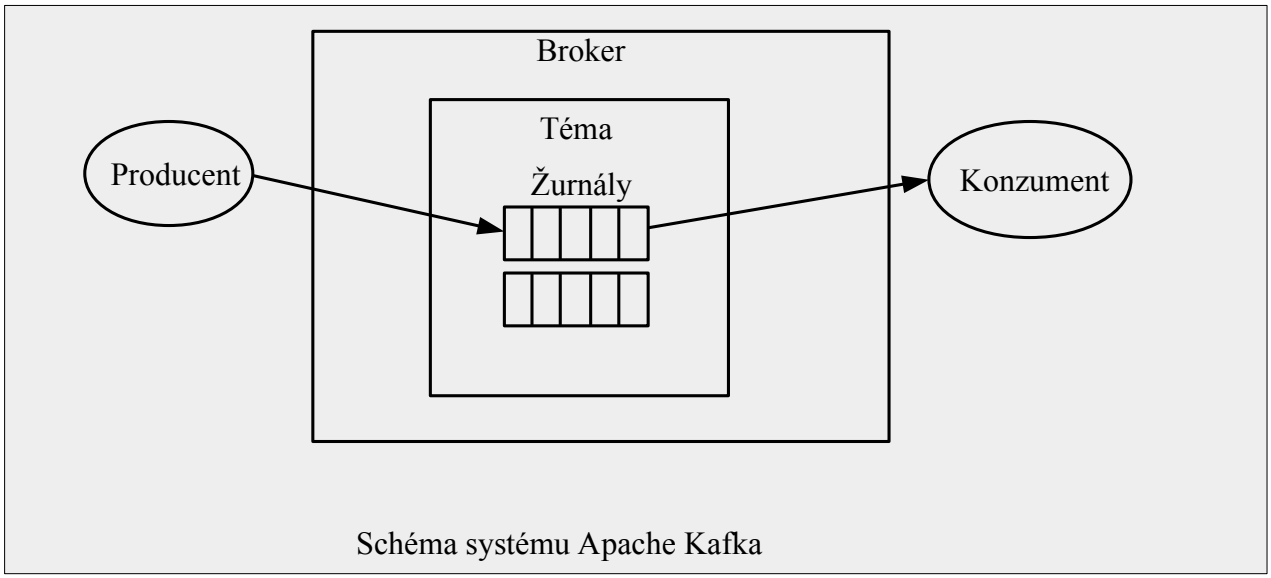


Schéma systému Apache Kafka

Spoľahlivosť

Na Kafku sa dá pozerať aj ako na distribuovanú databázu. Všetky správy prijaté na serveri sa ukladajú do perzistentného žurnálu (log-u). Na strane producenta je možné voliteľne čakať na potvrdenie o uložení správy. Ak je voľba zapnutá, producent čaká, dokým sa správa bezpečne uloží na server a o prípadnej chybe bude hneď informovaný. Nezávisle od tohto nastavenia sa správa, ktorá dorazí na server vždy uloží. Všetky témy, ktoré aplikácia vytvorí sú trvácne a prežijú aj reštart servera.

Pozícia v žurnále, z ktorej konzument číta správy, je voliteľne perzistentná. Ak je perzistencia offsetu zapnutá, offset sa periodicky ukladá na server. Pri reštarte, konzument pokračuje z pozície ktorá bola naposledy uložená. Aby toto fungovalo, musí mať konzument nastavený identifikátor group_id. Ak ho nemá nastavený, po reštarte dostane len nové správy a správy, ktoré boli poslané v čase nedostupnosti už nedostane.

Broker správy zo žurnálu nevymazáva hneď ako sa spracujú, pretože nevie, či sa ešte nepripojí ďalší konzument, ktorý by chcel staré správy prečítať. Keďže úložisko správ je obmedzené, Kafka postupne vymazáva najstaršie správy. Vymazávanie starých správ nie je zrovna ideálne pre potreby dávkového spracovania. Pretože, ak je správ veľa, alebo spracovanie trvá dlho, môže sa stať že systém vymaže aj ešte nespracované správy.

Zaujímavá vlastnosť je kompakcia žurnálov: Ak správy majú formát kľúč-hodnota, je možné broker nastaviť, tak aby pri vymazávaní správ ponechal vždy aspoň poslednú správu s konkrétnym kľúčom. Touto vlastnosťou sa Kafka veľmi približuje systému riadenia bázy dát (ref: log compaction).

Podpora vzoru žiadosť-odpoveď

Kafka nie je stavaná na použitie v zmysle dotaz-odpoveď. Komunikácia dotaz-odpoveď sa síce dá emulovať, ale ukladanie všetkých správ do žurnálov komunikáciu zbytočne spomaľuje.

Filtrovanie podľa vzoru

Vzor pubish-subscribe je veľmi dobre podporovaný. Rozosielanie správ funguje prakticky na úrovni skupín konzumentov. Každá správa sa pošle jednému konzumentovi z každej skupiny. Ak teda chceme aby každý konzument dostal kópiu správy, nemôžu byť združení v skupinách.

Filtrovanie správ podľa neúplnej zhody na tému, alebo podľa filtrovacieho vzoru, nie je podporované. Správa sa vždy posiela iba do konkrétnej témy. Konzument ale môže požiadať o odber tém podľa vzoru. Aplikačné rozhranie vyhľadá všetky témy vyhovujúce vzoru a tie začne odberať. Podpora tejto vlastnosti klientskými knižnicami je však slabá.

Riadenie toku

Flow control ako taký v Kafkovi nie je. Broker sa snaží správy uložiť čo najrýchlejšie. Ak sú producenti rýchlejší ako konzumenti, správy sa začnú hromadiť vo frontách a keď sa úložisko správ zaplní, začnú sa najstaršie správy vymazávať. Na druhej strane, broker ukladá na klientské spojenia určité kvóty, aby zaistil férové rozdelenie serverových prostriedkov (cpu&net bw).

Pre dávkové spracovanie úloh v Testose by bolo lepšie, aby v prípade preplnenia úložiska boli producenti pozastavení.

Nasadenie a podpora

Apache Kafka broker nie je dostupný v oficiálnych repozitároch Debianu, Ubuntu, Fedory ani openSuse. Je ale dostupný ako Java aplikácia, a teda je kompatibilný so systémami, kde beží Java. Klientské knižnice sa inštalujú samostatne, a okrem jednej, sú pod správou tretích strán. Vývojový tím Kafka spravuje iba klientsku knižnicu pre Javu. Tiež sú dostupné knižnice (väzby) pre Python, jazyk C/C++, JavaScript, Go, PHP, Ruby, Rust, a ďalšie.

Réžia

Proces brokera používa aj pri nečinnosti nemalé množstvo pamäte (800MiB), čo je viac ako ostatné messaging riešenia spomenuté v tejto práci. Ukladanie všetkých správ na disk môže pre niektoré aplikácie predstavovať zbytočnú réžiu.

2.3 Message broker RabbitMQ

RabbitMQ je pokročilý message broker založený na protokole AMQP (Aiyagari, 2008). Všetky správy, ktoré producenti posielajú tečú konzumentom cez broker. Úlohou brokera je správy smerovať konzumentom a prípadne ukladať, podľa konfigurácie. Model amqp definuje v brokeri tri entintné množiny (Klishin, 2019):

1. Queues – fronty, kde správy čakajú na spracovanie konzumentmi
2. Exchange – prijíma správy od producentov a umiestňuje ich do front
3. Bindings – prepojenia (väzby) medzi exchange a frontami.

RabbitMQ je flexibilný v tom zmysle, že tieto entity plne spravuje aplikácia. Typ a parametre entity exchange, spolu s parametrami väzieb určujú ako sa prijaté správy distribuujú do previazaných front. V Brokeri vždy existuje predvolená exchange, s ktorou sú implicitne zviazané všetky fronty podľa ich mena.

RabbitMQ broker je kompatibilný aj s MQTT klientami. MQTT protokol ponúka nižšiu réžiu oproti AMQP. Menšie hlavičky správ (iba 2 B) a jednoduchšie klientské knižnice, no nepodporuje všetky vlastnosti čo ponúka RabbitMQ broker.

Spoľahlivosť

V predvolenej konfigurácií fronty, ani správy neprežijú prípadný reštart brokera. Dokonca fronty sa automaticky odstránia po odpojení posledného konzumenta. Ale protokol poskytuje niekoľko nastavení na zvýšenie spoľahlivosti doručenia správ. Na strane konzumenta sú to trvácne fronty a potvrdenia o doručení. Nastavenie trvácnej fronty umožňuje konzumentovi sa opätovne pripojiť na frontu a ďalej pokračovať v spracovávaní správ. Zapnutie potvrdení o doručení spôsobí, že broker nepotvrdené správy znovu zaradí do fronty. Na strane odosielateľa sú to perzistentné správy a potvrdenia o zaradení správy do fronty. Perzistentné správy prežijú aj pád brokera pretože sa ukladajú na disk.

Každé z týchto opatrení ma určitý negatívny dopad na priepustnosť a vývojár má možnosť zvoliť vhodný kompromis medzi trvácnosťou a priepustnosťou podľa potrieb aplikácie.

Podpora vzoru žiadosť-odpoveď

RabbitMQ podporuje komunikačný vzor dotaz-odpoveď prostredníctvom dočasnej fronty na príjem odpovedí. Nepovinná perzistencia značne znižuje réžiu volania vzdialených služieb. Toto je výhda voči Apache kafkovi, ktorý na to nie je určený a perzistenciu vyžaduje.

Pred odoslaním správy so žiadosťou, klient najprv vytvorí dočasnú frontu na príjem odpovede a potom žiadosť odošle s odkazom na túto dočasnú frontu. Prípadne klient môže použiť tú istú dočasnú frontu na príjem odpovedí z viacerých dotazov,ktoré rozlýši pomocou identifikátora v hlavičke.
Server, aby mohol prijímať žiadosti, na začiatku deklaruje frontu s určitým názvom.

Load balancing žiadostí je implicitne poskytovaný brokerom, pretože opakovaná deklarácia fronty s rovnakým názvom vráti frontu, ktorá už bola deklarovaná predchádzajúcim poskytovateľom tej istej služby.

Preposielanie odpovedi od jednej služby druhej službe nieje zrovna efektívne. Lepšie je viacero služieb zreťaziť, spôsobom že odpoveď od jednej služby broker zašle priamo na vstup druhej služby. Dočasná fronta sa dá obdobne využiť aj pri zreťazenom volaní.

Filtrovanie podľa vzoru

RabbitMQ podporuje výber adresátov podľa čiastočnej zhody smerovacieho kľúča cez takzvanú topic exchange. Smerovací kľúč správ pre topic exchange musí mať podobu zoznamu slov, ktoré sú oddelené bodkou. Pri zviazaní novej fronty s topic exchange musí byť zadaný filtrovací vzor. Vzor je v podobnom formáte ako smerovací kľúč, ale môže obsahovať aj zástupné znaky, a to nielen na konci. Topic exchange ukladá prijaté správy do všetkých front, kde kľúč správy vyhovuje filtrovaciemu vzoru.

Ak Topic exchange skombinujeme s trvácnou frontou, eliminujeme chyby tzv. odpadnutého odberateľa (bližšie vysvetnené v časti event bus)

Direct exchange je jednoduchšia alternatíva k topic exhchange. Takisto podpodporuje publish-subscribe ale nepoužíva zástupné znaky. Ak aplikácia potrebuje triediť správy ku konzumentom podľa viacerých atribútov, môže použiť Headers Exchange, ktorá to umožňuje.

Riadenie toku

Ak do brokeru prúdi viac správ, ako sa stíha spracovať, môžu sa fronty aj preplniť. RabbitMQ ponúka niekoľko spôsobov, ako ošetriť túto situáciu. Prvá ochrana proti preplneniu je zablokovanie producentov správ. Ak brokerovi začne dochádzať pamäť alebo miesto na disku, či jednoducho nestíha spracovávať také množstvo správ, tak dočasne zablokuje producentov. Správy sa v tomto prípade nestrácajú, a budú prijaté neskôr, keď sa situácia zlepší. Druhý mechanizmus je realizovaný nastavením maximálnej dĺžky fronty. Popri tom je možné nastaviť, či sa pri pokuse o vloženie do plnej fronty má z fronty vymazať najstaršia správa, alebo či sa má nová správa odmietnuť. Ak je správa odmietnutá a producent používa potvrdenia o prijatí, dostane v druhom prípade namiesto potvrdenia prijatia chybovú odpoveď. Blokovanie producenta pri preplnenej fronte nie je podporované (cite). Toto nie je zrovna ideálne pre implementáciu spoľahlivých dávkových procesov, ktoré by preferovali blokovanie nad stratou správ. Negatívne potvrdenia o uložení správy aspoň umožňujú zopakovať odoslanie správy pri takejto chybe.

Nasadenie a podpora

RabbitMQ server je dostupný ako softvérový balík vo všetkých hlavných distribúciach Linuxu, tiež podporuje Windows ako i ďalšie platformy. Klientské knižnice sa inštalujú samostatne a okrem dvoch sú pod správou tretích strán. Tím RabbitMQ spravuje klientov pre Javu a dotNet. Tiež sú dostupné knižnice (väzby) pre Python, jazyk C/C++, JavaScript, Go, PHP, Ruby, Rust, a ďalšie.
Klient pre python (pika) podporuje spätné volania po prijatí správy a synchrónnu, ako aj asynchrónnu operáciu. Klient pre jazyk C je navrhnuyý tak, že sa dá použiť ako aj v reakčných (event loop) tak aj v sekvenčných programoch.

2.4 Zbernica správ D-Bus

D-Bus (Vermeulen, 2013) je komunikačné médium určené na komunikáciu medzi procesmi na jednom stroji. Patrí pod projekt freedesktop.org a používa sa hlavne ako middleware pre linuxové desktopové prostredia. Napriek tomu, podporuje aj komunikáciu cez sieť. Komunikácia prebieha zvyčajne iba medzi klientom a D-Bus démonom. Dva základné druhy správ sú signály - správy bez odpovede, a žiadosti - správy s odpoveďou. Správy sú štruktúrované a na opis tejto štruktúry D-Bus definuje pokročilý systém dátových typov (Carlsson, 2018).

Ďalšia zaujímavá vlastnosť je aktivácia služieb podľa potreby. D-Bus démon dokáže, pri správnej konfigurácií, spustiť službu vyžiadanú inou aplikáciou. Ak je démon správne nakonfigurovaný a klient sa pokúsi poslať žiadosť službe, ktorá nebež, túto službu démon automaticky a spustí tak, aby transparentne obslúžila žiadosť.

Podpora vzoru žiadosť-odpoveď

D-Bus plne podporuje request-reply (rpc) komunikačný vzor a je stavaná na toto použitie. Aplikácie na D-Buse zverejňujú objekty s metódami. Zverejnené objekty ako aj metódy majú aplikáciou určený názov. Ostatné aplikácie môžu potom tieto metódy vzdialene invokovať a prijať naspäť výsledok tejto invokácie.

Spoľahlivosť

Správy (signály a žiadosti) sa v D-Buse doručia iba tým klientom, ktorí sú v tom čase pripojení na zbernicu a vyjadrili o správy záujem. D-Bus nepodporuje perzistenciu.

D-Bus server (démon) garantuje doručenie signálov a RPC, ale iba kým všetko funguje. Pri neočakávanom alebo nekorektnom ukončení servera, producenta alebo konzumenta sa správy môžu stratiť. Toto nemusí byť problém ani pre spoľahlivé dávkové spracovanie úloh, pretože na jednom systéme vieme detekovať nekorektné ukončenie (crash) programov a podľa toho označiť výsledok ako neplatný.

Filtrovanie podľa vzoru

D-Bus filtruje správy podľa takzvaných Match Rules, ktoré špecifikuje príjemca. Filtrovať je možné podľa viacerých preddefinovaných položiek z hlavičky správy. Zaujímavá je položka path, ktorú je možné filtrovať nielen podľa úplnej zhody, ale aj podľa zhody predpony (cite dbus spec).

Riadenie toku

D-Bus nie je stavaný na veľký prúd správ a v špecifikácií nie je jasne definované, čo sa má stať v prípade, ak sa správy produkujú rýchlejšie ako sa stíhajú konzumovať. Súčasná implementácia D-Bus démona nedoručené správy ukladá do pamäti. Po dosiahnutí limitu tejto pamäti sú nové správy démonom odmietané.

Nasadenie a podpora

D-bus démon je implicitne nainštalovaný na každom linuxovom stroji s grafickým desktopovým prostredím, knižnice (bindingy), ktoré sú dostupné pre väčšinu programovacích jazykov. Aj keď D-Bus sa na Windows-e nepoužíva, D-Bus démon podporuje aj túto platformu.

2.5 Zhrnutie

Na záver zhrniem, že každý z týchto messaging systémov má svoje výhody. Niektoré ich vlastnosti uvádzam do tabuľky:

	Perzistencia správ a konfigurácie	Peer to peer	dotaz-odpoveď	Filtrovanie podľa vzoru	Flow Control
Apache Kafka	áno/vždy	nie	nie	čiastočne	nie
Zero MQ	nie	áno	áno	čiastočne	áno
Rabbit MQ	áno/voliteľne	nie	áno	áno	áno
D-Bus	nie	nie	áno	áno	čiastočne

Ďalej som zmeral výkonnosť týchto systémov v troch kategóriach a uvádzam aj spotrebu pamäte brokera v stave nečinnosti. Všetky testy boli spúšťané offline na lokálnom stroji, takže latencia sieťového spojenia nemá na výsledky vplyv. Odosielané správy boli krátke (8 bajtov). Položka latencia vyjadruje čas od odoslania krátkej správy jedným procesom do prijatia tej istej správy druhým. Výsledky meraní sú zhrnuté v tabuľke nižšie.

Položka dotaz-odpoveď vyjadruje koľko cyklov požiadavka-odpoveď prebehlo za sekundu. Cyklus požiadavka-odpoveď sa skladá z odoslania správy jedným procesom, prijatie druhým procesom, ktorý odošle druhú správu naspäť a následné prijatie tejto druhej správy prvým procesom.

Položka priepustnosť vyjadruje množstvo správ, ktoré systém dokáže preniesť za jednotku času.

	Latencia	dotaz-odpoveď	Priepustnosť	Broker RAM
Apache Kafka	7.0 ms	157,9 ž/s	13 035 ž/s	750 MiB
Zero MQ	0.4 ms	19 941,7 ž/s	102 074 ž/s	n/a
Rabbit MQ	2.1 ms	949,5 ž/s	3 081 ž/s	80 MiB

3 Návrh Testos Bus

Pre nedostatky produktov opísaných v predchádzajúcej kapitole som sa začal pracovať na návrhu Testos Bus.

3.1 Publish-Subscribe (TBus)

Správy v Testos Bus-e sú doručované konzumentom na základe predchádzajúcej žiadosti konzumenta. Filtrovanie správ je založené na položke *topic* (téma) v hlavičke správy. *Téma* type je tvorená bodkou oddelenými skupinami číslic a písmen anglickej abecedy. Prvé skupiny znakov vyjadrujú obecnú tému, a každá ďalšia skupina tému konkretizuje.

Jedna z výhod tohto formátu je možnosť vytvárať menné priestory a tak jednoducho predchádzať konfliktom názvu. Príjemca sa nemusí prihlásiť iba na konkrétne typy správ, ale môže použiť takzvanú hviezdičkovú konvenciu. Hviezdička slúži v žiadosti o prihlásenie ako zástupný znak pre ľubovolný text. Takýto formát sa ukázal sa byť dostatočne zrozumiteľný a užitočný, pretože ho používa aj jazyk Java na identifikáciu balíkov tried (Gosling, 2014) a, aj keď v opačnom poradí, aj služba DNS.

Správy, ktoré dátová zbernica testos podporuje môžeme rozdeliť na niekoľko typov:

- *Signal*. Signál je základný druh správy bez odpovede. Hlavička signálu neobsahuje žiadne ďalšie položky okrem tých spoločných pre dátové správy. Signál je po odoslaní eventuálne doručený všetkým odberateľom daného typu správy. Ak nieje žiadny odberateľ, signál sa zahodí.
- *Request*. Žiadosť je správa s povinnou odpoveďou. Správa je doručená jednému odberateľovi daného typu. Odpoveď na žiadosť je doručená odosielateľovi žiadosti.
- *MultiRequest*. Násobná žiadosť, podobne ako obyčajná žiadosť je správa s povinnou odpoveďou. Takáto žiadosť je doručená všetkým vyhovujúcim odberateľom a teda odosielateľ dostane viacero odpovedí.
- *Response*. Správa odpoveď môže byť poslaná iba ako odpoveď na predchádzajúcu žiadosť. Odpovede nemajú tému ale miesto toho majú v hlavičke identifikátor žiadosti, ktorý slúži na doručenie odpovede späť k odosielateľovi žiadosti.
- *MultiResponse*. Viacnásobná odpoveď je odpoveď na viacnásobnú žiadosť. V hlavičke je navyše uložený príznak označujúci poslednú zo série odpovedí na žiadosť.

Prihlásenie na odber

Prihlasovanie na odber je dôležitý aspekt zbernice. Aplikácie vyjadrujú svoj záujem o správy určitého typu pomocou API funkcií *Subscribe* alebo *Provide***.** Operácia Subscribe sa používa na odber signálov a Provide na odber žiadostí. Slovo provide bolo zvolené, pretože aplikácia vlastne poskytuje službu tým, že odpovedá na žiadosti.

Žiadosti o odber

Aplikácie musia mať možnosť odberať správy alebo poskytovať služby na zbernici. Táto funkcionalita je umožnená žiadosťami o odber. Keď aplikácia chce ničo odberať, vytvorí žiadosť o odber, ktorá sa najprv spracuje lokálne a potom sa rozpošle ostatným uzlom na segmente zbernice. Spracovanie žiadosti zahŕňa kontrolu a následnú úpravu smerovacej tabuľky. V smerovacej tabuľke sa priradí identifikátor príjemcu k predmetu. Pri doručení správy sa v smerovacej tabuľke hľadá dva krát. Najprv na odosieľajúcom uzle, aby vedel na ktoré susedné uzly sa má správa poslať. Na uzle adresáta sa z smerovacej tabuľky prečíta, aká rutina sa má vyvolať pri prijatí tej konkrétnej správy.

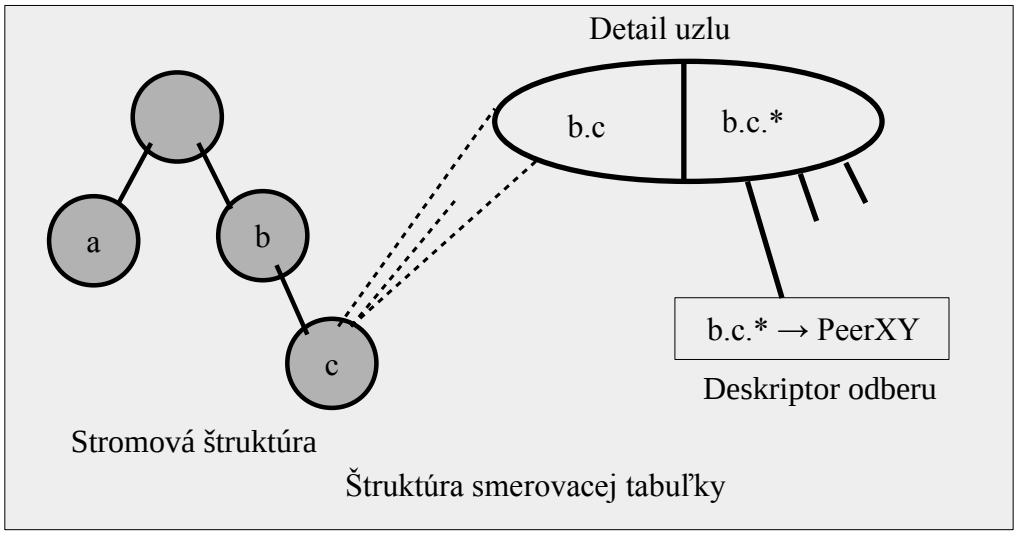
Pri pripojení nového uzla, mu ostatné uzly pošlú informácie o svojich odberoch.

Špeciálny typ žiadosti o odber je odber samotných žiadostí o odber. Smerovacia tabuľka každého uzla je implicitne prihlásená na odber žiadostí o odber. Takýto odber bude tiež užitočný pri implementácií proxy služby. Tento odber výnimočne posiela správy retrospektívne. A to preto, aby informácie o odberoch boli poslané každému novo-pripojenému uzlu.

Žiadosť musí obsahovať aj informáciu o tom, aký komunikačný vzor aplikácia očakáva. Aby sa nestalo, že sa napríklad doručí signál, na ktorý sa odpoveď neposiela, rutine ktorá chce odoslať odpoveď. Odber môže mať aj takzvaný exkluzívny príznak. Tento príznak hovorí, že na daný predmet môže byť prihlásený iba jeden odberateľ. Ak by sa pokúsil prihlásiť ďalší odberateľ, vygeneruje sa chyba. Ďalšia hodnota odberu je priorita, ktorá pomáha určiť, kam sa správa pošle, ak predmet správy vyhovuje viacero odberom.

Žiadosť o odber žiadostí môže byť prípadne označená ako perzistentná. V takom prípade bude špecifikovať maximálny čas, ako dlho tú žiadosť bude Testos Bus evidovať po nečakanom odpojení odosielateľa. Táto vlastnosť je užitočná, ak je predmetom odberu dôležitá služba, pretože ak služba spadne, klienti nevyhlásia hneď chybu, ale počkajú, dokým sa služba nereštartuje.

3.2 Návrh smerovacej tabuľky



Informácie o odberoch sa budú ukladať do smerovacej tabuľky. Každý uzol má svoju vlastnú smerovaciu tabuľku. Informácii o odbere správ, ktorá sa ukladá do smerovacej tabuľky, budem ďalej nazývať deskriptor odberu, alebo iba deskriptor. Obsah tabuliek jednotlivých uzlov nemusí byť zhodný, ale do určitej miery sa medzi uzlami synchronizuje. Informácie o odberoch sú v tabuľke organizované podľa hodnoty ich predmetu.

Tabuľka musí podporovať tieto operácie:

1. Vloženie nového deskriptoru,
2. Odstránenie deskriptoru,
3. Vyhľadanie všetkých deskriptorov, ktoré sa aplikujú na konkrétny predmet správy,
4. Vyhľadanie všetkých deskriptorov, ktorých predmet zdieľa určitú predponu.

Operácia 3 je určená na vyhľadanie adresátov pri smerovaní správy. Operácia 4 je určená na preposielanie žiadostí o odber.

Keďže aplikácia môže voliteľne požiadať o odber všetkých pod-tém konkrétnej témy, deskriptory sú buď viazané s konkrétnou témou, alebo s akýmsi vzorom, ktorému vyhovujú rôzne pod-témy. Vzor teda môže mať dva druhy: Konkrétny, napríklad *org.testos.db.storeA*, čo vyhovuje iba správe poslanej do témy *org.testos.db.storeA*. Alebo obecný, napríklad org.testos.db.*, čo vyhovuje správam poslaným do tém ako sú: *org.testos.db*, *org.testos.db.a*, alebo *org.testos.db.a.b.c*. Zástupný znak hviezdička, sa môže vo vzore vyskytovať iba ako posledný element. Pri vyhľadávaní (operácia 3 alebo 3), musí implementácia nájsť nielen deskriptory bez hviezdičky, ale aj všetky vyhovujúce deskriptory s hviezdičkou. Dôsledkom tejto požiadavky je, že intervalový výber (range query) v zoradenej štruktúre nieje dostatočný na vyhľadanie všetkých vyhovujúcich deskriptorov.

Ani vyhľadávacia štruktúra založená na zoradenom indexe, ani Haš-tabuľka, nieje sama dostatočná na vyhľadávanie v smerovacej tabuľke. Ale vyhľadávacia štruktúra *trie* (Allison, 2007) umožňuje takéto vyhľadávanie. Trie je vyhľadávací strom, kde každá hrana je ohodnotená jedným znakom z kľúča. Keďže počet hrán vychádzajúcich z uzla stromu nieje obmedzený, bude treba tieto hrany a ich ohodnotenia ukladať v ďalšej dátovej štruktúre. Stačí, ak táto štruktúra bude podporovať bodové (exact match) vyhľadávanie. Štruktúru trie môžeme použiť nasledovne:

- Hodnotu témy rozdelíme na elementy oddelené bodkou. Usporiadanú n-ticu z týchto elementov nazveme vyhľadávací kľúč. Zástupný znak hviezdička nieje súčasťou kľúča.
- Ak budeme nasledovať hrany ohodnotené elementami z vyhľadávacieho kľúča, dostaneme sa do uzla, ktorý je danému vyhľadávaciemu kľúču vlastný.
- Každý deskriptor v smerovacej tabuľke bude uložený práve v uzle stromu, ktorý je vlastný kľúču deskriptora. Pri vkladaní sa neexistujúce uzly vytvárajú.
- Výsledok vyhľadávania v tabuľke je zoznam všetkých deskriptorov z vlastného uzla, zjednotený s zoznamom deskriptorov s hviezdičkou z všetkých nadradených uzlov.

Vyhľadávanie v takomto trie strome je časovo úmerné k počtu elementov v vyhľadávacom kĺúči.

Implementácia smerovacej tabuľky

V minulej kapitole boli identifikované požiadavky na smerovaciu tabuľku bola navrhnutá dátová štruktúra, ktorá týmto požiadavkám vyhovuje. V tejto kapitole sa detailnejšie pozriem na implementáciu.

Do tabuľky sa ukladajú informácie o odberoch, tiež zvané deskriptory. Keďže môže byť viac druhov deskriptorov, definujem abstraktnú triedu ITap, z ktorej budú všetky konkrétne deskriptory odvodené. Generalizácia všetkých deskriptorov do abstraktnej triedy ITap zároveň oddeľuje smerovaciu tabuľku od detailov deskriptoru.

Ako som v minulej kapitole naznačil, smerovacia tabuľka je založená na vyhľadávacom strome. Uzol stromu predstavuje štruktúra TRouteNode. Štruktúra uzlu je tvorená množinou vlastných deskriptorov bez hviezdičky, množinou vlastných deskriptorov s hviezdičkou, názvom uzlu, ukazateľom na nadradený uzol a mapou hrán smerujúcich k podradeným uzlom. Deskriptory sú ukladané do dvoch množín, pre potreby vyhľadávania. Mapu a množiny v uzle stromu som sa rozhodol implementovať kontajnermi zo štandardnej knižnice.

Samotná smerovacia tabuľka je zabalená v triede CRoutingTable. Položka root ukazuje na koreňový uzol stromu, ktorý nemá názov. Presnejšie povedané, názov koreňového uzlu je prázdny reťazec.

Vkladanie nového deskriptoru do tabuľky je implementované metódou AddTap. Vkladaný deskriptor odberu musí byť potomok triedy ITap a nesmie už byť vložený do žiadnej tabuľky. Operácia vkladania je rozložená na dve pod-operácie: nájdenie vlastného uzlu a vloženie do príslušnej množiny. Hľadanie vlastného uzla prebieha iteratívne. Začína sa v koreni stromu a postupne pre každý element vyhľadávacieho kľúča sa cez mapu hrán prejde na nižší uzol stromu. Ak v mape hrán nieje hrana ohodnotená aktuálnym elementom kľúča, vytvorí sa nový uzol pre daný element a do mapy hrán aktuálneho uzlu pridá názov nového uzlu a ukazateľ na nový uzol. Názov nového uzlu, je samozrejme zhodný s hodnotou aktuálneho elementu vyhľadávacieho kľúča. Keďže novo-vytvorený uzol je prázdny a nemá žiadne hrany, preto pre každý ďalší element vyhľadávacieho kľúča sa vytvorí nový uzol.

Vyhľadávací kľúč som definoval v teoretickej kapitole. Zástupný znak, hviezdička, na konci témy netvorí samostatný element vo vyhľadávacom kľúči a tiež sa nevytvára žiadny uzol, ktorý by mal názov iba hviezdička.

Objekte ITap má ukazateľ na uzol smerovacej tabuľky, kde bol vložený. Nasledovaním tohto ukazateľa a ukazateľov na nadradené uzly je možné ľahko zrekonštruovať vyhľadávací kľúč. Pre to názov témy na ktorú je deskriptor podpísaný nieje nutné uchovávať v deskriptore a može sa uvoľniť.

Názov uzlu, a hodnotenie hrany k nemu vedúcej, je zhodné. Stačí teda ukladať názov uzlu iba v ohodnotení hrany (pár string-pointer v mape hrán) a z uzla na tento reťazec viesť iba pointer.

Uzol smerovacej tabuľky tiež obsahuje príznak exclusive, ktorý slúži na obsluhu rovnomenného príznaku v žiadostiach o odber. Tento príznak sa nastaví, keď sa do uzlu vloží deskriptor s príznakom exclusive. Pri pokuse o vloženie ďalšieho deskriptoru odberu operácia skončí s chybov.

Objekt ITap obsahuje ukazateľ na uzol, do ktorého bol vložený, aby ho bolo možné zase z uzlu odstrániť. Odstránenie deskriptoru je implementované metódou RemoveTap. Najprv sa ITap objekt odstráni z príslušnej množiny z uzla TRouteNode. Ak sa uzol po odstránení deskriptora stane prázdnym, čo znamená že neobsahuje už žiadne deskriptory a ani nemá žiadne pod-uzly, môže sa takýto uzol vymazať z nadradeného uzla. Analogicky sa skontrolujú a odstránia prázdne uzly až po koreň stromu. Koreň stromu nemá nadradený uzol a preto sa neuvažuje jeho uvoľnenie pri odstraňovaní deskriptoru.

Vyhľadanie všetkých deskriptorov, ktoré sa aplikujú na konkrétny predmet správy je implementované metódou Lookup. Postupuje sa od koreňa stromu až po vlastný uzol témy, tak že sa nasledujú hrany postupne ohodnotené elementami vyhľadávacieho kľúča. Ak v určitom kroku taká hrana nieje, žiadny uzol sa nevytvára, ale vyhľadávanie končí. V každom kroku sa z aktuálneho uzla skopírujú ukazatele na deskriptory s hviezdičkou do výsledného vektora. Na záver sa z vlastného uzla, ak bol dosiahnutý, skopírujú do výsledku aj deskriptory bez hviezdičky.

3.3 Tok Správ, Fronty

Správy, ktoré vygeneruje aplikácia, musí uzol určitým spôsobom doručiť adresátovi. Po prijatí akejkoľvek správy, či už od aplikácie alebo zo siete sa pomocou smerovacej tabuľky určí, kam všade sa má správa poslať. Smerovacia tabuľka môže vrátiť žiadny alebo viac odkazov na lokálnu rutinu, alebo odkaz na iný uzol v sieti. Potom sa správa musí poslať cez všetky tieto odkazy. Samozrejme, ak smerovacia tabuľka vrátila viac odkazov na jeden vzdialený uzol, stačí tam správu poslať raz.

To je základná schéma ako by mali správy tiecť cez uzol. Do tohto toku môžeme pridať fronty. Fronty majú tú výhodu, že zvyšujú asynchrónnosť. Dovoľujú aplikácií pokračovať v činnosti, zatiaľ čo správy čakajú na doručenie. Fronty sa ale môžu zaplniť, a potom iba zvyšujú latenciu systému. Netreba zabúdať že operačný systém už má fronty na soketoch.

Fronta by mohla byť umiestnená na vstupe správy do uzla, alebo v bode pred odoslaním správy ďalšiemu uzlu. Fronta na vstupe zo siete do uzla nedáva zmysel, pretože sokety, ktorými bude uzol komunikovať, majú už príchodziu frontu. Fronta dáva zmysel v bode odosielania správ cez sieť. Sokety TCP neumožňujú určiť prioritu určiť prioritu odosielaných správ, pretože TCP je protokol, ktorý pracuje s usporiadaným prúdom bajtov. Ak by implementácia uzlu obsahovala prioritnú frontu v bode odosielania, aplikácie by to mohli využiť na prenos prioritných správ. Prioritná správa by mohla byť napríklad správa vyjadrujúca postup v plnení určitej úlohy. Kapacita jednej fronty, alebo spoločná kapacita front musí byt obmedzená, ak nechceme aby došlo k vyčerpaniu operačnej pamäte. Správne ošetrenie preplnenej fronty môže byť komplikované. Teda ak nepotrebujeme prioritné doručovanie, bude lepšie nechať fronty na operačný systém.

Aj tak je ale treba správne ošetriť problém pomalého konzumenta. Najjednoduchšie riešenie je blokovať pri odosielaní. To môže zablokovať celí program dokým sa správa neodošle alebo nevyprší čas. Alternatívou je v takomto stave prestať obsluhovať zdroj blokovanej správy.

3.4 Analýza sieťovej architektúry

Základná architektúra služby je peer-to-peer rozšírená o voliteľnú serverovú štruktúru. Aplikáciam sú služby testos zbernice prezentované prostredníctvom rozhrania zvaného adaptér.

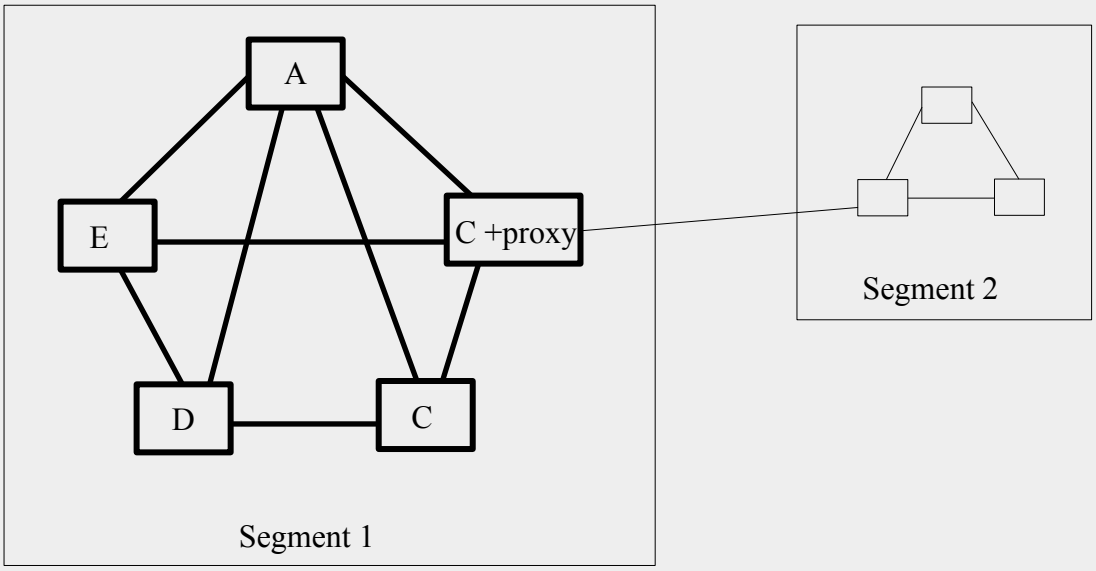
Pokiaľ je to možné, Testos Bus by mal prenášať správy priamo. Infraštruktúra Internetu dokáže efektívne smerovať *unicast* správy, čo sa aj dlhodobo potvrdilo. Preto dátová zbernica by nemala replikovať smerovacie algoritmy na aplikačnej úrovni. Technológie nižšej vrstvy (TCP/IP) už zvládajú smerovanie, zavedenie viac-krokového smerovacieho protokolu do režimu peer-to-peer by bolo redundantné.

Každú správu, ktorú adaptér obdrží, buď lokálne doručí aplikácií alebo zahodí. Keďže sa správy neposielajú ďalej, nemôžu vzniknúť smerovacie slučky. Ale nadviazanie priamich spojení medzi klientmi môžu komplikovať NAT-y a Firewall-y.

Naskytli sa dve alternatívne sieťové architektúry:

3.4.1 Úplné prepojenie uzlov v segmente

Jedna z možností je prepojiť všetky inštancie testos bus adaptéra so všetkými v zmysle úplného Kn-grafu. Súbor takýchto prepojených uzlov budeme nazývať *segment zbernice*.



Uplné prepojenie uzlov v segmente. Vyobrazené je aj proxy spojenie do segmentu 2.

Nadviazanie a udržiavanie spojení môže byť automatické. Po pripojení k jednému uzlu zo segmentu sa adaptér dozvie o ostatných uzloch v segmente a tiež sa na ne pripojí.

Jedna z výhod tejto architektúry je, že nieje treba žiadne servere na prevádzku malej zbernice. Propagácia informácií o odberoch je tiež jednoduchá. Uzol, ktorý chce odoberať niakú tému, rozpošle túto informáciu všetkým ostatným uzlom. A obdobne po pripojení nového uzlu, mu ostatné uzly pošlú informácie o všetkých aktuálnych odberoch.

Nevýhodou tohto návrhu je množstvo nečinných spojení, ktoré narastá kvadraticky s počtom uzlov v segmente.

Táto nevýhoda sa dá čiastočne riešiť rozdelením systému na viac segmentov komunikáciu medzi segmentami sprostredkovať inak. Komunikáciu s medzi segmentami zbernice zabezpečuje adaptér so službou *proxy*. Proxy servere sú dobý kandidáti na randevú - bod pre zavedenie peer-to-peer režimu.

Proxy uzol je jedným zo spôsobov ako nadviazať komunikáciu medzi segmentami zbernice. Jeden proxy uzol môže byť spojený s niekoľkými segmentmi, prípadne s ďalšími proxy uzlami. Proxy, na rozdiel od normálneho uzla, pre-posiela správy. Pri preposielaní správy sa však môže stať, že výstupný soket je plný. Normálne uzly si môžu dovoliť riešiť takýto problém zablokovaním aplikácie, no pozastavenie proxy by zablokovalo celú komunikáciu.

Jednoduchá implementácia proxy pre-posiela všetky žiadosti o odber všetkým okrem odosielateľa a pre-posiela dátové správy podľa smerovacej tabuľky. Zložitejšia implementácia môže správy a odbery filtrovať. Zreťazením proxy serverov je možné vytvoriť hierarchiu pod-segmentov.

3.4.2 Service discovery server

Ďalší spôsob komunikácie je použiť smerovací server. Smerovací server slúži ako server DNS. Eviduje všetky odbery v svojej kompetencií ale nepreposiela správy. Namiesto toho, odpovedá klientom, kam majú správu odoslať až keď ju budú chcieť odoslať. Výhoda tohto spôsobu je, že žiadosti o odber sa nebudú zbytočne preposielať uzlom, ktoré nikdy takú správu nepošlú. Smerovacia tabuľka uzlov bude ako dôsledok menšia, no odosielanie správ bude o niečo zložitejšie. Server môže tiež plniť úlohu reverznej proxy a tak asistovať pri prekonávaní NAT a Firewall-ov.

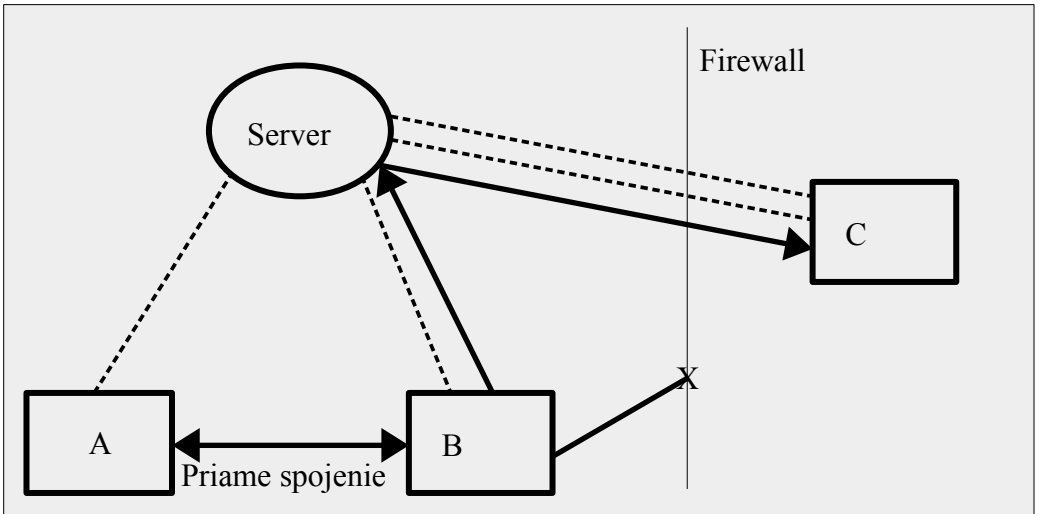


Schéma uzlov využívajpcich service discovery server.

Uzol udržuje spojenie s smerovacím serverom. Informácie o smerovaní mimo segment získa dotazom k smerovaciemu serveru. Ak sa má správa poslať mimo segmentu na uzol, ku ktorému zrovna nieje spojenie, nadviaže sa dočasné spojenie, cez ktoré sa pošle správa.

Výhod priameho pripojenia je niekoľko: nevznikajú fronty na uzloch po ceste, ak sa spojenie preruší, odosielateľ sa o tom dozvie hneď, správy sa zbytočne nezdržiavajú na proxy serveroch, proxy nemusí riešiť férové zaplnenie front na aplikačnej úrovni.

Kontaktovať smerovací server pred odoslaním každej správy by značne zdržiavalo prevádzku a zbytočne zaťažovalo server, bolo by vhodné odpovede servera uchovávať určitú dobu vo vyrovnávacej pamäti. Preto smerovací server ku každej odpovedi priloží časový údaj, ako dlho budú je odpoveď platná. Kým platnosť odpovede nevyprší, server sa zaväzuje aktualizovať údaje. Pri odosielaní ďalšej správy, testos bus uzol skontroluje dobu platnosti informácií v svojej vyrovnávacej pamäti a ak už vypršala, alebo je tesne pred vypršaním, musí kontaktovať smerovací server znovu. Ak ale je záznam stále platný, server sa nemusí kontaktovať. Okrajová situácia nastane, nikto neodoberá tému na, ktorú sa uzol pokúša poslať správu. V tom prípade by mal server vrátiť prázdnu odpoveď. Teda aj prázdna odpoveď je platná, a mala by sa uložiť do vyrovnávacej pamäte. Smerovacia tabuľka uzlu môže slúžiť ako vyrovnávacia pamäť. Pri pokuse odoslať ďalšiu takúto správu, uzol nájde v svojej smerovacej tabuľke keš-deskriptor a bude vedieť, že ostatné údaje v tabuľke sú na túto tému platné a môže ich použiť bez dotazu k serveru.

Expirované keš-deskriptory a informácie o odberoch by sa pre šetrenie miesta mali zo smerovacej tabuľky eventuálne vymazať. Server môže, pre zníženie počtu dotazov, poslať odpoveď s dlhšou dobou platnosti a môže do odpovede zahrnúť väčšiu časť svojej smerovacej tabuľky ako je nutné.

Keš deskriptory tiež slúžia na odhalenie neplatných údajov. Ak klient napríklad posielal správu *a.b.c* a teda má v smerovace tabuľke od servera informácie o odberateľoch takých správ. Napríklad deskriptory odberu: *a.b.c* a *a.b.**. Je jasné, že ak sa teraz klient pokúsi odoslať správu *a.b.d*, deskriptor pre *a.b.** sa objavý vo výsledku, lenže to nieje správne, pretože server má aj deskriptor *a.b.d*. Riešenie s keš deskriptorom je jednoduché: klient bude považovať lokálne údaje za platné, iba ak vo výsledku je aj platný keš deskriptor. V spomenutom príklade, klient bude mať *a.b.c* keš. Pre správu *a.b.d* nemá keš, tak bude musieť informácie o adresátoch pre správ a.b.d vyžiadať z servera.

Server by mal mať plnú kontrolu nad keš deskriptormi, ktoré posiela. Napríklad môže optimalizovať, a na dotaz *a.b.c* poslať všetky informácie z pod-stromu *a.b*, vrátane keš deskriptoru *a.b.** a tým ušetriť klientovi na informácie v spomínanom pod-strome. Čas platnosti by tiež mal učiť server.

Oproti proxy má tento prístup nevýhodu v tom, že dotaz na smerovací server, a nasledovné pripojenie na uzly mimo segmentu trvá určitý čas navyše. Vyrovnávacia pamäť toto zdržanie do určitej miery zakrýva. Výhoda je, že správy prúdia pokiaľ možno priamo – bez množstva zbytočných spojení.

Po nadviazaní spojenia medzi uzlami iniciátor spojenia hneď pošle správu, pre ktorú bolo toto spojenie nadviazané. Iniciátor musí udržať spojenie otvorené minimálne dokým nedostane odpoveď na všetky poslané žiadosti. Nadviazanie spojenia v prípade TCP vyžaduje určitý čas, preto by bolo vhodné udržať dočasné spojenia určitý čas po tom, ako naplnili svoju úlohu.

Cez spojenie cez ktoré bola prijatá žiadosť, nesmie príjemca žiadosti poslať iné správy ako odpoveď na prijatú žiadosť. Ak by to bolo dovolené, mohlo by dôjsť k uviaznutiu.

Pre niektoré správy by bolo výhodné použiť aj protokol UDP pretože nadviazanie spojenia je veľmi rýchle. Keďže UDP je protokol nespoľahlivý a bez riadenia toku, jedine jednoduché žiadosti ho môžu využiť. Ak by sa žiadosť alebo odpoveď stratili, iniciátor žiadosti by mal žiadosť opakovať.

3.5 Sieťový protokol

Na komunikácií sa podieľajú následovný aktéri:

- smerovací server
- poskytovatelia služieb
- klienti služieb

TestosBUS server, ktorý tiež nazývam smerovací server, slúži ako register služieb a odberov na sieti. Poskytovatelia služieb, alebo odberatelia správ oznamujú serveru, aké témy odoberajú. Užívatelia služieb naopak od serveru žiadajú zoznam odberateľov, keď chcú poslať správu. Klienti zbernice môžu súčasne poskytovať ja používať služby.

Komunikácia prebieha medzi klientmi a serverom, ďalej medzi jednotlivými klientmi a v prípade niekoľkých serverov aj navzájom medzi nimi.

Protokol, ktorým klienti komunikujú so serverom, je asynchrónny a založený na správach. Každá správa má spoločnú hlavičku dlhú 6 bajtov tvorenú nasledovnými položkami:

Length	32-bit unsigned	Dĺžka správy v bajtoch (okrem hlavičky)
Type	8-bit unsigned	Typ správy
Flags	8-bit	Príznaky

Bezprostredne po hlavičke nasleduje zbytok správy dlhý Length bajtov.

Okrem základných numerických typov ďalej v popise protokolu používam aj niekoľko zložitejších. Bajty numerických typov *signed* a *unsigned* sú v sieťovom poradí, t.j. big-endian. Zložené typy definujem tu:

- *ShortString*: predstavuje reťazec. Je uložený ako 16-bit unsigned hodnota predstavujúca dĺžku (v bajtoch) nasledovaná príslušným počtom znakov.
- *TIdentifier* je ďalší reťazcový typ, ktorý vša má dĺžku uloženú ako 8-bit unsigned a môže obsahovať iba znaky povolené v C99 identifikátoroch.
- *TTopic* slúži na uloženie témy správy a je zložený z jedného alebo viac TIdentifier. Typ TTopic musí byť ukončený identifikátorom nulovej dĺžky.
- *TAddress* obsahuje sieťovú adresu, prípadne niekoľko, v rôznych formátoch a má nasledovnú štruktúru:

Názov	Príznak	Typ	
addrflags		8-bit	Príznaky.
Port		16-bit unsigned	Číslo TCP/UDP portu.
IPv4	F_IP4 (1)	32-bit	IPv4 adresa.
IPv6	F_IP6 (2)	128-bit (16 byte)	IPv6 adresa.
UnixSocket	F_UNIX (3)	ShortString	Cesta k unix-domain soketu.

V Type TAddress môžú byť uložené naraz aj všetky tri typy adries, pokiaľ sú nastavené príslušné príznaky. Ďalšie dva príznaky F_TCP (4) a F_UDP (5) značia či je možné použiť protokol TCP, UDP alebo oboje na pripojenie k uvedenej adrese.

Ďalej uvádzam prehľad jednotlivých správ s uvedením funkcie, štruktúry a spôsobu ich reálneho používania v prebiehajúcej komunikácii.

Typ Správy	Smer	Odpoveďe
ClientHello	Klient -> Server	SetID
Listen	Klient -> Server	
ReverseRelay	Server -> Klient	
RelayHello	Klient -> Relay	
Subscribe	Klient -> Server	SubAck, SubErr
Unsubscribe	Klient -> Server	
Lookup	Klient -> Server	RplSub, RplUnsub, RplCache
RplSub	Server -> Klient	
RplUnsub	Server -> Klient	
RplCache	Server -> Klient	
LookupExact	Klient -> Server	ReplyExact
Signal	Klient -> Klient	
Request	Klient -> Klient	Reply, NoReply
Reply	Klient -> Klient	

Správa ClientHello (Typ 1)

Túto správu posiela klient ihneď po nadviazaní spojenia so serverom.

Version	16-bit unsigned	Verzia protokolu s ktorou je kompatibilný klient (hodnota 1)
SelfID	TIdentifier	Identifikácia klienta

Prvá verzia protokolu má číslo 1. SelfID je identifikácia určená klientom samotným, server rozhodne, či klientovi dovolí používať vybrané ID alebo mu pridelí iné. Na túto správu server odpovie správou SetID.

Správa SetID (Typ 2)

Version	16-bit unsigned	Verzia protokolu s ktorou je kompatibilný server (hodnota 1)
ClientID	TTopic	Pridelená identifikácia klienta.

ClientID je vygenerované serverom pre klienta. Server môže pri generovaní použiť SelfID zo správy ClientHello. ID klienta spravidla začína identifikátorom servera a pokračuje diskriminačným identifikátorom pre klienta. Identifikátor servera je z menného priestoru `org.testos.bus`.

Správa Listen (Typ 3)

Správu Listen klient odosiela následne po správe ClientHello na začiatku komunikácie, aby serveru oznámil, na akom porte je pripravený prijať prichádzajúce spojenia. Táto správa nie je povinná v prípade, že klient nepotrebuje alebo nechce prijímať žiadne signály ani žiadosti.

Príznaky:

F_IP4	1	Klient počúva na IPv4 sokete
F_IP6	2	Klient počúva na IPv6 sokete
F_TCP	3	Klient počúva na TCP/IP sokete
F_UDP	4	Klient počúva na UDP/IP sokete
F_UNIX	5	Klient počúva na Unix-Domain sokete

Položky:

IPPort	F_IP4, F_IP6	16-bit unsigned	Port IPv6 alebo IPv4 pasívneho soketu na príjem spojení
SelfAddr	F_IP4, F_IP6	TAddress	Adresa soketu, cez ktorý bola táto správa odoslaná
Unix	F_UNIX	ShortString	Cesta k unix-domain soketu na príjem spojení

Položka SelfAddr v tejto správa slúži na detekciu NAT zariadení medzi serverom a klientom. Ak server zistí, že sa na soket klienta nedá pripojiť, či už z dôvodu NAT-u alebo Firewallu, ponúkne klientovi riešenie správou ReverseForward. Ak je klient pripojený k serveru cez IP spojenie, IP adresa klienta je zhodná s adresou odosielateľa správy Listen. Táto adresa spolu s položkou IPPort tvorí kompletnú adresu klienta. Ak je klient pripojený cez Unix soket, server použije svoju IP adresu, pretože unix-domain spojenie je platné iba na jedom stroji. Server tiež použije svoju adresu, ak je klient pripojený z loopback adresy (::1, 127.0.0.1, ::ffff:7f00:1).

Ak je nastavený príznak F_UNIX server môže poskytnúť adresu z položky Unix ostatným klientom na tom istom stroji. Unix-domain sokety môžu byť na niektorých systémoch rýchlejšie ako lokálne TCP/IP spojenia (cite).

Server môže overiť funkčnosť pasívneho soketu klienta pomocou správy Ping, ale nemusí, ak hodnota SelfAddr nenaznačuje, že klient nieje za NAT-om.

Správa ReverseRelay (Typ 4)

Ak z predchádzajúcej správy Listen server zistil, že sa klient nachádza za NAT-om, server poskytne preposielanie správ pomocou tejto správy.

RelayAddr		TAddress	Adresa relay služby.
RelayID		64-bit	Identifikátor klienta pre relay službu.

Po prijatí tejto správy sa klient pripojí na TCP/IP RelayAddr, cez ktorú mu bude server preposielať signály a žiadosti. Po nadviazaní spojenia, klient pošle RelayHello správu, čím inicializuje službu relay.

Správa ReverseRelay sa posiela asynchrónne, preto server musí pozdržať spracovanie žiadostí o odber dokým klient správu ReverseRelay nedostane a nepripojí sa na relay, alebo, dokým nevyprší čas na túto operáciu.

Správa RelayHello (Typ 5)

Správou RelayHello sa klient identifikuje relay službe.

ClientID		TTopic	Identifikátor klienta.
RelayID		64-bit	Identifikátor klienta pre relay službu.

Správa Subscribe (Typ 6)

Príznaky:

F_exclusive	1	Na rovnakú tému nesme byť prihlásený žiaden iný klient.
F_nosubtopic	2	Žiaden klient nesmie byť prihlásený na pod-tému danej témy
F_request	3	Príznak odberu požiadaviek.
F_global	4	Odber má byť propagovaný iba klientom tohto servera.
F_subsub		

Položky:

Položka	Typ	
Topic	TTopic	Téma odberu.
SubID	32-bit	Identifikátor odberu.
Priority	16-bit signed	Priorita.
timeout	32-bit unsigned	Čas perzistencie v centi-sekundách

Správa Subscribe predstavuje žiadosť o odber. Posiela ju klient keď sa chce prihlásiť na odber určitej témy. Položka SubID jednoznačne identifikuje odber v rámci klienta a klient si ju môže určiť podľa svojho uváženia. Položka topic vyjadruje tému odberu a môže voliteľne obsahovať zástupný znak. Položka Priority ovplyvňuje výber adresáta ak je ich viac dostupných pri posielaní požiadavky. Význam zástupného znaku, priority, timeout-u a príznakov odberu je vysvetlený v predchádzajúcich kapitolách.

Následne server skontroluje platnosť žiadosti o odber a po jej úspešnom spracovaní odpovie správou SubAck. Spracovanie žiadosti serverom zahŕňa úpravu smerovacej tabuľky a propagáciu žiadosti. V prípade neúspechu, server odpovie správou SubErr.

Správa SubAck (Typ 7)

Príznaky:

F_resumed	1	Úspešné obnovenie predchádzajúceho odberu pred vypršaním času.

Položky:

Položka	Typ	
SubID	32-bit	Identifikátor odberu.

Správa vyjadruje úspešné spracovanie žiadosti o odber. Server odoslaním tejto správy vyjadruje, že všetky smerovacie tabuľky boli upravené a ďalšia vyhovujúca správa bude poslaná klientovi.

Správa SubErr (Typ 8)

Príznaky:

F_conflict	1	Na rovnakú tému bol už prihlásený iný klient.
F_subtopic	2	Na pod-tému už bol prihlásený iný klient.
F_nolisten	3	Ku klientovi sa nepodarilo pripojiť.
F_serverfail	4	Interná chyba serveru.

Položky:

Položka	Typ	
SubID	32-bit	Identifikátor odberu.
Message	ShortString	Doplňujúca chybová správa.

Správa *SubErr* vyjadruje neúspešné spracovanie žiadosti o odber. Príznaky tejto správy bližšie špecifikujú možné bežné chybové stavy:

Ak v žiadosti bol nastavený príznak F_exclusive a server už má v smerovacej tabuľke odber na danú tému, bude v odpovedi nastavený príznak F_conflict. Podobne príznak F_subtopic vyjadruje porušenie príznaku F_nosubtopic zo žiadosti. Príznak F_nolisten bude nastavený buď ak klient neposlal správu Listen, alebo sa naň nedá pripojiť.

Správa Unsubscribe (Typ 9)

Položka	Typ	
Topic	TTopic	Téma odberu
SubID	32-bit	Identifikátor odberu.
Timeout	32-bit unsigned	Čas perzistencie v centi-sekundách.

Klient pošle túto správu, keď chce zrušiť svoj predchádzajúci odber. Ak klient nie je prihlásený na daný odber, správa je ignorovaná. Správa sa spracuje asynchrónne a server neposiela odpoveď. Kým sa všetky smerovacie tabuľky neaktualizujú, klient môže dostávať extra správy zo zrušeného odberu. Extra signály môže bezpečne ignorovať a žiadosti zamietať.

Správa Lookup (Typ 10)

Položka	Typ	
Topic	TTopic	Téma vyhľadávania.

Keď klient chce odoslať signál alebo žiadosť a nemá vo vyrovnávacej pamäti informácie o odberoch, vyžiada si tieto informácie od serveru pomocou správy *Lookup*. Položka Topic je nastavená na tému správy, ktorú chce odoslať. Klient by nemal posielať ďalšie správy Lookup ohľadom rovnakej témy, dokým neodstane od serveru odpoveď. Server ako odpoveď najprv pošle správy *RplSub* s informáciami o odberoch a potom správu *Cache*.

Správa RplSub (Typ 11)

Príznaky:

F_exclusive	1	Na rovnakú tému nesme byť prihlásený žiaden iný klient.
F_nosubtopic	2	Žiaden klient nesmie byť prihlásený na pod-tému danej témy
F_request	3	Príznak odberu požiadaviek.

Položky:

Položka	Typ	
Topic	TTopic	Téma odberu.
SubID	32-bit	Identifikátor odberu.
Priority	16-bit signed	Priorita.
Expire	32-bit unsigned	Čas exspirácie údajov
Remote	TAddress	

Správu RplSub posiela server ako odpoveď na Lookup, alebo ako priebežnú aktualizáciu vyrovnávacej pamäti klienta. Rovnomenné položky a príznaky majú rovnaký význam ako v správe Subscribe.

Správa RplUnsub (Typ 12)

Položka	Typ	
Topic	TTopic	Téma odberu.
SubID	32-bit	Identifikátor odberu.
Remote	TAddress	

Správu *RplSub* posiela server ako priebežnú aktualizáciu vyrovnávacej pamäti klienta. Správa značí, že odber s konkrétnym SubID a Remote už nieje platný.

Správa Cache (Typ 13)

Položka	Typ	
Topic	TTopic	Téma Keš deskriptoru.
Expire	32-bit unsigned	Čas exspirácie údajov

Správa vkladá keš deskriptor do tabuľky klienta.

Správa LookupExact (Typ 14)

Príznaky:

F_signal	1	Iba informácie o odbere signálov.
F_request	2	Iba informácie o odbere žiadostí.
F_multircpt	3	

Položky:

Položka	Typ	
CorrID	32-bit	Korelačné ID
Topic	TTopic	Téma vyhľadávania.
Timeout	32-bit unsigned	Čas v centi-sekundách.
Max	16-bit unsigned	Obmedzenie počtu deskriptorov v odpovedi.

Správu *LookupExact* môžu poslať tzv. odľahčení klienti na zistenie adresátov. Odľahčeným klientom sa neoplatí spravovať smerovacia tabuľka, pretože potrebujú odoslať len malé množstvo správ na malý počet tém. Topic bude nastavený na tému správy, ktorú chce odľahčený klient odoslať a príznaky podľa toho, či chce informácie o odberoch signálov, žiadostí alebo oboje. CorrID si môže klient nastaviť ľubovolne. Timeout značí, ako najdlhšie má server čakať na obnovenie dostupnosti perzistentnej služby ak nieje zrovna dostupná. Server odpovie správou ReplyExact.

Správa ReplyExact (Typ 15)

Srávu *ReplyExact* posiela server ako odpoveď na správu *LookupExact*. Táto správa má iný formát ako ostatné, pretože obsahuje už niekoľko deskriptorov odberov. Za univerzálnou 6-bajtovou hlavičkou nasleduje ďalšia hlavička:

Položka	Typ	
Topic	TTopic	
Expire	32-bit unsigned	
CorrID	32-bit	Korelačné ID
SubN	16-bit unsigned	Počet deskriptorov odberou v správe.

Po druhej hlavičke nasleduje nula až niekoľko deskriptorov odberov:

Položka	Typ	
Flags2	8-bit	príznaky
PrefixN	8-bit	Počet zhodných elementov témy.
SubID	32-bit	Identifikátor odberu.
Priority	16-bit signed	Priorita.
Remote	TAddr	

Príznaky (Flags2):

F_exclusive	1	
F_nosubtopic	2	
F_request	3	

Rovnomenné položky a príznaky majú rovnaký význam ako pri správe Subscribe. Remote. CorrID je určené na spárovanie odpovede s požiadavkou. Položka PrefixN vyjadruje počet zhodných elementov zľava medzi témou požiadavky a témou deskriptoru odberu. Klient použije PrefixN, spolu s položkou Priority pri rozhodovaní, ktorému z adresátov žiadosť pošle.

Správa Error (Typ 16)

Položka	Typ	
Message	ShortString	Popis chyby.

Správa Error značí neopraviteľnú chybu a posiela sa ako posledná správa pred uzatvorením spojenia. Táto správa slúži primárne na diagnostiku a ladenie, preto jediná jej položka, Message, obsahuje krátky popis chyby.

3.5.1 Komunikácia Klient-Klient.

Po tom ako klient od servera získa smerovacie informácie môže správu odoslať. Iba kontrolné správy sa posielajú cez server. Samotné žiadosti, signály a odpovede sa posielajú medzi klientmi priamo.
Ak ešte neexistuje spojenie medzi uzlami, ktoré si chcú poslať správu, spojenie vytvorí iniciátor a hneď pošle svoju správu.

V komunikácii klientov navzájom tiež použitý protokol, ktorého základ tvoria správy a využíva rovnakú hlavičku ako klient-server protokol. Jedine správy prenášané medzi klientmi sú iné:

Správa Signal (Typ 17)

Položka	Typ	
Topic	TTopic	Téma odberu.
SubID	32-bit	Identifikátor odberu.
Body	binárny	Telo správy

Správa reprezentuje užívateľskú správu typu signál. Položka SubID musí mať rovnakú hodnotu ako rovnomenná položka zo správy RplSub alebo ReplyExact.

Správa Request (Typ 18)

Príznaky:

F_multi	2	Odosielateľ dokáže spracovať viac ako jednu odpoveď.

Položky:

Položka	Typ	
Topic	TTopic	Téma odberu.
SubID	32-bit	Identifikátor odberu.
CorrID	32-bit	Korelačné ID.
Timeout	32-bit unsigned	Obmedzenie trvania žiadosti v centi-sekundách.
Body	binárny	Telo správy

Správa reprezentuje užívateľskú správu typu žiadosť. Položka SubID musí mať rovnakú hodnotu ako rovnomenná položka zo správy RplSub alebo ReplyExact. Korelačný identifikátor si zvolí odosielateľ a slúži na priradenie budúcich odpovedí k tejto správe. Na túto správu sa očakáva odpoveď Reply alebo NoReply.

Správa Reply (Typ 19)

Príznaky:

F_last	3	Posledná odpoveď zo série

Položky:

Položka	Typ	
CorrID	32-bit	Korelačné ID.
Body	binárny	Telo správy

Správa NoReply (Typ 20)

Príznaky:

F_nosub	3	Klient už neodoberá danú tému.
F_error	4	Chyba klienta.

Položky:

Položka	Typ	
CorrID	32-bit	Korelačné ID.

Táto správa sa posiela ako odpoveď na správu Request, ak ju nebolo možné obslúžiť.

3.6 Aplikačné rozhranie

Klienti používajúci testosbus ho používajú prostredníctvom rozhrania testos bus knižnice. Aby nemusel každý klient implemetovať čítanie argumentov, testosbus knižnica číta adresu na priojenie k smerovaciemu serveru z premennej prostredia alebo z konfiguračného súboru. Základná trieda tohto rozhrania je **CNode** (testos::bus::CNode), ktorý reprezentuje logické pripojenie na dátovú zbernicu. Tento objekt má sadu konfiguračných premenných, ktoré upresňujú správanie knižnice.

routerAddr	string	Adresa smerovacieho servera.
timeout	integer	Časový limit v centi-sekundách.
listen	bool	Príznak, či chce aplikácia prijímať žiadosti a signály. (true)
id	string	Identifikátor klienta.
maxRequests		
maxOutSignalB		

Po nastavení premennej routerAddr aplikácia zavolá metódu CNode::connect, čím iniciuje pripojenie k smerovaciemu serveru a zbytku siete. Po pripojení objekt CNode slúži na vytvorenie ďalších objektov.

Odosielanie signálu

Signály sa odosielajú pomocou triedy CSignal. Od aplikácie sa očakáva, že vykoná nasledovné kroky:

1. pomocou metódy CNode::Signal vytvorí nový objekt
2. nastaví položky hlavičky, predovšetkým tému signálu
3. naplní telo signálu
4. metódov Csignal::Send pripravenú správu odošle

Metóda CNode::Signal vráti ukazateľ na nový objekt reprezentujúci signál. Väčšina objektov v zbernici testos má životný cyklus riadený počítaním referencií. Signál nieje výnimkou, preto ukazateľ vrátený metdov CNode::Signal je takzvaný *smart pointer*.

Ak aplikácia vytvára správy rýchlejšie ako sa stíhajú odosielať, môže dôjsť k vyčerpaniu prostriedkov (pamäti). Aby sa tomu predišlo nastavenie CNode::maxOutSignalB obmedzuje koľko pamäte môžu obsadiť signály pripravené na odoslanie. Ak je tento limit prekročený, vytvorenie nového signálu je bude blokované (pozastavené), dokým sa situácia nezlepší. Keďže Event Loop vlákno nemôže byť blokované, pretože by došlo k uviaznutiu (deadlock), takýto pokus vráti chybu.

Najdôležitejšia položka hlavičky signálu je **topic**, čiže téma. Topic je dátového typu CTopic, ktorý je však implicitne zameniteľný s jednoduchým reťazcom.

Ďalšie položky sú *timeout*, exclusive a local. **Timeout** určuje maximálny čas na doručenie správy, v centi-sekundách. Keďže od signálu sa neočakáva odpoveď, timeout sa kontroluje iba odosielaní na sieť. Príznak **exclusive** spôsobí, že sa vyvolá chyba, ak by sa mal signál doručiť viac ako jednému príjemcovi.

Po vyplnení hlavičky je možné nastaviť telo správy. Testos BUS sa nestará veľmi čo je telom správy, ale poskytuje niekoľko primitív na binárnu serializáciu. Pomocou metódy setBody(ptr,size) môže aplikácia nastaviť ľubovoľný binárny reťazec ako telo správy. Oblasť pamäti vymedzená argumentami ptr a size sa do tela správy skopíruje.

Alternatívne, je možné telo správy zapísať postupne pomocou sady serializačných funkcií:

Funkcia	Dátový typ	Veľkosť
w1	číslo	1
w2	číslo	2
w4	číslo	4
w6	číslo	6
w8	číslo	8
writeString	reťazec	max 64KiB
writeZString	reťazec	Neobmedzené, null-terminated
writeStringAll	reťazec	Neobmedzené, po koniec správy
write	pamäť	neobmedzené

Na čítanie tela správy existujú obdobné funkcie (namiesto w, r a namiesto write, read) a funkcia seek.

Objekt CRequest

Žiadosti, ktoré posiela aplikácia sú reprezentované objektom CRequest. Nová žiadosť sa vytvorí tak, že aplikácia zavolá metódu Request nad inštanciou Node. Rquest má niekoľko položiek, ktoré môžu byť nastavené pred tým, než aplikácia začne písať telo správy.

topic	Téma, do ktorej sa má správa poslať.
timeout	Maximálna doba čakania na odoslanie a odpoveď.
exclusive	Príznak, že má systém vrátiť chybu, ak existuje viac jako jeden
multi_target	Príznak, že aplikácia spracuje odpovede od viacerých poskytovateľov
multi_result	Príznak, že aplikácia spracuje viac odpovedí od poskytovateľa.

Po nastavení týchto hodnôt, môže aplikácia nastaviť telo správy a to buď priamo:

setBody(data, size, own) – Nastaví telo správy podľa pamäti data s veľkosťou size a ak bol zadaný ukazateľ own, tak tento ukazateľ uvolní, keď už nebude potrebný.

Alebo postupne (prúdovo) použitím funkcií prúdového zápisu.

Keď je žiadosť hotová, odošle sa zavolaním metódy Send. Žiadosti na rozdiel od signálov majú odpoveď a táto odpoveď sa dá prečítať tromi spôsobmi.

Prvý spôsob je blokujúce čakanie na odpoveď pomocou metódy GetReply, ktorá vráti ukazateľ na objekt CReply. Druhý spôsob je presmerovanie odpovede do fronty CQueue alebo CReplyQueue, z ktorej môže aplikácia neskôr odpoveď vyzdvihnúť. Prevzatie odpovede cez asynchrónne spätné volanie (callback) je tretí spôsob. Ak aplikácia zvolila tento spôsob, testos bus knižnica invokuje aplikačný callback, hneď ako bude odpoveď prijatá. Spôsob prevzatia odpovedi aplikácia vyberie pri vytváraní žiadosti:

CNode::Request()	Bez argumentov	Vyzdvihnutie odpovedi cez GetReply
CNode::Request(queue)	Referencia na CQueue	Odpovede budú vložené do fronty.
CNode::Request(cb)	Lambda obslužná funkcia	Spätné volanie pri prijatí odpovede.

CRequest je objekt, ktorého životný cyklus je riadený počítadlom referencií. Jeden typ referencií, takzvané externé, sú držané aplikáciou a druhý typ, interné, sú držané vnútornými štruktúrami knižnice. Ak aplikácia zruší poslednú referenciu na Request, spracovávanie žiadosti bude knižnicou zastavené a žiadne ďalšie odpovede nebudú doručené. Samotný objekt Request bude však uvoľnený z pamäti až po uvoľnení interných aj externých referencií.

Počet žiadostí nieje treba obmedzovať na úrovni knižnice, pretože sa očakáva, že aplikácie budú čakať na odpoveďe.

CReplyQueue

CReplyQueue je lokálna fronta, do ktorej sa ukladajú odpovede. Aplikácia môže odoslať niekoľko konkurentných žiadostí, a potom pomocou tejto fronty spracovávať odpovede tak ako prichádzajú. Je to rýchlejšie ako najprv čakať na odpoveď jednej žiadosti a až potom poslať ďalšiu žiadosť. Samozrejme, záleží na aplikácií, či, a aké žiadosti dokáže paralelizovať. Na vyzdvihnutie odpovede z fronty slúži blokujúca metóda GetReply.

Objekt CRecvRequest

Keď do uzlu príde žiadosť, systém vyvolá obslužnú rutinu nastavenú pri vytvorení odberu (registrácií služby). Ako argument, je obslužnej rutine predaný ukazateľ na objekt RecvRequest.

body	Pamäť, kde je uložené telo správy
bodySize	Veľkosť tela správy
topic	Téma, do ktorej bola správa poslaná.
multi_result	Príznak, že aplikácia môže poslať viac odpovedí na túto žiadosť

Odpoveď na prijatú žiadosť aplikácia odošle pomocou objektu CReply, ktorý vytvorí metóda CRecvRequest::Reply. Ak bol v žiadosti nastavený príznak multi_result, aplikácia môže poslať viac odpovedí, a to tak, že pri odosielaní každej, okrem poslednej, nastaví nepovinný príznak multipart.

CProvide a CSubscribe sú veľmi podobné triedy a reprezentujú prihlásenie sa na odber. CSubscribe prihlasuje na odber signálov a CProvide na odber žiadostí. Správy budú doručené, podobne ako pri odpovedi na žiadosť, buď do fronty CQueue, alebo bude vyvolaná aplikáciou špecifikovaná obslužná rutina.

3.7 Príklady použitia

Prístup do bázy dát

Jedným z komponentov testosu je dátové úložisko Testos DB. Niektoré komponenty testosu potrebujú pracovať s Testos DB a mohli by použiť TestosDB na komunikáciu s databázou. Predpokladajme, že TestosDB poskytuje dve vzdialené procedúry: ulož do bázy dát a prečítaj z bázy dát. Tieto procedúry môžu korešpondovať dvom typom správ na ktoré sa databáza prihlási pomocou metódy *Provide* zbernice. Typy týchto správ môžu mať tvar napríklad: org.testos.database.store a org.testos.database.get. Aplikácie, ktoré chcú využiť služby databázy, jednoducho pošlú cez testos bus žiadosť s príslušným typom a dostanú od bázy dát odpoveď. Aplikácie vďaka testos busu nemusia riešiť na akej adrese, alebo ceste beží ktorá služba, ani riešiť chybové stavy spôsobené sieťou.

Inštrumentácia programu

Testos obsahuje modul na inštrumentáciu programov. Inštruované programy spravidla potrebujú poslať inštrumentačný výstup na ďalšie spracovanie. Tento vedľajší výstup má väčšinou podobu krátkych správ, ktoré nesú informáciu o vývoji stavu programu počas jeho behu. Práve *signály* dátovej zbernice sú vhodné na prenos takýchto správ. Parameter typ správy môže byť do programu priamo zakódovaný inštrumentačným nástrojom, alebo predaný za behu, napríklad cez premennú prostredia.

Detektory Widget Recognition

Modelový prípad s detektormi je asi ten, kde predávanie správ cez zbernicu asi najviac pomôže. Modul WiRec na rozpoznávanie prvkov grafického rozhrania pozostáva z sady detektorov a riadiaceho programu. Riadiaci program zadá detektorom snímku obrazovky testovaného programu a detektory hľadajú v obrázku prvky, na ktoré boli naprogramované. Ale niektoré detektory potrebujú výstup predchádzajúcich aby mohli pracovať. Napríklad modul detekcie hrán nájde v obrázku ohraničené oblasti. Modul detekcie textu, zistí že je v oblasti text a čo je tam napísané. Modul rozpoznávania tieňov zistí, že oblasť opticky vystupuje z povrchu. A posledný modul vyvodí záver, že detekovaný útvar je napríklad tlačítko „OK“. Formát týchto výsledkov detektorov zodpovedá *signálom* testos bus-u. Jednotlivé detektory nevedia, kto má o ich výsledky záujem, naopak, výsledky dávajú iba tak obecne na vedomie. Toto tiež sedí s vlastnosťami signálov. Typ správy by v tomto prípade mal niesť informáciu o type zistenia, napríklad: ohraničená oblasť, text alebo tieň. Jednotlivé detektory sa môžu prihlásiť na odber iba tých typov, ktoré dokážu spracovať a tým znížiť množstvo doručených správ. Ale problémy popísané v kapitole 1.1.4 sú tu relevantné, preto by asi bolo vhodné, použiť iné riešenie.

4 Záver

V prvej časti som analyzoval bežné komunikačné vzory. Preskúmal som možnosti použitia zdielanej zbernice na integráciu modulov. Ukázalo sa že zdielaná a distribuovaná zbernica prináša niekoľko problémov, ktoré boli analyzované.

V druhej časti boli vyzdvihnuté výhody a nevýhody vybraných existujúcich nástrojov. Zdá sa že všeobecné riešenie pre systém rozosielania neexistuje. Každý systém sa pokúša riešiť problém z inej strany a má svoje prípady použitia. Zaujala ma predovšetkým efektívnosť knižnice ZeroMQ, nad ktorou sa dajú poskladať distribuované aplikácie bez nutnosti spravovať centrálny broker.

V tretej časti som navrhol komunikačný systém pre projekt Testos. Bol navrhnutý sieťový protokol a aplikačné rozhranie. Pri návrhu rozhrania bol kladený dôraz na ochranu pred vyčerpaním pamäte.

Literatúra

Skupina TESTOS: Domovská stránka projektu Testos [online]. FIT VUT v Brně. 2019, [cit. 14.12.2018]. URL http://testos.org

PECINOVSKÝ, Rudolf, 2017: *Návrhové vzory*. Brno: Computer Press, ISBN 8025145107

HOHPE, Gregor, WOOLF Bobby, 2012: Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions. Londýn: Addison-Wesley, ISBN 0321200683

MATOUŠEK, Petr, 2014: Síťové aplikace a jejich architektura. Brno: VUTIUM, ISBN 978-80-214-3766-1

HINTJENS, Pieter, 2013: *ZeroMQ Messaging for Many Applications*. Sebastopol: O'Reilly Media

NARKHEDE, Neha, SHAPIRA, Gwen, PALINO, Todd, 2017: *Kafka: The Definitive Guide.* Sebastopol: O'Reilly Media

AIYAGARI, Sanjay et al., 2008: AMQP Advanced Message Queuing Protocol [online]. Verzia: 0-9-1. Dostupné z: https://www.rabbitmq.com/resources/specs/amqp0-9-1.pdf

KLISHIN Michael et al., 2019: *RabbitMQ documentation* [online]. Pivotal Software Inc. © 2007-2019 [cit. 20.4.2019]. Dostupné z: https://www.rabbitmq.com/documentation.html

CARLSSON, Anders, HERZBERG, Sven, LARSSON, Alexander, MCVITTIE, Simon, PENNINGTON, Havoc, ZEUTHEN, David, 2018: *D-Bus Specification* [online]. Verzia: 0.34. Dostupné z: https://dbus.freedesktop.org/doc/dbus-specification.html

VERMEULEN, Jeroen, 2013: *Introduction to D-Bus* [online]. FreeDesktop.org. [cit. 20.4.2019]. Dostupné z: https://www.freedesktop.org/wiki/IntroductionToDBus/

GOSLING, James, JOY, Bill, STEELE, Guy L., BRACHA, Gilad, BUCKLEY, Alex, 2014: *The Java Language Specification, Java SE 8 Edition.* Boston: Addison-Wesley, ISBN 9780133900699

ALLISON, Lloyd, 2007: *Tries* [online]. [cit. 20.4.2019]. Dostupné z: http://www.allisons.org/ll/AlgDS/Tree/Trie/

Obsah CD

- text/ - digitálna verzia technickej správy a jej zdrojová podoba
- testosbus/ - zdrojový text TestosBus
- eval/ - programy na ohodnotenie a vyskúšanie existujúci riešení z kapitoly 2
- Readme.txt – poznámky k obsahu CD