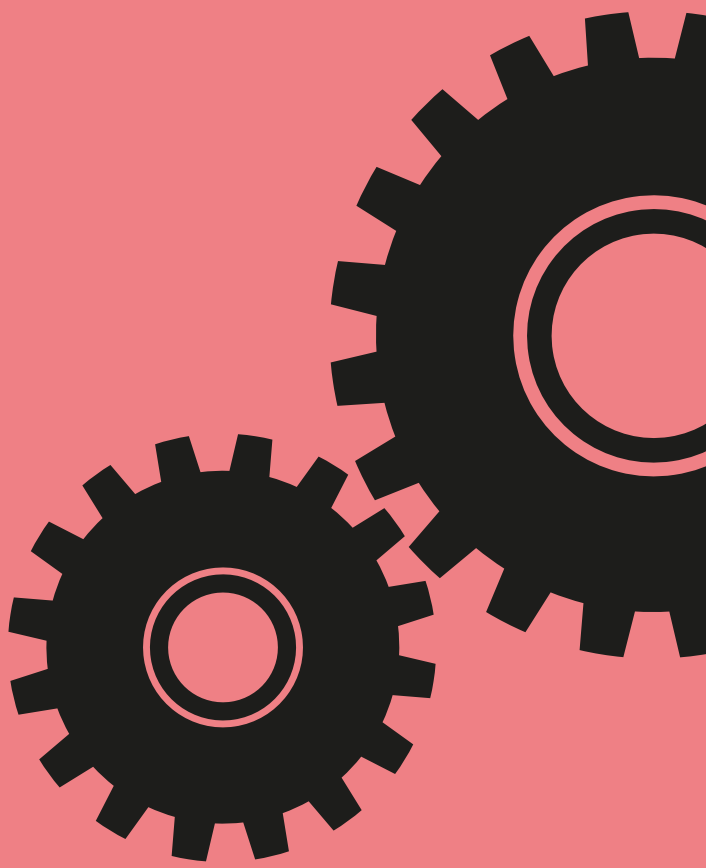
Editoval Tavish Armstrong

Výkonnost open source aplikací

Rychlost, přesnost a trocha štěstí



Edice CZ.NIC

VÝKONNOST OPEN SOURCE APLIKACÍ
Rychlost, přesnost a trocha štěstí

Editoval Tavish Armstrong

Vydavatel:
CZ.NIC, z. s. p. o.
Milešovská 5, 130 00 Praha 3
Edice CZ.NIC
www.nic.cz

1. vydání, Praha 2016
Kniha vyšla jako 13. publikace v Edici CZ.NIC.
Přeloženo z anglického originálu knihy *The Performance of Open Source Applications*.

CZ.NIC | SPRÁVCE DOMÉNY CZ

ISBN 978-80-88168-14-0

— Editoval Tavish Armstrong

Výkonnost open source aplikací

Rychlost, přesnost a trocha štěstí

— Edice CZ.NIC

Předmluva vydavatele

Vážený čtenáři,

kniha, kterou právě držíte v ruce anebo čtete na svém mobilním zařízení, se zabývá v porovnání s dosavadními tituly Edice CZ.NIC poměrně specializovaným tématem – laděním výkonnosti softwarových aplikací.

Ačkoliv rychlost procesorů i kapacita paměti v posledním čtvrtstoletí rostly fenomenálním tempem, svižně běžící aplikace nejsou ani dnes ničím samozřejmým a bezpracným. Je to tím, že musí zpracovávat stále větší a složitější data a prezentovat výsledky stále náročnějším uživatelům. Vývojáři softwaru by proto měli výkonnostním aspektům věnovat pozornost jak při návrhu aplikací, tak i při jejich testování a ladění.

Kniha *Výkonnost open source aplikací* obsahuje dvanáct případových studií, jejichž autoři popisují řadu metod a triků vedoucích ke zlepšení výkonu. Škála je opravdu široká: počínaje spekulativními optimalizacemi, díky nimž nás moderní webové prohlížeče udivují svou předvídavostí, přes sofistikované algoritmy syntaktické analýzy, až po specifika mobilních zařízení. Pro ostříleného softwarového profesionála může být poměrně překvapivá předposlední kapitola, která demonstruje, že i „nepraktické“ funkcionální jazyky mohou nabízet elegantní řešení pro zvýšení výkonu. Sekundární, ale rovněž velmi zajímavou problematikou, které se tak či onak dotýká většina kapitol, jsou postupy, jimiž lze změřit reálnou výkonnost softwarových systémů a porovnat ji s alternativními implementacemi.

Přeji vám příjemné a inspirativní čtení.

Ladislav Lhotka, CZ.NIC
České Budějovice, 22. června 2016

Úvod

(Tavish Armstrong)

Úvod

Neustále se setkáváme s názorem, že počítačový hardware je v současné době tak rychlý, že si většina vývojářů nemusí dělat starosti s jeho výkonem. Douglas Crockford dokonce odmítl napsat kapitolu pro tuto knihu, a to z následujícího důvodu:

„Pokud bych měl přispět nějakou kapitolou do knihy, týkala by se anti výkonu: většina úsilí při honu za výkonem je vynaložena marně. Nemyslím si, že právě tohle hledáte.“

Donald Knuth k tomu poznamenal už před třiceti lety:

„Měli bychom zapomenout na nízkou výkonnost, řekněme v 97 % času: zdrojem všeho zla je totiž předčasná optimalizace.“[1]

Avšak u mobilních zařízení s omezeným výkonem a omezenou pamětí, stejně jako u projektů analýzy dat, které vyžadují zpracování terabajtů dat, potřebuje stále větší počet vývojářů zrychlit svůj kód, zmenšit datové struktury a zkrátit dobu odezvy. Zatímco stovky učebnic popisují principy operačních systémů, sítí, počítačové grafiky a databází, jen několik málo (pokud vůbec nějaké) vysvětlují, jak nacházet a opravovat chybné věci ve skutečných aplikacích, které jsou jednoduše zatraceně pomalé.

Tato sbírka případových studií je naší snahou zvětšit počet takových knih. Každou kapitolu napsali skuteční vývojáři, jejichž úkolem bylo zrychlit stávající systém, nebo přímo navrhnout něco rychlého. Pokrývají mnoho různých druhů softwaru a výkonnostních cílů. Mají společné detailní chápání toho, co se kdy děje, a jak se k sobě hodí různé části velkých aplikací. Doufáme, že vám tato kniha - stejně jako kniha *The Architecture of Open Source Applications*[2] - pomůže stát se lepším vývojářem, a to díky tomu, že vás necháme nahlížet těmto odborníkům přes rameno.

– Tavish Armstrong

Spolupracovníci

Tavish Armstrong (editor): Tavish studuje softwarové inženýrství na Concordia University a doufá, že na jaře roku 2014 absolvuje závěrečné zkoušky. Jeho domovská stránka je http://tavisharmstrong.com.

1: Poznámka vydavatele: Donald Knuth v roce 1974 napsal, že předčasná optimalizace je počátkem všech problémů. Viz D.E. Knuth, „Structured Programming with go to Statements“, ACM Computing Surveys, vol. 6, pp. 261–301, 1974

2: Poznámka vydavatele: Kniha *The Architecture of Open Source Applications* je součástí série knih AOSA (www.aosabook.org).

Michael Snoyman (Warp): Michael je vedoucím softwarovým inženýrem ve firmě FP Complete. Je zakladatelem a hlavním vývojářem webového frameworku Yesod, který poskytuje prostředky pro vytváření masivních, vysoce výkonných webových aplikací. Oficiálně studuje pojistnou matematiku a dříve pracoval ve Spojených státech v oblasti pojištění automobilů a domácností, kde analyzoval velké sady dat.

Kazu Yamamoto (Warp): Kazu je vedoucím výzkumníkem institutu IIJ Innovation Institute. Na open source softwaru pracuje již zhruba 20 let. Mezi jeho produkty patří Mew, KAME, Firemacs a Mighty.

Andreas Voellmy (Warp): Andreas je doktorandem informatiky na Yale University. Andreas při svém výzkumu softwarově definovaných sítí používá Haskell a publikoval open source Haskell balíčky, jako je například Nettle OpenFlow používaný k ovládání směrovačů pomocí Haskell programů. Andreas také přispívá do projektu GHC a udržuje jeho správce vstupu a výstupu.

Ilya Grigorik (Chrome): Ilya je inženýrem pro webový výkon, vývojář a zastánce týmu Make The Web Fast ve společnosti Google, kde tráví dny a noci zrychlováním webu a řízením adaptace best practices v oblasti výkonnosti. Ilyu najdete online na jeho blogu igvita.com a na Twitteru jako @igrigorik.

Evan Martin (Ninja): Evan pracuje devět let jako programátor ve společnosti Google. V životopisu má uvedené tituly z počítačových věd a lingvistiky. Pomáhal na mnoha malých projektech svobodného softwaru a na několika větších, včetně projektu LiveJournal. Jeho webová stránka má adresu http://neugierig.org.

Bryce Howard (výkonnost mobilních zařízení): Bryce je softwarový architekt, jenž je posedlý zrychlováním věcí. Pohybuje se v oboru více než 15 let a podílel se na řadě startupů, o kterých jste nikdy neslyšeli. Aktuálně se pokouší o psaní a je autorem úvodní knihy o webových službách Amazon pro společnost O'Reilly Associates.

Kyle Huey (Memshrink): Kyle pracuje ve společnosti Mozilla Corporation na zobrazovacím modulu Gecko, na němž je založen webový prohlížeč Firefox. Před přestěhováním se do San Francisca získal bakalářský titul na matematických studiích na University of Florida. Bloguje na adrese blog.kylehuey.com.

Clint Talbert (Talos): Clint se zabývá už téměř deset let projektem Mozilla, začínal nejprve jako dobrovolník a později se stal zaměstnancem. V současnosti vede tým pro automatizaci a nástroje a má povolení automatizovat cokoli, co automatizovat lze. Vyhlásil osobní vendetu všem nečinným cyklům procesoru u jakýchkoliv automatizačních strojů. Jeho dobrodružství ve světě open source a psaní můžete sledovat na adrese clinttalbert.com.

Joel Maher (Talos): Joel má více než 15 let zkušeností s automatizací softwaru. V posledních pěti letech se ve společnosti Mozilla zabýval automatizací a nástroji pro zdokonalování mobilních

telefonů. Také převzal odpovědnost za Talos, aby rozšířil testování, zvýšil spolehlivost a vylepšil detekci regresí. V době, kdy běží jeho automatizace, chodí Joel rád ven a řeší nové výzvy v životě. Další informace o jeho dobrodružstvích na poli automatizace najdete na adrese elvis314.wordpress.com.

Audrey Tang (Ethercalc): Programátorka-samostudentka a překladatelka Audrey žijící na Tchaj-wanu v současnosti pracuje ve společnosti Socialtext na pozici „Stránka bez názvu" a zároveň ve společnosti Apple v oblasti lokalizace a uvolňování inženýrství. Předtím Audrey navrhovala a vedla projekt Pugs, první fungující implementaci jazyka Perl 6, a pracovala v komisích pro design jazyků Haskell, Perl 5 a Perl 6, v jejichž rámci mnohokrát přispěla do repositářů CPAN a Hackage. Sledujte Audrey na jejím Twitteru @audreyt.

C. Titus Brown (Khmer): Titus se zabývá evolučním modelováním, fyzickou meteorologií, vývojovou biologií, genomikou a bioinformatikou. Nyní je docentem na Michigan State University, kde rozšířil své zájmy na několik nových oblastí, včetně reprodukovatelnosti a udržitelnosti vědeckého softwaru. Je také členem nadace Python Software Foundation a bloguje na adrese http://ivory.idyll.org.

Eric McDonald (Khmer): Eric McDonald je vývojářem vědeckého softwaru se zaměřením na vysoce výkonné výpočty (HPC), což je oblast, ve které pracoval po většinu času z uplynulých 13 let. Dříve pracoval s různými fyziky a nyní pomáhá bioinformatikům. Je držitelem bakalářského titulu v počítačové vědě, matematice a fyzice. Eric je už od poloviny devadesátých let fanouškem FOOF (svobodného a open source softwaru).

Douglas C. Schmidt (DaNCE): Dr. Douglas C. Schmidt je profesorem informatiky, místopředsedou programu informatiky a inženýrství a vedoucím výzkumníkem Institutu softwarově integrovaných systémů, to vše na Vanderbilt University. Doug publikoval 10 knih a více než 500 technických článků, které se týkají široké škály softwarových témat. Během posledních dvou desetiletí vedl vývoj ACE, TAO, CIAO a CoSMIC.

Aniruddha Gokhale (DaNCE): Dr. Aniruddha S. Gokhale je docentem na Katedře elektroinženýrství a informatiky a vedoucím vědeckým výzkumníkem v Institutu softwarově integrovaných systémů (ISIS), obojí na Vanderbilt University. Na svém kontě má více než 140 technických článků a jeho současný výzkum se soustředí na vývoj novátorských řešení výzev vznikajících v oblasti cloud computing a kyberneticko-fyzikálních systémů.

William R. Otte (DaNCE): Dr. William R. Otte je vědeckým výzkumníkem v Institutu softwarově integrovaných systémů (ISIS) na Vanderbilt University. Má takřka deset let zkušeností v oblasti vývoje open source middlewaru a modelovacích nástrojů pro distribuované systémy, systémy reálného času a vestavěné systémy, spolupracoval jak s vládními, tak průmyslovými partnery, mezi něž patří agentury DARPA, NASA a společnosti Northrup Grumman a Lockheed-Martin. Dosud publikoval množství technických článků a zpráv popisujících pokroky a podílel se na vývoji otevřených standardů pro middleware komponenty.

Manik Surtani (Infinispan): Manik je hlavním inženýrem pro výzkum a vývoj v JBoss, divizi middlewaru společnosti Red Hat. Je zakladatelem projektu Infinispan a architektem platformy JBoss Data Grid. Je také vedoucím pro specifikace JSR 347 (datové gridy pro platformu Java) a zastupuje společnost Red Hat ve skupině odborníků pro JSR 107 (dočasné kešování pro jazyk Java). Jeho zájmy se týkají cloudových a distribuovaných výpočtů, velkých dat a NoSQL, autonomních systémů a vysoce dostupných systémů.

Arseny Kapoulkine (Pugixml): Arseny strávil celou svou kariéru programováním grafických a nízko-úrovňových systémů v oblasti videoher, od malých úzce specializovaných titulů až po multiplatformní trháky nejvyšší třídy, jako je například FIFA Soccer. Miluje zrychlování pomalých věcí a další zrychlování rychlých věcí. Můžete ho kontaktovat na adrese mail@zeuxcg.org nebo na jeho Twitteru @zeuxcg.

Arjan Scherpenisse (Zotonic): Arjan je jedním z hlavních architektů frameworku Zotonic a zvládá pracovat na desítkách projektů naráz, obvykle s využitím frameworku Zotonic a jazyka Erlang. Arjan překlenuje mezeru mezi back-end a front-end projekty v Erlangu. Kromě problémů, jako jsou škálovatelnost a výkonnost, se Arjan často zabývá kreativními projekty. Navíc pravidelně přednáší na různých akcích.

Marc Worrell (Zotonic): Marc je respektovaným členem komunity jazyka Erlang a byl iniciátorem projektu Zotonic. Marc tráví svůj čas konzultacemi velkých projektů v Erlangu, vývojem projektu Zotonic a také je technickým ředitelem společnosti Maximonster, kde vytvořili MaxClass a LearnStone.

Poděkování

Tato kniha by nevznikla bez pomoci Amy Brown a Grega Wilsona, kteří mě požádali, abych knihu redigoval, a přesvědčili mě, že je to vůbec možné. Rád bych poděkoval také Tonymu Arklesovi za jeho pomoc v raných fázích redigování a našim technickým korektorům:

Colinu Morrisovi,
Coreymu Chiversovi,
Gregu Wilsonovi,
Julii Evans,
Kamalovi Marhubimu,
Kim Moir,
Laurie MacDougall Sookraj,
Loganu Smythovi,
Monice Dinculescu,
Nikitovi Pchelinovi,
Natalie Black,
Pierre-Antoinu Lafayetteovi.

Díky patří i malé armádě redaktorů a pomocníků, kterých bylo zapotřebí k tomu, aby tato kniha vyšla ještě v tomto desetiletí:

Adamu Fletcherovi,
Amy Brown,
Danielle Pham,
Eriku Habbingovi,
Jeffu Schwabovi,
Jessice McKellar,
Michaelu Bakerovi,
Natalie Black,
Alexandře Phillips,
Peteru Roodovi.

Amy Brown, Bruno Kinoshita a Danielle Pham si zasluhují zvláštní poděkování za pomoc při přípravě, grafickém zpracování a sazbě knihy.[3]

Redigování knihy je obtížný úkol, avšak je snazší, pokud máte přátele, kteří vás podporují. Natalie Black, Julia Evans a Kamal Marhubi byli po celou dobu trpěliví a zapálení.

Příspěvky

Na tvorbě této knihy tvrdě pracovaly desítky dobrovolníků, avšak stále toho ještě zbývá hodně udělat. Pokud chcete pomoci, můžete se zapojit hlášením chyb, překladem obsahu do jiných jazyků nebo popisem dalších open source systémů. Pokud se chcete zapojit, prosím obraťte se na nás na adrese aosa@aosabook.org.[4]

3, 4: Poznámka vydavatele: Vztahuje se k originálu knihy *The Performance of Open Source Applications*.

Obsah

1 Vysoká síťová výkonnost prohlížeče Chrome

(Ilya Grigorik)

1 Vysoká síťová výkonnost prohlížeče Chrome

1.1 Historie a hlavní principy prohlížeče Google Chrome

Prohlížeč Google Chrome byl poprvé vydán v druhé polovině roku 2008 jako beta verze pro platformu Windows. Kód vytvořený společností Google, který prohlížeč Chrome pohání, byl také zpřístupněn s liberální licencí BSD - a je znám také jako projekt Chrome. Pro mnohé pozorovatele byl tento vývoj událostí překvapivý: máme čekat návrat tzv. válek prohlížečů? Bude prohlížeč Google opravdu o tolik lepší?

„Byl tak dobrý, že jsem byl v podstatě nucen změnit názor…“
– Eric Schmidt o svém počátečním odporu k nápadu začít vyvíjet prohlížeč Google Chrome.

Časem se ukázalo, že opravdu je. Dnes je Chrome jedním z nejrozšířenějších prohlížečů na webu (podle nástroje StatCounter má více než 35% podíl na trhu) a je nyní k dispozici pro Windows, Linux, OS X, Chrome OS a také pro platformy Android a iOS. Uživatelé si oblíbili jeho funkce a vlastnosti, a mnohé z inovací prohlížeče Chrome si našly cestu i do ostatních populárních prohlížečů.

Původní komiks o 38 stranách vysvětlující nápady a inovace prohlížeče Google Chrome přináší skvělý přehled procesu myšlení a navrhování, který se za populárním prohlížečem skrývá. To byl však teprve jen začátek. Klíčové principy, kterými byl vznik prohlížeče motivován, jsou i nadále hlavními principy jeho neustálého vylepšování.

Rychlost: Vytvořit ten **nejrychlejší** prohlížeč.
Bezpečnost: Poskytovat uživateli **nejbezpečnější** prostředí.
Stabilita: Poskytovat **odolnou a stabilní** platformu pro webové aplikace.
Jednoduchost: Vytvořit sofistikovanou technologii skrytou za **snadno použitelným uživatelským rozhraním**.

Jak tým zjistil, mnoho stránek, které v dnešní době používáme, nejsou jen webové stránky, jsou to aplikace. A stále ambiciózněji aplikace vyžadují rychlost, bezpečnost a stabilitu. Každý z těchto požadavků by si zasloužil vlastní kapitolu, ale protože naším tématem je výkonnost, zaměříme se primárně na rychlost.

1.2 Mnoho aspektů výkonnosti

Moderní prohlížeč je, podobně jako operační systém, platformou a stejně tak je navržen i Google Chrome. Před prohlížečem Google Chrome byly všechny hlavní prohlížeče postaveny jako monolitické aplikace v rámci jediného procesu. Všechny otevřené stránky sdílely stejný adresní prostor a soupeřily o stejné zdroje. Chyba na kterékoliv stránce nebo v prohlížeči přinášela riziko pokažení celkového uživatelského dojmu.

Chrome používá model více procesů, který poskytuje každé záložce vlastní paměť, jakož i oddělený adresní prostor procesu a jeho bezpečnostní kontext. V dnešním světě vícejádrových procesorů se jasně ukazuje, že schopnost izolace na úrovni procesů a samostatné ochrany každé otevřené záložky od ostatních problémových stránek přinesla Chrome významný náskok před konkurencí. Je potřeba poznamenat, že ve skutečnosti většina ostatních prohlížečů se k těmto principům už také hlásí, anebo je v procesu migrace na podobnou architekturu.

Spuštění a běh webové aplikace v procesu prohlížeče primárně vyžaduje získání zdrojů, sestavení a zobrazení stránky, a vykonání JavaScriptu. Zobrazování a vykonávání skriptu se řídí jednovláknovým prokládaným modelem výpočtu – není možné provádět souběžné úpravy výsledného objektového modelu dokumentu (DOM). Zčásti je to způsobeno faktem, že pro JavaScript až do nedávna neexistovalo standardizované API pro práci s vlákny. Proto je optimalizace vzájemného prokládání vykreslování a vykonávání JavaScropti extrémně důležitá, a to jak pro webové vývojáře vytvářející aplikace, tak pro vývojáře pracující na prohlížeči.

Prohlížeč Chrome využívá k vykreslování stránek Blink, což je rychlé jádro prohlížeče vykreslující webové stránky, které je open source a splňuje všechny potřebné normy. Pro jazyk JavaScript používá prohlížeč Chrome svůj vlastní, velmi optimalizovaný runtime V8 JavaScriptu, který byl také vydán jako samostatný open source projekt a našel si cestu do mnoha dalších populárních projektů – např. do runtime systému Node.js. Optimalizace vykonávání JavaScriptu skriptovacím jádrem V8 anebo parsovacích a vykreslovacích procedur Blinku by ovšem byla málo platná, pokud by se prohlížeč zablokoval čekáním na uvolnění síťových zdrojů.

Schopnost prohlížeče optimalizovat pořadí a prioritu přístupu k jednotlivým URL (jednotná adresa zdroje v Internetu) je jedním z nejdůležitějších faktorů celkového dojmu uživatele. Možná si to neuvědomujete, ale síťová architektura prohlížeče Chrome je, a to v podstatě doslova, den ode dne chytřejší a pokouší se snižovat důsledky zpoždění každého zdroje: předpovídá pravděpodobné DNS dotazy (převod textových, doménových jmen zdrojů v Internetu na číselné IP adresy), zapamatovává si navštívenou topologii webu, předběžně se připojuje k serverům a dělá mnoho dalšího. Navenek se prezentuje jako jednoduchý mechanismus získávání zdrojů, ale uvnitř se jedná o propracovaný a fascinující příklad, jak optimalizovat výkonnost webu a maximálně uspokojit uživatele.

Pojďme tedy na to.

1.3 Jak vypadá moderní webová aplikace?

Než se dostaneme k zásadním podrobnostem toho, jak optimalizovat naši interakci se sítí, pomůže nám, pokud pochopíme trendy a kontext problému, před kterým stojíme. Jinými slovy, *jak vypadá moderní webová stránka či aplikace?*

Projekt HTTP Archive sleduje, jak se buduje web, a může nám při hledání odpovědi na tuto otázku pomoci. Namísto prohledání webu kvůli obsahu pravidelně prochází nejpopulárnější stránky a pro každou z nich zaznamenává a shromažďuje analýzy počtu použitých zdrojů, typy obsahu, hlavičky a další metadata. Statistiky k lednu 2013 vás možná překvapí. Průměrná stránka mezi 300 000 destinacemi na webu má následující charakteristiky:

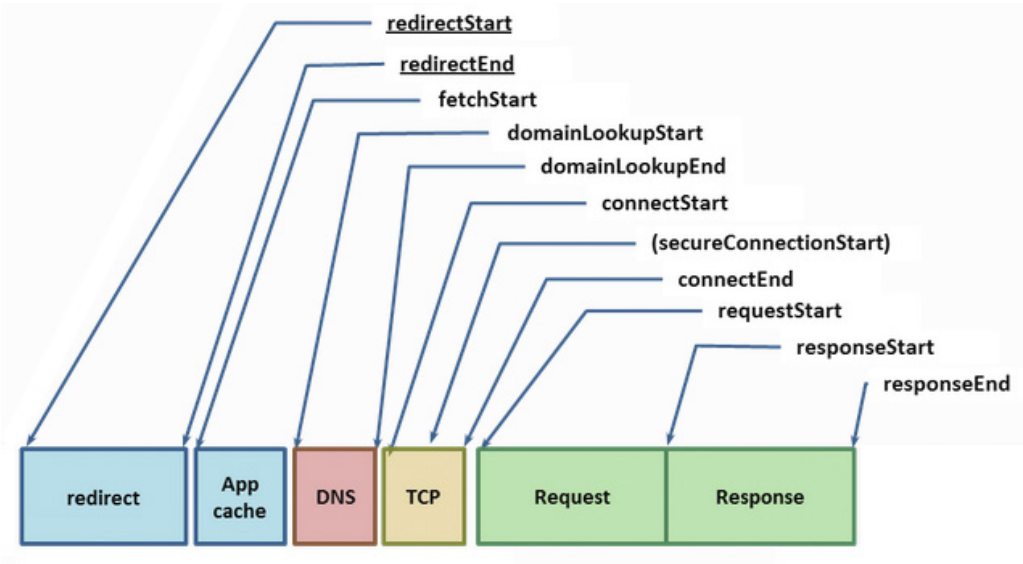
- její velikost je 1 280 KB,
- skládá se z 88 zdrojů,
- připojuje se k více než 15 různým serverům.

Pojďme si to přepočítat. Zhruba 1 MB průměrné velikosti se skládá z 88 zdrojů, jako jsou obrázky, JavaScript a CSS, a tyto zdroje dodává 15 různých serverů, jak vlastních, tak i třetích stran. Navíc každé z těchto čísel v průběhu několika uplynulých let neustále narůstá a nic nenaznačuje tomu, že by se tento růst měl zastavit. Budujeme stále větší a ambicióznější webové aplikace.

Pokud na číselné údaje z HTTP Archive použijeme základní matematiku, zjistíme, že průměrný zdroj má velikost zhruba 15 KB (1280 KB/88 zdrojů), což znamená, že většina síťových přenosů je v prohlížeči krátká a nárazová. Tento samotný fakt přináší řadu komplikací, protože původní HTTP protokol navazoval jedno spojení pro jeden zdroj, což se časem stalo úzkým hrdlem datového přenosu, jak se webové stránky začaly skládat z čím dál tím většího množství zdrojů. Podívejme se tomuto tématu pod kůži a rozeberme si podrobně jeden z těchto síťových požadavků.

1.4 Životní cyklus požadavku na zdroje na síti

Specifikace časování navigace konsorcia W3C poskytuje prohlížeči aplikační rozhraní (API) a přístup k datům o časování a výkonnosti během životního cyklu každého požadavku v prohlížeči. Podívejme se podrobně na komponenty, z nichž každá přispívá důležitým dílem k dosažení optimálního dojmu uživatele:



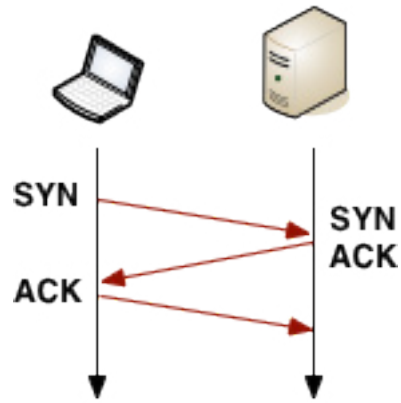
Obrázek 1.1: Časování navigace

Po zadání URL webového zdroje začne prohlížeč tím, že zkontroluje svou lokální a aplikační vyrovnávací mezipaměť (keš). Pokud jste tento zdroj stáhli už dříve a máte k dispozici příslušné údaje o jeho platnosti (doba expirace, politiku uložení v mezipaměti prohlížeče, atd.), lze k vyřízení požadavku použít lokální kopii z mezipaměti. V opačném případě, pokud musíme opětovně ověřit, zda zdroj vypršel, anebo jsme ho jednoduše dosud neviděli, je nutné vyslat nákladný síťový požadavek.

Po zadání jména hostitele a cesty ke zdroji prohlížeč Chrome nejprve prověří stávající otevřená spojení, která může použít opětovně – sokety (programátorská abstrakce konce síťového spojení mezi dvěma počítači) se sdružují podle parametrů {schéma URL (protokol), server, komunikační port protokolu TCP/IP}. V případě, že jste nakonfigurovali proxy server (server mezi lokálním a vzdáleným počítačem, přes který prochází komunikace mezi nimi) nebo specifikovali skript

automatické konfigurace proxy (PAC), prohlížeč Chrome zkontroluje připojení prostřednictvím příslušného proxy serveru. Skripty PAC umožňují použít různé proxy na základě adresy obsažené v URL nebo jinak specifikovaných pravidel, přičemž každý z nich může mít svou vlastní sadu soketů. Pokud není splněna ani jedna z výše uvedených podmínek, musí požadavek začít převedením doménového jména serveru na IP adresu protokolem DNS.

Pokud máme štěstí, záznam o převodu doménového jména na IP adresu je už uložen v mezipaměti DNS. V takovém případě je odezva obvykle dána jediným systémovým voláním. Pokud tam uložen není, pak je nutné před jakoukoli další činností vyslat DNS dotaz z lokálního počítače na DNS server. Čas, který zabere vyhledání v DNS, se může lišit podle vašeho poskytovatele internetového připojení, popularity stránky a pravděpodobnosti toho, zda bude jméno serveru nalezeno v DNS mezipaměti některého z postupně dotazovaných DNS serverů, stejně jako i podle doby odezvy autoritativních serverů dané domény. Jinými slovy, roli hraje velké množství proměnných, takže vyhledání DNS mnohdy zabere několik stovek milisekund. Ach jo.



Obrázek 1.2: Trojcestné navázaní spojení

Jakmile získá IP adresu, může prohlížeč Chrome otevřít nové spojení protokolem TCP/IP k cíli, což znamená, že musíme provést tzv. „trojcestné navázaní spojení“: pakety IP protokolu SYN, SYN-ACK a ACK v uvedeném pořadí. Tato výměna paketů zvýší každému nově vytvářenému TCP/IP spojení zpoždění o celou jednu obousměrnou cestu – bez možnosti využít jakoukoliv zkratku. Podle vzdálenosti mezi klientem a serverem a také podle vybrané cesty směrování můžeme nabrat desítky až stovky, občas dokonce i tisíce, milisekund zpoždění. A tato práce je vynaložena, která dobou svého trvání zvyšuje celkové zpoždění získání zdroje, ještě předtím, než se jediný bajt aplikačních dat rozběhne do sítě.

Pokud se připojujeme k zabezpečené destinaci (HTTPS), musí po navázání TCP/IP spojení dojít ještě k navázání spojení SSL - SSL šifruje a dešifruje data http protokolu, která v zašifrované podobě přenáší protokol TCP/IP. Tím se může přidat další latence v délce dvou obousměrných cest mezi klientem a serverem. Pokud je relace SSL uložena ve vyrovnávací paměti, můžeme z toho „vyváznout" pouze s jednou dodatečnou obousměrnou cestou.

Konečně je prohlížeč Chrome schopen vyslat HTTP požadavek (requestStart na Obrázku 1.1). Jakmile je požadavek přijat, server jej může zpracovat a poté proudově zaslat data odezvy zpět do klienta. Tím je vyvolána minimálně jedna obousměrná cesta, plus doba zpracovávání na serveru. Poté máme konečně hotovo – pokud však není skutečná odezva přesměrováním na novou adresu požadovaného zdroje: v takovém případě musíme celý cyklus opakovat ještě jednou. Pokud máte na vašich stránkách nějaká bezdůvodná přesměrovávání, budete je teď zřejmě chtít přehodnotit.

Sčítali jste všechna ta zpoždění? Abychom si problém jasně znázornili, předpokládejme, že dojde k nejhoršímu možnému scénáři u typického širokopásmového připojení: v lokální DNS mezipaměti záznam nenajdeme, takže následuje síťový dotaz DNS protokolem (50 ms), navázání spojení TCP/IP, vyjednávání SSL a relativně rychlá doba odezvy serveru (100 ms) s obousměrným zpožděním (RTT – round trip time) 80 ms (průměrná obousměrná cesta napříč kontinentálními USA):

- 50 ms na DNS
- 80 ms na TCP/IP navázání spojení (jedno RTT)
- 160 ms na SSL navázání spojení (dvě RTT)
- 40 ms pro požadavek na server
- 100 ms pro zpracování na serveru
- 40 ms na odezvu ze serveru

Celkem 470 milisekund na jediný požadavek, což znamená, že 80 % zpoždění padá na režii síťového spojení a jenom 20 % tvoří skutečná doba zpracovávání požadavku na serveru – s tím musíme něco udělat. A to těch 470 milisekund může být ještě optimistický odhad:

- Pokud se odezva serveru nevejde do velikosti 4–15 KB (tzv. congestion window protokolu TCP/IP), zvýší se komunikační zpoždění o jednu nebo dvě další obousměrné cesty.[1]

- Zpoždění způsobené SSL může být ještě větší, pokud potřebujeme získat chybějící certifikát nebo provést kontrolu stavu online certifikátu (OCSP), přičemž obojí bude vyžadovat úplně nové spojení TCP, které může přidat stovky nebo dokonce tisíce milisekund dalšího zpoždění.

1: Kapitola 10 popisuje problém detailněji.

1.5 Co znamená „dostatečně rychlý“?

Síťová režie DNS, navázání spojení a obousměrná zpoždění jsou v našem dřívějším příkladu hlavními faktory celkové doby – doba odezvy serveru má na svědomí pouze 20 % celkového zpoždění. Když to ale vezmeme kolem a kolem, mají tato zpoždění nějaký faktický význam? Pokud tuto knihu čtete, tak již pravděpodobně znáte odpověď: ano, mají, a velký.

Dosavadní výzkum dojmu uživatelů vytváří konzistentní obraz toho, jak jako uživatelé vnímáme délku odezvy jakékoli aplikace, ať už offline nebo online:

Zpoždění	Reakce uživatele
0–100 ms	Okamžik
100–300 ms	Malé postřehnutelné zpoždění
300–1000 ms	Počítač pracuje
1 s a více	Přepnutí mentálního kontextu
10 s a více	Vrátím se později…

Tabulka 1.1: Vnímání latence uživatelem

Z tabulky 1.1 také jasně plyne, proč se komunita zabývající se webovou výkonností drží následujícího nepsaného pravidla: chcete-li udržet zájem uživatele, vykreslete své stránky, nebo alespoň poskytněte vizuální zpětnou vazbu za dobu kratší než 250 ms. V tomto případě se nejedná o samoúčelnou rychlost. Studie ve společnostech Google, Amazon, Microsoft i na tisícovkách dalších stránek prokázaly, že delší latence má přímý vliv na to, jak si bude vaše stránka stát: rychlejší stránky mají více zobrazení, větší zájem uživatelů a vyšší konverzní poměr.

Takže je to jasné: náš limit na zpoždění je 250 ms, a přitom, jak jsme viděli ve výše uvedeném příkladu, kombinace DNS vyhledávání, TCP/IP a SSL navázání spojení a doby přenosu zdroje přidá až 370 ms. Překračujeme limit o 50 %, a to jsme stále ještě nezapočítali dobu zpracovávání na serveru!

Zpoždění způsobená DNS, TCP/IP a SSL jsou ovšem mimo zorné pole většiny uživatelů a dokonce i vývojářů webu. Vznikají na takových úrovních sítě, na které sestoupila, nebo o kterých přemýšlí jen hrstka z nás. Přesto je každý z těchto kroků důležitým faktorem celkového dojmu uživatele, protože každý síťový požadavek navíc může přidávat desítky nebo stovky milisekund latence. To je důvod, proč je síťová architektura prohlížeče Chrome mnohem, mnohem více než pouze jednoduchý manipulační program pro sokety.

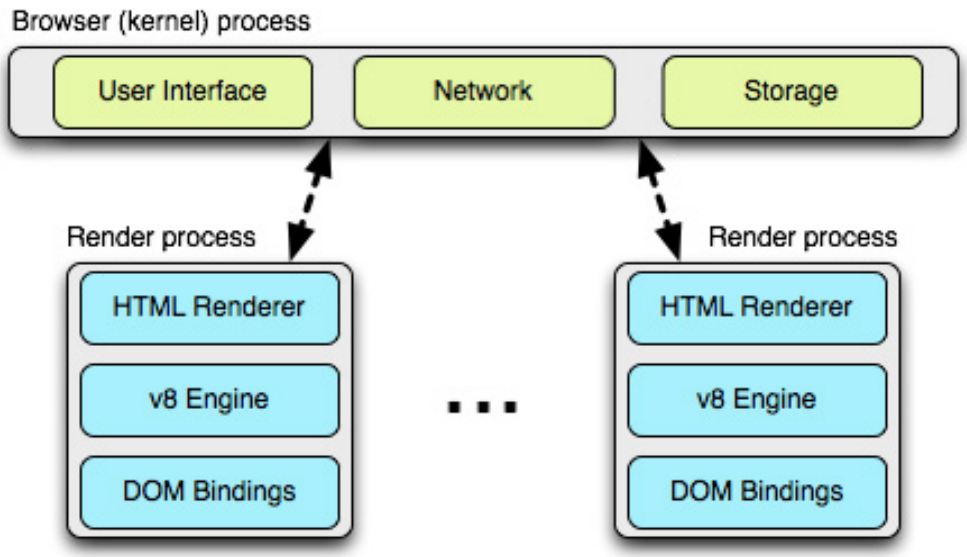
Teď, když jsme identifikovali problém, pojďme se podívat podrobněji na implementace.

1.6 Náhled na síťový stack prohlížeče Chrome

Architektura s více procesy

Prohlížeč Chrome používá architekturu s více procesy, což má velký vliv pro řešení každého síťového požadavku v rámci prohlížeče. Uvnitř prohlížeče Chrome se ve skutečnosti skrývá podpora čtyř různých modelů, které určují způsob, jakým se příslušný proces vytvoří.

Ve výchozím nastavení využívá prohlížeč Chrome na desktopech model jednoho procesu pro každou stránku. Tento model od sebe navzájem izoluje různé stránky, ale seskupuje všechny instance téže stránky do jednoho procesu. Pro jednoduchost ale předpokládejme, že máme pro každou otevřenou záložku samostatný proces. Z hlediska síťové výkonnosti nejsou rozdíly příliš podstatné, pro pochopení je ale model jednoho procesu na záložku mnohem snazší.



Obrázek 1.3: Architektura s více procesy

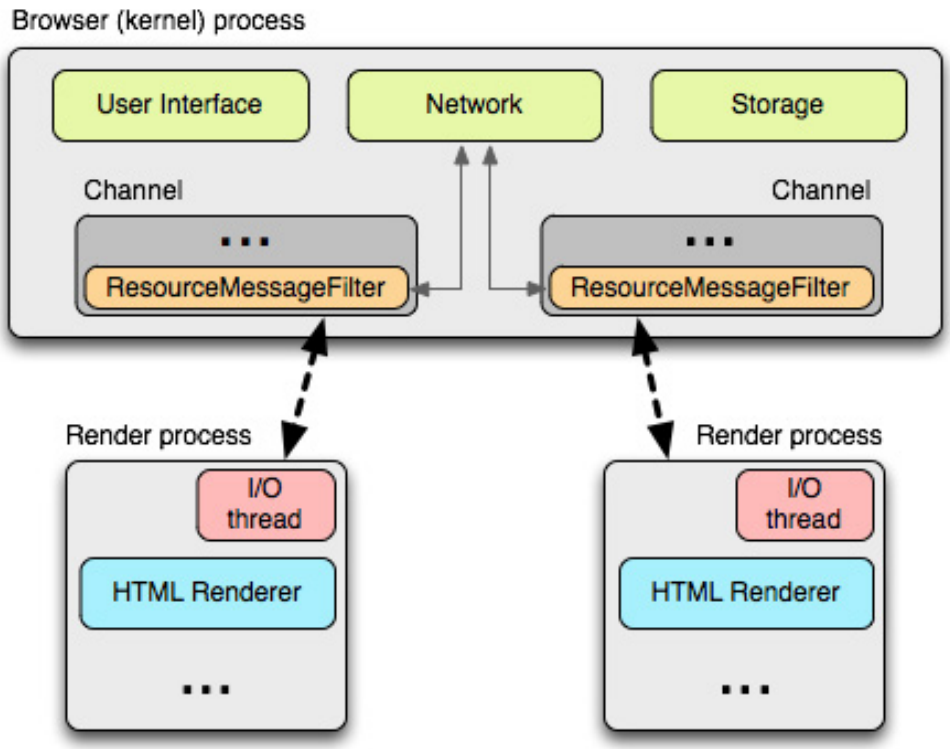
Architektura vyhrazuje každé záložce jeden vykreslovací proces. Ten obsahuje instance vykreslovacího jádra prohlížeče Blink a jádro V8 JavaScriptu, společně s vazebným kódem, který propojuje tyto a několik málo dalších komponent[2].

2: Pokud vás toto téma zaujalo, na wiki Chromium najdete výborný úvod k instalaci: http://www.chromium.org/developers/design-documents/multi-process-architecture.

Každý z vykreslovacích procesů je vykonáván v odděleném procesu, který má omezený přístup k počítači uživatele – a to včetně sítě. Aby získal přístup k těmto prostředkům, komunikuje každý vykreslovací proces s hlavním procesem prohlížeče (zvaným kernel), který je schopen každému vykreslovači vnutit bezpečnostní a přístupová pravidla.

Meziprocesová komunikace a načítání zdrojů pro více procesů

Veškerá komunikace mezi vykreslovačem a procesem kernelu se v prohlížeči Chrome provádí prostřednictvím meziprocesové komunikace (IPC). Na systémech Linux a OS X se používá funkce socketpair(). Každá zpráva z vykreslovače se postupně předá se vyhrazenému vstupně-výstupnímu vláknu, které ho pošle procesu hlavního prohlížeče. Na příjmu poskytuje proces kernelu filtrační rozhraní, které umožňuje prohlížeči Chrome zachycovat požadavky IPC na zdroje (viz ResourceMessageFilter), které má řešit síťová architektura.



Obrázek 1.4: Meziprocesová komunikace

Jednou z výhod této architektury je, že všechny požadavky na zdroje jsou celkově řešeny ve V/V vláknech a jak UI generovaná činnost, tak síťové události spolu navzájem neinterferují. Filtrování prostředků probíhá na V/V vlákně v procesu prohlížeče, zachycuje zprávy požadavků na zdroje a přeposílá je do singletonu ResourceDispatcherHost3 v procesu prohlížeče.

Unikátní rozhraní umožňuje prohlížeči nejen řídit přístup každého vykreslovače k síti, ale také účinné a konzistentní sdílení zdrojů, jako například:

- **Zásobník soketů a limity připojení:** prohlížeč je schopen omezit počet otevřených soketů na profil (256), proxy (32) a objekty {schéma, server, port} (6). Všimněte si, že díky tomu je možné zřídit až šest připojení HTTP a šest připojení HTTPS ke stejným objektům {server, port}.

- **Opětovné použití soketu:** perzistentní spojení TCP/IP jsou zachována v zásobníku soketů ještě určitou dobu po obsloužení požadavku, což umožňuje opětovné použití připojení a ušetření režie pro nastavení DNS, TCP/IP a SSL (pokud je vyžadováno), jež je spojená s každým novým připojením.

- **Pozdní vazba soketu:** požadavky jsou asociovány s navázaým spojením TCP/IP pouze tehdy, je-li soket připraven vyslat požadavek aplikace. To umožňuje lepší prioritizaci požadavků (např. příchod požadavku s vyšší prioritou ve chvíli, kdy se soket připojoval), lepší průchodnost (např. opětovné použití ještě „teplého“ TCP/IP spojení v případech, kdy se stávající soket uvolní během otevírání nového spojení), i obecně použitelný mechanismus k předběžnému připojení TCP/IP a řadu dalších optimalizací.

- **Konzistentní stav relace:** autentizace, cookies a data uložená ve vyrovnávací mezipaměti jsou sdílena mezi všemi vykreslovacími procesy.

- **Globální optimalizace prostředků a sítě:** prohlížeč je schopen rozhodovat se s ohledem na všechny vykreslovací procesy a nevyřízené požadavky. Příkladem může být přiřazení síťové priority požadavkům vyvolaným záložkou v popředí.

- **Prediktivní optimalizace:** díky sledování veškerého provozu v síti je prohlížeč Chrome schopen budovat a zpřesňovat prediktivní modely za účelem zlepšení výkonnosti.

Co se týká vykreslovacího procesu, tak jednoduše posílá požadavek s unikátním ID na zdroje do procesu prohlížeče přes IPC. Proces kernelu prohlížeče se pak postará o vše ostatní.

Načítaní zdrojů mezi platformami

Jedním z hlavních požadavků při implementaci síťového stacku prohlížeče Chrome byla přenositelnost mezi mnoha různými platformami: Linux, Windows, OS X, Chrome OS, Android a iOS. Síťová architektura je proto implementována z větší části formou jednovláknové (vyrovnávací mezipaměť a proxy mají vlastní vlákna) meziplatformní knihovny, která umožňuje prohlížeči Chrome opětovně využívat shodnou infrastrukturu a poskytovat stejné optimalizace výkonnosti. Tato architektura také poskytuje lepší možnosti pro optimalizaci napříč všemi platformami.

Komponenta	Popis
net/android	Vazby na runtime systému Android
net/base	Běžné síťové služby, jako je například vyhledání hostitele v DNS,cookies, detekce změny sítě a správa certifikátů SSL
net/cookies	Implementace ukládání, správy a načítání HTTP cookies
net/disk_cache	Implementace diskové a paměťové keše pro webové zdroje
net/dns	Implementace asynchronního DNS resolveru
net/http	Implementace protokolu HTTP
net/proxy	Konfigurace proxy (SOCKS a HTTP), rozpoznávání, načítání skriptů atd.
net/socket	Platformě nezávislé implementace TCP soketů, SSL proudů a zásobníků soketů
net/spdy	Implementace protokolu SPDY
net/url_request	Implementace URLRequest, URLRequestContext a URLRequestJob
net/websockets	Implementace protokolu WebSockets

Tabulka 1.2: Komponenty prohlížeče Chrome

Veškerý síťový kód je samozřejmě open source a naleznete jej v podadresáři src/net. Nebudeme probírat podrobně každou komponentu, uspořádání samotného kódu ale napovídá mnohé o jeho možnostech a struktuře. Několik příkladů je uvedeno v Tabulce 1.2.

Pro všechny zájemce představuje kód jednotlivých komponent skvělé počtení – je dobře zdokumentován a ke každé komponentě najdete spoustu unit testů.

Architektura a výkonnost na mobilních platformách

V současné době pozorujeme exponenciální nárůst používání mobilních prohlížečů. I podle skromných odhadů předčí už v blízké budoucnosti prohlížení na stolních počítačích. Pro tým prohlížeče Chrome jsou proto pochopitelně nejvyšší prioritou optimalizace zaměřené na uživatele mobilních zařízení . Na začátku roku 2012 byl spuštěn prohlížeč Chrome pro systém Android a o pár měsíců později jej následoval prohlížeč Chrome pro systém iOS.

První věc, které si na mobilní verzi prohlížeče Chrome všimnete, je, že se jednoduše nejedná o přímou adaptaci prohlížeče ze stolního počítače – ta by nebyla pro uživatele ideální. Už ze své přirozené povahy má mobilní prostředí jednak mnohem omezenější zdroje, a jednak zásadně odlišné provozní parametry:

- Uživatelé stolních počítačů je ovládají pomocí myši, mohou mít překrývající se okna, mají velkou obrazovku a většinou nejsou omezeni napájením. Obvykle mají také mnohem stabilnější připojení k síti a přístup k mnohem větším diskovým a paměťovým prostorům.

- Mobilní zařízení ovládají jejich uživatelé pomocí dotyků a gest, mají mnohem menší obrazovku, omezenou kapacitu baterií i příkon. Často též využívají dočasná připojení a mají omezené diskové a paměťové prostory.

Navíc neexistuje nic jako „typické mobilní zařízení“. Existuje široká škála zařízení s různými hardwarovými možnostmi, a pokud chce prohlížeč Chrome podávat ten nejlepší výkon, musí se přizpůsobit . Naštěstí přesně toto umožňují prohlížeči Chrome různé modely.

Na zařízeních se systémem Android využívá prohlížeč Chrome shodnou architekturu s více procesy jako verze pro stolní počítače – nachází se zde proces prohlížeče a jeden nebo více vykreslovacích procesů. Z důvodu paměťového omezení mobilních zařízení je jedinou odlišností to, že prohlížeč Chrome není schopen spouštět vyhrazený vykreslovací proces pro každou otevřenou záložku. Místo toho se na základě dostupné paměti a dalších omezení zařízení určí optimální počet vykreslovacích procesů, a ty se pak sdílejí mezi více záložkami.

V případech, kdy jsou k dispozici pouze minimální zdroje, nebo kdy nemůže prohlížeč Chrome spustit více procesů, může se přepnout na model zpracovávající více vláken v jednom procesu. Vlastně přesně takto se chová na zařízeních se systémem iOS – spouští jediný proces o více vláknech. Důvodem je odlišná politika izolace a spouštění procesů na této platformě.

A co síťová výkonnost? Zaprvé, prohlížeč Chrome používá u systémů Android a iOS síťovou architekturu shodnou se všemi ostatními verzemi. Tím jsou umožněny tytéž síťové optimalizace napříč všemi platformami, což dává prohlížeči Chrome značnou výhodu z hlediska výkonnosti. Některé faktory se ovšem liší a jsou často upravovány na základě možností zařízení a používané sítě. Příkladem může být třeba priorita spekulativních technik optimalizace, časový limit soketů a řídící logika, velikost vyrovnávací mezipaměti a další.

Například z důvodu šetření baterie může mobilní prohlížeč Chrome přistoupit k pomalému zavírání nevyužívaných soketů – sokety se zavřou jen při otevírání nových, aby se minimalizovalo využití rádiového přenosu. Stejně tak je povoleno předběžné vykreslování (viz níže), které vyžaduje značné síťové zdroje a výkon procesoru, pouze ve chvílích, kdy je uživatel připojen k Wi-Fi.

Optimalizace uživatelského dojmu mobilního prohlížení je pro vývojový tým prohlížeče Chrome jednou z položek s nejvyšší prioritou a můžeme očekávat, že v nadcházejících měsících a letech se dočkáme mnoha nových vylepšení. Ve skutečnosti by si toto téma zasloužilo vlastní samostatnou kapitolu – možná v dalším dílu ze série knih POSA.

Spekulativní optimalizace pomocí objektu Predictor prohlížeče Chrome

Prohlížeč Chrome se zrychluje tím, jak jej používáte. Toho je dosaženo za pomoci singletového objektu Predictor, který je vytvořen v rámci procesu kernelu prohlížeče. Prediktor sleduje typické způsoby využití sítě, učí se z nich, a předvídá pravděpodobné příští činnosti uživatele. Objekt Predictor zpracovává například tyto podněty:

- Kurzor vznášející se nad odkazem je dobrým ukazatelem pravděpodobné nadcházející navigační události, kterou může prohlížeč Chrome urychlit vysláním spekulativního dotazu DNS na jméno hostitele, a případně může také zahájit navazování TCP/IP spojení. V okamžiku, kdy uživatel klikne na odkaz, což průměrně zabere asi 200 ms, máme dobrou šanci, že jsme již dokončili první kroky k získání zdroje, které vyažďují použití protokolůDNS a TCP/IP. To nám umožňuje eliminovat stovky milisekund dalšího zpoždění potřebné pro tuto navigační událost.

- Zadávání textu do panelu Omnibox (URL) generuje vysoce pravděpodobné návrhy adres zdrojů, pro kterém může být zahájen DNS převod, předběžně navázáno TCP/IP spojení, nebo i případné vykreslení stránky na skryté záložce.

- Každý z nás má seznam oblíbených stránek, které navštěvujeme denně. Prohlížeč Chrome se dokáže naučit spekulativně rozpoznávat dílčí zdroje na těchto stránkách, případně je i předběžně načítat, aby zrychlil prohlížení.

Prohlížeč Chrome se učí topologii webu i vaše vlastní způsoby používání prohlížeče. V příznivém případě dokáže eliminovat stovky milisekund zpoždění z každé navigace a dostat uživatele blíže ke Svatému grálu jménem „okamžité načtení stránky“. K dosažení tohoto cíle využívá prohlížeč Chrome čtyři základní techniky optimalizace uvedené v Tabulce 1.3.

Technika	Popis
Předběžné rozpoznání DNS	Převádí doménová jména serverů předem, aby se redukovalo zpoždění DNS
Předběžné připojení TCP/IP	Připojí se k cílovému serveru předem, aby se vyhnul zpoždění spojenému s navázáním spojení TCP/IP
Předběžné načtení zdrojů	Načte kritické zdroje na stránce předem, aby zrychlil vykreslení stránky
Předběžné vykreslení stránky	Načte celou stránku se všemi zdroji předem, aby po spuštění uživatelem umožnil okamžitou navigaci

Tabulka 1.3: Techniky optimalizace sítí používané prohlížečem Chrome

Každé rozhodnutí vyvolat jednu nebo více těchto technik je optimalizováno vůči velkému množství omezení. Koneckonců, každá z nich je spekulativní optimalizací, což znamená, že pokud bude provedena špatně, může spustit zbytečnou akci a provoz v síti, nebo mít dokonce negativní účinek na dobu načítání při skutečné navigaci spuštěné uživatelem.

Jak prohlížeč Chrome tento problém řeší? Objekt Predictor využívá tolik podnětů, kolik jen může, a to včetně činností prováděných uživatelem, dat o historii prohlížení i informací z vykreslovače a samotného síťového stacku.

Podobně jako objekt ResourceDispatcherHost, který je odpovědný za koordinaci veškeré síťové činnosti v rámci prohlížeče Chrome, vytváří objekt Predictor v rámci prohlížeče Chrome řadu filtrů, jimiž se detekují činností vyvolané uživatelem nebo sítí:

- filtr kanálu IPC sledující signály z vykreslovacích procesů,
- ke každému požadavku se přidá objekt ConnectInterceptor tak, aby mohl sledovat schémata provozu a zaznamenávat metriky úspěšnosti pro každý požadavek.

Praktický příklad: Vykreslovací proces může poslat procesu prohlížeče zprávu obsahující kteroukoli z následujících pomocných informací, které jsou přehledně definovány v objektu ResolutionMotivation (url_info.h[3]):

```
enum ResolutionMotivation {
     MOUSE_OVER_MOTIVATED,        // Přejetí kurzorem iniciované uživatelem.
     OMNIBOX_MOTIVATED,           // Převod byl navržen- panelem Omnibox.
     STARTUP_LIST_MOTIVATED,      // Tento zdroj je v první desítce seznamu spouštěných.
     EARLY_LOAD_MOTIVATED,        // V některých případech používáme předběžné načtení
                                  // k přípravě připojení ještě před odesláním skutečného
                                  // požadavku.

// Následující informace se týkají prediktivního předběžného načtení spuštěného navigací.
// Při jejich použití se také nastaví referring_url_.

     STATIC_REFERAL_MOTIVATED,    // Tento převod navrhuje externí databáze.
     LEARNED_REFERAL_MOTIVATED,   // Tento převod nás naučila dřívější navigace.
     SELF_REFERAL_MOTIVATED,      // Odhad potřeby druhého připojení.

     // <snip> ...
};
```

Pokud Prediktor dostane takovýto podnět, je jeho úkolem vyhodnotit pravděpodobnost úspěchu a následně spustit činnost, pokud jsou pro ni k dispozici zdroje. Každá pomocná informace může mít přiřazenou pravděpodobnost úspěchu, prioritu a časové razítko vypršení platnosti, jejichž kombinaci lze použít k údržbě interní fronty priority spekulativních optimalizací. Navíc u každého požadavku vyslaného z této fronty je objekt Predictor také schopen sledovat jeho úspěšnost, což umožňuje dále optimalizovat jeho budoucí rozhodnutí.

3: http://code.google.com/searchframe#OAMlx_jo-ck/src/chrome/browser/net/url_info.h&l=35

Síťová architektura prohlížeče Chrome v kostce

- Prohlížeč Chrome využívá architekturu s více procesy, která izoluje vykreslovací proces od procesu prohlížeče.
- Prohlížeč Chrome udržuje jedinou instanci dispečera zdrojů, která je sdílena všemi vykreslovacími procesy, a jež je spuštěna v rámci procesu kernelu prohlížeče.
- Síťová architektura je meziplatformní, z větší části jednovláknová knihovna.
- Síťová architektura využívá k řízení všech síťových operací neblokující operace.
- Síťová architektura umožňuje účinnou prioritizaci prostředků, jejich opětovné používání a dovoluje prohlížeči provádět globální optimalizaci napříč všemi spuštěnými procesy.
- Každý vykreslovací proces komunikuje s dispečerem zdrojů prostřednictvím IPC.
- Dispečer zdrojů zachycuje požadavky na zdroje prostřednictvím vlastního filtru IPC.
- Objekt Predictor zachycuje požadavky na zdroje a síťový provoz spojený s odpověďmi, čímž se učí a optimalizuje budoucí síťové požadavky.
- Objekt Predictor může spekulativně plánovat dotazy DNS, navázání spojení TCP/IP a dokonce stažení zdrojů na základě naučených schémat provozu a šetřit tak stovky milisekund po spuštění navigace uživatelem.

1.7 Životní cyklus práce s prohlížečem

Když už známe obecný popis architektury síťového stacku prohlížeče Chrome, podívejme se blíže na druhy uživatelsky orientovaných optimalizací, které prohlížeč nabízí. Konkrétně si představme, že jsme si právě vytvořili nový profil v prohlížeči Chrome a jdeme na věc.

Optimalizace prvního startu

Když poprvé spustíte svůj prohlížeč, bude o vašich oblíbených stránkách nebo způsobech navigace vědět velmi málo. Řada z nás ale postupuje po prvním startu prohlížeče stejně – přejdeme do naší složky s doručenými e-maily, na oblíbenou novinovou stránku, stránku sociální sítě, interní portál atd. Konkrétní stránky se budou lišit, ale podobnost těchto relací umožňuje objektu Predictor v prohlížeči Chrome zrychlit váš dojem z prvního startu.

Prohlížeč Chrome si pamatuje deset nejpravděpodobnějších doménových jmen serverů, ke kterým uživatel po spuštění prohlížeče přistupuje – uvědomte si, že se nejedná globálně o deset

nejnavštěvovanějších destinací, ale o specifické destinace přicházející na řadu po prvním spuštění prohlížeče. Při spouštění prohlížeče může Chrome zahájit předběžný převod doménových jmen pravděpodobných destinací. Pokud vás to zajímá, můžete si prohlédnout svůj vlastní seznam názvů hostitelů otevřením nové záložky a navigací na chrome://dns. V horní části stránky najdete seznam deseti nejpravděpodobnějších kandidátů po spuštění k vašemu profilu.

Future startups will prefetch DNS records for 10 hostnames

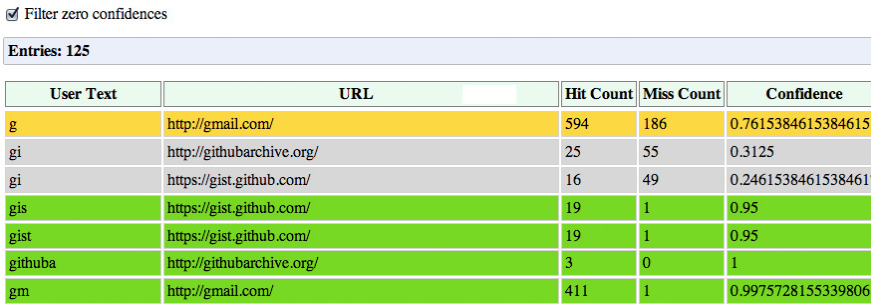
Host name	How long ago (HH:MM:SS)	Motivation
http://www.google-analytics.com/	15:31:33	n/a
https://a248.e.akamai.net/	15:31:30	n/a
https://csi.gstatic.com/	15:31:16	n/a
https://docs.google.com/	15:31:18	n/a
https://gist.github.com/	15:31:34	n/a
https://lh6.googleusercontent.com/	15:31:16	n/a
https://secure.gravatar.com/	15:31:29	n/a
https://ssl.google-analytics.com/	15:31:29	n/a
https://ssl.gstatic.com/	15:31:16	n/a
https://www.google.com/	15:31:16	n/a

Obrázek 1.5: DNS po spuštění

Snímek obrazovky na Obrázku 1.5 je příkladem mého vlastního profilu v prohlížeči Chrome. Čím obvykle začínám své prohlížení? Často navigací na Google Docs, pokud pracuji na článku, jako je tento. Asi není moc překvapivé, že v seznamu vidíme mnoho jmen serverů obsahujících Google.

Optimalizace interakcí s panelem Omnibox

Jednou z inovací prohlížeče Chrome bylo uvedení panelu Omnibox, který na rozdíl od svých předchůdců zvládá mnohem víc, než jen cílové adresy URL. Kromě zapamatování adres URL stránek, které uživatel v minulosti navštívil, nabízí také vyhledávání v plném textu vaší historie, i úzkou integraci s vyhledávačem dle vašeho výběru. Jak uživatel zadává text, panel Omnibox automaticky navrhuje akci, a to ať už se jedná o adresu URL na základě vaší navigační historie, nebo o vyhledávaný dotaz. Uvnitř je každá navrhovaná akce ohodnocena vzhledem k dotazu i k výkonnosti v minulosti. Prohlížeč Chrome nám umožňuje prohlížet tato data na stránce chrome://predictors.



☑ Filter zero confidences

Entries: 125

User Text	URL	Hit Count	Miss Count	Confidence
g	http://gmail.com/	594	186	0.7615384615384615
gi	http://githubarchive.org/	25	55	0.3125
gi	https://gist.github.com/	16	49	0.24615384615384617
gis	https://gist.github.com/	19	1	0.95
gist	https://gist.github.com/	19	1	0.95
githuba	http://githubarchive.org/	3	0	1
gm	http://gmail.com/	411	1	0.9975728155339806

Obrázek 1.6: Predikce adres URL v Omniboxu

Prohlížeč Chrome udržuje historii uživatelem zadaných prefixů, činností, které navrhl, i úspěšnost každé z nich. V mém vlastním profilu můžete vidět, že kdykoli zadám do panelu Omnibox „g“, je 76% šance, že mířím na Gmail. Jakmile přidám „m“ (tedy „gm“), pravděpodobnost vzroste na 99,8 % – ve skutečnosti jsem ze 412 zaznamenaných návštěv neskončil na Gmailu po zadání „gm“ pouze jednou.

Co to má společného se síťovou architekturou? Žlutá a zelená[4] barva pravděpodobných kandidátů jsou také důležitým signálem pro objekt ResourceDispatcher. Pokud máme pravděpodobného kandidáta (žlutá), prohlížeč Chrome může spustit předběžný převod doménového jméno na IP adresu pomocí DNS. Pokud máme vysoce pravděpodobného kandidáta (zelená), pak prohlížeč Chrome může, jakmile je rozpoznán název serveru, spustit také předběžné připojení TCP/IP. Konečně, pokud se obě činnosti dokončí, přičemž uživatel stále váhá, může prohlížeč Chrome dokonce předběžně vykreslit celou stránku ve skryté záložce.

Pokud pro zadaný prefix neexistuje v historii navigace z minulosti dobrá shoda, může prohlížeč Chrome odeslat předběžný dotaz DNS a navázat předběžné TCP/IP spojení s poskytovatelem vyhledávací služby v očekávání pravděpodobného požadavku na vyhledávání.

4: Poznámka vydavatele: Pro černobílou verzi knihy platí, že 1. řádek označuje *„pravděpodobného kandidáta (žlutá)“*, 4.–7. řádek *„vysoce pravděpodobného kandidáta (zelená)“*.

Průměrnému uživateli trvá vyplnění dotazu a vyhodnocení nabízených návrhů automatického dokončování stovky milisekund. Na pozadí je prohlížeč Chrome schopen předběžně získat, připojit a v některých případech i předběžně vykreslit stránku, takže v okamžiku, kdy je uživatel připraven stisknout klávesu „enter“, je již většina síťového zpoždění eliminována.

Optimalizace výkonnosti vyrovnávací paměti

Když mluvíme o výkonnosti, nesmíme nikdy zapomenout na vyrovnávací paměť – ke všem zdrojům na vašich webových stránkách přece do odpovědi přidáváte hlavičky Expires, ETag, Last-Modified, a Cache-Control, že ano? Pokud ne, tak to napravte. Čekáme na vás.

Prohlížeč Chrome má dvě různé implementace interní vyrovnávací paměti: jedna používá lokální disk a druhá ukládá všechno v paměti. Implementace v paměti se používá k prohlížení v anonymním režimu a po zavření okna se vyčistí. Obě však implementují shodné interní rozhraní (disk_cache::Backend, a disk_cache::Entry), které velice zjednodušuje architekturu, a – máte-li takové spády – umožňuje snadno experimentovat s vlastními pokusnými implementacemi vyrovnávací paměti.

Disková vyrovnávací paměť interně implementuje své vlastní datové struktury, které jsou všechny uloženy v jediné složce vyrovnávací paměti pro váš profil. Uvnitř této složky se nacházejí indexové soubory, které jsou namapovány do paměti při spuštění prohlížeče, a datové soubory, které uchovávají skutečná data spolu s hlavičkami HTTP a dalšími režijními informacemi.[5] Nakonec disková vyrovnávací paměť udržuje mezipaměť typu LRU (Least Recently Used – nejčastější naposledy použité položky jsou v ní uloženy nejdéle). K ohodnocení položky v mezipaměti se přitom berou v úvahu takové metriky, jako je frekvence přístupu k uloženému zdroji a jeho stáří.

Obrázek 1.7: Zjištění stavu vyrovnávací paměti prohlížeče Chrome

5: Zdroje do velikosti 16 KB jsou uloženy ve sdílených datových blokových souborech, větší soubory mají své vlastní dedikované soubory na disku.

Pokud vás někdy zajímal stav vyrovnávací paměti prohlížeče Chrome, můžete otevřít novou záložku a přejít na chrome://net-internals/#httpCache. Případně, chcete-li vidět skutečná metadata HTTP a odezvu uloženou ve vyrovnávací paměti, můžete přejít také na chrome://cache, kde se vypíšou veškeré zdroje aktuálně dostupné ve vyrovnávací paměti. Kliknutím na adresu URL určitého zdroje v tomto seznamu zobrazíte přesné hlavičky uložené ve vyrovnávací paměti a bajty odezvy.

Optimalizace DNS předběžným načtením

Již jsme při několika příležitostech zmínili předběžný převod pomocí protokolu DNS. Předtím, než se ponoříme do implementace, projděme si případy, ve kterých se může tento převod spustit, a proč:

- Parser dokumentů Blink, který spouští vykreslovací proces, může poskytnout seznam doménových jmen serverů pro všechny odkazy na aktuální stránce. Prohlížeč Chrome se může rozhodnout a předem převést tato jména na IP adresy.

- Vykreslovací proces může spustit událost přejetí kurzorem nebo stisknutí klávesy coby včasný signál uživatelova úmyslu provést navigaci.

- Panel Omnibox může spustit požadavek na resoluci na základě vysoce pravděpodobného návrhu.

- Objekt Predictor může vyslat požadavek na převod doménového jména serveru na základě navigace v minulosti a dat obsažených v požadavku na zdroj.

- Vlastník stránky může explicitně označit doménová jména serverů, která má prohlížeč Chrome předběžně rozpoznat.

Ve všech případech je předběžný převod pomocí DNS pokládán za nápovědu. Prohlížeč Chrome nezaručuje, že k předběžnému převodu dojde, spíše využívá každý podnět v kombinaci s vlastním objektem Predictor k vyhodnocení nápovědy a rozhodnutí o další činnosti. V „nejhorším případě“, kdy by nebyl prohlížeč Chrome schopen předběžně převést doménové jménovčas, počká uživatel na explicitní převod DNS, následované dobou připojení TCP/IP a nakonec na vlastní načtení zdroje. Pokud ovšem k tomuto dojde, objekt Predictor si udělá poznámku a podle ní upraví svá budoucí rozhodnutí – tím, jak jej používáte, se stává rychlejším a chytřejším.

Jednou z optimalizací, které jsme se zatím nevěnovali, je schopnost prohlížeče Chrome učit se topologii každé stránky a pak tyto informace využít ke zrychlení budoucích návštěv. Vzpomeňte si třeba na onu průměrnou stránku sestávající z 88 zdrojů, které byly získány od více než 15 různých serverů. Pokaždé, když provedete navigaci, prohlížeč Chrome si může zaznamenat jména serverů u oblíbených zdrojů na stránce a během budoucí návštěvy se může rozhodnout spustit předběžný převod DNS a dokonce předběžně navázat spojení TCP/IP pro některé nebo všechny z nich.

Host for Page	Page Load Count	Subresource Navigations	Subresource PreConnects	Subresource PreResolves	Expected Connects	Subresource Spec
https://plus.google.com/	688	6	4	17	0.013	https://apis.google.com/
		2	3	8	0.065	https://csi.gstatic.com/
		152	27	33	0.194	https://lh3.googleusercontent.com/
		2	3	1	0.509	https://lh6.googleusercontent.com/
		896	296	386	1.853	https://plus.google.com/
		79	22	18	0.194	https://ssl.gstatic.com/

Obrázek 1.8: Statistiky dílčích zdrojů

Chcete-li si prohlédnout jména hostitelů dílčích zdrojů, které si prohlížeč Chrome uložil, přejděte na chrome://dns a vyhledejte jméno hostitele libovolné oblíbené destinace ve vašem profilu. V příkladu výše vidíte názvy hostitelů šesti dílčích zdrojů, které si prohlížeč Chrome zapamatoval u Google+, i statistiky počtu případů, v nichž došlo ke spuštění předběžného rozpoznání DNS nebo navázání předběžného spojení TCP, i očekávaný počet požadavků, které každý z nich obslouží. Tento interní výpočet je tím, co umožňuje objektu Predictor prohlížeče Chrome provádět optimalizace.

Kromě všech interních podnětů může také vlastník stránky vložit na své stránky další značku, kterou požádá prohlížeč o předběžné rozpoznání názvu hostitele:

```
<link rel="dns-prefetch" href="//host_name_to_prefetch.com">
```

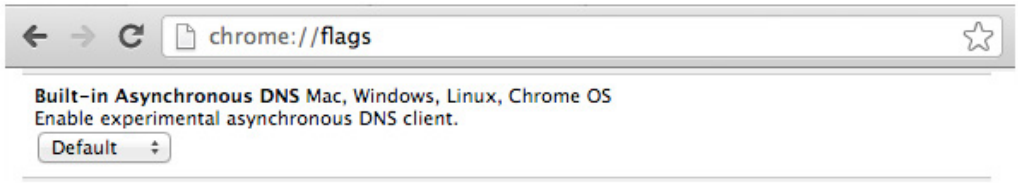
Proč se jednoduše nespolehnout na automatiku prohlížeče? V některých případech můžete chtít předběžně rozpoznat jméno serveru, který není nikde na stránce zmíněn. Učebnicovým příkladem je přesměrování: odkaz může odkazovat na server – jako je například analytická sledovací služba –, který následně přesměruje uživatele na skutečnou destinaci. Prohlížeč Chrome nedokáže odvodit tento vzor sám od sebe, ale můžete mu pomoci tím, že mu poskytnete manuální nápovědu a necháte prohlížeč předem převést jméno serveru skutečné destinace.

Jak je to vše implementováno uvnitř? Odpověď na tuto otázku, ostatně jako u všech ostatních optimalizací, které jsme probrali, závisí na verzi prohlížeče Chrome, protože tým stále experimentuje s novými a lepšími způsoby, jak zlepšit výkonnost. Každopádně, zevrubně řečeno, infrastruktura DNS v rámci prohlížeče Chrome má dvě hlavní implementace. V minulosti se prohlížeč Chrome spoléhal na systémové volání getaddrinfo(), které je nezávislé na platformě, a delegoval tak skutečnou odpovědnost za vyhledávání na operační systém. Tento přístup je však postupně nahrazován vlastní implementací asynchronního DNS překladače.

Původní implementace, která se spoléhala na operační systém, měla své výhody: kratší a jednodušší kód, a také možnost využívat DNS mezipaměť operačního systému. Avšak getaddrinfo() je blokujícím voláním systému, což znamenalo, že prohlížeč Chrome musel vytvořit a udržovat speciální zásobník pracovních vláken, aby mohl provádět více převodů najednou. Tento zásobník byl omezen na šest pracovních vláken, což je empirické číslo založené na nejmenším společném jmenovateli hardwaru – ukázalo se totiž, že vyšší počet paralelních požadavků dokáže přetížit směrovače některých uživatelů.

Předběžnou převod pomocí pracovního zásobníku prohlížeč Chrome jednoduše provedl voláním getaddrinfo(), které blokovalo pracovní vlákno, dokud nebyla připravena odezva, a v tu chvíli pouze zahodil vrácený výsledek a začal zpracovávat další požadavek na předběžný převod. Výsledek je uložen ve vyrovnávací mezipaměti operačního systému pro DNS, která v budoucnu vrátí okamžitou odezvu skutečného vyhledávání voláním getaddrinfo(). Je to jednoduché, účinné a v praxi to dostatečně funguje.

Ano, je to účinné, ale ne dost dobré. Volání getaddrinfo() ukrývá spoustu užitečných informací, například časová razítka doby platnosti (TTL) každého záznamu a také stav vyrovnávací mezipaměti DNS samotné. Ke zlepšení výkonnosti se tým prohlížeče Chrome rozhodl implementovat svůj vlastní, na platformě nezávislý asynchronní DNS resolver.

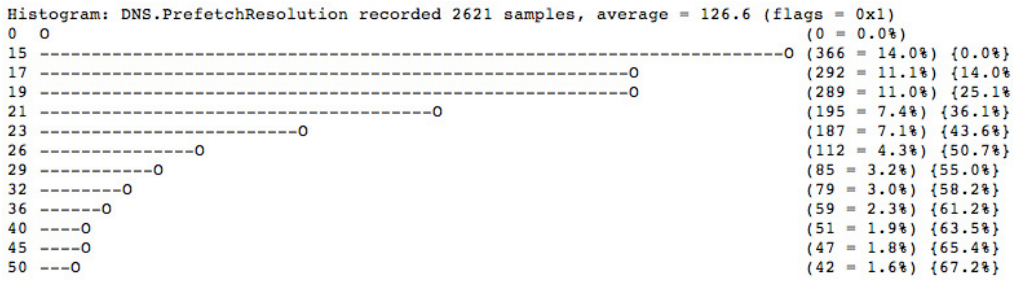


Obrázek 1.9: Povolení asynchronního DNS překladače

Přesunem převodu DNS do prohlížeče Chrome umožnil nový asynchronní překladač řadu nových optimalizací:

- lepší ovládání časovačů opakovaného přenosu a schopnost vykonávat více dotazů paralelně,
- viditelnost TTL záznamu, což umožňuje prohlížeči Chrome aktualizovat oblíbené záznamy předem,
- lepší chování implementací (protokoly IPv4 a IPv6),
- přepnutí na jiné servery v případě poruchy na základě RTT nebo jiných signálů.

Vše výše uvedené a mnoho dalšího jsou nápady pro pokračující experimentování s prohlížečem Chrome a jeho vylepšování. To nás přivádí k jasné otázce: jak poznáme a změříme vliv těchto nápadů? Jednoduše, protože prohlížeč Chrome zaznamenává podrobné statistiky výkonnosti sítě a histogramy pro každý jednotlivý profil. Chcete-li si prohlédnout sesbírané metriky DNS, otevřete novou kartu a přejděte na chrome://histograms/DNS (viz Obrázek 1.10).



Obrázek 1.10: Histogramy předběžného načtení DNS

Výše uvedený histogram zobrazuje distribuci zpoždění u požadavků na předběžnou resoluci DNS: zhruba 50 % (sloupec úplně vpravo) dotazů na předběžné načtení bylo dokončeno do 20 ms (sloupec úplně vlevo). Všimněte si, že se jedná o data založená na nedávném použití prohlížeče (9 869 vzorků) a jsou soukromá pro uživatele. Pokud se uživatel rozhodl hlásit své statistiky využití v prohlížeči Chrome, pak je souhrn těchto dat anonymizován a pravidelně zasílán týmu inženýrů, kteří vidí dopad svých experimentů a mohou je příslušným způsobem upravit.

Optimalizace řízení spojení TCP/IP pomocí předběžného navázání

Předběžně jsme převádeli doménové jméno serveru a s vysokou pravděpodobností se chystá navigační událost, přesně jak odhadl panel Omnibox nebo objekt Predictor prohlížeče Chrome. Proč tedy nepokročit o krok dále a také se předem spekulativně nepřipojit k cílovému hostiteli a nedokončit navázání TCP spojení ještě předtím, než uživatel vyšle požadavek? Pokud tak učiníme, můžeme eliminovat další zpoždění z důvodu latence na obousměrné cestě, které nám jednoduše ušetří stovky milisekund pro uživatele. Přesně tohle je totiž předběžné připojení TCP a přesně takto funguje.

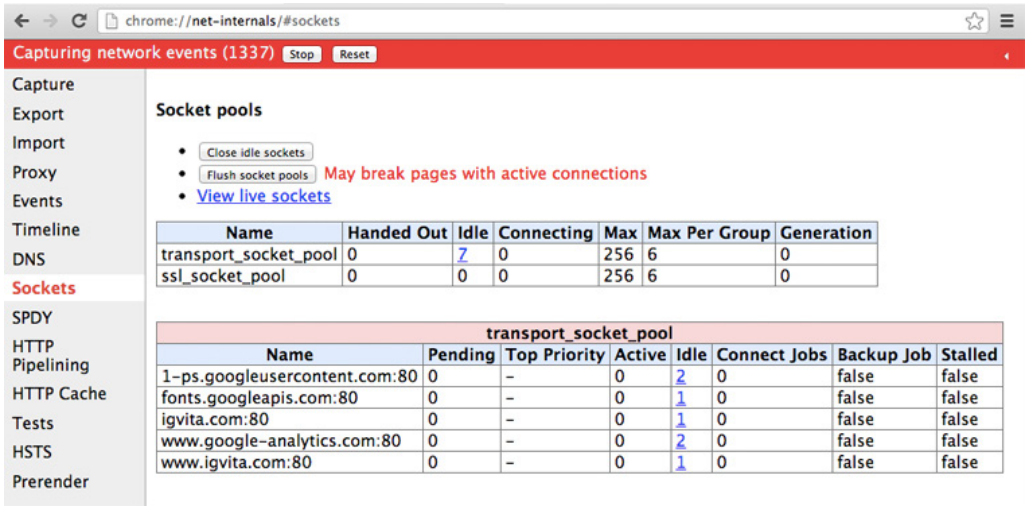
Chcete-li vidět hostitele, u kterých bylo spuštěno předběžné připojení TCP, otevřete novou záložku a navštivte chrome://dns.

Host for Page		Subresource Navigations	Subresource PreConnects	Subresource PreResolves	Expected Connects	Subresource Spec
https://plusone.google.com/	51	36	23	18	1.215	https://plusone.google.com/

Obrázek 1.11: Zobrazení hostitelů, u kterých bylo spuštěno předběžné připojení TCP

Prohlížeč Chrome nejprve zkontroluje své zásobníky soketů, aby zjistil, zda je pro jméno hostitele k dispozici soket, který by Chrome mohl opětovně použít – živé sokety jsou udržovány v zásobníku pouze po určitou dobu, aby se neuplatnily náklady na navázání TCP spojení a pomalý start. Pokud není k dispozici žádný soket, může se spustit navázání TCP spojení a uložit nový soket do zásobníku. Poté, jakmile uživatel začne s navigací, je možné požadavek HTTP vyslat okamžitě.

Pokud byste rádi zjistili více informací, prohlížeč Chrome je vybaven nástrojem chrome://net-internals#sockets, kterým lze prozkoumat stav všech soketů otevřených v prohlížeči Chrome. Snímek obrazovky je zobrazen na Obrázku 1.12.



Obrázek 1.12: Otevřené sokety

Všimněte si, že můžete hlouběji prozkoumat každý soket a prohlédnout si časovou osu: doby připojení a proxy, čas přijetí každého paketu a další informace. A také můžete tato data exportovat pro pozdější analýzu nebo hlášení o chybách. Dobré vybavení nástroji je klíčem k jakékoli

optimalizaci výkonnosti a na stránce chrome://net-internals se scházejí informace o tom, co prohlížeč Chrome na síti dělá – pokud jste jej ještě neprozkoumali, měli byste se na to vrhnout!

Optimalizace načítání zdrojů pomocí nápovědy pro předběžné načtení

Někdy je autor stránky schopen poskytnout další navigaci nebo kontext stránky na základě struktury či rozvržení své stránky, a pomoci tak prohlížeči optimalizovat uživatelský dojem. Prohlížeč Chrome podporuje dvě takové nápovědy, které lze vložit jako značku na stránku:

```
<link rel="subresource" href="/javascript/myapp.js">
<link rel="prefetch"    href="/images/big.jpeg">
```

Obě vypadají velmi podobně, ale mají odlišnou sémantiku. Pokud odkaz specifikuje svůj vztah jako „předběžné načtení" (prefetch), jedná se o indikaci prohlížeči, že by tento zdroj mohl potřebovat při budoucí navigaci. Jinými slovy, jedná se prakticky o nápovědu pro křížový odkaz mezi stránkami. Naproti tomu, pokud odkaz specifikuje vztah jako „dílčí zdroj" (subresource), jedná se o časnou indikaci prohlížeči, že se na aktuální stránce použije zdroj, a že by měl vyslat požadavek předtím, než se s ním setká později v dokumentu.

Jak se dá očekávat, rozdílná sémantika nápověd vede k velice odlišnému chování načítávače zdrojů. Zdroje označené předběžným načítáním se považují za nízkoprioritní a prohlížeč je může stáhnout až ve chvíli, kdy je načtena aktuální stránka. Zdroje typu dílčí zdroj jsou získávány s vysokou prioritou ihned, jakmile se s nimi prohlížeč setká, a budou zápolit se zbytkem zdrojů na aktuální stránce.

Obě nápovědy, pokud jsou použity dobře a ve správném kontextu, mohou výrazně pomoci s optimalizací dojmu uživatele na vaší stránce. Nakonec je také důležité zmínit, že předběžné načtení je součástí specifikace[6] HTML5, které dnes podporují prohlížeče Firefox a Chrome, zatímco dílčí zdroj je v současnosti k dispozici pouze v prohlížeči Chrome.

Optimalizace načtení zdrojů pomocí předběžného obnovení prohlížeče

Bohužel ne všichni vlastníci stránek jsou schopni nebo ochotni poskytnout prohlížeči ve svém kódu nápovědu pro dílčí zdroje. Navíc, i když to udělají, musíme počkat, než nám ze serveru dorazí HTML dokument, abychom mohli analyzovat tyto nápovědy a začít načítat potřebné dílčí zdroje – v závislosti na době odezvy serveru i latenci mezi klientem a serverem to může trvat stovky či dokonce tisíce milisekund.

Jak jsme však viděli dříve, prohlížeč Chrome se již učí názvy hostitelů oblíbených zdrojů, aby mohl provádět předběžný převod DNS. Tak proč bychom nemohli udělat to samé, ale jít ještě o krok dál a provést převod DNS, předběžně navázat spojení TCP/IP a poté i spekulativně předběžně načíst zdroj? Tak přesně toto dokáže předběžné obnovení:

6: http://www.whatwg.org/specs/web-apps/current-work/multipage/links.html#link-type-prefetch

- Uživatel spustí požadavek na cílovou adresu URL.
- Prohlížeč Chrome vyšle dotaz svému objektu Predictor na naučené dílčí zdroje asociované s cílovou adresou URL a spustí sekvenci předběžného převodu DNS, předběžného připojení TCP/IP a předběžného obnovení zdroje.
- Pokud je naučený dílčí zdroj ve vyrovnávací paměti, načte se z mezipaměti.
- Pokud naučený dílčí zdroj chybí nebo vypršela jeho platnost, získá se ze sítě.

Předběžné obnovení zdroje je skvělým příkladem postupu práce na každé experimentální optimalizaci v prohlížeči Chrome – teoreticky by mělo zajistit lepší výkonnost, ale je zde také mnoho kompromisů. Existuje pouze jeden způsob, jak spolehlivě určit, zda daná optimalizace uspěje a dostane se do prohlížeče Chrome: implementovat a spustit jako A/B experiment na některém kanálu pro předčasné vydávání mezi skutečnými uživateli, na skutečných sítích a se skutečnými průběhy prohlížení.

Již na začátku roku 2013 zahájil tým prohlížeče Chrome prvotní fáze diskuzí o implementaci. Pokud to na základě sesbíraných výsledků projde, dočkáme se předběžného obnovení v prohlížeči Chrome někdy koncem roku. Proces vylepšování síťové výkonnosti prohlížeče Chrome nikdy nekončí – tým stále experimentuje s novými přístupy, nápady a technikami.

Optimalizace navigace pomocí předběžného vykreslování

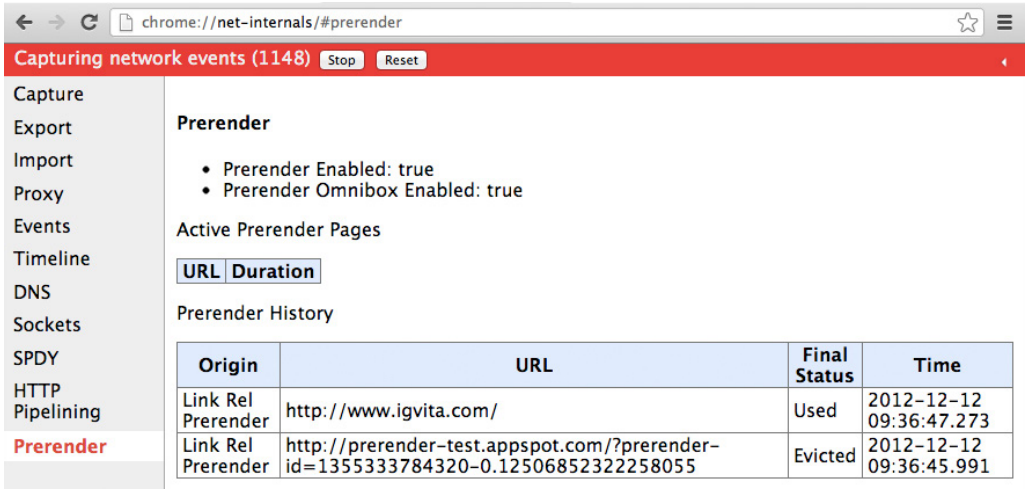
Každá optimalizace, kterou jsme až dosud probrali, pomáhá snižovat zpoždění mezi přímým požadavkem uživatele na navigaci a výsledným vykreslením stránky v jeho záložce. Nicméně, co je potřeba pro opravdu okamžitý dojem? Na základě dat z UX (user experience - dojem uživatele), které jsme viděli dříve, má taková interakce méně než 100 ms, což nenechává vůbec žádný prostor pro zpoždění sítě. Co můžeme udělat pro to, abychom dokázali dodat vykreslovanou stránku za méně než 100 ms?

Samozřejmě už znáte odpověď, protože se jedná o běžný vzorec používaný mnoha uživateli: pokud otevřete více záložek, pak je přepínání mezi nimi okamžité a je rozhodně mnohem rychlejší než čekání na navigaci mezi shodnými zdroji na jedné záložce v popředí. Tak co kdyby pro toto poskytoval prohlížeč rozhraní API?

```
<link rel="prerender" href="http://example.org/index.html">
```

Uhádli jste, jedná se o předběžné vykreslení v prohlížeči Chrome. Namísto pouhého stahování jednoho zdroje, jako je tomu v případě nápovědy „předběžné načtení", indikuje atribut „předběžné vykreslení" prohlížeči Chrome, že by měl předběžně vykreslit stránku na skryté záložce spolu se všemi jejími dílčími zdroji. Skrytá záložka sama o sobě je uživateli neviditelná, avšak jakmile uživatel spustí navigaci, záložka se přepne z pozadí a poskytne „okamžitý dojem".

Chtěli byste si to vyzkoušet? Na stránce prerender-test.appspot.com najdete praktickou ukázku. Historii a stav předběžně vykreslených stránek vašeho profilu si pak můžete prohlédnout na stránce: chrome://net-internals/#prerender. (Viz Obrázek 1.13.)



Obrázek 1.13: Předběžně vykreslené stránky v aktuálním profilu

Dá se očekávat, že vykreslení celé stránky ve skryté záložce dokáže spotřebovat hodně zdrojů, jak strojového času procesorů, tak síťových prostředků, a mělo by se tedy používat pouze v případech velké pravděpodobnosti, že skrytou záložku využijeme. Například při používání panelu Omnibox se předběžné vykreslení může spustit u vysoce pravděpodobných návrhů. Podobně služba Google Search občas přidává nápovědu pro předběžné vykreslení mezi své značky, pokud odhaduje, že první výsledek hledání je vysoce pravděpodobnou destinací (známé také pod názvem Google Instant Pages).

I vy můžete přidat nápovědu pro předběžné vykreslení na své vlastní stránky. Než tak učiníte, uvědomte si, že předběžné vykreslení má celou řadu omezení, na která byste měli myslet:

- Napříč všemi procesy je povolena maximálně jedna předběžně vykreslená záložka.
- HTTPS a stránky s autentizací HTTP nejsou povoleny.
- Předběžné vykreslení se neudělá, pokud požadovaný zdroj nebo jakýkoli z jeho dílčích zdrojů vyžadují provedení neidempotentního požadavku (jsou povoleny pouze požadavky GET).

- Veškeré zdroje jsou načteny s nejnižší síťovou prioritou.
- Stránka je vykreslena s nejnižší prioritou - tj. vykresluje se pouze tehdy, když procesor nemá na práci nic jiného.
- Předběžné vykreslení se zastaví, pokud požadavky na paměť přesáhnou 100 MB.
- Inicializace pluginu je odložena a předběžné vykreslení je zastaveno, pokud se vyskytne mediální prvek HTML5.

Jinak řečeno, není zaručeno, že dojde k předběžnému vykreslení, a použije se pouze u stránek, kde je to bezpečné. Vzhledem k tomu, že se na skryté záložce může provést JavaScript nebo jiná akce, je nejlepší využít rozhraní Page Visibility API ke zjištění, zda je stránka viditelná – což byste měli dělat tak jako tak.

1.8 Prohlížeč Chrome se zrychluje vaším používáním

Není nutné zmiňovat, že síťová architektura prohlížeče Chrome je mnohem víc, než jen jednoduchý návrh pro manipulaci se sokety. Naše rychlá prohlídka se týkala mnoha úrovní potenciálních optimalizací, které se provádějí transparentně v pozadí při procházení webu. Čím více se prohlížeč Chrome naučí o topologii webu a vašich způsobech prohlížení, tím lépe dokáže dělat svou práci. Vypadá to skoro jako kouzlo – prohlížeč Chrome se zrychluje tím, jak jej používáte. Až na to, že to není kouzlo – protože teď už víte, jak to funguje.

Nakonec je důležité si uvědomit, že tým prohlížeče Chrome pokračuje v iteracích a experimentech s novými nápady, jak vylepšit výkonnost – tento proces nikdy neskončí. Zatímco čtete tuto knihu, je velká šance, že probíhají nové experimenty a vyvíjejí se, testují se nebo se nasazují nové optimalizace. Možná jednou dosáhneme našeho cíle okamžitého načítání stránek (< 100 ms) u každé stránky a pak si konečně budeme moci odpočinout. Do té doby před námi bude pořád nějaká ta práce.

2 Od SocialCalc k EtherCalc

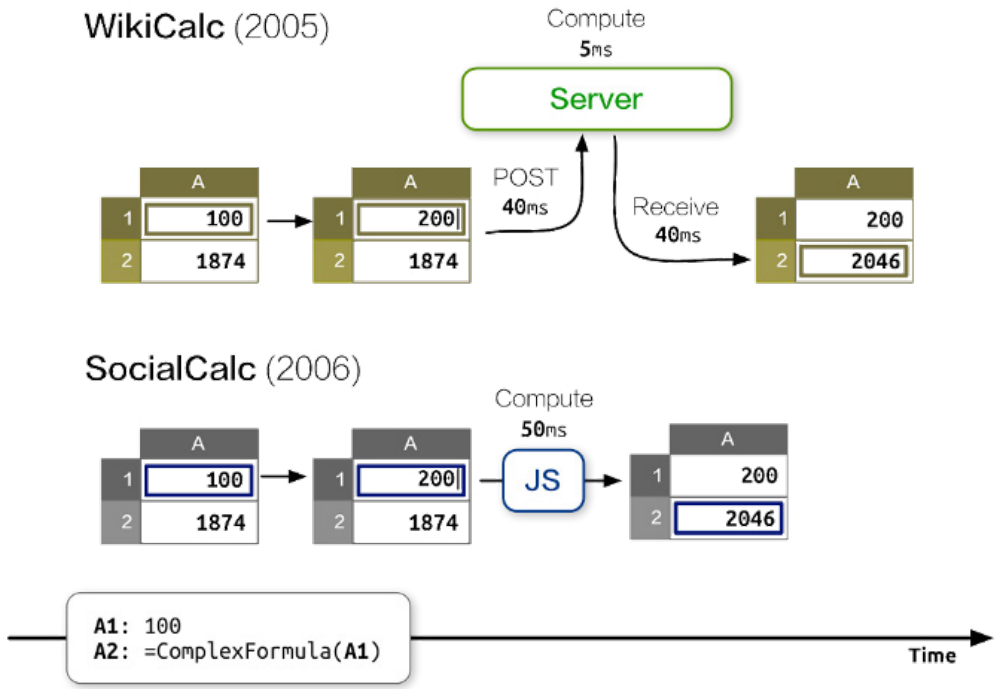
(Audrey Tang)

2 Od SocialCalc k EtherCalc

EtherCalc je online tabulkový procesor optimalizovaný pro souběžnou editaci, který používá SocialCalc k tomu, aby mohl běžet v prohlížeči. Byl navržen Danem Bricklinem (vynálezcem tabulkového procesoru), a to jako součást platformy Socialtext, což je sada nástrojů pro spolupráci firemních uživatelů.

Když v roce 2006 začínal tým Socialtext pracovat na vývoji SocialCalc, byla pro něj primárním cílem vysoká výkonnost.

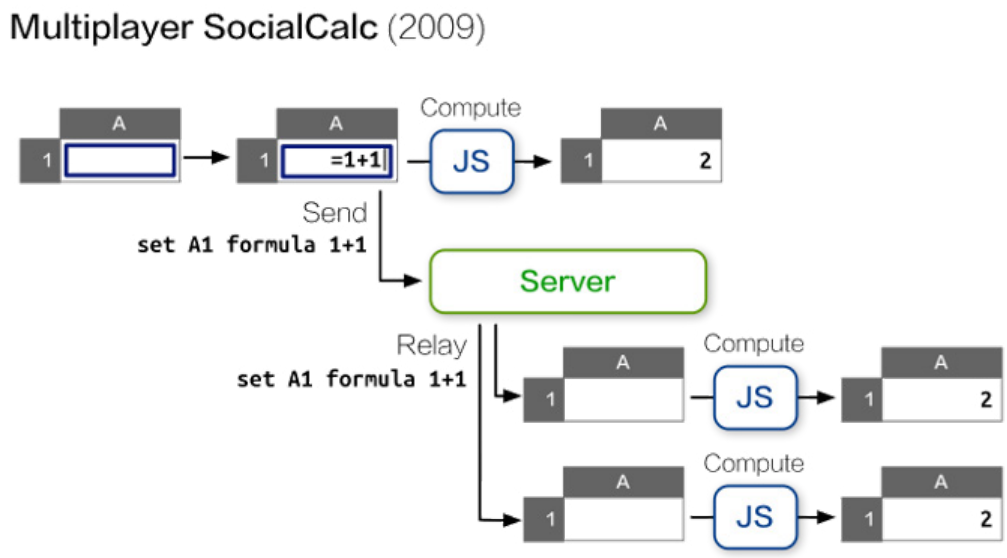
Klíčovým zjištěním bylo, že výpočty v prohlížeči, a za pomoci JavaScriptu, i když byly řádově pomalejší než výpočet na serverové straně v jazyce PERL, trvaly kratší dobu než síťové zpoždění vznikající během obousměrné komunikaci se serverem při použití technologie AJAX.



Obrázek 2.1: Výkonnostní model WikiCalc a SocialCalc. Od roku 2009 pokrok v implementaci běhového prostředí JavaScriptu zredukoval z 50 ms na méně než 10 ms.

SocialCalc provádí všechny své výpočty v prohlížeči; server se stará jenom o načtení a uložení tabulek.

Na konci knihy *The Architecture of Open Source Applications*[1] [BW11] jsme v kapitole o SocialCalc představili simultánní spolupráci na tabulkách použitím jednoduché architektury podobné diskuzní místnosti.



Obrázek 2.2: SocialCalc pro více hráčů

Nicméně když jsme ji začali testovat pro produkční nasazení, objevili jsme několik nedostatků v její výkonnosti a škálovatelnosti, které vedly k sérii rozsáhlých celosystémových úprav, abychom dosáhli akceptovatelnou výkonnost. V této kapitole se podíváme, jak jsme se dostali k výsledné architektuře, jak jsme používali profilovací nástroje, a jak jsme vytvořili nové nástroje, abychom překonali výkonnostní problémy.

1: Poznámka vydavatele: Kniha *The Architecture of Open Source Applications* je součástí série knih AOSA (www.aosabook.org).

Omezení při návrhu

Socialtext platformu je možné nasadit jak za firewallem, tak na cloudu. Každá z možností má specifické omezení na výkonnostní požadavky systému EtherCalc.

V době psaní této knihy vyžadoval Socialtext 2 CPU jádra a 4 GB RAM pro VMWare vSphere intranetového nasazení. Při cloudovém hostování měla Amazon EC2 instance více jak dvojnásobnou kapacitu - 4 jádra a 7,5 GB paměti RAM.

Nasazení za firewallem znamená, že nemůžeme prostě jenom přidat více hardware jako u vícestranných hostovaných systémů (např. docVerse, který se později stal součástí GoogleDocs); k dispozici máme jenom malou část serverové kapacity.

Ve srovnání s intranetovým nasazením, cloudové instance nabízejí větší kapacitu a rozšíření na přání, ale síťové připojení z prohlížeče je obvykle pomalejší a s častým odpojením a opětovným připojením.

Proto následující omezení výpočetních zdrojů udalo směrování architektury systému EtherCalc:

Paměť: Server reagující na události umožňuje škálovat souběžná připojení i s malou kapacitou paměti RAM.

CPU: Na základě původního návrhu SocialCalc přesuneme většinu výpočtů a vykreslování na klientskou stranu pomocí JavaScriptu.

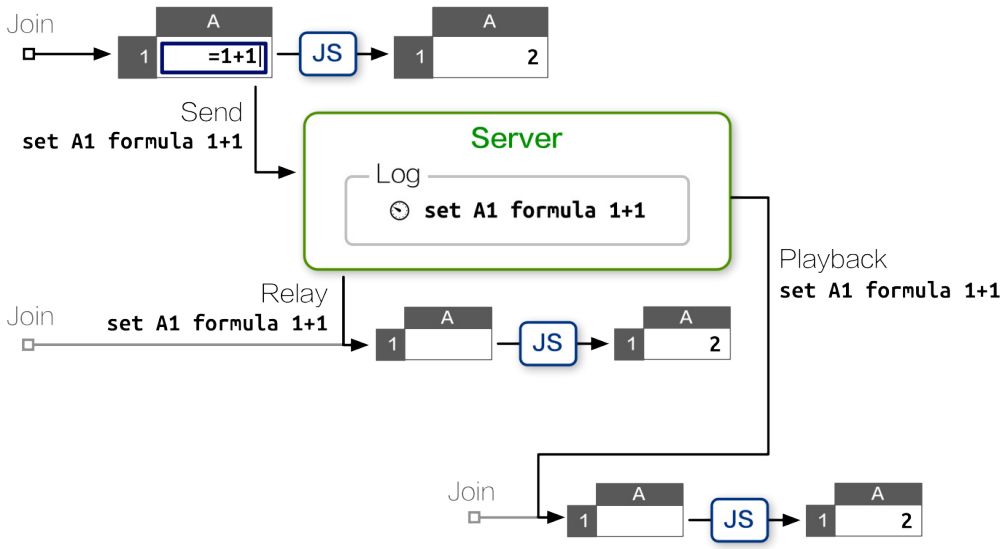
Síť: Odesíláním tabulkových operací, místo obsahu tabulek, zredukujeme požívání připojení a umožníme obnovení síťového spojení na nespolehlivých linkách.

2.1 Úvodní prototyp

Začali jsme serverem WebSocket implementovaným v Perl 5, zálohovaným web-serverem Feersum, což je neblokující web server postavený na libev a vyvinutý v Socialtext. Feersum je velmi rychlý, umožňuje zpracovat více jak 10 000 požadavků za sekundu na jednom CPU. Kromě Feersum jsme použili PocketI middleware, abychom využili oblíbeného klienta Socket.io JavaScript, který poskytuje zpětnou kompatibilitu u starších prohlížečů bez podpory WebSocket.

Prvotní prototyp se podobá chat serveru. Každá relace pro spolupráci je diskuzní místností; klienti odešlou své lokálně vykonávané příkazy a pohyby kurzoru na server, který je předává všem ostatním klientům ve stejné místnosti.

Diagram níže ukazuje typický tok operací:



Obrázek 2.3: Prototyp serveru s logováním a přehráváním

Server zaznamenává každý příkaz s časovým razítkem. Pokud se klient odpojí a znovu připojí, může pokračovat tak, že požádá o záznam všech požadavků od doby svého odpojení, tyto příkazy si lokálně přehraje, a dostane se do stejného stavu jako ostatní účastníci.

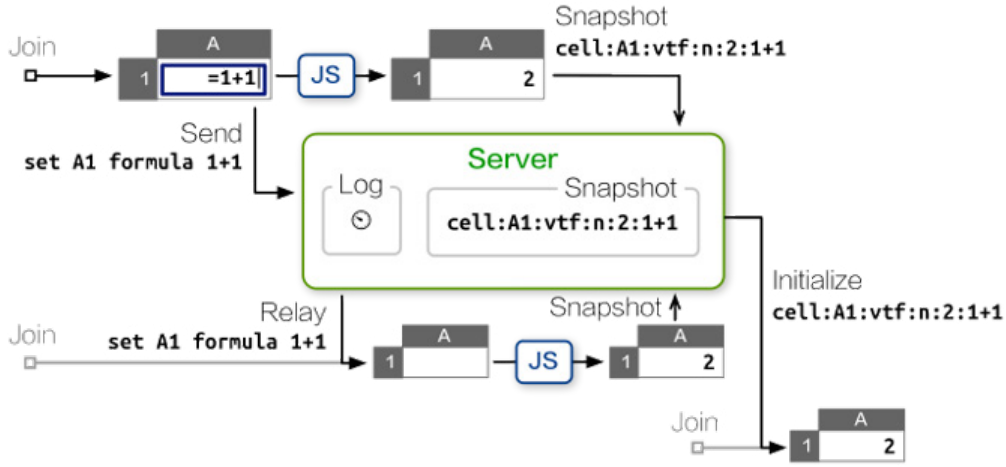
Tento jednoduchý návrh redukuje požadavky na CPU a RAM na straně serveru, a je rozumně odolný vůči výpadkům sítě.

2.2 První úzké místo

Když jsme v červnu 2011 dali prototyp na testování, rychle jsme zjistili výkonnostní problém s dlouho běžícími relacemi. Tabulky jsou dokumenty s dlouhou životností, a relace pro spolupráci může akumulovat tisíce modifikací v průběhu několika týdnů editování. Když se klient přidá k této editovací relaci s jednoduchým zálohovacím modelem, musí si přehrát tisíce příkazů, což způsobí významné zpoždění po spuštění dřív, než začne dělat jakoukoliv modifikaci.

Pro zmírnění tohoto problému jsme zavedli snímkovací mechanismus. Po každých 100 příkazech zaslaných do místnosti shromáždí server stavy všech aktivních klientů a uloží je jako poslední

snímek do zálohy. Nově připojený klient obdrží tento snímek včetně nového příkazu vloženého po tom, co byl pořízen snímek, takže musí odpovědět nanejvýš na 99 příkazů.



Obrázek 2.4: Prototyp serveru se snímkovacím mechanismem.

Tato alternativa řešila problém se zatížením CPU pro nové klienty, ale vytvořila vlastní výkonnostní problém na síti tím, že výrazně zvýšila datový objem komunikace s klientem. V případě pomalého připojení to zpožďovalo příjem následných příkazů od klientů.

Navíc neměl server možnost ověřit konzistenci snímků předložených klienty.

Tudíž všechny chybné či škodlivé snímky mohly poškodit stav všech nově příchozích tím, že se nemohly synchronizovat se stávajícími účastníky.

Bystrý čtenář si může uvědomit, že oba problémy jsou způsobeny neschopností serveru vykonávat tabulkové příkazy. Pokud by byl server schopen aktualizovat svůj vlastní stav po přijetí každého příkazu, vůbec by nepotřeboval udržovat zálohu příkazů.

V prohlížeči běžící jádro SocialCalc je napsané v JavaScriptu. Uvažovali jsme o překladu této logiky do Perlu, to by ale přineslo velký nárůst nákladů na udržování dvou zdrojových kódů. Experimentovali jsme s vloženými JavaScriptovými jádry (V8, SpiderMonkey), což ale přinášelo poklesy výkonosti při běhu na serveru Feersum.

Nakonec jsme to v srpnu 2011 vyřešili tak, že jsme přepsali server do Node.js.

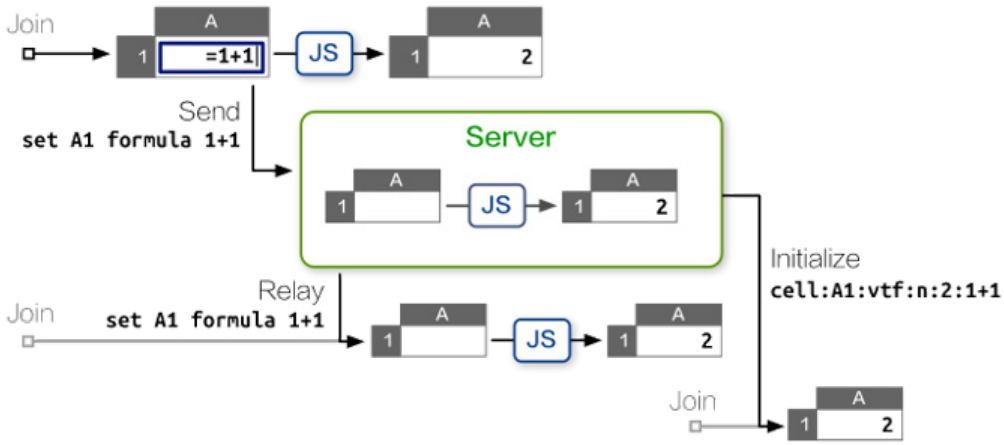
2.3 Přenesení do Node.js

Prvotní přepsání proběhlo hladce, protože jak Feersum, tak Node.js jsou založeny na stejném událostním modelu knihovny libev a API Pocket.io přesně odpovídá Socket.io. Stačilo nám jedno odpoledne, abychom napsali funkčně ekvivalentní server o 80 řádcích kódu, díky stručnému API nabízeném frameworkem ZappaJS.

Prvotní mikro srovnání nám ukázalo, že přenesení na Node.js nás stojí kolem poloviny maximální propustnosti. Na typickém procesoru Intel Core i5 v roce 2011 obsloužila původní architektura s Feersum 5 000 požadavků za sekundu, zatímco Express na Node.js maximálně 2 800 požadavků za sekundu.

Toto snížení výkonu bylo pro náš první JavaScript port považováno za akceptovatelné, protože nezpůsobilo významné zvýšení zpoždění pro uživatele, a očekávali jsme zlepšení v budoucnu.

Následně jsme pokračovali v práci na redukci zatížení CPU na straně klienta a minimalizaci objemu po síti přenášených dat sítě sledováním každého odchozího stavu relace se serverovou stranou tabulek SocialCalc.



Obrázek 2.5: Udržování stavu tabulek s Node.js serverem

2.4 Serverová strana SocialCalc

Klíčovou technologií umožňující naši práci je jsdom, kompletní implementace W3C objektového modelu dokumentu (DOM), který umožňuje, aby Node.js načítal knihovny JavaScript na klientské straně v simulovaném prostředí prohlížeče.

Použitím jsdom je triviální vytvořit libovolný počet SocialCalc tabulek na serverové straně, každou o cca 30 KB paměti RAM a běžící ve vlastním odděleném prostředí:

```
require! <[ vm jsdom ]>
create-spreadsheet = ->
     document = jsdom.jsdom \<html><body/></html>
     sandbox = vm.createContext window: document.createWindow! <<< {
          setTimeout, clearTimeout, alert: console.log
     }
     vm.runInContext """
          #packed-SocialCalc-js-code
          window.ss = new SocialCalc.SpreadsheetControl
     """ sandbox
```

Každá relace pro spolupráci je asociovaná s řadičem SocialCalc, kde je izolovaná od ostatních, a vykonává příkazy v pořadí, v jakém přijdou. Server následně přenáší tento aktualizovaný stav řadiče nově připojeným klientům a zcela odstraňuje potřebu zálohování.

Byli jsme spokojeni s výsledky porovnání, a tak jsme naprogramovali persistentní vrstvu v Redis a uvolnili EtherCalc.org k veřejnému beta testování. V následujících šesti měsících si vedl pozoruhodně dobře, vykonal miliony tabulkových operací bez jediného incidentu.

V dubnu 2012, po přednášce o EtherCalc na konferenci OSDC.tw, jsem byl pozván firmou Trend Micro, abych se účastnil jejich hackathonu a přizpůsobil EtherCalc do programovatelného vizualizačního nástroje pro jejich systém monitorování webového provozu v reálném čase.

Pro tento případ jsme vytvořili REST API pro přístup k jednotlivým buňkám s GET/PUT a také POST příkazy přímo do tabulek. V průběhu hackathonu přijal úplně nový REST obsluhovač stovky volání za sekundu, aktualizoval grafy a obsah vzorce buňky v prohlížeči bez jediného náznaku zpomalení nebo úniku paměti (alokované paměti, která nebyla korektně uvolněna).

Nicméně, konec konců šlo o demo, když jsme poslali větší objem dat do EtherCalc a začali psát vzorce do tabulek v prohlížeči, server se najednou uzamkl a zamrzla všechna aktivní spojení. Restartovali jsme Node.js proces, jenom abychom zjistili, že spotřebovává 100 % CPU a za chvíli se znovu uzamkl.

Vyvedeni z míry jsme se vrátili zpátky k menšímu datovému vzorku, který pracoval správně a umožnil nám dokončit demo. Neustále jsem však přemýšlel nad tím, co skutečně způsobilo uzamčení.

2.5 Profilování Node.js

Abychom zjistili, proč byl procesor tak vytížený, potřebovali jsme profilovat náš kód (zjistit, které části kódu nejvíce zatěžují procesor).

Profilování prvotního Perl prototypu bylo velmi přímočaré, zejména díky proslulému NYTProf profilovači, který poskytuje časové informace pro jednotlivé funkce, řádky, operační kód a bloky, s detailní grafickou vizualizací volání a HTML reportů. Kromě NYTProf jsme sledovali dlouho běžící procesy v Perlu zabudovaném podpůrném modulu DTrace, abychom tak získali statistiky vstupu a výstupu z funkcí v reálném čase.

Naproti tomu profilovací nástroje pro Node.js nejsou zatím dostupné. V době psaní této knihy je podpora DTrace stále limitovaná na operační systém illumos v 32 bitovém módu procesoru, a tak jsme se hlavně spoléhali na Node Webkit Agenta, který poskytuje dostupné profilovací rozhraní, ačkoliv jenom se statistikami na úrovni funkcí.

Typická profilovací relace vypadá takto:

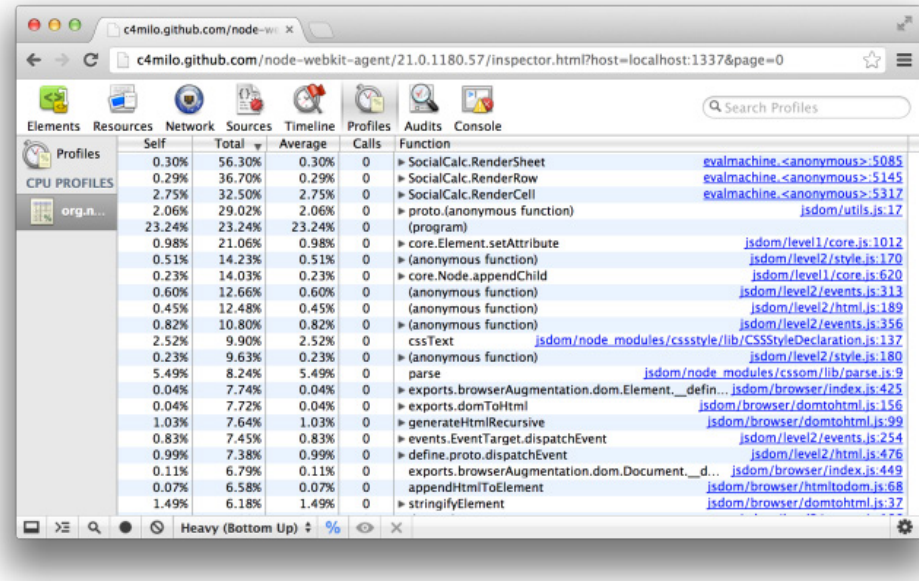
```
# "lsc" is the LiveScript compiler
# Load WebKit agent, then run app.js:
lsc -r webkit-devtools-agent -er ./app.js
# In another terminal tab, launch the profiler:
killall -USR2 node
# Open this URL in a WebKit browser to start profiling:
open http://tinyurl.com/node -8-agent
```

Abychom znovu vytvořili velkou zátěž s velkým počtem požadavků, provedli jsme mnoho souběžných volání REST API funkcí s nástrojem Apache ab. Pro simulaci operací na straně prohlížeče, jako například pohyb kurzoru a aktualizace vzorců, jsme použili Zombie.js, který poskytuje simulované prostředí pro testování JavaScriptu a je postavený na jsdom a Node.js. Paradoxně se úzké místo objevilo v samotném jsdom.

Z reportu na Obrázku 2.6 můžeme vidět, že RenderSheet dominuje při využití CPU. Pokaždé, když server přijme příkaz, zabere několik mikrosekund překreslení innerHTML buněk, aby se projevil efekt každého příkazu.

Protože celý jsdom kód běží v jednom vlákně, následující REST API volání jsou blokovaná až do dokončení předcházejícího příkazu na vykreslení. Při velké souběžných příkazů způsobí jejich fronta latentní chybu, která nakonec skončí uzamčením serveru.

Když jsme tento problém více zkoumali, uvědomili jsem si, že vlastně na straně serveru nepotřebujeme nic v reálném čase vykreslovat.Vždy můžeme rekonstruovat vykreslení innerHTML v každé buňce ze struktury tabulky v paměti pro odkaz získaný přes HTML export API.



Obrázek 2.6: Snímek obrazovky profilovače (s jsdom)

A tak jsme odstranili jsdom z funkce RenderSheet, znovu jsme implementovali minimální DOM ve 20 řádcích LiveScript[2] pro HTML export, a poté znovu spustili profilování kódu (viz Obrázek 2.7).

Mnohem lepší! Zlepšili jsme propustnost o faktor 4, export HTML je 20krát rychlejší a problém uzamčení zmizel.

2: https://github.com/audreyt/ethercalc/commit/fc62c0eb#L1R97

2.6 Vícejádrové škálování

Po tomto kole zlepšení jsme se konečně cítili dostatečně spokojeni a připraveni na to, abychom integrovali EtherCalc do platformy Socialtext, umožňující souběžnou editaci wiki stránek a tabulek.

Abychom zajistili rozumnou dobu odezvy v produkčních prostředích, nasadili jsme reverzní proxy server nginx a použili jeho příkaz limit_req[3], abychom se dostali na horní limit rychlosti volání API. Tato technika se osvědčila jak pro scénář za firewallem, tak pro scénář hostingové dedikované instance.

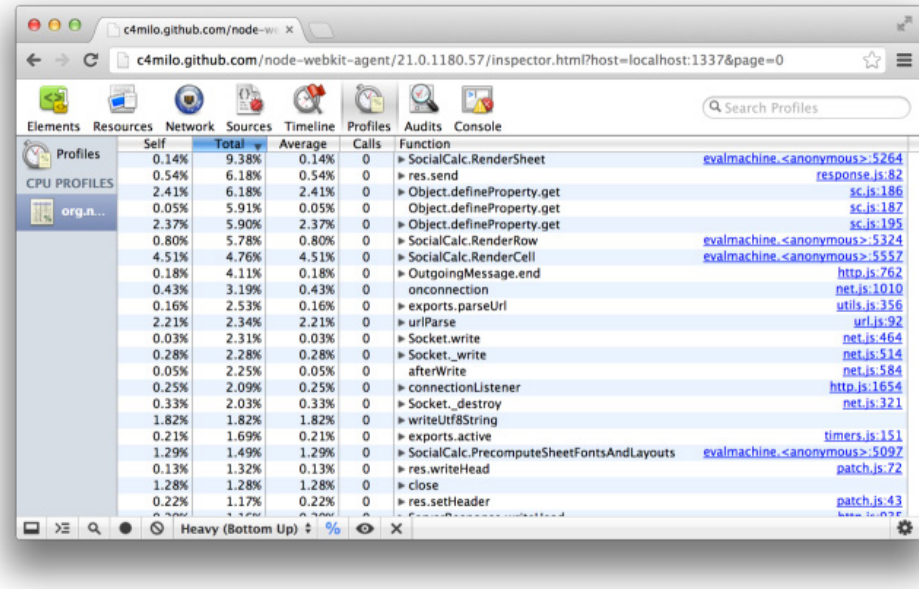
Pro zákazníky z malých a středních firem umožňuje Socialtext i třetí možnost nasazení: víceuživatelský hosting. Jeden velký server hostuje víc jak 35 000 společností, každá má průměrně 100 uživatelů.

V tomto víceuživatelském scénáři je rychlostní limit sdílen všemi zákazníky volajícími REST API. To dále zvyšuje efektivní limit pro každého klienta – kolem 5 požadavků za sekundu. Jak bylo uvedeno v předcházející části, tato limitace je způsobena tím, že Node.js používá jenom jedno procesorové jádro pro všechny výpočty.

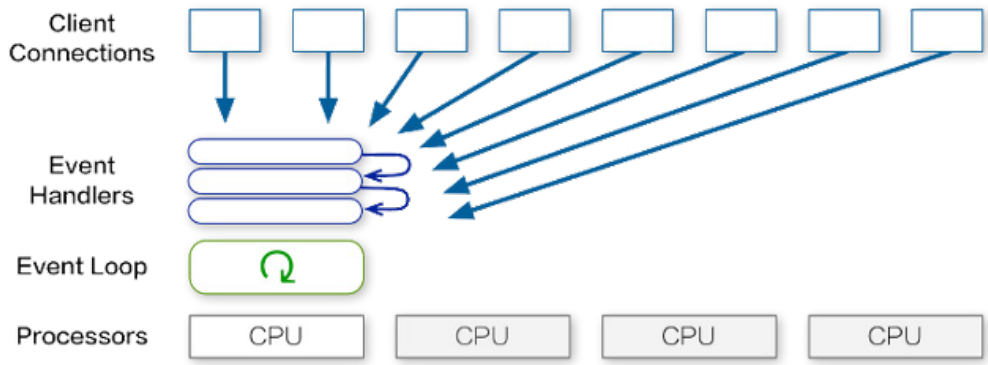
Existuje nějaký způsob, jak použít všechna procesorová jádra v multiuživatelském serveru?

Pro další služby Node.js běžící na vícejádrových hostových serverech jsme použili předem spuštěné procesy klastrového server, který vytváří proces pro každé jádro procesoru.

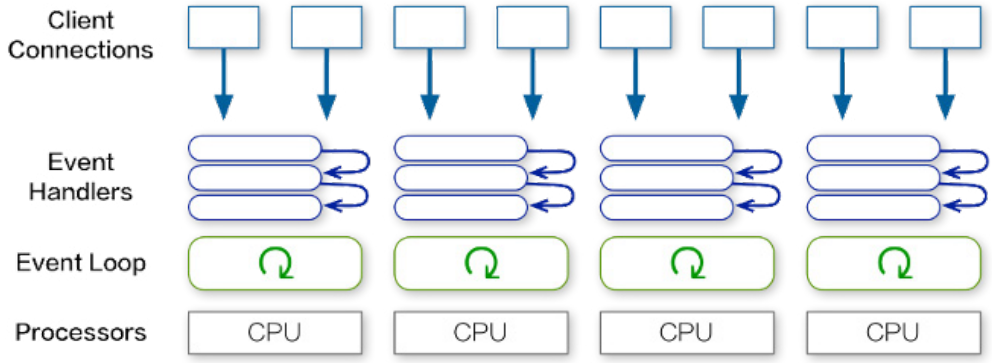
3: http://wiki.nginx.org/HttpLimitReqModule#limit_req



Obrázek 2.7: Aktualizovaný snímek profilovače (bez jsdom)



Obrázek 2.8: Událostní server (jedno jádrový)

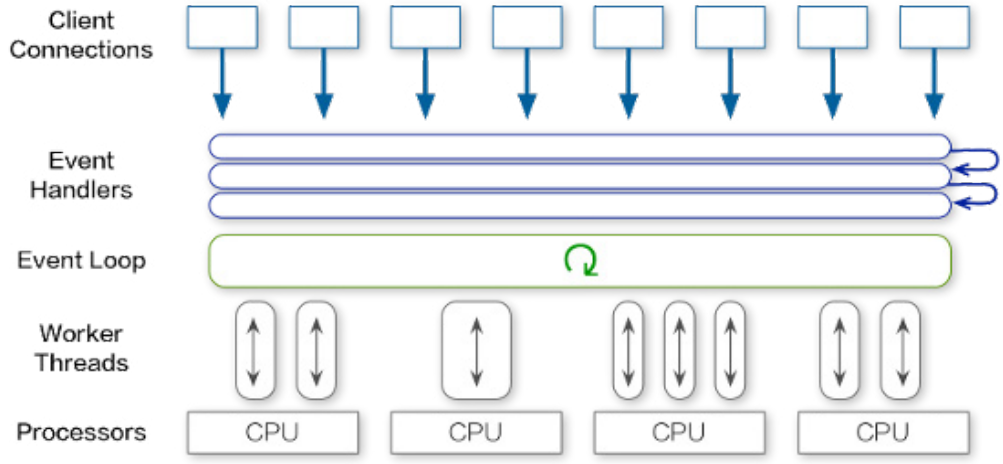


Obrázek 2.9: Událostní klastrový server (více jádrový)

Nicméně, zatímco EtherCalc podporuje více serverové škálování prostřednictvím Redis-u, nasadit Socket.io klastrování spolu s RedisStore na jednom serveru by hodně zkomplikovala celou logiku našeho systému a jehož ladění by se tak stalo mnohem komplikovanějším.

Navíc, pokud by byly všechny procesy v klastru by byly omezeny na jedno procesorové jádro, následující spojení by se stejně zablokovala.

Místo spouštění předem daného počtu počtu procesů jsme hledali způsob, jak vytvořit jedno vlákno na pozadí pro každou tabulku na straně serveru a distribuovat tak práci vykonávání příkazů mezi všechna procesorová jádra.



Obrázek 2.10: Událostní vláknový server (více jádrový)

Pro naše účely se perfektně hodil W3C Web Worker API. Původně byl zamýšlený pro prohlížeče a definuje tak způsob, jak nezávisle spustit skripty na pozadí. To umožňuje, aby byly rozsáhlé úlohy vykonány samostatně, aniž by zatěžovaly hlavní vlákno.

Takže jsme vytvořili webworker vlákna jako implementaci Web Worker API pro Node.js napříč více platformami.

Použitím webworker vláken lze přímo vytvořit nové vlákno SocilCalc a komunikovat s ním:

```
{ Worker } = require \webworker-threads
w = new Worker \packed-SocialCalc.js
w.onmessage = (event) -> ...
w.postMessage command
```

Toto řešení poskytuje to nejlepší z obou světů: dává nám svobodu přidělit EtherClac více strojového času procesoru kdykoliv je to potřebné, a režie na vytvoření vlákna v pozadí zůstává zanedbatelná v prostředí s jedním procesorovým jádrem.

2.7 Ponaučení

Omezení jsou osvobozující

Fred Brooks v jeho knize The Design of Design (Design designu) tvrdí, že zmenšení prostoru designéra pro hledání může pomoci soustředit se a urychlit proces návrhu. To zahrnuje dobrovolně přijaté omezení:

„Umělá omezení pro úlohu designéra mají tu pěknou vlastnost, že je možné si je svobodně užívat. V ideálním případě tlačí do neprozkoumaného koutu designu a stimulují kreativitu.“

V průběhu vývoje systému EtherCalc byla taková omezení nezbytná pro udržení koncepční integrity po celou dobu různých iterací.

Například by se mohlo zdát, že je dobré přijmout tři různé souběžné architektury, každou ušitou na míru jedné z variant serveru (za firewallem, v cloudu a multiuživatelský hosting). Nicméně každá taková ukvapená optimalizace by vážně narušila koncepční integritu.

Místo toho jsem soustředila pozornost na to, aby byl systém EtherCalc výkonný bez kompromisů na požadavky jednoho zdroje na úkor druhého, čímž se minimalizovalo použití strojového času procesoru, RAM a síťového přenosu současně. Jelikož je požadováno méně než 100 MB RAM, můžou dokonce i vestavěné platformy jako Raspberry Pi lehce hostovat.

Tato dobrovolně přijatá omezení umožňují nasadit EtherCalc v PaaS prostředí (např. DotCloud, Nodejitsu a Heroku), kde místo jednoho zdroje jsou omezeny všechny tři. Proto je velmi snadné vytvořit si osobní služby tabulkového procesoru pro běžné lidi, což podnítí více příspěvků od nezávislých integrátorů.

Nejhorší je nejlepší

Na konferenci YAPC::NA 2006 v Chicago jsem byla pozvaná, abych předpověděla vývoj open-source prostředí, a zde je můj příspěvek[4]:

„Myslím si, ale nemohu to zatím dokázat, že příští rok se stane JavaScript 2.0 sám bootstrapem, kompletně samo-hostingovým, se zpětným kompilátorem do JavaScriptu 1, a nahrazení Ruby se stane, další ‚velkou věcí' pro všechna prostředí.

Myslím si, že CPAN a JSAN se sloučí; JavaScript se stane společným backendem pro všechny dynamické jazyky, takže můžete psát Perl pro spuštění v prohlížeči, na serveru a uvnitř databází, to vše se stejnou sadou vývojových nástrojů.

Vzhledem k tomu, že víme, že horší je lepší, je ten nejhorší skriptovací jazyk odsouzen stát se nejlepším.“

4: Viz http://pugs.blogs.com/pugs/2006/06/my_yapcna_light.html

Vize se proměnila v realitu kolem roku 2009 s příchodem nových JavaScript jader běžících rychlostí nativních strojových instrukcí. V době psaní této knihy se stal JavaScript virtuálním strojem typu „Napiš jednou, spusť kdekoli“, ostatní hlavní jazyky do něho můžou kompilovat téměř bez jakékoliv výkonnostní penalizace.

Kromě prohlížečů na straně klienta a Node.js na serveru dosáhl JavaScript také pokroku v oblasti databáze Postgres a těší se zde velké sbírce volně a opakovaně použitelných modulů sdílených těmito runtime prostředími.

Co umožnilo takový náhlý růst v komunitě? V průběhu vývoje systému EtherCalc, díky účasti v začínající NPM komunitě jsem si uvědomila, že to je právě proto, že JavaScript předepisuje jen velmi málo a sám se hodí k různým účelům, takže se inovátoři mohou soustředit na slovník a idiomy (např. jQuery a Node.js), každý tým může abstrahovat jejich vlastní „dobré části“ z obecného, volného jádra.

Novým uživatelům se nabízí velmi jednoduchá dílčí sada pro začátek; zkušení vývojáři stojí před výzvou vyvinout lepší konvence, než ty stávající. Místo spoléhání se na jádro týmu designérů, aby dokončili jazyk tak, aby byl vhodný pro všechny předpokládané možnosti užití, vývoj JavaScriptu od základů rezonuje s dobře známou předpovědí Richarda P. Gabriela „Horší je lepší“.

LiveScript, Redux

Na rozdíl od přímočaré Perl syntaxe Coro:: AnyEvent, zpětné volání na bázi API Node.js vyžaduje hluboce vnořené funkce, které je těžké opětovně použít.

Poté, co jsem experimentovala s různými knihovnami pro řízení běhu programu, nakonec jsem vyřešila tento problém tím, že jsem se rozhodla pro LiveScript, nový jazyk, který se kompiluje do JavaScriptu, se syntaxí silně inspirovanou Haskellem a Perlem.

Ve skutečnosti byl systém EtherCalc portován přes linii čtyř jazyků: Javascript, CoffeeScript, Coco a LiveScript. Každá iterace přináší více expresivity, při zachování plné zpětné a dopředné kompatibility, a to díky aktivitám jako jsou js2coffee a js2ls.

Vzhledem k tomu, že se LiveScript kompiluje do JavaScriptu, než aby se interpretoval jeho vlastní kód, zůstává plně kompatibilní se stávajícími nástroji pro profilování kódu. Vygenerovaný kód je stejně výkonný jako ručně vyladěný JavaScript v moderním běhovém prostředí.

Co se týče syntaxe, LiveScript eliminoval vnořená zpětná volání s novými konstrukcemi, jako jsou zpětná volání a kaskády. To nám poskytuje výkonné syntaktické nástroje pro funkční a objektově orientovanou kompozici.

Když jsem se poprvé setkala s LiveScriptem, označila jsem ho jako „menší jazyk v rámci Perlu 6, snažící se dostat se ven“. Cíle bylo dosaženo mnohem snáze tím, že přijal stejnou sémantiku jako samotný JavaScript a zaměřil se výhradně na syntaktickou ergonomii.

Závěr

Na rozdíl od projektu SocialCalc s dobře definovanou specifikací a procesy týmového vývoje, EtherCalc byl od poloviny roku 2011 až do konce roku 2012 sólo experimentem a sloužil jako základ pro posouzení připravenosti Node.js k produkčnímu použití.

Tato neomezená volnost dopřála vzrušující příležitost prozkoumat širokou škálu alternativních jazyků, knihoven, algoritmů a architektury. Jsem velmi vděčna všem přispěvatelům, spolupracovníkům a integrátorům, zejména Danovi Bricklinovi a mým kolegům z projektu Socialtext za povzbuzování, abych experimentovala s těmito technologiemi. Děkuji, přátelé!

3 Ninja

(Evan Martin)

3 Ninja

Ninja je sestavovací systém podobný příkazu Make. Jako jeho vstup popíšete příkazy potřebné ke zpracování zdrojových souborů do cílových souborů. Ninja používá tyto příkazy, aby vytvořil aktuální cílové soubory. Na rozdíl od mnoha jiných sestavovacích systémů byla hlavním cílem návrhu systému Ninja rychlost.

Systém Ninja jsem napsal při práci na Google Chrome. Začal jsem Ninju jako experiment, abych zjistil, zda by mohlo být sestavení prohlížeče Chrome rychlejší. Když jsem chtěl úspěšně vytvořit Chrome, vznikl další hlavní cíl návrhu Ninji: Ninja potřeboval být snadno vnořený do většího sestavovacího systému.

Ninja byl dostatečně úspěšný, postupně nahrazoval další sestavovací systémy používané prohlížečem Chrome. Po té, co byl Ninja vytvořen, byl zveřejněn jiný příspěvkový kód k vytvoření oblíbeného CMake sestavovacího systému, generujícího Ninja soubory – nyní Ninja umožňuje vytvářet také projekty na CMake bázi, jako je LLVM a ReactOS. Další projekty, jako TextMate, mají Ninju jako podporovaný cíl v uživatelském sestavení.

Pracoval jsem na Chrome od roku 2007 do roku 2012, a práci na Ninja jsem začal v roce 2010. Existuje mnoho faktorů, které přispívají k výkonnosti sestavování tak velkého projektu, jako Chrome (dnes kolem 40 000 souborů v C ++ kódu, které generují výstupní binární kód o velikosti kolem 90 MB). Během mého působení jsem se zaobíral mnohými z nich, od distribuce kompilace mezi více počítači přes triky při propojování. Ninja se primárně soustředí na jeden z nich, na začátek sestavování. Jedná se o dobu mezi spuštěním sestavení až do okamžiku, kdy začíná běžet první kompilace. Abychom porozuměli tomu, proč je to důležité, je nutné pochopit, jak jsme uvažovali nad výkonností samotného prohlížeče Chrome.

3.1 Stručná historie prohlížeče Chrome

Diskuse o všech cílech Chrome je mimo rozsah této knihy, ale jedním ze základních cílů projektu byla rychlost. Výkon je obsáhlý cíl, který se rozpíná přes celou informatiku, a Chrome používá téměř každý trik, který je k dispozici, od ukládání do mezipaměti, přes paralelizaci až po kompilaci just-in-time. Pak zde hrála roli také rychlost spouštění - jak dlouho trvalo programu, než se zobrazil na obrazovce po kliknutí na ikonu - která se ale zdá trochu nepodstatná.

Proč záleží na rychlosti spouštění? Rychlé uvedení do provozu vyvolává u prohlížeče pocit jednoduchosti, že práce na webu je triviální, tak jako třeba otevření textového souboru. Dále je v oblasti interakce člověka s počítačem dobře prostudovaný vliv zpoždění na vaši radost a na ztrátu toku myšlenek.

Na zpoždění jsou zaměřené internetové společnosti jako Google či Amazon, které mají dobré předpoklady k měření a experimentování vlivu zpoždění, a které provádějí experimenty ukazující, že zpoždění i v milisekundách má měřitelné vlivy na to, jak často lidé používají web nebo přes něj nakupují. Je to malá frustrace, která se ale podvědomě sčítá.

Přístup Chrome k rychlému spuštění byl chytrým trikem jednoho z prvních inženýrů na projektu Chrome.

Jakmile dostali kostru jejich aplikace do bodu, kdy se ukázalo okno na obrazovce, vytvořili benchmark pro měření rychlosti, aby měřil rychlost spouštění v procesu kontinuálního budování platformy. Slovy Brett Wilsonové „velmi jednoduché pravidlo: tento test se nemůže nikdy zpomalit".[1] Jak byl přidáván kód do prohlížeče Chrome, udržování tohoto benchmarku vyžadovalo úsilí[2] inženýrů navíc - v některých případech byla práce odložena do doby, než bylo skutečně potřeba udělat ji, nebo byly přepočteny údaje použité při startu, ale primární „Trik" s výkonem, a ten, který na mě udělal největší dojem, byl prostě dělat méně práce.

Připojil jsem se k týmu Chrome bez jakéhokoliv úmyslu pracovat na vytváření sestavovacích nástrojů. Mé zkušenosti a má volba byla platforma Linux, a já chtěl být Linuxový chlapík. Aby se omezil rozsah projektu, zpočátku byl pouze pro systém Windows; vzal jsem na sebe roli pomoct dokončit Windows implementaci, abych pak Chrome rozběhal na Linuxu.

Při zahájení práce na jiných platformách bylo první překážkou vytřídění sestavovacího systému. Z tohoto pohledu byl Chrome už velký (ve skutečnosti kompletní - Chrome pro Windows byl uvolněn v roce 2008 dříve, než začala jakákoliv portace), takže snaha přenést Windows sestavování z Visual Studia do jiného sestavovacího systému byla s probíhajícím vývojem celkově konfliktní. Připadalo mi to jako výměna základů už používané budovy.

Členové týmu Chrome přišli s inkrementálním řešením nazvaným GYP3[3], které by mohlo být použito pro generování sestavovacích souborů Visual Studio, jež už Chrome používá k sestavovacím souborům, a které by byly použity na jiných platformách. A to vždy pro jednu dílčí komponentu najednou.

Vstup do GYP je jednoduchý: požadovaný název výstupu je doprovázen textovými seznamy zdrojových souborů, příležitostně vlastní pravidlo jako „zpracuj každý IDL soubor pro vygenerování dodatečného zdrojového souboru", a některé podmíněné chování (například použij pouze některé soubory na některých platformách). GYP poté vezme tento vysokoúrovňový popis a vytváří pro danou platformu nativní sestavovací soubory.[4]

1: http://blog.chromium.org/2008/10/io-in-google-chrome.html

2: http://neugierig.org/software/chromium/notes/2009/01/startup.html

3: GYP zkratka pro Generate Your Projects.

4: Jedná se o stejný vzor používaný Autotools: Makefile.am je seznam zdrojových souborů, který je pak zpracován konfiguračním skriptem pro vygenerování konkrétnějších sestavovacích instrukcí.

Na Macu znamenaly „nativní sestavovací soubory“ soubory projektu Xcode. Nicméně na Linuxu nebyla žádná zřejmá jediná volba. Počátečním pokusem bylo použít Scons sestavovací systém, ale byl jsem zdrcen zjištěním, že GYPem generovanému Scons sestavení může trvat spuštění i 30 sekund, protože Scons zjišťuje, které soubory se změnily. Spočítal jsem, že Chrome má zhruba velikost Linuxového jádra, takže přístup použitý u Linuxu by měl fungovat.

Vyhrnul jsem si rukávy a napsal jsem kód, aby GYP generoval prosté Makefiles za použití triků z Makefiles pro Linuxové jádro.

Tak jsem se nechtěně začal propadat do šílenství sestavovacího systému. Existuje mnoho faktorů, které zaberou čas při vytváření sestavovacího softwaru, od pomalých linkerů až po špatnou paralelizaci, a já se potýkal se všemi z nich. Makefile přístup byl zpočátku docela rychlý, ale jak jsme přenášeli více z Chrome do Linuxu, rostoucí počet souborů použitých v průběhu sestavování způsobil jeho zpomalení.[5]

Při práci na přenosu jsem zjistil, že jedna část procesu sestavení je zejména frustrující: udělal jsem změnu jednoho souboru, spustil příkaz make, uvědomil si, že jsem vynechal středník, znovu jsem spustil příkaz make, a čekání bylo pokaždé tak dlouhé, že jsem zapomněl, co jsem vlastně dělal. Vzpomněl jsem si, jak tvrdě jsme bojovali se zpožděním u koncového uživatele. „Jak je možné, že to trvá tak dlouho,“ zajímalo mě, „to nemůže být přece tak moc práce.“ Pokusně jsem začal s Ninjou, abych zjistil, jak bych to mohl udělat jednodušeji.

3.2 Design systému Ninja

Zjednodušeně řečeno, každý sestavovací systém plní tři hlavní úkoly. Načítá (1) a analyzuje sestavovací cíle, (2) zjišťuje, jaké kroky musí udělat za účelem dosažení těchto cílů, a (3) musí provést tyto kroky.

Aby bylo spuštění prvního kroku (1) rychlé, Ninja musel udělat nějaké minimální množství práce při načítání sestavovacích souborů. Sestavovací systémy obvykle používají lidé, což znamená, že zadají vhodnou vysokoúrovňovou syntaxi pro vyjádření sestavovacích cílů. To také znamená, že když přijde čas, aby sestavovací systém skutečně sestavil projekt, musí zpracovat i další instrukce: například v určitém okamžiku se Visual Studio musí konkrétně rozhodnout, na základě konfigurace sestavení, kam má umístit výstupní soubory, nebo které soubory musí být sestaveny s C ++ nebo C kompilátorem.

Z tohoto důvodu byla práce GYP při generování Visual Studio souborů efektivně omezena na překládání seznamu zdrojových souborů do syntaxe Visual Studia a přenechání Visual Studiu dělat většinu práce.

5: Chrome sám o sobě také rychle roste. V současné době roste tempem kolem 1000 commitů týdně, z nichž většina jsou přírůstky kódu.

U Ninji jsem viděl příležitost udělat tolik práce, kolik je jen možné, v GYP. V jistém smyslu platí, že když GYP generuje Ninja sestavovací soubory, vykoná všechny výše uvedené výpočty jedenkrát. GYP pak uloží snímek tohoto přechodného stavu do formátu, který Ninja může rychle načíst pro každé následující sestavování.

Jazyk Ninja sestavovacího souboru je jednoduchý až do té míry, že pro lidi je nepohodlné v něm psát. Neexistují žádná podmiňovací pravidla nebo pravidla na základě rozšíření souboru. Místo toho je formát jenom seznamem, ze kterého vytvoří přesně souborové cesty pro konkrétní výstupy. Tyto soubory můžou být načteny rychle, což nevyžaduje téměř žádnou interpretaci.

Tento minimalistický design neintuitivně vede k větší flexibilitě. Vzhledem k tomu, že Ninja postrádá znalost obecných sestavovacích konceptů na vyšší úrovni, jako je výstupní adresář nebo aktuální konfigurace, lze Ninju jednoduše zapojit do větších systémů (např. CMake, jak jsme zjistili později), které mají odlišné názory na to, jak by měl být organizován proces sestavení. Například Ninja je agnostický v otázce, zda jsou sestavené výstupy (např. objektové soubory) umístěny vedle zdrojových souborů (některými považováno za špatnou hygienu), nebo v samostatném výstupním adresáři pro sestavení (považováno pro jiné za těžko pochopitelné). Dlouho po uvolnění Ninja jsem konečně našel správnou metaforu: zatímco ostatní sestavovací systémy jsou kompilátory, Ninja je assembler.

3.3 Co Ninja dělá

Pokud Ninja přesouvá většinu práce do generátoru na sestavení souborů, co zbývá ještě udělat? Uvedená ideologie je v zásadě pěkná, ale potřeby skutečného světa jsou vždy složitější. Vlastnosti systému Ninja se vytvářely (a také ztrácely) v průběhu jeho vývoje. V každém okamžiku byla vždy důležitá otázka „Můžeme dělat méně?“ Zde je stručný přehled o tom, jak funguje.

Lidi potřebují odladit tyto soubory když jsou sestavovací pravidla chybná. Proto jsou Ninja sestavovací soubory prostým textem, podobně jako Makefiles, a podporují několik abstrakcí, aby byly lépe čitelné.

První abstrakce je „pravidlo“, které představuje vyvolání jediného nástroje z příkazového řádku. Pravidlo se pak sdílí mezi různými kroky sestavení. Zde je příklad syntaxe Ninja pro deklarování pravidla s názvem „kompilace“, která spouští kompilátor gcc spolu se dvěma build příkazy, jež se použijí pro specifické soubory.

```
rule compile
command = gcc -Wall -c $in -o $out
build out/foo.o: compile src/foo.c
build out/bar.o: compile src/bar.c
```

Druhá abstrakce je proměnná. Ve výše uvedeném příkladu se jedná o identifikátor s dolarem v prefixu ($in a $out). Proměnné mohou reprezentovat jak vstupy, tak výstupy příkazu a mohou být použity k vytvoření krátkých názvů pro dlouhé řetězce. Zde je rozšířena definice kompilace, která používá proměnné pro příznaky kompilátoru:

```
cflags = -Wall
rule compile
command = gcc $cflags -c $in -o $out
```

Hodnoty proměnné použité v pravidle mohou být překryty v rámci jednoho build bloku odsazením jejich nové definice. Pokračováním výše uvedeného příkladu, hodnota cflags může být nastavena pro jeden soubor:

```
build out/file_with_extra_flags.o: compile src/baz.c
cflags = -Wall -Wextra
```

Pravidla se chovají téměř jako funkce a proměnné se chovají jako argumenty. Tyto dva jednoduché rysy jsou nebezpečně blízko k programovacímu jazyku - protiklad k cíli „nedělat žádnou práci". Ale mají významnou výhodu v tom, že redukují opakovaní řetězců, což není výhodné pouze pro člověka, ale také pro počítače, protože se snižuje množství textu, který je parsován.

Zparsovaný sestavovací soubor popisuje graf závislostí: finální binární výstup závisí na propojení s celou řadou objektů, z nichž každý je výsledkem kompilace zdrojů. Specificky se jedná o bipartijní graf, kde „uzly" (vstupní soubory) poukazují na „hrany" (sestavovací příkazy), které poukazují na uzly (Výstupní soubory).[6] Proces sestavení pak prochází tento graf.

Pro daný cílový výstup pro sestavení Ninja nejdříve projde graf, aby identifikovat hrany každého vstupního souboru: to znamená, zda vstupní soubory existují a jaké jsou kdy byly naposledy změněny. Ninja pak spočítá plán. Plán je množina hran, které musí být provedeny s cílem aktualizovat finální cíl, podle časů modifikace dočasných souborů. Nakonec je plán vykonán procházením grafu a kontrolou hran, zda byly vykonány a úspěšně dokončeny.

Jakmile vše proběhlo, mohl jsem vytvořit základní benchmark pro Chrome: čas na další spuštění Ninji po úspěšném dokončení sestavení. Je to čas k načtení sestavovacích souborů, vyhodnocení stavu sestavení, a zjištění, zda je potřeba něco dělat. Doba běhu tohoto benchmarku byla těsně pod jednu sekundu. To byla moje nová hranice benchmarku pro spuštění. Nicméně, jak rostl Chrome, musel být i Ninja stále rychlejší, aby si udržel tuto hranici.

6: Tato extra bezcílnost umožňuje sestavení správného modelu příkazů, které mají více výstupů.

3.4 Optimalizace Ninja

Počáteční implementace Ninja optimalizovala datové struktury tak, aby bylo možné rychlé sestavení, ale nebyla nijak zvlášť chytrá z hlediska dalších optimalizací. Jakmile program fungoval, uvažoval jsem, že by profilování mělo odhalit, na které části kódu se mám dále zaměřit.[7]

Za ta léta ukazovalo profilování na různé části programu. Někdy tím nejhorším pachatelem byla jedna nejnovější funkce, kterou bylo možno mikro-optimalizovat. Jindy byla problémem alokace paměti a kopírování paměti. Byly také případy, kdy mělo význam přepsat funkci nebo kdy výkon zásadně ovlivňovala konkrétní datová struktura.

Dále bude následovat procházka implementací systému Ninja a některé zajímavé příběhy o jeho výkonnosti.

Parsování

Zpočátku Ninja používal ručně psaný lexer a rekurzivní sestupní parser. Podle mého názoru byla syntaxe dost jednoduchá. Ukázalo se, že pro poměrně velký projekt, jako je Chrome[8], může jednoduchá parsování sestavovacích souborů (s příponou .ninja v názvu) trvat až překvapivě hodně času.

Původní funkce pro analýzu jednoho znaku se brzy objevila v profilech:

```
static bool IsIdentifierCharacter(char c) {
return
('a' <= c && c <= 'z') ||
('A' <= c && c <= 'Z') ||
// and so on...
}
```

Jednoduchá oprava - úspora času 200 ms - tuto funkci jsme nahradili vyhledávací tabulkou s 256 vstupy, které by mohly být indexovány na základě vstupního znaku. Takovou věc je triviální vygenerovat pomocí kódu Python, jako je:

```
cs = set()
for c in string.ascii_letters + string.digits + r'+,-./\_$':
cs.add(ord(c))
for i in range(256):
print '%d,' % (i in cs),
```

7: Ninja má velkou testovací sadu 164 testovacích případů, která sama o sobě běží za méně než sekundu, což znamená, že vývojáři mohou mít důvěru v to, že změny výkonnosti neovlivní správnost programu.

8: Dnes sestavení Chrome generuje více jak 10 MB .ninja souborů.

Tento trik udržel po nějakou dobu systém Ninja rychlým. Nakonec jsme se přesunuli na něco principiálnějšího: na lexikální generátor re2c používaný jazykem PHP. Ten může vytvářet složitější převodní tabulky a stromy nesrozumitelného kódu. Například:

```
if (yych <= 'b') {
     if (yych == '`') goto yy24;
     if (yych <= 'a') goto yy21;
     // and so on...
```

Zůstává však nezodpovězenou otázkou, zda je dobrým nápadem mít textový vstup. Možná, že nakonec budeme potřebovat vygenerovat vstupní soubory do Ninja v strojově přívětivém, binárním formátu, který by nám umožnil vyhnout se z větší části parsování.

Převod do kanonické formy

Ninja nepoužívá řetězce pro identifikaci cest. Místo toho Ninja mapuje každou cestu, narazí-li na unikátní Node objekt, a objekt Node se pak používá ve zbytku kódu. Užití tohoto objektu zajistí, že daná cesta je vždy kontrolována na disku jen jednou a výsledek této kontroly (tj. modifikace času) může být znovu použit v jiném kódu.

Ukazatel na objekt Node slouží jako jedinečná identita pro tuto cestu. Chcete-li otestovat, zda dva Node objekty odkazují na stejnou cestu, postačí porovnat ukazatele, než provádět nákladnější porovnání řetězců. Například, jak Ninja prochází graf vstupů do sestavení, udržuje zásobník závislých Node objektů pro kontrolu smyčky závislosti (tj. cyklického grafu): jestliže A závisí na B a B závisí na C a C závisí na A, nemůže sestavení pokračovat. Tento zásobník, který reprezentuje soubory, může být implementován pomocí jednoduchého pole ukazatelů a rovnost ukazatelů lze použít pro kontrolu duplicity.

Chcete-li vždy používat stejný Node objekt pro jeden soubor, musí Ninja spolehlivě mapovat všechny možné názvy souboru do stejného Node objektu. To vyžaduje převod do kanonické formy všech cest uvedených ve vstupním souboru, který transformuje cestu jako foo /../ bar.h do bar.h. Zpočátku systém Ninja požadoval, aby byly všechny cesty poskytovány v kanonické formě, ale to nakonec z několika důvodů nefungovalo. Jedním z nich je, že lze důvodně předpokládat, že uživatelem specifikované cesty (např. z příkazového řádku ninja ./bar.h) budou fungovat správně. Dalším důvodem je, že se mohou kombinovat proměnné a vytvořit nekanonické cesty. A nakonec i informace o závislosti emitované kompilátorem gcc mohou být nekanonické.

A tak většina toho, co systém Ninja nakonec dělá, je zpracování cesty, tudíž je převod cesty do kanonické formy dalším výkonnostně kritickým bodemprogramu. Původní implementace byla psána pro přehlednost, ne pro výkonost, takže standardní optimalizační techniky, jako odstranění dvojité smyčky nebo vyhnutí se přidělení paměti, výrazně pomohly.

Log sestavení

Výše uvedené mikro-optimalizace mají často menší dopad než strukturální optimalizace, u kterých změníte algoritmus nebo přístup. To byl i případ log sestavení systému Ninja.

Jedna část jádra sestavovacího systému Linuxu sleduje příkazy používané ke generování výstupů. Vezměme si motivující příklad: kompilujete vstupní foo.c na výstupní foo.o a potom změníte sestavovací soubor tak, že by měl být soubor znovu sestaven s odlišnými příznaky kompilace. Aby sestavovací systém věděl, že je třeba znovu sestavit výstup, musí se buď uvést, že foo.o závisí na samotném sestavovacím souboru (což by v závislosti na organizaci projektu mohlo znamenat, že změna v sestavovacím souboru by způsobila opětovné sestavení celého projektu), nebo zaznamenávat příkazy používané ke generování každého výstupu a porovnat je pro každou verzi.

Jádro (a následně Makefiles prohlížeče Chrome a systému Ninja) představuje druhý přístup. Zatímco se provádí sestavení, Ninja vytváří log, ve kterém zaznamenává veškeré příkazy používané k vygenerování každého výstupu[9]. Pak pro každé následující sestavení načte Ninja předchozí log sestavení a porovnává nové sestavovací příkazy se sestavovacími příkazy v logu, aby zjistil změny. Toto byl, stejně jako načtení sestavovacích souborů nebo kanonizace cest, další důležitý bod profilování.

Po několika menších optimalizacích implementoval Nico Weber, plodný přispěvatel systému Ninja, nový formát pro log sestavení. Ninja místo záznamů příkazů, které jsou často velmi dlouhé a zabere dost času parsovat je, zaznamená hash otisk příkazů. V následujících sestaveních porovnává Ninja hash otisk příkazu, který má být spuštěn, s hash otiskem příkazem v logu. Pokud se tyto dva hash otisky liší, je výstup zastaralý. Tento přístup byl velmi úspěšný. Použitím hash otisků se dramaticky snížila velikost logu sestavení - z 200 MB na méně než 2 MB na Mac OS X – a to umožnilo více než 20krát rychlejší načítání.

Soubory závislostí

Existuje další úložiště metadat, která musí být zaznamenána a použita napříč sestaveními. Chcete-li správně sestavit C / C ++ kód, musí sestavovací systém pojmout závislosti mezi hlavičkovými soubory. Předpokládejme, že foo.c obsahuje řádek #include „bar.h“ a samotný bar.h obsahuje řádek #include „baz.h“. Všechny tři tyto soubory (foo.c, bar.h, baz.h) pak ovlivňují výsledek kompilace. Například změny v baz.h by měly vyvolat opětovné sestavení foo.o.

Některé sestavovací systémy používají „skener hlaviček“ k extrakci těchto závislostí okamžiku sestavení, ale tento přístup může být pomalý a jeho správnost je za přítomnosti direktivy #ifdef obtížná. Jinou alternativou je požadovat, aby sestavovací soubory správně zaznamenávaly všechny závislosti, včetně hlaviček, což je ale obtížné pro vývojáře: pokaždé, když budete přidávat nebo odebírat příkaz #include, je třeba upravit nebo znovu provést sestavení.

9: To také ukládá, kdy se každý příkaz spustí a ukončí, což je užitečné pro profilování sestavení z mnoha souborů.

Lepší přístup se opírá o skutečnost, že v době kompilace může kompilátor gcc (nebo Microsoft Visual Studio) poskytnou výstupní informace o tom, které hlavičky byly použity k sestavení výstupu. Tyto informace, podobně jako příkaz použitý ke generování výstupu, lze zaznamenat a znovu načíst pomocí sestavovacího systému, což umožňuje přesné vysledování těchto závislostí. U prvního sestavení budou ještě předtím, než existuje nějaký výstup, sestaveny všechny soubory, takže není nutná jakákoliv závislost hlaviček. Po první kompilaci způsobí modifikace výstupem kteréhokoli používaného souboru (včetně modifikace, které přidávají nebo odebírají další závislosti) opětovné sestavení, a to uchováváním aktuálních informací o závislostech.

Při sestavování zaznamenává kompilátor gcc závislosti hlaviček ve formátu Makefile. Systém Ninja má parser pro (zjednodušená podmnožina) Makefile syntaxi a načte všechny tyto informace o závislostech při příštím sestavení. Načítání těchto údajů je hlavním slabým místem. V nedávném sestavení Chrome dosáhly informace o závislostech produkované gcc 90 MB ve formátu Makefile, přičemž všechny referenční cesty musí být kanonizované před použitím.

Stejně jako u jiných parsování, výkonnosti pomohlo použití re2c a vyhýbání se kopiím jak jen to bylo možné. Nicméně, přesunutím do GYP se parsování dalo posunout na později, než aby bylo na kritické cestě pro spuštění. Naší poslední prací na Ninja (v době psaní této knihy je tato funkce kompletní, ale dosud není uvolněna) bylo, aby proběhla co nejdříve během sestavení.

Jakmile Ninja začne provádět sestavovací příkazy, všechny výkonově kritické práce jsou dokončeny a Ninja je většinou nečinný, neboť čeká na příkazy pro dokončení. V tomto novém přístupu pro hlavičkové závislosti používá Ninja tento čas na zpracování Makefile souborů vytvořených kompilátorem gcc tak, jak jsou napsány, kanonizováním cest a zpracováním závislostí na rychle deserializovatelném binárním formátu. Pro další sestavení Ninja potřebuje pouze načíst tento soubor. Dopad je dramatický, zejména pokud jde o Windows. (O tom později v této kapitole.)

„Log závislostí“ potřebuje uložit tisíce cest a závislostí mezi těmito cestami. Načítání tohoto logu a jeho rozšiřování proto musí být rychlé. Připojení k tomuto logu by měla být bezpečná, a to i v případě přerušení, kterým je například zrušení sestavování.

Po zvážení mnoha databázových návrhů jsem konečně našel triviální implementaci: soubor je posloupnost záznamů a každý záznam je buď cesta, nebo seznam závislostí. Každé cestě zapsané do souboru je přiřazen identifikátor typu, sekvenční celé číslo. Závislosti jsou tak seznamy celých čísel. Chcete-li přidat závislosti do souboru, Ninja nejdříve zapíše nové záznamy pro každou cestu bez identifikátoru a pak zapíše záznam o závislosti použitím uvedených identifikátorů. Při načítání souboru za následného běhu může Ninja použít jednoduché pole pro mapování identifikátorů k jejich Node ukazatelům

Vykonání sestavení

Proces provádění příkazů s ohledem na výkonnost je nezbytné provádět podle závislostí uvedených výše. Je to poměrně nezajímavé, protože většinu práce, kterou je třeba provést, se provádí v příkazech (tj., v kompilátoru, propojovacích modulech atd.), nikoli v samotném[10] systému Ninja.

Ninja spouští sestavovací příkazy ve výchozím nastavení paralelně, na základě počtu dostupných procesorových jader v systému. Vzhledem k současnému běhů příkazů jsou jejich výstupy prokládané, Ninja proto ukládá veškeré výstupy příkazů do vyrovnávací paměti, dokud se příkaz nedokončí, aby je teprve poté vypsal. Výsledný výstup se pak objeví jako by byly příkazy provedeny sériově.[11]

Uvedená kontrola výstupu příkazu umožňuje systému Ninja pečlivě řídit jeho celkový výstup. Během sestavování zobrazuje Ninja jeden řádek o stavu; je-li sestavení úspěšně dokončeno, celkový zobrazený výstup systému Ninja je jediný řádek.[12] To sice neumožňuje rychlejší běh systému Ninja, ale vytváří pocit rychlosti, což je téměř stejně důležité vzhledem k původnímu cíli, jako skutečná rychlost.

Verze pro Windows

Systém Ninja jsem napsal pro Linux. Nico (zmíněn výše) dělal na tom, aby fungoval i na Mac OS X. Jak se Ninja začal více používat, lidé se začali ptát na verzi pro Windows.

Z prvotního pohledu nebyla verze pro Windows příliš těžká. Byly nutné některé jednoduché úpravy, jako změnit oddělovač v cestě na zpětné lomítko, nebo změna syntaxe Ninja, aby byly dvojtečky v cestě (například c: \ foo.txt). Poté, co byly uvedené změny zavedeny, vynořily se větší problémy. Ninja byl navržen tak, že předpokládal Linuxové chování; Windows je naproti tomu odlišný v malých, ale zato důležitých ohledech.

Například má systém Windows relativně malý limit pro délku příkazu, což je problém, který se vynoří při stavbě příkazu pro předání k finálnímu propojovacímu kroku, který lze uvést u většiny souborů v projektu. Řešením Windows pro tento problém jsou soubory typu „odpověď", a pouze Ninja (nikoliv generátor před Ninja) je umí zpracovat.

Mnohem důležitější výkonnostní problém je, že souborové operace ve Windows jsou pomalé a Ninja pracuje s mnoha soubory. Kompilátor Visual Studia zaznamenává hlavičkové závislosti pouhým tiskem při kompilaci, takže Ninja ve Windows v současné době obsahuje nástroj, kte-

10: Jeden menší přínos je to, že uživatelé v systémech s malým počtem procesorových jader si všimli, že jejich kompletní sestavení jsou rychlejší kvůli relativně malé výpočetní náročnosti systému Ninja při řízení sestavování, což uvolní jádro pro použití sestavovacích příkazů.

11: Nejúspěšnější sestavovací příkazy nemají žádný výstup, takže se to stane pouze tehdy, když více příkazů selže paralelně: jejich chybové zprávy se zobrazují sériově.

12: Původ názvu „Ninja": tichý a rychle udeří.

rý obalí kompilátor tak, že produkuje makefile soubory v stylu kompilátoru gcc se seznamem závislostí požadovaných systémem Ninja. Velký počet souborů, již zmíněné problémové místo v systému Linux, je mnohem horší ve Windows, kde trvá otevření souboru mnohem delší dobu. Výše uvedený nový přístup k parsování závislostí v okamžiku sestavování se perfektně hodí pro Windows, což nám umožňuje zcela upustit od použití přechodného nástroje: Ninja již ukládá výstup příkazu do vyrovnávací paměti, takže může parsovat závislosti přímo ve vyrovnávací paměti, vynecháním přechodného Mikefile na disku používaného gcc kompilátorem.

Získání času modifikace - GetFileAttributesEx () ve Windows[13] a stat () na jiných platformách – se zdá asi 100 krát pomalejší v systému Windows, než je tomu u Linux.[14] Je možné, že je tomu tak kvůli „nefér" faktorům, jako je antivirový software, ale v praxi tyto faktory existují v systémech koncových uživatelů, takže výkon systému Ninja trpí. Git verzovací řídicí systém, který rovněž potřebuje zjistit stav mnoha souborů, může použít více ve Windows vláken pro paralelní kontrolu souborů. Tuto funkcionalitu by bylo vhodné doplnit do systému Ninja.

3.5 Závěry a alternativní návrhy

Občas na mailing listu někdo navrhuje, že by Ninja měl fungovat jako v paměti rezidentní daemon nebo server, obzvláště v kombinaci s monitory pro modifikaci souboru (např. inotify na Linuxu). Všechny tyto obavy o dobu potřebnou k načtení a zapsání dat by nebyly problémem, pokud by Ninja byl použit jenom u sestavení.

Ve skutečnosti byl tento návrh mým původním plánem pro Ninja. Poté, co jsem viděl, že první sestavení pracuje rychle, uvědomil jsem si, že by bylo možné, aby Ninja pracoval bez nutnosti serverových komponent. To ale může být ještě v budoucnu potřebné, protože Chrome stále roste. Ale nejatraktivnější pro mě vždy bude jednodušší přístup, když získáme rychlost tím, že uděláme méně práce, než implementovat složitější technikuy. Mojí nadějí je, že budou stačit některé další restrukturalizace (jako úpravy, kdy jsme použili lexikální generátor nebo nový formát pro závislosti ve Windows).

Jednoduchost je v oblasti softwaru ctností; otázkou vždy je, jak daleko to může zajít. Systému Ninja se podařilo snížit složitost sestavování delegováním určitých nákladných úkolů na jiné nástroje (GYP nebo CMake), a z tohoto důvodu je vhodný i v jiných projektech než jen v tom, pro který byl vytvořen. Jednoduchý kód systému Ninja snad povzbudil přispěvatele - většina práce na podporu OS X, Windows, CMake a další funkce byla provedena přispěvateli. Jednoduchá sémantika systému Ninja vedla k dalším experimentům o implementaci v jiných jazycích (Scheme a Go, pokud je mi známo).

13: stat() ve Windows je ještě pomalejší než GetFileAttributesEx().

14: Toto platí, když je připravena disková mezipamět, takže by neměl být důležitý výkon disku.

Opravdu záleží na milisekundách? Mezi většími softwarovými starostmi se může zdát hloupé zajímat se o milisekundy. Avšak poté, co jsem pracoval na projektech s pomalejším sestavováním, zjistil jsem, že více produktivity jsem získal rychlou odezvou, která dává projektu pocit lehkosti, a dělá mi radost si s ním hrát. A kód, který je zábavné hackovat, je v první řadě důvodem, proč píšu software. V tomto smyslu má rychlost prvořadý význam.

3.6 Poděkování

Zvláštní dík patří mnohým přispěvatelům systému Ninja, z nichž některé si můžete najít na GitHub stránkách projektu Ninja.

4 Parsování XML rychlostí světla

(Arseny Kapoulkine)

4 Parsování XML rychlostí světla

4.1 Předmluva

XML je standardizovaný značkovací jazyk, který definuje soubor pravidel pro kódování hierarchicky strukturovaných dokumentů v čitelném textovém formátu. XML je hojně využíván, od velmi krátkých a jednoduchých dokumentů (jako jsou SOAP dotazy) po několika gigabytové dokumenty (OpenStreetMap) se složitými datovými vztahy (COLLADA). Za účelem zpracování XML dokumentů uživatelé obvykle potřebují speciální knihovnu: XML parser, který převádí dokument z textu na vnitřní vyjádření. XML je kompromisem mezi výkonem parsování, lidskou čitelností a složitostí kódu – proto rychlý XML parser umožní volbu XML jako preferovaného základního formátu pro datový model aplikace.

Tato kapitola popisuje různé výkonnostní triky, které umožnily autorovi napsat vysoce výkonný parser v C ++: pugixml. I když tyto techniky byly použity pro XML parser, většina z nich může být aplikována na parsery jiných formátů, nebo dokonce jiný software (např. algoritmy řízení paměti jsou široce použitelné i mimo parsery).

Vzhledem k tomu, že existuje několik podstatně odlišných přístupů k XML parsování, a parser musí udělat další zpracování, o kterém netuší ani lidi dobře obeznámení s XML zpracováním, před ponořením se do detailů implementace je důležité nejprve nastínit celý úkol.

4.2 Model XML parsování

Každý z různých modelů analýzy XML je vhodný v odlišných situacích, a každý z nich má vliv na výkon a spotřebu paměti. Tyto modely se široce používají:

- U SAX (Simple API pro XML) parserů uživatel předkládá parseru proud dokumentů jako vstup a callback funkcí „začátek tagu“, „konec tagu“, „znaková data uvnitř tagu“. Parser volá callback funkce podle údajů v dokumentu. Kontext potřebný pro provádění parsování je omezen hloubkou stromu aktuálního prvku, což znamená, že požadavky na paměť jsou mnohem nižší. Tento typ analýzy může být použit pro proudy dokumentu, kde je k dispozici pouze část dokumentu v každém daném okamžiku.

- Pull analýza je podobná SAX co se týče procesu analýzy – to znamená, že dokument je zpracován postupně po jednom prvku - ale řízení je obrácené: u SAX analýzy je parser řízen přes volání callback funkcí, zatímco u Pull analýzy uživatel řídí analytický proces přes objekt podobný iterátoru.

- U DOM (Document Object Model) parserů uživatel poskytuje parseru celý dokument jako textový proud/zásobník, z kterého parser generuje objekt dokumentu - zobrazení celého stromu dokumentu v paměti, který má jednotlivé objekty pro každý specifický XML element nebo atribut, a sadu povolených operací (např. „přečíst všechny podřízené prvky tohoto uzlu“). Pugixml následuje tento model.

Volba analytického modelu obvykle závisí na velikosti a struktuře dokumentu. Vzhledem k tomu, že pugixml je DOM parserem a je efektivní pro dokumenty, které:

- jsou dostatečně malé, aby se vešly do paměti,
- mají komplikovanou strukturu s odkazy mezi uzly, které je třeba přejít, nebo
- potřebují složité transformace dokumentů.

4.3 Designové volby v pugixml

Pugixml se zaměřuje na problematiku DOM analýzy z velké části proto, že zatímco rychlé a lehké SAX parsery (jako Expat) už byly k dispozici, všechny XML DOM parsery připravené k produkci v době tvorby pugixml (2006) buď nebyly dostatečně lehké, nebo příliš rychlé nebo standardně ani jedno. Proto hlavním cílem pugixml je být velmi rychlou a lehkou knihovnou pro manipulování s XML na bázi DOM.

XML je definován podle doporučení W3C, které specifikují dva odlišné typy analýzy: s validací a bez validace (jinými slovy, persery bez validace zkontrolují syntaxi XML, zatímco parser s validací kontroluje také datovou sémantiku). Dokonce i parser bez validace musí provést relativně náročnou validaci.

I když je výkon primárním cílem, musí být dosaženo kompromisu mezi výkonem a shodou. U pugixml je kompromis následující: každý správně vytvořený XML dokument bude plně parsován, včetně všech požadovaných transformací, s výjimkou dokumentu typu deklarace.[1] Vzhledem k tomu, že jsou kontrolována rychle ověřitelná pravidla, i nesprávně vytvořený dokument může být někdy úspěšně parsován.

Data v dokumentu XML mají být často nějakým způsobem transformována, než se dostanou k uživateli. Tyto transformace zahrnují zpracování konce řádku, normalizaci na základě hodnoty atributu a rozšíření znakové reference. Tyto transformace přinášejí dodatečné zpoždění; pugixml je optimalizuje tak, jak je to jen možné, a zároveň poskytuje možnost jejich vypnutí pro dosažení maximálního výkonu.

1: Dokumenty (DOCTYPE) typu deklarace jsou parsovány, ale jejich obsah je ignorován. Toto rozhodnutí je přijato na základě problémů s výkonností, implementační složitostí a poptávkou po této funkci.

Prvním úkolem je vytvořit rychlý DOM parser, který úspěšně parsuje odpovídající XML dokumenty s požadovanými transformacemi, a je tak rychlý, jak je to rozumně možné, a zároveň je připraven pro produkci. Pro účely výkonnosti znamená „připravený pro produkci" především to, že je odolný vůči nesprávně vytvořeným datům. Není možné obětovat kontroly na překročení vyrovnávací paměti ke zlepšení výkonnosti.

Dále budeme diskutovat o procesu analýzy použitém v pugixml. V poslední části se popisuje datová struktura používaná v pugixml k ukládání objektového modelu dokumentu a algoritmus použitý v pugixml k přidělení paměti pro uvedenou datovou strukturu.

4.4 Parsování

Cílem DOM parseru je vzít vstupní řetězec, který obsahuje XML dokument, a vytvořit strom objektů, který představuje tentýž dokument v paměti. Parser se obvykle skládá ze dvou fází: lexikální a parsovací. Vstupem lexeru je proud znaků a výstupem proud symbolů. (Pro XML parser může soubor symbolů obsahovat ostré závorky, uvozovky, názvy tagů a názvy atributů.) Parser zpracuje proud symbolů a vytváří syntaktický strom založený na gramatice, používající jeden z mnoha algoritmů parsování, jako je například rekurzivní sestup. Když se setká symbol s řetězcem dat, jako je tag pro jméno, lexer nebo parser zkopíruje obsah řetězce do haldy a uloží odkaz na řetězec uvnitř uzlu stromu.

Pro zlepšení výkonu parsování se pugixml liší od typických přístupů několika způsoby.

Proud symbolů vs. proud znaků

Jak již bylo zmíněno dříve, parsery tradičně využívají lexery k převedení proudu znaků do proudu symbolů. To může zlepšit výkon v případech, kdy musí parser provádět zpětné sledování, ale pro XML parser je lexikální etapa jen další vrstvou složitosti, která zvyšuje režii na jeden znak. Pugixml pracuje s proudem znaků místo symbolů.

Za normálních okolností se proud skládá z UTF-8 znaků a pugixml čte proud bajt po bajtu. Vzhledem k UTF-8 struktuře není nutné parsovat UTF-8 bajtovou sekvenci, pokud hledáte specifické ne-ASCII znaky, protože v platných UTF-8 proudech všechny bajty s hodnotou menší 128 jsou samostatné ASCII znaky.[2]

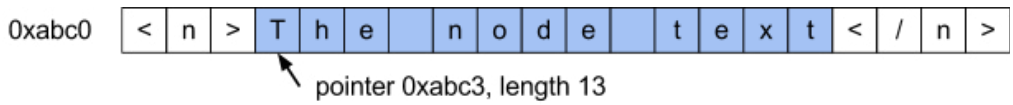
2: Všimněte si, že standardu odpovídající XML parsery jsou povinny odmítnout určité části Unicode. Pugixml obětuje tuto analýzu, aby dosahoval lepší výkonnosti.

Parsování na místě

V typické implementaci parseru existuje několik neefektivit. Jednou z nich je kopírování řetězců dat do haldy. To zahrnuje přidělování mnoha bloků různé velikosti, od bajtů po megabajty, a vyžaduje, abychom kopírovali všechny řetězce z původního proudu do haldy. Pokud se vyhneme operacím kopírování, umožní nám to eliminovat oba zdroje neefektivity. Použitím techniky známé jako analýza na místě (nebo in situ), může parser použít data přímo z proudu. Tuto analytickou strategii používá pugixml.

Základní parser na místě má vstupní řetězec uložený v souvislé vyrovnávací paměti, prohledává řetězec jako proud znaků a vytváří potřebnou stromovou strukturu. Po načtení řetězce, který je součástí datového modelu, jako je značka (tag) pro název, ukládá parser ukazatel na řetězec a jeho délku (místo uložení celého řetězce).[3]

Jedná seo kompromis mezi výkonem a využitím paměti. Analýza na místě je obvykle rychlejší ve srovnání s analýzou s kopírováním řetězce do haldy, ale může spotřebovat více paměti. Kromě vlastních údajů popisujících strukturu dokumentu musí parser držet původní proud v paměti. Jiný typ parseru může místo toho uložit příslušné části původního proudu.



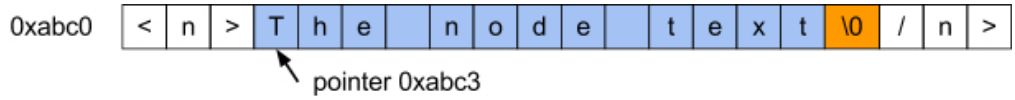
Obrázek 4.1: Příklad základní analýzy textu na místě

Většina parserů na místě se bude muset vypořádat s dalšími problémy. V případě pugixml jsou dva: zjednodušení přístupu k řetězci a transformace XML dat během analýzy.

Přístup k řetězcům, které jsou parsovány na místě, je těžký, protože nejsou ukončeny znakem null (znak s hodnotou nula). To znamená, že znak za řetězcem není null, ale je tam místo toho další znak v XML dokumentu, jako je otevřená ostrá závorka (<). Díky tomu je těžké používat standardní C / C ++ řetězcové funkce, které očekávají ukončení řetězce znakem null.

Abychom umožnili použití těchto funkcí, budeme muset ukončit řetězce znakem null během analýzy. Vzhledem k tomu, že nemůžeme snadno vkládat nové znaky, znak za posledním znakem každého řetězce bude muset být přepsán znakem null. Naštěstí to můžeme vždy udělat v XML: znak, který navazuje na konec řetězce, je vždy značkovací a není relevantní pro vyjádření v paměti.

3: To vytváří závislost na celý životní cyklus, celý zdrojový zásobník musí přežít všechny uzly dokumentu, aby technika fungovala.



Obrázek 4.2: Úprava pro ukončení řetězců null znakem při analýze na místě

Druhý problém je složitější: často se odlišuje hodnota řetězce a jeho vyjádření v XML souboru. U vyhovujícího parseru se očekává dekódování této reprezentace v XML. Pokud bychom ho prováděli s přístupen k uzlovým objektům, byl by výkon přístupu k objektům nepředvídatelný, proto preferujeme provedení během analýzy. V závislosti na typu obsahu by mohl XML parser provést následující transformace:

- Zpracování konce řádku: Vstupní dokument může obsahovat různé typy ukončení řádků a parser by je měl normalizovat takto: jakoukoliv sekvenci znaků návrat vozíku (ASCII 0xD) a posun řádku (ASCII 0xa) a každý samostatný návrat vozíku by měl být nahrazen znakem pro odřádkování. Například, řádek

  ```
  'line1\xD\xAline2\xDline3\xA\xA'
  ```

  by měl být transformován na

  ```
  'line1\xAline2\xAline3\xA\xA'.
  ```

- Rozšíření znakové reference: XML podporuje escape znaky pomocí jejich kódu Unicode buď v desítkovém, nebo šestnáctkovém vyjádření. Například, a by se mělo převést na a a ø by se měla převést na ø.

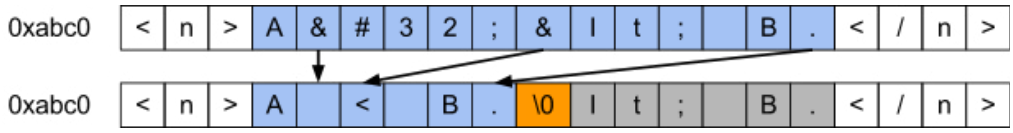
- Rozšíření reference entit: XML podporuje obecné odkazy na entity, kde se &name; nahradí hodnotou názvu entity. Existuje pět předdefinovaných entit: < (<), > (>), " („), ' (') a & (&).

- Normalizace podle hodnoty atributu: kromě rozšiřovacích referencí by měl parser provést normalizaci prázdných znaků při analýze hodnoty atributů. Všechny prázdné znaky (mezera, tabulátor, návrat vozíku a posun na další řádek) by měly být nahrazeny mezerou. Kromě toho, v závislosti na typu atributu, by měly být prázdné znaky na začátku a konci odstraněny a sekvence prázdných znaků uprostřed řetězce by měly být nahrazeny jednou mezerou.

V parseru na místě je možné podporovat libovolnou transformaci změnou obsahu řetězce, ale s důležitým omezením: transformace nesmí nikdy zvětšit délku řetězce.

Pokud by k tomu došlo, výsledek by se mohl překrývat s významnými daty v dokumentu. Naštěstí všechny z výše uvedených transformací splňují tento požadavek.[4]

Obecné rozšíření entity nesplňuje toto omezení. Vzhledem k tomu nepodporuje pugixml deklarace DOCTYPE, je ale nemožné specifikovat vlastní entitu bez DOCTYPE, každý podporovaný dokument lze plně parsovat s použitím analýzy na místě.[5]



Obrázek 4.3: Textové transformace s analýzou textu na místě

Je zajímavé, že analýza na místě může být použita i s I/O souborem mapovaným v paměti.[6] Podpora null zakončení a transformace textu vyžaduje speciální režim mapování paměti, známý jako kopírování při zápisu (copy on write), aby se zabránilo úpravám souboru na disku.

Použití I/O souboru mapovaného v paměti s analýzou na místě má následující výhody:

- Jádro obvykle může mapovat stránky vyrovnávací paměti přímo do adresního prostoru procesu, čímž se eliminuje paměťová kopie, která by se vytvořila v případě standardní práce se souborem.
- Pokud soubor už není ve vyrovnávací paměti, může jádro předběžně načíst části souboru z disku, což umožní paralelní provádění souborových operací a parsování.
- Protože pouze modifikované stránky potřebují přidělit fyzickou paměť, spotřeba paměti se může značně snížit pro dokumenty s velkými sekcemi textů.

4: To je obvyklé pro všechny transformace, vyjma Unicode transformace. V takovém případě jsou UTF-8 a UTF-16 kódování kompaktnější, než hexadecimální nebo desítkové reprezentace Unicode kódových bodů, což je důvod, proč nahrazením jednoho druhým se nikdy nezvýší délka.

5: Je možné podporovat obecné reference entit při analýze na místě tím, že se rozdělí každý řetězec s referencí entity do tří uzlů: prefix řetězec před referencí entity, uzel zvláštního druhu, který obsahuje referenční ID a suffix řetězec, který může být dále rozdělen. Tento postup se používá v Microsoft XML parseru (z různých důvodů).

6: Viz http://en.wikipedia.org/wiki/Memory-mapped_file

Optimalizace operací s ohledem na znaky

Eliminace kopií řetězců není to jediné, co můžeme udělat pro optimalizaci výkonu parseru. Pro porovnání výkonu parseru je užitečnou metrikou průměrný počet procesorových cyklů použitých pro každý znak. I když se toto liší v závislosti od dokumentu a architektury procesoru, je dostatečně stabilní pro dokumenty s podobnou strukturou. Dává smysl optimalizovat podle této metriky a pro začátek jsou vhodným místem operace vykonávané pro každý znak.

Nejdůležitější operací je detekce znakové sady.

Užitečným postupem je vytvoření tabulky Boolean příznaků, kde pro hodnotu každého znaku je uložena hodnota pravda/nepravda v závislosti na tom, zda znak patří do znakové sady. V závislosti na kódování dávají různé datové struktury a velikosti tabulek smysl takto:

- Pro kódování, kde každý znak nezabírá více než 8 bitů, je dostatečná tabulka o velikosti 256.

- Pro UTF-8 bychom chtěli použít bytově indexovanou tabulku, aby se zabránilo náročnému dekódování znaků; to funguje pouze tehdy, pokud všechny znaky s hodnotou větší než 127 patří do dané sady, anebo žádný znak s hodnotou větší než 127 nepatří do této sady. Pokud platí některá z podmínek, pak je dostačující tabulka velikosti 256. Prvních 128 záznamů v tabulce je vyplněno hodnotami pravda/nepravda (v závislosti na tom, zda je znak v cílové sadě) a posledních 128 záznamů v tabulce sdílí stejnou hodnotu. Vzhledem k tomu, jak UTF-8 kóduje data, budou všechny znaky kódu s hodnotou větší než 127 zastoupeny jako sekvence bajtů s hodnotami většími než 127, tj. včetně prvního znaku.

- Pro UTF-16 nebo UTF-32 jsou tabulky velkých rozměrů obvykle nepraktické. Se stejnými omezeními jako v případě optimalizované UTF-8 můžeme ponechat tabulku o velikosti 128 nebo 256 záznamů a přidat další porovnání, abychom se vypořádali s hodnotami mimo rozsah.

Všimněte si, že budeme potřebovat pouze jeden bit pro uložení hodnoty pravda/nepravda, takže můžeme použít bitové masky k ukládání osmi různých znakových sad v jedné 256 bajtové tabulce. Pugixml používá tento přístup k úspoře prostoru vyrovnávací paměti: na architektuře x86 má zjištění boolean hodnoty stejnou cenu jako zjištění hodnoty bitu v rámci bajtu, za předpokladu, že pozice bitu v bytuje konstantou v době kompilace. Následující kód v jazyce C demonstruje tento přístup:

```
enum chartype_t {
     ct_parse_pcdata      = 1, // \ , &, \\r, \<
     ct_parse_attr        = 2, // \ , &, \\r, ', „
     ct_parse_attr_ws = 4, // \ , &, \\r, ', „, \\n, tab
     // ...
};
static const unsigned char table[256] = {
     55, , , , , , , , 12, 12, , , 63, , , //  -15
     // ...
};
bool ischartype_utf8(char c, chartype_t ct) {
     // note: unsigned cast is important to
     // guarantee that the value is in ..255 range
     return ct & table[(unsigned char)c];
}

bool ischartype_utf16_32(wchar_t c, chartype_t ct) {
     // note: unsigned cast is important to
     // guarantee that the value is not negative
     return ct & ((unsigned)c < 128 ? table[(unsigned)c] : table[128]);
}
```

V případě, že testovaný rozsah zahrnuje všechny znaky v určitém intervalu, mohlo by být smysluplné použít místo vyhledávací tabulky porovnání. Při pečlivém používání aritmetiky bez znamének je potřeba jen jedno porovnání. Například test znaku, zda je číslicí:

```
bool isdigit(char ch) { return (ch >= ' ' && ch <= '9'); }
může být přepsán pomocí pouze jednoho porovnání:
bool isdigit(char ch) { return (unsigned)(ch - ' ') < 1 ; }
```

Pokud budeme pracovat znak po znaku, je zlepšení na základě výše uvedených přístupů obvykle nemožné. Nicméně, někdy je možné pracovat na skupině znaků a použít vektorové instrukce k provedení kontroly. Pokud je v cílovém systému k dispozici nějaká forma SIMD instrukcí, můžete obvykle použít tyto pokyny pro rychlou práci na skupinách o 16 a více znacích.

Dokonce, aniž bychom použili platformově specifickou sadu instrukcí, je někdy možné vektorizovat operace se znaky. Například existuje způsob, jak zjistit, jestli čtyři po sobě jdoucí bajty v UTF-8 z bytového proudu představují symboly ASCII[7]:

7: Samozřejmě, že údaje musí být vhodně sladěny, aby to fungovalo; navíc, tato metoda porušuje přísná pravidla alias pravidla C / C ++ standardu, která v praxi mohou nebo nemusí představovat problém.

A ať už děláte cokoliv, vyhněte se ve výkonově citlivém kódu funkcím is.*() ze standardní knihovny (například IsAlpha ()). I ty nejlepší implementace zjišťuj aktuální regionální nastavení, což je časově náročnější, než samotné vyhledávací tabulky, a nejhorší implementace můžou být ještě o dva řády pomalejší.[8]

Optimalizace řetězcových transformací

V pugixml jsou časově náročné zejména čtení a transformace hodnot. Pojďme se například podívat na čtení dat typu prostý znak (PCDATA); například text mezi tagy XML. Jakýkoliv standardní vyhovující parser, jak již bylo uvedeno, by měl provést referenční rozšíření a normalizaci konce řádku při zpracování obsahu PCDATA[9].

Jako příklad následující text v XML:

```
A&#32;&lt; B.
```

Měl by být transformován na:

```
A < B.
```

Funkce pro analýzu PCDATA nasměruje ukazatel na počáteční hodnotu PCDATA, pokračuje čtením zbytku hodnoty, konvertuje hodnotu dat na místě a ukončuje výsledek znakem null.

Vzhledem k tomu, že existují dva boolean příznaky, máme čtyři varianty této funkce. Aby se předešlo nákladné runtime kontrole, používáme pro příznaky boolean šablonu - budeme tedy kompilovat čtyři varianty funkce z jedné šablony. Parser volá funkci pomocí uvedeného ukazatele na funkci.

To umožňuje kompilátoru odstranit kontroly podmínek pro příznaky a nepřekládat nepoužitý kód pro každou variantu funkce. Důležité je, že uvnitř analytické smyčky funkce používáme rychlý test znakové sady, abychom přeskočili všechny znaky, které jsou součástí obyčejného PCDATA obsahu, a zpracujeme pouze znaky, o které se zajímáme. Zde je ukázka, jak vypadá kód:

8: Viz Kapitola 12 pro jiný příklad tohoto problému.

9: Pugixml umožňuje uživateli jeden z nich za běhu vypnout, a to z důvodu udržení výkonu a ochrany dat. Například se můžete zabývat dokumentem, kde je důležité zachovat přesný typ sekvence nových řádků, nebo tam, kde by reference entit měly zůstat bez rozšíření XML parserem, aby mohly být zpracovány až následně.

```
template <bool opt_eol, bool opt_escape> struct
strconv_pcdata_impl {
     static char_t* parse(char_t* s) {
          gap g;
          while (true) {
               while (!PUGI__IS_CHARTYPE(*s, ct_parse_pcdata)) ++s;
               if (*s == '<') { // PCDATA ends here
                 *g.flush(s) = 0;
                 return s + 1;
               } else if (opt_eol && *s == '\r') { // 0x0d or 0x0d 0x0a pair
                 *s++ = '\n'; // replace first one with 0x0a
                 if (*s == '\n') g.push(s, 1);
               } else if (opt_escape && *s == '&') {
                 s = strconv_escape(/s, g);
               } else if (*s == 0) {
                 return s;
               } else {
                 ++s;
               }
          }
     }
};
```

Další funkci dostane ukazatel na vhodnou implementaci, která je založena na příznacích runtime; např., &strconv_pcdata_impl<false, true>::parse.

Jedna neobvyklá položka v tomto kódu je instance třídy gap. Jak je ukázáno výše, pokud budeme provádět transformaci řetězce, výsledný řetězec bude kratší, protože některé znaky musí být odstraněny. Existuje několik způsobů, jak to udělat.

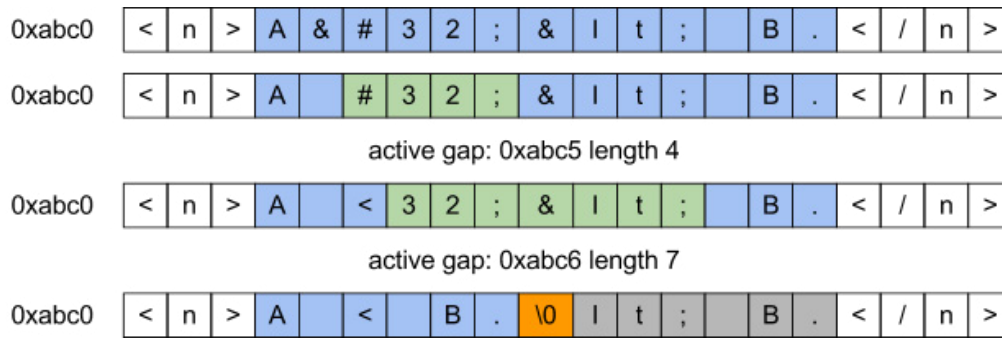
Jednou ze strategií (které pugixml nepoužívá) je mít oddělené ukazatele pro čtení a zápis tak, že obojí ukazují do stejné mezipaměti. V tomto případě ukazatel pro čtení sleduje aktuální pozici ke čtení a ukazatel pro zápis sleduje aktuální pozici pro zápis. Za všech okolností musí platit invariant (podmínka, která je vždy splněna) pozice pro zápis <= pozice ke čtení. Libovolný znak, který má být součástí výsledného řetězce, musí být explicitně zapsán podle ukazatele pro zápis. Tato technika zabrání kvadratickým nákladům na naivní přemísťování znaků, ale je stále neefektivní, protože pokaždé čteme a zapisujeme všechny znaky v řetězci, i když není nutné řetězec měnit.

Zřejmým rozšířením tohoto nápadu je vynechání prefixu původního řetězce, který nemusí být pozměněn, a jen začít psát znaky za prefixem. A vlastně takto běžně pracují algoritmy, jako např.: std :: remove_if ().

Pugixml používá odlišný přístup (viz Obrázek 4.4). V každém okamžiku je nanejvýš jedna mezera (gap) v řetězci. Mezera je posloupnost znaků, které již nejsou platné, protože již nejsou součástí finálního řetězce. Když má být přidána nová mezera, protože byla provedena další substituce (např. nahrazením " znakem „ vytváří mezeru 5 znaků), stávající mezera (pokud existuje) se vymaže posunutím dat mezi dvěma mezerami na začátek první mezery a pak zapamatováním si nové mezery. Z hlediska složitosti je tento přístup ekvivalentní k přístupu s ukazateli na pozice pro čtení a zápis; nicméně nám umožňuje použít rychlejší funkce k vymazání mezery (Pugixml používá memmove, který může kopírovat efektivněji ve srovnání se znakovou smyčkou, v závislosti na délce mezery a na implementaci C knihovny).

Optimalizace toku řízení

Samotný Pugixml parser lze považovat za rekurzivně sestupní parser. Nicméně je rekurze transformována do smyčky pro zlepšení výkonu. Uzlový kurzor se chová jako ukazatel na zásobník. Je-li nalezen počáteční tag, nový uzel se připojí ke kurzoru a stává se novým kurzorem; když je nalezena koncová značka, kurzor se přesune do nadřazeného uzlu vzhledem k aktuálnímu kurzoru. Tím je spotřeba místa v zásobníkukonstantní bez ohledu na vstupní dokument, což zlepšuje robustnost a vyhýbá se potenciálně nákladnému volání funkcí.



Obrázek 4.4: Příklad operací s mezerami v průběhu PCDATA konverze.

Parser používá řídicí smyčku, která čte znak z proudu, čte nula nebo více znaků, aby (v závislosti na prvním znaku) určila typ XML značky, a pak pokračuje do kódu, který parsuje příslušnou značku. Například v případě, že první znak je <, musíme přečíst alespoň jeden znak, abychom odlišili mezi počáteční a koncovou značkou, komentářem, nebo jiným typem značky. Pugixml

také používá goto příkazy, aby se v některých případech zabránilo procházení řídicí smyčkou - například analýza textového obsahu zastaví na konci proudu nebo znaku <. Nicméně pokud je další znak <, nemusíme procházet řídicí smyčkou jenom proto, abychom přečetli znovu znak a zjistili, že je to <; můžeme skočit rovnou do kódu, který značku parsuje.

Dvě důležité optimalizace pro takový kód jsou pořadí větvení a lokalita kódu.

V parseru různé části kódu zpracovávají různé typy vstupů. Některé z nich (jako je název značky nebo analýza atributu) vykonává často, zatímco jiné (například DOCTYPE analýza) spustí zřídka. Dokonce i uvnitř malé části kódu mají různé vstupy různé pravděpodobnosti. Například když parser narazí na levou ostrou závorku (<), s největší pravděpodobností se dále objeví znak tagu pro název. Další nejvíce pravděpodobný je znak /[10], a s menší pravděpodobností následují ! a ?.

Je možné přeuspořádat kód s ohledem na výše uvedené a zrychlit tak jeho provedení. Za prvé, všechny „studené" kódy; to znamená, že je nepravděpodobné, že by byl kód někdy proveden, nebo je nepravděpodobné, že se bude provádět často - v případě pugixml to zahrnuje veškerý obsah XML vyja značek s atributy a textovým obsahem - musí být přesunuta ze smyčky parseru do samostatné funkce. V závislosti na obsahu funkce a kompilátoru by mohlo pomoci označit překladači tuto funkci atributem noinline, nebo jeho ekvivalentem. Cílem je omezit velikost „horkého" (často vykonávaného) kódu v hlavní funkci parseru. To pomáhá kompilátoru optimalizovat kritickou část kódu tak, aby se vešla do instrukční vyrovnávací mezipaměti procesorového jádra.

Má smysl seřadit na základě podmíněné pravděpodobnosti všechny podmíněné řetězce. Například kód níže není efektivní pro typický obsah XML:

```
if (data[0] == '<')
{
    if (data[1] == '!') { ... }
    else if (data[1] == '/') { ... }
    else if (data[1] == '?') { ... }
    else { /* start-tag or unrecognized tag */ }
}
```

10: Důvodem, že / je méně pravděpodobné než znak tagu pro jméno, je to, že pro každý koncový tag existuje počáteční tag, ale jsou zde i tagy prázdných elementů, jako je <node/>.

Lepší verze by vypadala následovně:

```
if (data[ ] == '<')
    {
        if (PUGI__IS_CHARTYPE(data[1], ct_start_symbol)) { /* start-tag */ }
        else if (data[1] == '/') { ...   }
        else if (data[1] == '!') { ...   }
        else if (data[1] == '?') { ...   }
        else { /* unrecognized tag */ }
    }
```

V této verzi jsou větve seřazeny podle pravděpodobností od nejvíce po nejméně frekventované. Tím se minimalizuje průměrný počet podmíněných testů a provedených podmíněných skoků.

Zajištění bezpečnosti paměti

Bezpečnost paměti je důležitým faktorem pro parser. U libovolného vstupu (včetně škodlivého vstupu) nesmí parser nikdy číst nebo zapisovat do paměti za koncem vstupního zásobníku. Existují dva způsoby, jak toto implementovat. První možnost je zajistit, aby parser všude zjišťoval aktuální pozici čtení oproti koncové pozici. Druhou možností je použít řetězec s null ukončením jako vstup a zajistit, že parser správně zpracovává ukončení znakem null. Pugixml využívá rozšířenou druhou variantu.

Dodatečné zjišťování pozice čtení má za následek znatelnou výkonnostní režii, zatímco zakončení znakem null je často přirozeně zahrnuto ve stávajících kontrolách. Například smyčka

```
while (PUGI__IS_CHARTYPE(*s, ct_alpha))
    ++s;
```

přeskočí běh alfanumerických znaků a zastaví se na null zakončení nebo na dalším neabecedním znaku bez nutnosti dodatečných kontrol. Ukládání koncové polohy v mezipaměti také snižuje celkovou rychlost, protože to obvykle vyžaduje další registr procesoru. Volání funkce je také nákladnější, protože je třeba místo jednoho projít až dva ukazatele (aktuální a koncovou pozici).

Nicméně požadavek na null ukončený vstup je méně vhodný pro uživatele knihovny: často se XML data načtou do mezipaměti, ve které nemusí být místo pro přidání znaku null. Z pohledu uživatele by mezipaměť měla mít velikost bez zakonečení znakem null

Mezipaměť musí být možné přepisovat, aby bylo možné provést analýzu namist. Pugixml řeší tento problém jednoduchým způsobem. Před analýzou nahradí poslední znak v mezipaměti znakem null a uchovává si hodnotu původního znaku. Tímto způsobem jsou jedinými místy, která musíme uvažovat kvůli hodnotě posledního znaku, místa, kde chceme ukončit dokument.

V XML jich není mnoho[11], takže výsledky tohoto přístupu jsou pozitivní.[12]

Toto shrnuje nejzajímavější triky a návrhové rozhodnutí, které pomáhají udržet pugixml parser rychlý pro širokou škálu dokumentů. Nicméně je zde ještě poslední výkonově citlivá součást parseru, která stojí za diskusi.

4.5 Datové struktury pro objektový model dokumentu

XML dokument je stromová struktura. Obsahuje jeden nebo více uzlů; každý uzel může obsahovat jeden nebo více uzlů; uzly mohou představovat různé typy XML dat, jako jsou například elementy nebo texty; element uzlu může také obsahovat jeden nebo více atributů.

Každá reprezentace dat uzlu je obvykle kompromisem mezi spotřebou paměti a výkonem různých operací. Například sémanticky uzel obsahuje soubor podřízených uzlů; toto seskupení může být reprezentováno ve struktuře dat. Přesněji řečeno, tato data mohou být uložena jako pole nebo jako spojový seznam. Reprezentace formou pole by umožnila rychlý přístup založený na indexaci; reprezentace formou spojového seznamu by umožnila konstantní čas vložení nebo zrušení.[13]

Pugixml reprezentuje jak uzel, tak seskupení atributů jako spojový seznam. Proč ne jako pole? Dvě hlavní výhody polí jsou rychlý přístup založený na indexaci (což není nijak zvlášť důležité pro pugixml) a paměťová lokalita (což může být dosaženo prostřednictvím jiných prostředků).

Rychlý přístup na bázi indexu obvykle není potřeba, protože kód, který zpracovává XML strom, buď musí iterovat přes všechny podřízené uzly, nebo dostane specifický uzel, který je identifikován hodnotou atributu (např. „dostat podřízený uzel s ‚id' atributem rovným ‚X'").[14] Tento přístup je také křehký v měnitelném XML dokumentu: například přidávání XML komentářů změní indexy následných uzlů ve stejném podstromu.

11: Například, pokud je skenování tagu pro jméno zastavené u null zakončení, pak je dokument neplatný, protože neexistují žádné platné XML dokumenty, kde je předposlední znak součástí tagu pro jméno.

12: Samozřejmě se parsování kódu stává složitějším, protože je potřeba počítat s některými porovnáními posledního znaku, a všechny ostatní budou muset být přeskočené z výkonnostních důvodů. Sada dobře naplánovaných unit testů a fuzzy testování pomáhá udržovat parser funkční pro všechny typy vstupních dokumentů.

13: Strom modifikace je důležitý, zatímco existují způsoby, jak reprezentovat neměnné stromy mnohem efektivněji ve srovnání s tím, co pugixml dělá, mutace stromu představuje velice potřebnou vlastnost a to, jak pro stavbu dokumentů od nuly, tak pro modifikaci existujících dokumentů.

14: Komplexnější logika může být také použita.

Paměťová lokalita závisí na algoritmu přidělování (alokaci) paměti. Se správným algoritmem mohou být spojové seznamy efektivní jako pole v případě, že seznam uzlů je alokován postupně. (Více si o tom povíme později).

Základní datová struktura stromu s listy uloženými v poli (což není to, co pugixml používá) obvykle vypadá takto:[15]

```
struct Node {
     Node* children;
     size_t children_size;
     size_t children_capacity;
};
```

Základní datová struktura stromu, která používá spojové seznamy (což není přesně to, co pugixml používá), vypadá takto:

```
struct Node {
     Node* first_child;
     Node* last_child;
     Node* prev_sibling;
     Node* next_sibling;
};
```

V tomto případě je třeba last_child ukazatel nezbytný, aby umožnil zpětnou iterací a připojení dalších listů v O (1) čase.

Všimněte si, že u této konstrukce je snadné podporovat různé typy uzlů pro snížení spotřeby paměti; Například uzel elementu potřebuje seznam atributů, ale textový uzel ne. Tento přístup typu pole nás nutí udržovat stejnou velikost všech typů uzlů, což brání efektivitě této optimalizace.

Pugixml používá přístup na základě spojového seznamu. Tímto způsobem je modifikace uzlu vždy O (1). Navíc by nás přístup typu pole nutil alokovat bloky různých velikostí, od desítek bajtů po MB v případě jednoho uzlu s mnoha dětmi; zatímco u přístupu v propojeném seznamu existuje pouze několik různých alokačních velikostí potřebných pro uzlovou strukturu. Navrhování rychlých alokátorů pro přidělení fixní velikosti je obvykle snazší, než navrhování rychlých alokátorů pro libovolnou velikost, což je další důvod, proč pugixml zvolil tuto strategii.

15: Kapacitní pole je vyžadováno pro implementaci amortizovaného přírůstku konstantního v čase.
Pro víc informací viz http://en.wikipedia.org/wiki/Dynamic_array

Pro zachování spotřeby paměti, která se blíží k přístupu založenému na poli, pugixml vynechá last_child ukazatel, ale ponechává k dispozici přístup k poslednímu listu v O (1) čase částečnou cyklizací seznamu s prev_sibling_cyclic:

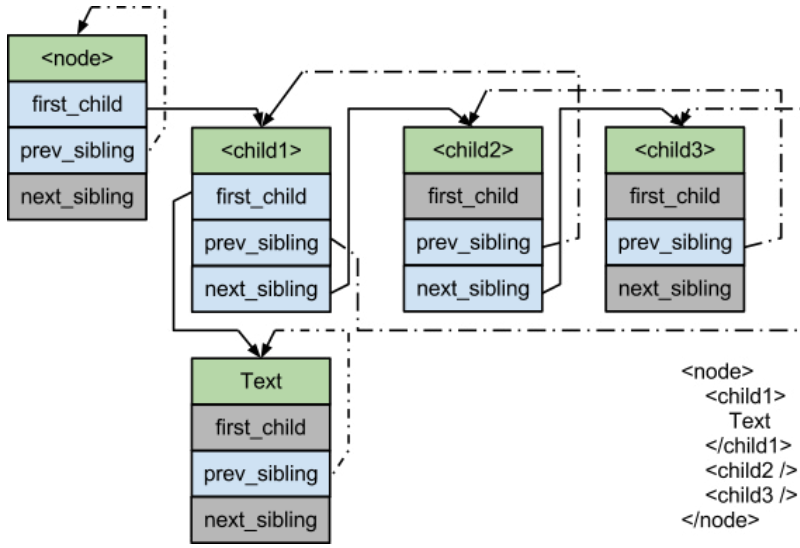
```
struct Node {
     Node* first_child;
     Node* prev_sibling_cyclic;
     Node* next_sibling;
};
```

Tato datová struktura je organizovaná následovně:

1. first_child ukazuje na první list uzlu, nebo je NULL, pokud uzel nemá list.

2. prev_sibling_cyclic ukazuje na levý list uzlu (list rodičovského uzlu, které je bezprostředně před uzlem v dokumentu). Když je uzel nejvíce vlevo (tj., když je uzel prvním listem jeho nadřízeného uzlu), prev_sibling_cyclic ukazuje na poslední list nadřízeného uzlu, nebo sám na sebe, pokud je jediným listem. prev_sibling_cyclic nemůže být NULL.

3. next_sibling ukazuje na pravý list, nebo je NULL, pokud je list posledním listem nadřízeného uzlu.



Obrázek 4.5: Příklad podstromu reprezentovaného použitím částečně cyklických propojených seznamů.

V této struktuře jsou všechny tyto operace konstantní v čase:

```
Node* last_child(Node* node) {
     return (node->first_child) ?
          node->first_child->prev_sibling_cyclic : NULL;
}

Node* prev_sibling(Node* node) {
     return (/node->prev_sibling_cyclic->next_sibling) ?
          node->prev_sibling_cyclic : NULL;
}
```

Přístup na základě pole a přístup propojeného seznamu se s trikem částečně cyklického seznamu listů stávají rovnocenné z hlediska spotřeby paměti. Použití 32 bitových typů pro velikost/kapacitu činí uzel v poli menší na 64 bitových systémech.[16] V případě pugixml převažují další výhody spojenýhc seznamů nad náklady.

V případě datových struktur je čas mluvit o posledním kousku skládanky - algoritmu pro přidělování paměti.

4.6 Alokace paměti na bázi zásobníku

Rychlý paměťový alokátor je rozhodující pro výkon DOM parseru. Analýza na místě eliminuje alokaci pro řetězec dat, ale ještě je potřeba alokovat DOM uzly. Je rovněž třeba alokovat řetězce různých velikostí, aby bylo možné modifikovat prvkystrom. Zachování paměťové lokality je důležité pro výkon při procházení stromu: pokud po sobě jdoucí žádosti o přidělení vrátí sousední bloky paměti, lze snadno zajistit lokalitu stromu během jeho konstrukce. A navíc je důležitá rychlost destrukce: kromě mazání v konstantním čase, schopnost rychle uvolnit všechnu paměť alokovanou pro daný dokument bez mazání každého uzlu zvlášť mohou významně zlepšit čas potřebný k zrušení rozsáhlých dokumentů.

Před diskusí o alokačním schématu používaném v pugixml se pojďme podívat na schéma, které by se dalo použít.

Vzhledem k tomu, že DOM uzly mají malou sadu požadovaných velikostí paměti, bylo by možné použít standardní paměťový zásobník založený na volných seznamech pro každou velikost. Pro takový zásobník by existoval jeden propojený seznam volných bloků, kde každý blok má stejnou velikost. Pokud je seznam volných bloků při žádosti o alokaci prázdný, je do něj alokovaná nová

16: Toto předpokládá omezení počtu uzlů dětí na hodnotu 232.

stránka s polem bloků. Tyto bloky jsou spolu spojeny do jednoho spojeného seznamu. Pokud seznam volných bloků není prázdný, první blok je vyjmut ze seznamu a je předán uživateli. Požadavek na dealokaci prostě zařadí uvolňovaný blok zpět do seznamu volných bloků.

Toto schéma alokace je velmi dobré na opakované použití – alokování jednoho uzlu po uvolnění jiného uzlu by okamžitě znovu použilo paměť. Nicméně přidání podpory pro uvolnění paměťových stránek zpět do haldy vyžaduje další sledování použitých bloků na jednu stránku. Lokalita alokací se také mění dle předchozího využití alokátoru, což může skončit poklesem výkonnosti při procházení ukazatelů.

Vzhledem k tomu, že pugixml podporuje modifikaci stromu, vyžaduje podporu pro alokace o libovolné velikosti. Nebylo jasné, zda alokátor může být snadno rozšířen o podporu libovolně velkých alokací a dalších pugixml funkcí bez dopadu na výkon parseru. Použití komplikovaného univerzálního schématu alokace blízkého algoritmům realizovaných v dlmalloc a dalších univerzálních paměťových alokátorech také nebylo správnou variantou – tyto alokátory mají tendenci být o něco pomalejší, než jednoduché volné seznamy, nemluvě o složitějších. Pugixml potřeboval něco jednoduchého a rychlého.

Ukazuje se, že nejjednodušší možné alokační schéma je alokátor používající zásobník. Tento alokátor funguje následovně: pro daný paměťový zásobník a offset uvnitř zásobníku potřebuje alokace pouze zvýšení offsetu o velikost požadované paměti. Samozřejmě, že je nemožné předpovědět velikost paměťového zásobníku předem, takže alokátor musí být schopen alokovat nové zásobníky na požádání.

Tento kód ilustruje obecnou představu:

```
const size_t allocator_page_size = 32768;
struct allocator_page {
     allocator_page* next_page;
     size_t offset;
     char data[allocator_page_size];
};
struct allocator_state {
     allocator_page* current;
};

void* allocate_new_page_data(size_t size) {
     size_t extra_size = (size > allocator_page_size) ?
          size - allocator_page_size :      ;
     return malloc(sizeof(allocator_page) + extra_size);
}
```

```
void* allocate_oob(allocator_state* state, size_t size) {
     allocator_page* page = (allocator_page*)allocate_new_page_data(size);
     // add page to page list
     page->next_page = state->current;
     state->current = page;
     // user data is located at the beginning of the page
     page->offset = size;
     return page->data;
}

void* allocate(allocator_state* state, size_t size) {
     if (state->current->offset + size <= allocator_page_size) {
          void* result = state->current->data + state->current->offset;
          state->current->offset += size;
          return result;
     }
     return allocate_oob(state, size);
}
```

Podpora alokací, které jsou větší než velikost stránky, je snadná. Alokujeme velký blok paměti, ale budeme s ním zacházet stejným způsobem, jako bychom zacházeli s malou stránkou.[17]

Tento alokátor je velmi rychlý. Vzhledem k omezením je to pravděpodobně nejrychlejší alokátor. Srovnávací testy ukazují, že je rychlejší než alokátor seznamu volných bloků, který musí udělat více práce pro určení správného seznamu na základě velikosti stránky a musí propojit všechny bloky na stránce dohromady. Náš alokátor také vykazuje téměř dokonalou paměťovou lokalitu. Jediným případem, kdy po sobě jdoucí alokace spolu nesousedí, je alokace nové stránky.

V případě malých alokací neplýtvá alokátor pamětí. Je však možné navrhnout hypotetický vzor pro alokaci paměti (který by mohl nastat v praxi), který plýtvá pamětí. Sekvence alokovaných velikostí v rozsahu od 64 do 65 536 by způsobila alokaci nové stránky na každé zavolání, což má za následek 30 % nevyužitého místa. Z tohoto důvodu se implementace alokátoru v pugixml chová mírně odlišně: pokud je alokace větší než jedna čtvrtina výchozí velikosti stránky, přiděluje celou stránku, a místo jejího přidání do přední části seznamu stránek se přidává za první položkou. Tímto způsobem jdou malé alokace, které nastanou po velkých, ještě do neúplné stránky.

Všimněte si, že allocate_oob () je „studený“ kód - to znamená, že bude vykonán jenom v pří-

17: Z důvodů výkonu tato implementace neupraví offset, který má být zarovnán. Namísto toho se očekává, že všechny uložené typy potřebují zarovnání typu ukazatel, a že všechny žádosti o alokaci určí velikost dle velikosti ukazatele.

padě, kdy vyčerpáme aktuální stránku, což by měla být vzácná událost. Z tohoto důvodu může atribut kompilátoru noinline zlepšit výkon.[18] To také znamená, že pokud máme složitější logiku v allocate_oob () - například logiku, která zpracovává velké alokace odlišně - nemá to žádný vliv na celkový výkon alokátoru.

A konečně, protože všechny alokace jsou obsaženy v nějaké stránce a alokátor udržuje celý seznam stránek jako stav, je velmi snadné zrušit celý seznam stránek, a tím i uvolnit celou přidělenou paměť. Je to velmi rychlé, protože se to dotkne jenom hlaviček jednotlivých stránek v paměti.

4.7 Podpora dealokace v alokátoru na bázi stacku

Implementace diskutovaná v předchozí sekci nemá žádný způsob, jak uvolnit a znovu použít paměť.

Je zajímavé, že pro mnoho případů užití to ve skutečnosti není velký problém. Vzhledem k tomu, můžeme uvolnit paměť po zrušení dokumentů tím, že odstraníme všechny stránky[19], analýza dokumentu nebo vytvoření nového dokumentu nespotřebovává další paměť. Problém však nastává, když odstraníme podstatnou část dokumentu a pak přidáváme další uzly do dokumentu. Protože opakovaně nepoužíváme paměť, může se stát velmi významným dosažení vrcholu spotřeby paměti.

Zdá se nemožná implementace jemnozrnného opakovaného použití při zachování výkonu alokace. Nicméně, může být dosaženo kompromisu. Během alokace budeme počítat počet alokací v každé uvažované stránce. Žádosti o dealokaci pak budou muset dostat odkaz na stránku zrušeného ukazatele a snížit tento počet. Pokud počet dosáhne nuly, stránka dále není potřeba a může být odstraněna.

Aby to bylo možné, musíme vědět, jaké stránce byl každý objekt alokován. To je možné bez uložení ukazatele na stránku, ale je to těžké.[20] Z tohoto důvodu se pugixml uchyluje k ukládání ukazatele stránky souběžně s každou alokací.

Pugixml používá dva odlišné přístupy ke snížení zatížení paměti spojené s ukládáním ukazatele stránky s každou alokací.

První přístup je ukládání ukazatele stránky v jednom poli o velikosti ukazatele s několika nesouvisejícími daty, které budeme muset uchovávat tak jako tak. Alokátor zajišťuje zarovnání stránek

18: Toto zlepšení je měřitelné v pugixml.

19: To samozřejmě znamená, že uzly a atributy nemůžou existovat bez dokumentů, což je v C ++ rozumné návrhové rozhodnutí pro vlastnictví uzlu.

20: Je to možné udělat bez ukládání dalších dat pomocí zarovnání stránky, která je větší než velikost stránky (tj. alokovat všechny stránky pomocí 64k alokací s 64k zarovnáním), ale není možné používat velké zarovnání alokací přenosným způsobem bez obrovské režie paměti.

na 32 bajtů, takže to znamená, že pět nejméně významných bitů každého ukazatele stránky je nulových; jako takové mohou být použity k ukládání libovolných dat. Pět bitů je dobré číslo, protože metadata XML uzlu jsou: tři bity jsou použity pro typ uzlu a dva bity se používají k určení, zda název a hodnota uzlu pobývají v zásobníku na místě.

Druhý přístup je ukládat offset přiděleného prvku relativně k začátku stránky, což nám umožňuje získat adresu ukazatele stránky následujícím způsobem:

```
(allocator_page*)((char*)(object) -
object->offset - offsetof(allocator_page, data))
```

Je-li naše velikost stránky limitována 216 = 65 536 bajty, tento offset se vejde do 16 bitů, takže pro uložení nám stačí 2 bajty místo 4. Pugixml používá tento přístup pro řetězce alokované v haldě.

Zajímavou vlastností výsledného algoritmu je, že respektuje referenční lokalitu vykazovanou kódem, který používá alokátor. Lokalita žádostí o alokaci nakonec vede k lokalitě v prostoru alokovaných dat. Lokalita žádostí o dealokaci prostoru vede k úspěšnému uvolnění paměti. V případě uložení stromu to znamená, že vymazání velkého podstromu obvykle uvolňuje většinu paměti použitou podstromem.

Samozřejmě, že pro některé vzory použití není nic vymazáno, dokud celý dokument není zrušen. Například, pokud je velikost stránky 32 000 bajtů, můžeme udělat jeden milion 32 bajtových alokací, kterým by bylo alokováno 1 000 stránek. Pokud budeme držet každý 1 000. objekt naživu a odstraníme zbývající objekty, každé stránce zůstane přesně jeden objekt, což znamená, že i když kumulativní velikost živých objektů je nyní 1 000 · 32 = 32 000 bajtů, stále udržujeme všechny stránky v paměti (spotřebovávají 32 milionů bajtů). To má za následek velmi vysokou režii paměti. Avšak takové použití je krajně nepravděpodobné a nad tímto problémem převažují výhody algoritmu pro pugixml.

4.8 Závěr

Optimalizace softwaru je těžká. Abychom byli úspěšní, zahrnuje optimalizační úsilí téměř vždy kombinaci nízkoúrovňové optimalizace, vysokoúrovňových výkonově orientovaných návrhových rozhodnutí, pečlivého výběru algoritmu a ladění, vyvažování mezi pamětí, výkonem, složitostí implementace a další. Pugixml je příklad knihovny, která potřebuje všechny tyto přístupy pro vytvoření velmi rychlého, k produkci připraveného XML parseru, i když byly nutné kompromisy k dosažení tohoto cíle. Spousta implementačních detailů může být přizpůsobena různým projektům a úkolům, ať už je to další parsovací knihovna nebo něco úplně jiného. Autor doufá, že prezentované triky byly zábavné, a že některé z nich budou užitečné pro další projekty.

— 3 Ninja (Evan Martin)

5 MemShrink

(Kyle Huey)

5 MemShrink

5.1 Úvod

O Firefoxu se již dlouho traduje, že používá příliš mnoho paměti. Pravdivost této reputace se v průběhu let měnila, ale je spojena s prohlížečem. Každé vydání Firefoxu se v průběhu posledních několika let setkávalo u skeptických uživatelů s otázkou „Opravili už úniky paměti?“. V březnu 2011 jsme po dlouhém beta cyklu a několika promeškaných termínech vydání vydali Firefox 4, a opět se setkal se stejnými otázkami. I když byl Firefox 4 významným krokem vpřed pro web v oblastech, jako jsou open video, výkonnost JavaScript, zrychlená grafika, byl ale bohužel významným krokem zpět ve využití paměti.

V průběhu uplynulých let se oblast webových prohlížečů stala velmi konkurenční. S rozmachem mobilních zařízení, uvolněním Google Chrome a investicemi Microsoftu do Internetu, se samotný Firefox začal potýkat s řadou vynikajících a dobře zafinancovaných konkurentů místo jen skomírajícího Internet Exploreru. Zejména Google Chrome zašel při poskytování rychlého a štíhlého procházení poměrně daleko. Zjistili jsme, že být dobrým prohlížečem již je málo; potřebovali jsme být vynikajícím prohlížečem. Jak řekl Mike Shaver, v té době VP pro inženýrství v Mozilla a dlouholetý přispěvatel Mozilly: „Je to svět, který jsme chtěli, a je to svět, který jsme vytvořili.“

To je místo, kde jsme se ocitli počátkem roku 2011. Tržní podíl Firefoxu byl malý nebo dokonce klesající, zatímco Google Chrome se těšil rychlému růstu k prominenci. I když jsme začali zacelovat mezeru na výkonu, měli jsme stále významnou konkurenční nevýhodu ve spotřebě paměti. Firefox 4 investoval do rychlejšího JavaScriptu a zrychlené grafiky často za cenu zvýšené spotřeby paměti. Po vydání Firefox 4 začala skupina inženýrů pod vedením Nicholase Nethercoteho pracovat na projektu MemShrink ke snížení spotřeby paměti. V současné době, po téměř roce a půl, toto koordinované úsilí zásadně změnilo spotřebu paměti Firefoxu a jeho pověst.

V myslích většiny uživatelů je „únik paměti“ věcí minulosti a Firefox často vychází ve srovnáních jako jeden z prohlížečů s nejmenšími nároky na paměť. V této kapitole budeme zkoumat úsilí, které bylo vynaloženo s cílem zlepšit využití paměti Firefoxu a ponaučení, které jsme u toho získali.

5.2 Celkový pohled na architekturu

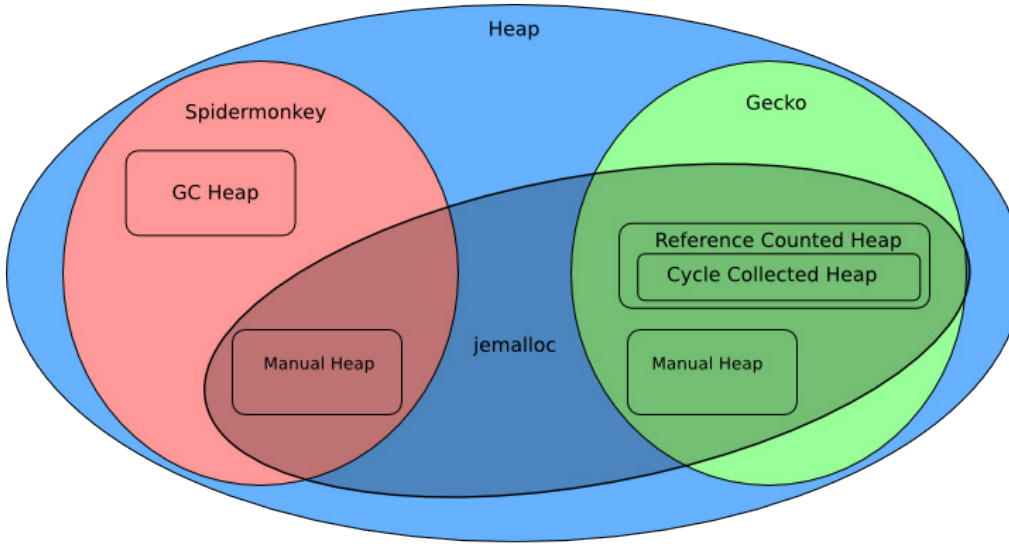
Abyste porozuměli problémům, na které jsme narazili, a jejich následnému řešení, budete potřebovat základní přehled o tom, co Firefox dělá a jak funguje.

Moderní webový prohlížeč je v podstatě virtuální stroj pro spuštění nedůvěryhodného kódu. Tento kód je nějakou kombinací HTML, CSS a JavaScriptu (JS), poskytovaných třetími stranami.

K dispozici je také kód z Firefox doplňků a modulů doplňků. Virtuální stroj poskytuje funkce pro výpočet, uspořádání a styl textu, obrázků, přístup k síti, offline úložiště, a dokonce i přístup k hardwarově akcelerované grafice. Některé z těchto schopností jsou poskytovány prostřednictvím rozhraní API určených pro daný úkol; mnoho dalších jsou k dispozici prostřednictvím rozhraní API, která byla zamýšlena pro zcela nové využití. Kvůli způsobu, jakým se web vyvinul, musí být webové prohlížeče velmi liberální v tom, co přijímají. Prohlížeče navržené před 15 lety dnes již nemusí být relevantní pro poskytnutí dojmu vysoké výkonnosti.

Firefox je vybaven Gecko vykreslovacím jádrem a SpiderMonkey JS jádrem. Oba jsou primárně vyvinuta pro Firefox, ale jsou oddělené a nezávisle znovupoužitelné části kódu. Podobně jako všechna často používaná vykreslovací a JS jádra, jsou napsána v jazyce C ++. SpiderMonkey implementuje JS virtuální stroj, včetně garbage collectoru a více druhů just-in-time kompilace (JIT). Gecko implementuje většinu API viditelných pro webovou stránku, včetně DOM, vykreslení webové stránky s možností hardwarové akcelerace, rozložení stránky a textu, celé síťové architektury a mnoho dalšího. Společně poskytují platformu, na které je Firefox postaven. Uživatelské rozhraní Firefoxu, včetně adresního řádku a navigačních tlačítek, je prostě řada speciálních webových stránek, které běží s rozšířenými oprávněními. Tyto výsady jim umožňují přístup ke všem druhům funkcí, které běžné webové stránky nevidí. Říkáme jim speciální, vestavěné oprávnění chromové stránky (žádný vztah k Google Chrome), na rozdíl od obsahových nebo normálních webových stránek.

Pro naše účely je nejzajímavější informací o SpiderMonkey a Gecko to, jak řídí paměť. Paměť v prohlížeči můžeme kategorizovat na základě dvou vlastností: jak je alokována a jak je uvolňována. Dynamicky alokovaná paměť (halda) se získává po velkých kusech z operačního systému a je rozdělena do požadovaných velikostí alokátorem haldy. Existují dva hlavní alokátory haldy: specializovaný garbage collected alokátor haldy použitý pro garbage collector paměti (GC heap) v SpiderMonkey, a jemalloc, který je používán vším ostatním v SpiderMonkey a Gecko. Existují také tři způsoby uvolnění paměti: ručně, pomocí počítání referencí a přes garbage collection.



Obrázek 5.1: Řízení paměti ve Firefox

GC halda v SpiderMonkey obsahuje objekty, funkce a většinu dalších věcí vytvořených běžícím JS. V GC haldě také ukládáme implementační detaily, jejichž životní cyklus je spojen s těmito objekty. Tato halda používá zcela standardní inkrementální kolektor typu „označ a uvolni“ (mark and sweep), který byl silně optimalizován na výkon a rychlost odezvy. To znamená, že se každou chvíli garbage collector probudí a podívá se na celou paměť v GC haldě. Počínaje sadou „kořenů“ (jako je globální objekt stránky, kterou prohlížíte), „označí“ všechny objekty v haldě, které jsou dosažitelné. Pak uvolní všechny objekty, které nejsou označeny, a v případě potřeby znovu použije tuto paměť.

Většina paměti v Gecko je spravována pomocí počítání odkazů. U počítání odkazů je sledován počet odkazů na daný kus paměti. Když toto číslo klesne na nulu, paměť se uvolní. Zatímco počítání odkazů je technicky forma garbage collection, pro tuto diskusi ho odlišujeme od systémů garbage collection, který vyžaduje speciální kód (tj. garbage collector), aby pravidelně označil už nepoužívanou paměť jako volnou. Jednoduché počítání odkazů není schopno se vypořádat s cykly, kdy se jeden kus paměti A odkazuje na jiný kus paměti B a naopak. V této situaci, jak A, tak i B, mají počet odkazů 1 a nikdy se neuvolní. Gecko má specializovaný sledovací garbage collector určený pro vyhledání těchto cyklů a nazývá se cycle collector. Pracuje pouze s těmi třídami, o kterých víme, že vytvářejí cykly, takže můžeme uvažovat o cycle collectoru haldy jako o podmnožině haldy spravované pomocí počítání odkazů. Cycle collector pracuje také s garbage collectorem v SpiderMonkey při obsluze řízení mezi-jazyčné paměti tak, že C ++ kód může obsahovat odkazy na objekty JS a vice versa.

Jak v SpiderMonkey, tak v Gecko existuje také velké množství ručně řízené paměti. To zahrnuje vše od interní paměti polí a rozptylových tabulek, po zásobníky obrazových a textových zdrojových dat. Na vrcholu ručně řízené paměti existují i jiné specializované alokátory. Jedním z příkladů je Arena Allocator. Arény se používají, když může být najednou uvolněn velký počet samostatných alokací. Arena allocator získává kusy paměti z hlavního alokátoru haldy a dále je rozdělí tak, jak je požadováno. Když už arény není třeba, vrátí kusy paměti hlavnímu alokátoru haldy, aniž by se muselo jednotlivě uvolnit mnoho menších alokací. Gecko používá aréna allocator pro data na vykreslení stránky, která lze vyhodit najednou, když již není zapotřebí stránky. Alokace arény nám také umožňuje implementovat bezpečnostní funkce, jako je poisoning, při které nejprve přepíšeme obsah uvolňované paměti, takže nemůže být bezpečnostně zneužita.

Existuje několik dalších systémů pro správu paměti v malých částech Firefoxu, které se používají pro celou řadu různých důvodů, ale nejsou relevantní pro naši diskusi. Nyní, když máte stručný přehled o paměťové architektuře Firefoxu, můžeme diskutovat o problémech, které jsme našli, a jak je vyřešit.

5.3 Děláte to, co měříte

Prvním krokem k vyřešení problému je zjištění, v čem je problém. Dle striktní definice úniku paměti se jedná o alokaci paměti z operačního systému (OS) a její neuvolnění zpátky do operačního systému. Ta ale nepokrývá všechny situace, které chceme zlepšit. Některé situace, se kterými se setkáváme, nejsou „úniky“ v užším slova smyslu:

- Datová struktura vyžaduje dvakrát tolik paměti, kolik potřebuje.
- Paměť, která se již nepoužívá, není uvolněna, dokud nevyprší časový limit.
- Mnoho kopií stejně velké mezipaměti (řetězce, obrazová data atd.) existují po celou dobu programu.

To vše se komplikuje ještě tím, že většina paměti v haldě Firefoxu podléhá určité formě garbage collection, a tak paměť, která již není používaná, nebude uvolněna až do příštího běhu GC. Používáme termín „únik“ velmi volně tak, aby zahrnoval jakékoli situace, které vedou ve Firefoxu k menší paměťové efektivitě, než by rozumně mohla být. Je to v souladu s tím, jak tento termín používají také naši uživatelé: většina uživatelů a dokonce i webových vývojářů nemůže říct, jestli je vysoké využití paměti dáno skutečným únikem nebo nějakými jinými faktory při práci v prohlížeči.

Když MemShrink začal, neměli jsme moc vhledu do využití paměti prohlížeče. Identifikace povahy problémů s pamětí často vyžaduje použití komplexních nástrojů, jako je Massif,nebo nástrojů nižší úrovně, jako je GDB. Tyto nástroje mají několik nevýhod:

- Jsou určeny pro vývojáře a není snadné je používat.
- Neznají interní specifika Firefoxu (například detaily implementace různých druhů hald).
- Nejsou „vždy dostupné“ - musíte je použít, když se vyskytuje problém.

Výměnou za tyto nevýhody dostanete některé velmi mocné nástroje. K řešení uvedených nevýhod jsme postupem času vybudovali sadu vlastních nástrojů, abychom za menší úsilí získali větší vhled do chování prohlížeče.

První z těchto nástrojů je about:memory. Nejprve byl zaveden do Firefoxu 3.6, původně zobrazil jednoduché statistické údaje o haldě, jako je například přehled o paměti procesu aktuálně přítomného v RAM (tzv. reserved, mapped a committed memory). Později byly přidány některá, pro vývojáře zajímavá, měření, jako je například použití paměti vloženého SQLite databázového stroje a množství paměti používané grafickým akcelerátorem. Těmto měřením říkáme paměťoví reportéři. Vyjma těchto jednorázových přírůstků, about:memory zůstal primitivním nástrojem představujícím několik souhrnných statistik o využití paměti. Většina paměti neměla paměťového reportéra a nebyla specificky vyčíslena v about:memory. about:memory může dokonce použít kdokoli bez použití speciálního nástroje nebo verze Firefoxu, a to pouhým jeho napsáním do adresního řádku prohlížeče. Tak by se stala „bezkonkurenční funkcí“.

Dlouho před projektem MemShrink bylo jádro JavaScriptu ve Firefoxu přepracováno tak, aby rozdělilo monolitickou globální GC haldu do menších seskupení, subhald nazvaných kompartmenty kompartmenty. Tyto kompartmenty kompartmenty oddělují věci, jako je chromová a obsahová (respektive privilegované a neprivilegované kódy) paměť, stejně jako paměť různých webových stránek. Primární motivací pro tuto změnu byla bezpečnost, ale ukázalo se, že je velmi užitečná pro MemShrink. Krátce po této implementaci byl prototypován nástroj nazvaný about:compartments, který zobrazuje všechny kompartmenty, kolik používají paměti, a jak ji využívají. about:compartments nebyl nikdy integrován přímo do prohlížeče Firefox, ale poté, co začal MemShrink, byl upraven a zakombinován do about:memory.

Při přidávání tohoto reportingu kompartmentů kompartmentů do about:memory jsme si uvědomili, že začlenění podobného reportingu pro jiné alokace by umožnilo užitečné profilování haldy bez specializovaných nástrojů jako je Massif. about:memory byl změněn tak, že místo vytváření řady souhrnných statistik se zobrazí strom rozkládající využití paměti do velkého počtu různých použití. Pak jsme začali přidávat reportéry pro jiné druhy velkých alokací haldy, jako je vykreslovací subsystém.

Jednou z našich prvních, metrikou řízených snah bylo snížení množství neklasifikované haldy, tj. paměti bez paměťového reportéra. Vybrali jsme si docela libovolný počet, 10 % z celkové haldy, a stanovili jsme si cíl snížit neklasifikované haldy na tuto hodnotu v průměrném scénáři použití.

V konečném důsledku se ukázalo, že 10 % bylo příliš nízké číslo. Existuje prostě příliš mnoho malých jednorázových alokací v prohlížeči na to, abychom snížili neklasifikovanou haldu spolehlivě pod přibližně 15 %. Snížení neklasifikované haldy zvyšuje vhled do toho, jak se paměť používá prohlížečem.

Abychom snížili velikost neklasifikované haldy, vytvořili jsme nástroj a pokřtili ho jako „detektor temné hmoty" (DMD). Pomohl vystopovat nereportované alokace haldy. Funguje tak, že nahradí alokátor haldy a vloží se do about:memory procesu reportování paměti a paměťových bloků odpovídajících k alokovaným blokům. Pak shrne nereportované alokace paměti podle místa volání.

Běh DMD na Firefox relaci vytváří seznamy volání míst odpovědných za neklasifikovanou haldu. Jakmile byl identifikován zdroj alokací, byla nalezena odpovědná komponenta a vývojář pro ni rychle přidal paměťového reportéra. Během několika měsíců jsme měli nástroj, který vám mohl říct věci, jako např. „všechny Facebook stránky ve vašem prohlížeči používají 250 MB paměti, a zde je rozpis toho, jak je tato paměť používaná."

Také jsme vyvinuli další nástroj (nazývaný Measure and Save) pro odladění identifikovaných problémů s pamětí. Tento nástroj vypíše do souboru reprezentaci jak JS haldy, tak cycle-collected C++ haldy. Pak jsme napsali sérii analytických skriptů, které mohou procházet kombinované haldy a odpovědět na otázky typu „co drží tento objekt naživu?" To umožnilo spoustu užitečných technik ladění, od jednoduchého zkoumání grafu haldy pro odkazy, které by měly být zrušeny, až po nastavení míst zastavení ladicího programu na specifických objektech zájmu.

Velkou výhodou těchto nástrojů je, že na rozdíl od nástrojů, jako je např. Massif, můžete před jejich použitím počkat, až se problém objeví. Mnoho profilovačů hald (včetně Massifu) musí být spuštěných při spouštění programu, a ne až poté, co se objeví problém. Další výhodou těchto nástrojů je, že informace mohou být analyzovány a používány, aniž byste měli problém s jejich reprodukcí. Umožňují uživatelům jak zachytit informace o identifikovaném problému, tak je i odeslat vývojářům, když nemůžou problém reprodukovat. Pro běžné uživatele webového prohlížeče, a to i ty pro ty sofistikovanější, kteří umí zapisovat chyby v systému sledování chyb, je použití GDB nebo Massifu v prohlížeči příliš náročné. Ale načítání about:memory nebo běh malého fragmentu JavaScriptu pro získání dat a jejich připojení k hlášení o chybě je mnohem méně náročným úkolem. Obecné profilovače hald zachytí velké množství informací, ale s vysokými náklady. Byli jsme schopni napsat sadu nástrojů přizpůsobených našim specifickým potřebám, což nám přineslo významné výhody oproti běžně používaným nástrojům.

Ne vždy se vyplatí investovat do vlastních nástrojů; existuje důvod, proč používáme GDB místo psaní nového ladicího nástroje pro každý kus vyvíjeného softwaru. Ale zjistili jsme, že pro situace, kdy stávající nástroje nedokáží zajistit potřebné informace požadovaným způsobem, můžou být vlastní nástroje opravdovou výhrou. Trvalo nám asi rok práce na částečný úvazek dostat about:memory do bodu, kdy jsme ho považovali za kompletní. I dnes, pokud je to nutné, stále přidáváme nové funkce a reportéry.

Vlastní nástroje představují významnou investici. Toto téma je velkou odbočkou nad rámec této kapitoly, ale ještě před vytvářením vlastních nástrojů byste měli pečlivě zvážit jejich přínosy a náklady.

5.4 Snadno dostupné výsledky

Nástroje, které jsme vytvořili, nám poskytly významně větší viditelnost využití paměti v prohlížeči, než jsme měli předtím. Po jejich delším používání jsme získali cit pro to, co je normální, a co ne. Všimnout si věcí, které nebyly normální a byly pravděpodobně chybami, se stalo velmi jednoduché. Velké množství neklasifikovaných hald poukázalo na použití skrytých webových funkcí, pro které jsme dosud nepřidali paměťové reportéry, nebo na úniky ve vnitřních funkcích jádra Gecko. Velké využívání paměti v podivných místech jádra JS by mohlo naznačovat, že kód narazil na nějakou neoptimalizovanou nebo patologickou část. Byli jsme schopni tyto informace použít k vystopování a opravě nejhorších chyb ve Firefoxu.

Na začátku jsme zaznamenali jednu anomálii, že se někdy nezrušil kompartment kompartment spojený s již zavřenou stránkou i poté, co jsme přinutili garbage collector běžet opakovaně. Někdy tyto kompartmenty zmizely samy od sebe, a někdy vydržely do nekonečna. Pojmenovali jsme tyto úniky jako zombie kompartmenty. Byly to jedny z našich nejvážnějších úniků, protože množství paměti, které webová stránka mohla použít, bylo neomezené. Opravili jsme velký počet těchto chyb jak v Gecko, tak ve Firefox UI kódech, ale brzy se ukázalo, že největším zdrojem zombie kompartmentů byly doplňky. Práce s úniky v doplňcích nám zabrala několik měsíců, než jsme našli řešení, které je popsáno dále v této kapitole. Většina z těchto zombie kompartmentů, a to jak ve Firefoxu, tak v doplňcích, byla způsobena udržováním odkazů dlouhodobých objektů JS do krátkodobých objektů JS. Dlouhodobé objekty JS jsou obvykle objekty spojené s oknem prohlížeče, nebo dokonce globálně unikátní objekty, zatímco krátkodobé objekty JS můžou být objekty z webových stránek.

Kvůli způsobu, jakým DOM a JS pracují, udrží odkaz na jeden objekt z webové stránky celou stránku a její globální objekt (a cokoli z toho dosažitelné) při životě. To může snadno přidat mnoho megabajtů paměti. Jeden z rafinovaných aspektů systému garbage collection je, že GC vrací jen paměť, když je nedostupná, ne když se program provádí jejím použitím. Je na programátorovi, aby zajistil, že se paměť, která nebude znovu použita, řádně uvolní.

Selhání při odstraňování všech odkazů na objekt má ještě vážnější následky, když je významně odlišná očekávaná životnost odkazujícího a odkazovaného objektu. Paměť, která by měla být uvolněna relativně rychle (jako např. paměť používaná pro webové stránky), je místo toho vázána na dobu trvání odkazujícího objektu s delší životností (například v okně prohlížeče nebo samotné aplikaci).

Fragmentace v JS haldě byla pro nás z podobného důvodu také problémem. Často jsme viděli v reportech operačního systému, že uzavírání mnohých webových stránek nezpůsobilo významné snížení paměti využívané Firefoxem. Jádro JS přiděluje paměť z operačního systému ve velkých

megabajtových kusech a tyto kusy rozděluje podle potřeby mezi různé kompartmenty. Tyto kusy mohou být uvolněny zpět do operačního systému, pouze když jsou zcela nevyužité. Zjistili jsme, že přidělení nových kusů bylo téměř vždy způsobeno webovým obsahem požadujícím více paměti, ale tím posledním, co neumožnilo uvolnění kusu paměti, byl často chromový kompartment. Míchání několika dlouhodobých objektů do kusu plného krátkodobých objektů nám zabránilo uvolnit tento blok po zavření webové stránky. Vyřešili jsme to oddělením chromového a obsahových kompartmentů tak, aby každý daný kus měl buď chromovou, nebo obsahovou alokaci. To významně zvýšilo množství paměti, kterou jsme mohli vrátit do operačního systému po zavření záložky prohlížeče.

Objevili jsme další problém částečně způsobený technikou pro snížení fragmentace. Primární alokátor haldy Firefoxu je jemalloc verze pozměněná pro práci ve Windows a Ma OS X.

Jemalloc je navržen s cílem snížit ztráty paměti v důsledku fragmentace. Jednou z používaných technik, jak to udělat, je zaokrouhlení alokací do různých velikostních tříd, a poté alokování těchto velikostních tříd v souvislých blocích paměti. To zajišťuje, že po uvolnění může být tento prostor později znovu použitý pro alokaci podobné velikosti. To také představuje plýtvání prostoru z důvodu zaokrouhlování. U některých velikostních tříd může nejhorší případ znamenat plýtvání téměř 50 % alokovaného prostoru. Kvůli způsobu, jak jsou v jemalloc strukturovány velikostní třídy, se to obvykle stává hned po překročení druhé mocniny (např. 17 se zaokrouhlí na 32 a 1 025 se zaokrouhlí na 2 048).

Co se týče požadovaného množství, při přidělování paměti často nemáte moc na výběr. Přidání dalších bajtů k alokaci pro novou instanci třídy je jen zřídka užitečné. Jindy je možná jistá flexibilita. Pokud alokujete prostor pro řetězce, můžete použít více prostoru, abyste nemuseli alokovat paměť navíc při zvětšení řetězce. Když se prezentuje samotná flexibilita, má smysl žádat o množství, které se přesně shoduje s velikostí třídy. Tímto způsobem by nevyužitá, ale přesto alokovaná paměť byla k dispozici pro použití bez dalších nákladů. Obvykle je kód napsán tak, aby požadoval velikosti o mocnině dvou, protože to dobře vyhovuje pro skoro každý alokátor, který kdy byl napsán, a nevyžaduje speciální znalosti o alokátorech.

Našli jsme spoustu kódů v Gecko, který byl napsán za použití této techniky, a několik míst, které se o to snažily, ale bylo to špatně. Několik kusů kódů se pokoušelo alokovat optimálně zaokrouhlený kus paměti, ale použily špatně matematiku, a skončily alokaci těsně za tím, co bylo zamýšleno. Vzhledem ke konstrukci velikostních tříd v jemalloc to často vede k plýtvání téměř 50 % alokované paměti. Jeden zvláště neslýchaný příklad byl při implementaci arena alokátoru používaného pro vykreslovací datové struktury. Aréna alokátor se pokusil získat o pár bytů více, než 4 KB z haldy. To mělo za následek, že žádal o něco více než 4 KB, které byly zaokrouhleny na 8 KB. Oprava takovéto chyby ušetřila více než 3 MB paměti jenom na samotný GMail. Na testovacím případu se zvlášť náročným vykreslováním ušetřilo více než 700 MB paměti, což snižuje celkovou spotřebu paměti prohlížeče z 2 GB na 1,3 GB.

S podobným problémem jsme se setkali v SQLite. Gecko používá SQLite jako databázový stroj pro funkce, jako jsou historie a záložky. SQLite je napsán tak, aby umožnil vnořeným aplikacím dostatek kontroly nad přidělováním paměti, a je velmi puntičkářský při měření vlastního využívání paměti. Aby se zachovala tato měření, přidává pár bytů, které tlačí alokaci do další velikostní třídy. Ironicky řečeno, nástroje potřebné ke sledování spotřeby paměti způsobí zdvojnásobení její spotřeby a zároveň způsobí významně nižší reportování. Poukazujeme na tyto druhy chyb jako na „boty klauna", protože jsou jak komicky špatné, tak mají za následek hodně plýtvaného prázdného prostoru, přesně jako jsou boty klauna.

5.5 To, že to není vaše chyba, neznamená, že to není váš problém

V průběhu několika měsíců jsme dosáhli velkého pokroku ve snížení spotřeby paměti a oprav úniků paměti ve Firefoxu. Nicméně ne všichni naši uživatelé viděli výsledky této práce. Bylo jasné, že významný počet problémů s pamětí, které viděli naši uživatelé, vznikaly v doplňcích. Naše sledování únikových chyb v doplňcích nakonec spočetlo více než 100 potvrzených hlášení o doplňcích způsobujících úniky.

Co se týče doplňků, historicky to Mozilla zkoušela hrát na obě strany. Prodávali jsme Firefox jako rozšiřitelný prohlížeč s bohatým výběrem doplňků. Ale když uživatelé hlásili problémy s výkonem u těchto doplňků, jednoduše jsme uživateli řekli, aby je nepoužíval. Pouhý počet doplňků, které způsobily úniky paměti, udělal tuto situaci dále neudržitelnou. Mnoho doplňků Firefoxu je distribuováno prostřednictvím Mozilla stránek addons.mozilla.org (AMO). AMO přezkoumal politiku určenou k zachycení běžných problémů v doplňcích. Když AMO recenzenti začali testování doplňků na úniky paměti s nástroji jako about:memory, začali jsme získávat představu o rozsahu problému. Ukázalo se, že řada testovaných doplňků má problémy, jako např. zombie kompartmenty. Začali jsme oslovovat autory doplňků a dali jsme dohromady seznam osvědčených postupů a obvyklých chyb, které způsobily úniky. Bohužel to mělo poměrně malý úspěch. I když některé doplňky byly svými autory opraveny, většina však ne.

Existuje množství důvodů, proč to bylo neefektivní. Ne všechny doplňky jsou pravidelně aktualizovány. Autoři doplňků jsou dobrovolníci s vlastními plány a prioritami. Ladění úniků paměti může být těžké, zvláště pokud nelze reprodukovat původní problém. Nástroj pro sledování haldy popsaný výše je velmi silný a umožňuje snadné shromažďování informací, ale analýza výstupu je i tak složitá, a je přehnané očekávat, že to autoři doplňků udělají. Koneckonců neexistovaly žádné silné pobídky k opravě úniků. Nikdo nechce dodávat špatný software, ale nelze vždy všechno opravit. Lidé se spíše zajímají o to, aby udělali to, co chtějí dělat, než to, co po nich chceme, aby dělali.

Dlouho jsme si povídali o vytvoření pobídek pro opravu úniků paměti. Doplňky způsobily také další problémy s výkonem Mozilly, takže jsme diskutovali možnosti, jak zviditelnit údaje o výkonu v AMO nebo v samotném Firefoxu. Teoreticky, pokud budeme schopni informovat uživatele

o vlivu doplňků, či už instalovaných nebo připravovaných k instalaci, na výkon, pomůžeme jim činit informovaná rozhodnutí týkající se používaných doplňků. První problém je, že uživatelé softwaru orientovaného na spotřebitele, jako jsou webové prohlížeče, obvykle nejsou schopni učinit informovaná rozhodnutí o takových kompromisech. Kolik ze 400 milionů uživatelů Firefoxu chápe, co to únik paměti je a může vyhodnotit, zda to stojí za to utrpení, aby mohl používat nějaký náhodný doplněk? Za druhé, vypořádání se s výkonovými dopady doplňků tímto způsobem vyžaduje souhlas mnoha různých částí komunity Mozilla. Například lidé tvořící komunitu kolem doplňků nebyli nadšení myšlenkou zákazu doplňků. A konečně, velké procento Firefox doplňků není vůbec nainstalováno přes AMO, ale jsou spojeny s jiným softwarem. Máme jen minimální vliv na tyto doplňky a možnost zablokovat je. Z těchto důvodů jsme upustili od našich pokusů o vytvoření takovéto pobídky.

Další důvod, proč jsme upustili od vytváření pobídek pro opravu úniků paměti v doplňcích, je, že jsme našli úplně jiný způsob, jak vyřešit tento problém. Nakonec se nám podařilo najít způsob, jak „uklidit" po doplňcích s úniky paměti ve Firefoxu. Dlouhou dobu jsme si mysleli, že je to neproveditelné, aniž bychom narušili spoustu doplňků, ale stále jsme experimentovali. Nakonec jsme byli schopni implementovat techniku, která uvolňovala paměť, aniž by měla nepříznivý vliv na většinu doplňků. Využili jsme hranice mezi kompartmentmi na „ořezání" referencí z chromových kompartmentů do obsahových kompartmentů, když je stránka navigovaná nebo je zavřená záložka. To ponechává kolující objekty v chromové přihrádce, ale již bez odkazů. Původně jsme si mysleli, že by mohl být problém, když se kód snaží použít tyto objekty, ale zjistili jsme, že ve většině případů nejsou tyto objekty později použity. Ve skutečnosti doplňky náhodně a nesmyslně dočasně ukládaly do dočasné paměti věci z webových stránek, a jejich automatický úklid měl malou nevýhodu. Hledali jsme sociální řešení pro technický problém.

5.6 Věčné trvání je cenou za dokonalost

Projekt MemShrink dosáhl značného pokroku v řešení paměťových problémů Firefoxu, ale ještě zbývá udělat hodně práce. Většina jednoduchých problémů byla do této chvíle opravena, to, co zbylo, vyžaduje značné množství inženýrského úsilí. Máme v plánu pokračovat ve snižování fragmentace JS haldy pohyblivým garbage collectorem, který může konsolidovat haldu. Přepracováváme způsob zpracování obrázků, aby byl více paměťově efektivní. Na rozdíl od mnoha dokončených změn, tyto vyžadují rozsáhlé přepracování složitých subsystémů.

Neméně důležité je, že jsme neměli regresi na již provedených zlepšeních. Mozilla má silnou kulturu regresního testování od roku 2006. Jak jsme dosáhli pokroku v zeštíhlení využití paměti Firefoxu, naše touha po regresním testování systému na využití paměti se zvýšila. Testování výkonnosti je náročnější, než testování funkcí. Nejtěžší částí budování tohoto systému bylo přijít s realistickou pracovní zátěží prohlížeče. Stávající paměťové testy prohlížeče dost okázale selhaly na realitě. MemBuster například načte řadu wiki a blogů do nového okna prohlížeče vždy

v rychlém sledu za sebou. Většina uživatelů dnes používá záložky místo nových oken a prochází složitější věci, než wiki a blogy. Jiné srovnávací testy načítají všechny stránky do stejné záložky, což je také zcela nereálné pro moderní webové prohlížeče. Navrhli jsme vytížení, o kterém jsme přesvědčeni, že je poměrně realistické. Načte 100 stránek do fixní sady 30 záložek se zpožděním mezi načteními, abychom se co nejvíce přiblížili způsobu čtení stránek uživatelem. Z Mozilly jsou použity stránky z Tp5 sady. Tp5 je sada stránek z Alexa Top 100, které se používají k testování výkonu načtení stránky v naší stávající infrastruktuře pro testování výkonu. Tato pracovní zátěž se ukázala být vhodnou pro naše účely testování.

Jiný aspekt testování je přijít na to, co měřit. Náš testovací systém měří spotřebu paměti ve třech různých bodech testování: před načtením stránky, po načtení všech stránek a po zavření všech záložek. Na každém místě také provádíme měření po 30 sekundách nečinnosti a poté, co vynutíme spuštění garbage collectoru. Tato měření pomáhají zjistit, zda se neopakuje některý z problémů, se kterým jsme se setkali v minulosti. Například významný rozdíl mezi měřením +30 sekund a měřením po vynucení garbage collectoru může znamenat, že naše heuristiky garbage collectoru jsou příliš konzervativní. Významný rozdíl mezi měřením provedeným před načtením a měřením provedeným po zavření všech záložek může znamenat, že máme úniky paměti. Ve všech uvedených bodech měříme řadu hodnot, včetně velikosti rezidentní paměti (část paměti procesu, která je fyzicky v RAM a ne např. v odkládacím souboru), „explicitní" velikosti (velikost paměti, o kterou bylo požádáno přes malloc (), mmap (), atd.), a množství paměti, která spadá do některé kategorie v about:memory, např. neklasifikovaná halda.

Poté, co jsme tento systém dokončili, nastavili jsme jeho pravidelný běh na nejnovějších vývojových verzích Firefoxu. Také jsme ho spustili na předchozích verzích Firefoxu, zpětně zhruba od Firefox 4. Výsledkem je pseudo-kontinuální integrace s bohatou sadou historických dat. S výrazným přispěním Webdev jsme skončili areweslimyet.com, veřejné webové rozhraní ke všem údajům shromážděným naší infrastrukturou pro testování paměti. areweslimyet.com od svého ukončení detekoval několik regresí způsobených prací na různých částech prohlížeče.

5.7 Komunita

Konečným faktorem přispívajícím k úspěchu úsilí projektu MemShrink byla podpora ze strany širší komunity Mozilly. Zatímco většina (ale jistě ne všichni) z inženýrů pracujících na Firefoxu jsou v těchto dnech zaměstnáni Mozillou, komunita vibrujících dobrovolníků Mozilly přispěla podporou ve formě testování, lokalizace, QA, marketingu a dalších, bez nichž by se projekt Mozilla se skřípáním zastavil. Úmyslně jsme strukturovali MemShrink, abychom získali podporu komunity, což se značně vyplatilo. Jádro MemShrink týmu se skládalo z několika málo placených inženýrů, ale podpora ze strany komunity, kterou jsme obdrželi prostřednictvím hlášení chyb, testování a oprav doplňků znásobila naše úsilí.

Dokonce i v rámci komunity Mozilly bylo využití paměti již dlouho zdrojem frustrace. Někteří zažili problémy z první ruky. Jiní měli přátele nebo rodinu, kteří viděli problémy. Šťastlivci, kteří se tomu vyhnuli, nepochybně viděli stížnosti týkající se využití paměti Firefoxu či připomínky ptající se „je už únik opraven?“ na nových verzích, na kterých tvrdě dál pracovali. Nikoho netěší, když je jeho tvrdá práce kritizována, zvláště když jsou to věci, na kterých nepracuje. Adresování dlouhodobého problému, který se mohl týkat většiny členů komunity, byl prvním krokem k budování podpory.

Říkat, že tyto věci opravíme, ale nebylo dost. Museli jsme ukázat, že to myslíme vážně a můžeme dosáhnout skutečného pokroku v řešení problémů. Každý týden jsme pořádali veřejné schůzky ohledně třídění reportů o chybách a diskutovali jsme projekty, na kterých jsme pracovali. Nicholas rovněž blogoval report o progresu z každé schůzky, a tak lidé, kteří tam nebyli, mohli vidět, co jsme dělali. Zvýraznění realizovaných zlepšení, změny v počtech chyb a nově zaregistrované chyby jasně ukázali úsilí, které jsme do projektu MemShrink vkládali. A první zlepšení, která jsme byli schopni dosáhnout díky snadno proveditelným opravám, ukázala, že bychom mohli tyto problémy vyřešit.

Poslední částí bylo uzavření zpětnovazební smyčky mezi širší komunitou a vývojáři pracujícími na MemShrink. Nástroje, které jsme diskutovali dříve, odhalily chyby, které by byly uzavřeny jako nereprodukovatelné a zapomenuté pro reporty chyb, které by měly být a byly opraveny. Také jsme zahrnuli stížnosti, komentáře a odpovědi na naše blogované reporty o progresu do hlášení o chybách a snažili se získat potřebné informace pro jejich opravu. Všechny chybové reporty byly tříděny a byla jim přidělena priorita. Vynaložili jsme úsilí na prošetření všech zpráv o chybách, a to i těch, které byly určeny jako nepodstatné pro opravu.

Toto prozkoumání zvýšilo pocit ocenění u reportujících, a rovněž ponechalo chybu ve stavu, kdy někdo, kdo měl víc času, mohl přijít a opravit chybu později. Dohromady tyto akce vybudovaly silnou základnu podpory v komunitě, která nám poskytla skvělé hlášení chyb a neocenitelnou pomoc při testování.

5.8 Závěr

V průběhu dvou let, kdy byl projekt MemShrink aktivní, jsme dosáhli velkého zlepšení ve využití paměti Firefoxu. Tým projektu MemShrink změnil využití paměti z jedné z nejčastějších stížností uživatelů na výhodu prohlížeče a významně zlepšil uživatelský dojem mnoha uživatelů Firefoxu.

Rád bych poděkoval Justinu Lebarovi, Andrewovi McCreightovi, Johnu Schoenickovi, Johnnymu Stenbäckovi, Jetu Villegasovi a Timothymu Nikkelovi za všechnu jejich práci na projektu MemShrink, a stejně tak i dalším inženýrům, kteří pomohli opravit problémy s pamětí. Ze všeho nejvíc děkuji Nicholasu Nethercoteovi za rozběhnutí projektu MemShrink, intenzivní práci na snížení využití paměti jádra SpiderMonkey, vedení projektu po dobu dvou let a mnoho dalších věcí. Také bych rád poděkoval Jetu a Andrewovi za revizi této kapitoly.

6 Aplikování vzorů optimalizčních principů na nasazení komponent a konfigurační nástroje

(Doug C. Schmidt, William R. Otte a Aniruddha Gokhale)

6 Aplikování vzorů optimalizčních principů na nasazení komponent a konfigurační nástroje

6.1 Úvod

Distribuované, vestavěné systémy reálného času (DRE) jsou důležitou skupinou aplikací, které sdílejí vlastnosti jak podnikových distribuovaných systémů, tak vestavěných systémů s omezenými zdroji v reálném čase. Zejména aplikace v DRE systémech jsou podobné podnikovým aplikacím, to znamená, že jsou distribuovány přes velké oblasti. Navíc i aplikace v DRE systémech, stejně jako systémy v reálném čase a vestavěné systémy, jsou často kriticky důležité a odpovídají přísným servisním požadavkům na bezpečnost, spolehlivost a kvalitu (QoS).

Kromě komplexností popsaných výše, vytváří nasazení aplikačních a infrastrukturních komponent v DRE systémech vlastní sadu unikátních problémů. Za prvé, aplikace v DRE systémových doménách mohou mít konkrétní závislosti na cílovém prostředí, jako jsou zejména hardware / software (například GPS, senzory, pohony, speciální operační systémy reálného času atd.) Za druhé, zavedení infrastruktury DRE systému se musí vypořádat s přísnými požadavky na zdroje v prostředích s omezenými zdroji (například CPU, paměť, šířku pásma sítě atd.).

Softwarové inženýrství na bázi komponent (CBSE) [HC01] se stále častěji používá jako vzor pro vývoj aplikací jak v podnikových [ATK05], tak DRE systémech [SHS + 06]. CBSE usnadňuje systematické opětovné použití softwaru tím, že nabádá vývojáře vytvářet komponenty typu černá skříňka, které spolupracují vzájemně a s jejich prostředím přes dobře definovaná rozhraní. CBSE také ulehčuje nasazení vysoce komplexních distribuovaných systémů [WDS + 11] tím, že poskytuje standardizované mechanismy, které řídí konfiguraci a životní cyklus aplikací. Tyto mechanismy umožňují složení rozsáhlých, komplexních aplikací z menších, lépe řiditelných funkčních jednotek, například hotových komerčních komponentů a k použití předem připravených stavebních bloků. Tyto aplikace mohou být baleny společně s popisnými a konfiguračními metadaty a dány k dispozici pro nasazení do produkčního prostředí.

V návaznosti na odborné znalosti získané při vývoji ACE ORB (TAO) [SNG + 02] - open-source implementace Common Object Request Broker Architecture (CORBA) standardu - jsme v uplynulém desetiletí uplatnili CBSE principy na DRE systémech. V důsledku tohoto úsilí jsme vyvinuli vysoce kvalitní implementaci open-source OMG CORBA Component Model (CCM), kterou nazýváme Component Integrated ACE ORB (CIAO) [Insty]. CIAO implementuje takzvanou lehkou CCM [OMG04] specifikaci, která je podmnožinou plného CCM standardu, jež je vyladěn pro DRE systémy s omezenými zdroji.

V souvislosti s naší prací na uplatňování zásad CBSE na DRE systémech jsme také zkoumali neméně náročný problém usnadnění nasazení a konfigurace systémů na bázi komponent v těchto doménách.

Řízení nasazení a konfigurace aplikací na bázi komponent je závažný problém z následujících důvodů:

- *Závislost komponent a řízení verzí.* Můžou existovat složité požadavky a vztahy mezi jednotlivými komponentami. Aby správně fungovaly, komponenty mohou záviset na sobě navzájem, nebo specificky vyžadují nebo vylučují určité verze. Pokud tyto vztahy nejsou popsány a vyžadovány, nasazení komponentové aplikace může selhat; v horším případě může špatně fungovat omezeným a zhoubným způsobem.

- *Řízení konfigurace komponent.* Komponenta může být závislá na konfiguračních detailech, které mění její chování, a cílová infrastruktura musí řídit a používat všechny požadované konfigurační informace. Kromě toho může mít několik komponent při nasazení provázané konfigurační vlastnosti a cílová infrastruktura by měla zajistit, aby tyto vlastnosti zůstaly konzistentní napříč celou aplikací.

- *Distribuované propojení a řízení životního cyklu.* V případě podnikových systémů musí být komponenty nainstalovány a jejich propojení a aktivace řízeny na vzdálených hostitelích.

Tyto dva faktory měly na DAnCE několik vlivů. Například, úzce zaměřené případy užití často přiřadily nízkou prioritu vyhodnocení end-to-end výkonu reálně nasazených aplikací. Kromě toho nedostatek jednotné architektonické vize v kombinaci s pevnými lhůtami často znamenal, že ve jménu účelnosti byla přijata špatná architektonická rozhodnutí a nebyla později napravena. Na tyto problémy jsme se zaměřili, když jsme začali spolupracovat s našimi komerčními sponzory na aplikování DAnCE pro rozsáhlé nasazení, čítající stovky až tisíce komponent na desítkách až stovkách hardwarových uzlů. Zatímco úzce zaměřené případy užití by měly přijatelné časy nasazení, tato větší nasazení by trvala nepřijatelně dlouhou dobu, aby byla dokončena v řádu jedné hodiny nebo i více.

V reakci na tyto problémy jsme komplexně zhodnotili architektury, design a implementaci DAnCE a vytvořili novou implementaci, kterou nazýváme Locality-Enabled DAnCE (LE-dance) [OGS11] [OGST13]. Tato kapitola se zaměřuje na zdokumentování a aplikování vzorů optimalizačních principů, které tvoří jádro LE-DAnCE, z kterých jsou mnohé vhodné pro DRE systémy.

Tabulka 6.1 shrnuje obecné optimalizační vzory [Var05], z nichž mnohé aplikujeme v LE-DAnCE. Dodatečným cílem této práce bylo doplnit tento katalog o nové vzory, které byly identifikované v průběhu naší práci na LE-DAnCE.

Název	Princip	Příklad z oblasti sítí
Zamezení plýtvání	Vyhnout se zjevnému plýtvání	zero-copy [PDZ00]
Přesouvání v čase	Přesouvání výpočtů v čase (předběžný výpočet, líné vyhodnocování, sdílení výdajů, dávkování)	Zpracování integrované vrstvy[CT90], copy-on-write [ABB+86, NO88]
Specifikace uvolnění	Specifikace uvolnění (kompromis určitost za čas, kompromis přesnost za čas, a posunutí výpočtu v čase)	férové řazení [SV95], IPv6 fragmentace
Využití jiných komponent	Využití jiných systémových komponent (využití lokality, kompromis – paměť za rychlost, využití hardware)	Vyhledání Lulea IP [DBCP97], TCP/IP kontrolní součet
Přidání Hardware	Přidej hardware pro zlepšení výkonu	Seřazení IP vyhledávání [HV05], čítače
Efektivní postupy	Vytvoř efektivní postupy	UDP vyhledávání
Vyhýbání se obecnosti	Vyhni se zbytečným obecnostem	Fbufs [DP93]
Specifikace vs. Implementace	Nepleťte si specifikaci a implementaci	Upcalls [HP88]
Předávání nápověd	Předávání informací jako nápovědy v rozhraních	Filtry paketů [MJ93, MRA87, EK96]
Předávání informací	Předávání informací v hlavičkách protokolu	Přepínání tagů [RDR+97]
Očekávaný případ užití	Optimalizuj očekávaný případ	Predikce hlavičky [CJRS89]
Využívání stavu	Přidej nebo využij stavu pro získání rychlosti	Aktivní VC seznam
Stupně volnosti	Optimalizuj stupně volnosti	IP trie vyhledávání [SK03]
Využij konečné vesmíry	Použij speciální techniky pro konečné vesmíry	Časová kola [VL97]
Efektivní datové struktury	Použij efektivní datové struktury	4-úrovňové přepínání

Tabulka 6.1: Katalog optimalizačních principů a známých případů užití v síťové oblasti [Var05]

Zbývající část této kapitoly je organizována takto: oddíl 6.2 poskytuje přehled o OMG D&C specifikacích; oddíl 6.3 identifikuje nejvýznamnější zdroje výkonnostních problémů DAnCE (parsuje informace o nasazení z XML, analýza informací o nasazení v runtime a sériové provedení kroků nasazení) a používá je jako případové studie pro identifikaci optimalizačních principů, které (1) jsou obecně aplikovatelné na DRE systémy, a (2) jsme aplikovali na LE-DAnCE; a oddíl 6.4 prezentuje závěrečné poznámky.

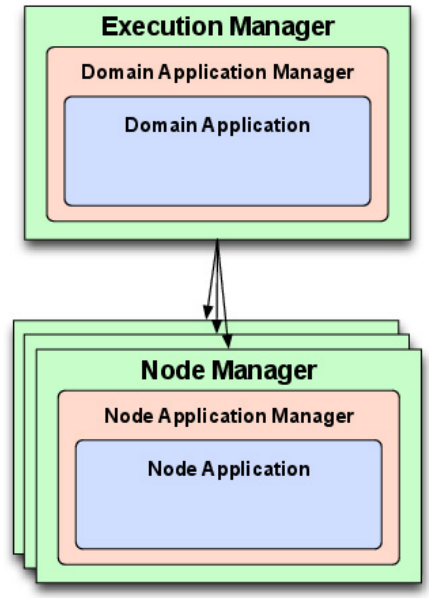
6.2 Přehled DAnCE

OMG D&C specifikace poskytuje standardní formáty pro vzájemnou výměnu metadat používanou v celém rozsahu životního cyklu vývoje aplikací na bázi komponent, stejně jako runtime rozhraní používané k balíčkování a plánování. Tato runtime rozhraní poskytují pokyny o nasazení cílové middleware infrastruktury pomocí plánu nasazení komponent, který obsahuje kompletní sadu informací o nasazení a konfiguračních informacích pro instance komponent a přidružené informace o jejich propojení. Během inicializace DRE systému musí být tyto informace parsovány, komponenty nasazeny na fyzické hardwarové prostředky a systém včas aktivován.

Tato část obsahuje stručné shrnutí základních architektonických prvků a postupů, které musí být provedeny v implementaci D&C kompatibilní se standardy. Toto shrnutí používáme jako základ k diskusi o značných výkonových a škálovacích problémech v DAnCE, který je naší open-source implementací OMG Deployment and Configuration (D&C) specifikace [OMG06], jak je uvedeno v části 6.1. Tento přehled je rozdělen do tří částí: (1) DAnCE runtime architektura, která popisuje démony a aktory, kteří jsou přítomni v systému, (2) datový model popisující strukturu „plánů nasazení" komponentních aplikací a (3) proces nasazení, který poskytuje vysokoúrovňový přehled procesu, jímž je nasazení distribuované aplikace realizováno.

Architektura Runtime D&C

Runtime rozhraní definovaného podle OMG D&C specifikace pro nasazení a konfiguraci komponent se skládá z dvouúrovňové architektury znázorněné na Obrázku 6.1.



Obrázek 6.1: OMG D&C architektonický přehled a oddělení funkcí

Tato architektura se skládá ze (1) souboru globálních (celosystémových) subjektů použitých ke koordinaci nasazení a (2) sady lokálních (úroveň uzlu) subjektů použitých k vytvoření instancí komponent a konfigurace jejich vzájemného propojení a QoS vlastností. Každý subjekt na těchto globálních a lokálních úrovních odpovídá jedné z následujících tří hlavních rolí:

- **Manažer:** Tato role (na globální úrovni známá jako VýkonnýManažer a na úrovni uzlu jako UzlovýManažer) je samostatným démonem, který koordinuje všechny entity nasazení v jednom kontextu. Manažer slouží jako vstupní bod pro všechny činnosti nasazení a jako továrna vytvářející instance, které implementují role Aplikačního manažera.

- **AplikačníManažer:** Tato role (na globální úrovni známá jako DoménovýAplikačníManažer a na úrovni uzlu jako UzlovýAplikačníManažer) koordinuje životní cyklus pro spuštěné instance aplikace na bázi komponent. Každý AplikačníManažer reprezentuje přesně jednu aplikaci na bázi komponent a slouží k zahájení nasazení a demontáže této aplikace. Tato role také slouží jako továrna na implementaci role Aplikace.

- **Aplikace:** Tato role (na globální úrovni známá jako DoménováAplikace a na úrovní uzlu jako UzlováAplikace) představuje nasazenou instanci aplikace na bázi komponent. Používá

se k finalizaci konfigurace propojených instancí komponent, které tvoří aplikaci a zahajují provádění nasazené aplikace na bázi komponent.

Datový model nasazení D&C

Kromě runtime objektů popsaných výše obsahuje D&C specifikace také rozsáhlý datový model, který se používá k popisu komponentních aplikací po celou dobu jejich životního cyklu nasazení. Metadata definovaná podle této specifikace jsou určena pro použití jako:

- formát pro vzájemnou výměnu mezi různými nástroji (například vývojovými nástroji, aplikacemi modelování a balíčkování, a nástroji pro plánování nasazení), využity pro tvorbu aplikací, a

- direktivy popisující konfiguraci a nasazení používané runtime infrastrukturou.

Většina entit v D&C metadatech obsahuje části, kde mohou být konfigurační informace obsaženy ve formě posloupnosti dvojic názvů / hodnot, kde může být hodnota libovolný datový typ. Tyto konfigurační informace lze použít k popisu všeho, od základních konfiguračních informací (například sdílené vstupní body knihovny a propojení komponenta/kontejner), až po složitější konfigurační informace (například QoS vlastnosti nebo inicializace atributů komponenty s uživatelsky definovanými datovými typy).

Tato metadata lze obecně rozdělit do tří kategorií: balíček, doména a nasazení. Popisovače balíčku jsou používány od začátku vývoje aplikace k určení rozhraní komponent, prostředků a požadavků. Poté, co byly vytvořeny implementace, tato metadata se dále používají k seskupení jednotlivých komponent do sestav, popisujících párování s implementačními artefakty, jako jsou například sdílené knihovny (také známy jako dynamicky linkované knihovny), a vytvoření balíčků, které obsahují jak metadata, tak implementace, které mohou být instalovány do cílového prostředí.

Popisovače domény jsou používány správci hardwaru pro popsání prostředků (například CPU, paměť, místo na disku a speciální hardware, jako jsou GPS přijímače) přítomné v doméně.

Proces nasazení OMG D&C

Nasazení komponentní aplikace jsou prováděna ve čtyřfázovém procesu kodifikovaném standardem OMG D&C. Manažer a AplikačníManažer jsou zodpovědní za první dvě fáze a Aplikace je odpovědná za poslední dvě fáze, jak je popsáno níže:

1. *Příprava plánu.* V této fázi je plán nasazení realizován VýkonnýmManažerem, který (1) analyzuje plán a určí, které uzly jsou zapojeny do nasazení a (2) rozdělí plány do „místně omezených“ plánů, jeden pro každý uzel obsahující informace pouze pro odpovídající uzel. Tyto místně omezené plány mají pouze instance a informace o připojení pro jeden uzel. Každý UzlovýManažer je pak kontaktován a opatřen místně omezeným plánem, což způsobí

vytvoření UzlovýchAplikačníchManažerů, jejichž odkaz je vrácen. Nakonec VýkonnýManažer vytvoří DoménovéhoAplikačníhoManažera s těmito odkazy.

2. *Start spuštění.* Když DoménovýAplikačníManažer obdrží pokyn pro start spuštění, deleguje na každém uzlu práci na UzlovéAplikačníManažery. Každý UzlovýAplikačníManažer vytváří UzelAplikace, který načte všechny instance komponent do paměti, provádí předběžnou konfiguraci a sbírá odkazy na všechny koncové body popsané v plánu nasazení. Tyto odkazy jsou pak uloženy v mezipaměti instance DoménovéAplikace vytvořené DoménovýmAplikačnímManažerem.

3. *Konec spuštění.* Tato fáze je zahájena operací na instanci DoménovéAplikace, která rozděluje v předchozí fázi shromážděné objektové reference ke každé UzlovéAplikaci a způsobuje, že tato zahájí tuto fázi. Všechny instance komponent přijmou konečnou konfiguraci a jsou tak vytvořena všechna propojení.

4. *Start.* Tato fáze je opět zahájena na DoménovéAplikaci, která ji deleguje instancím UzlovéAplikace a způsobuje, že tyto instruují všechny instance instalovaných komponent, aby začaly provedení.

6.3 Aplikování vzorů optimalizačních principů na DAnCE

Tento oddíl zkoumá tři z nejproblematičtějších výkonnostních problémů, které jsme identifikovali při použití DAnCE na komponentní aplikace v rozsáhlých produkčních DRE systémech. Nejprve popisujeme případovou studii, která se zaměřuje na mnohé z těchto výkonnostních problémů. Dále identifikujeme příčiny zhoršení výkonu a využíváme tuto diskusi na prezentaci principů optimalizace, což jsou hlavní směry, které mohou být použity i v jiných situacích a aplikacích k odstranění nebo zamezení problémů s výkonem.

Přehled platformy SEAMONSTER

Příkladem DRE systému, u kterého jsme použitím DAnCE odhalili významné výkonnostní problémy, byla spolupráce s University of Alaska na platformě *South East Alaska Monitoring Network for Science, Telecommunications, Education, and Research* (SEAMONSTER). SEAMONSTER je síť snímačů ledovce a povodí hostovaná na University of Alaska Southeast (UAS) [FHHC07]. Tato síť snímačů monitoruje a shromažďuje data o dynamice ledovce a hmotnostní bilance, hydrologii povodí, pobřežní mořské ekologie a vliv/hazardy člověka uvnitř a okolo povodí řeky Lemon Creek a ledovce Lemon Glacier. Shromážděná data jsou využívána ke studiu korelace mezi rychlostí ledovce, formováním a odvodněním ledovcového jezera, hydrologie povodí a kolísání teploty.

Síť snímačů SEAMONSTER obsahuje senzory a povětrnostní počítačové platformy, které jsou nasazeny na ledovec a na celém povodí shromažďují údaje zajímavé pro vědce. Údaje

shromážděné snímači jsou přenášeny prostřednictvím bezdrátových sítí do klastru serverů, které filtrují, korelují a analyzují data. Efektivní nasazení aplikací pro sběr a filtraci dat v poli hardwaru systému SEAMONSTER a dynamické přizpůsobování měnícím se podmínkám životního prostředí a dostupnosti zdrojů, představují významné softwarové výzvy pro efektivní provoz SEAMONSTER. Zatímco SEAMONSTER servery poskytují značnou výpočetní kapacitu, oblast hardware je výpočetně omezená.

Oblastní uzly v síti snímačů pozorují ve své oblasti často velké množství jevů. Druh, délka a četnost pozorování těchto jevů se mohou v čase měnit, a to na základě změn v životním prostředí, výskytu přechodných událostí v oblasti životního prostředí a měnících se záměrů a cílů v rámci vědeckého poslání sítě senzorů. Kromě toho, omezený výkon, kapacita zpracování, ukládání a šířka pásma sítě omezují schopnost těchto uzlů neustále provádět pozorování s požadovanou frekvencí a přesností. Dynamické změny v podmínkách prostředí ve spojení s omezenou dostupností zdrojů vyžadují, aby jednotlivé uzly v síti senzorů rychle revidovaly stávající provoz a plány do budoucna, aby co nejlépe využívaly svých zdrojů.

K řešení těchto problémů jsme navrhli převedení sběru dat a zpracování úloh na middleware platformu postavenou nad middleware CIAO a DAnCE popsané v části 6.1, resp. části 6.2. Vyvinuli jsme runtime plánovač [KOS +08], který analyzoval fyzikální pozorování senzorových uzlů. Na základě těchto informací, jakož i operativních cílů sítě, vytváří plánovač plány nasazení popisem požadované softwarové konfigurace.

Použití DAnCE k aplikaci změn nasazení požadovaných runtime plánovačem však odhalilo řadu nedostatků v jeho výkonu. Tyto nedostatky byly umocněny omezeným výkonem hardwaru v terénu, relativní pomalostí sítě spojující uzly a přísnými požadavky na reálný čas systému. Každý z těchto nedostatků je popsán níže.

Optimalizace parsování plánu nasazení

Kontext

Nasazení komponentní aplikace pro OMG D&C jsou popsána datovou strukturou obsahující všechny podstatné konfigurační metadata pro instance komponent, jejich mapování do jednotlivých uzlů, a všechny informace potřebné k připojení. Tento plán nasazení je serializovaný na disku ve formátu souboru XML, jehož struktura je popsána pomocí XML schématu definovaného D&C specifikací. Tento formát XML dokumentu představuje významné výhody tím, že poskytuje jednoduchý formát pro vzájemnou výměnu plánů nasazení mezi modelovacími nástroji [GNS + 02].

Například v případové studii SEAMONSTER poskytuje tento formát vhodný formát pro vzájemnou výměnu mezi plánovaným front-endem a infrastrukturou pro nasazení. Tento formát lze také snadno generovat a manipulovat s ním za použití běžně dostupných XML modulů pro populární programovací jazyky. Navíc umožňuje jednoduchou úpravu a dolování dat zpracováním textu v nástrojích jako jsou Perl, grep, sed a awk.

Problém
Zpracování těchto souborů s plány nasazení během nasazení a dokonce za běhu však mohou vést k podstatným výkonovým sankcím. Tyto výkonové sankce vyplývají z následujících zdrojů:

- Velikosti XML souborů s plánem nasazení výrazně rostou, když v nasazení roste počet instancí komponent a připojení, což způsobuje významnou I / O režii na načtení plánu do paměti a ověření struktury proti schématu k zajištění jeho správného sestavení.

- Formát XML dokumentu nelze přímo použít pro cílovou infrastrukturou, protože tato infrastruktura je CORBA aplikací, která implementuje rozhraní OMG Interface Definition Language (IDL). Z tohoto důvodu musí být XML dokument nejdříve převeden do formátu IDL, který používá rozhraní běhového systému.

V DRE systémech není neobvyklé nasazovat komponenty v řádech tisíců. Navíc instance komponent v těchto doménách vykazují vysoký stupeň propojení. Oba tyto faktory přispívají k rozsáhlým plánům. Nicméně plány nemusí být rozsáhlé, aby významně ovlivnily provoz systému. Ačkoli plány v SEAMONSTER případové studii popsané výše byly významně menší, extrémně omezené výpočetní zdroje znamenaly, že režie na zpracování i malých plánů byla často příliš časově náročná.

Vzory optimalizačních principů v parsování konfiguračních metedat

Existují dva obecné přístupy k řešení problému analýzy XML uvedené v části 6.3.

1. Optimalizovat schopnost zpracování XML do IDL.
DAnCE, nástroj nazvaný XML Schema Compiler (XSC), který využívá slovník specifický pro XML [WKNS05]. XSC čte D&C XML schémata a generuje rozhraní založené na C ++ pro XML dokumenty postavené nad programovým API Document Object Model (DOM) XML. DOM je časově/ prostorově náročný přístup, protože celý dokument musí být nejprve zpracován tak, aby byla před zahájením procesu překladu XML do IDL sestavena reprezentace dokumentu na bázi stromu. Vzhledem k tomu, že datové struktury plánu nasazení obsahují rozsáhlé interní křížové odkazy, nepřinesla by alternativa k DOM na zpracování plánů nasazení včetně událostmi řízených mechanismů, jako je Simple API pro XML (SAX), také žádné podstatné zlepšení.

XSC generuje C ++ kód, který obsahuje určitý počet tříd (na základě obsahu schématu XML), které poskytují silně typovaný objektově orientovaný přístup k datům v dokumentu XML. Navíc toto rozhraní využívá vlastností C ++ STL, aby tak pomohlo programátorům napsat kompaktní a efektivní kód pro interakci s jejich daty. Obecný postup pro naplnění těchto tříd je: 1) parsovat dokument XML pomocí DOM XML parseru; 2) parsovat DOM strom k naplnění generované hierarchie tříd. Za účelem zvýšení kompatibility se STL algoritmy a funktory, XSC ukládá data ve specializovaných STL kontejnerech (tj. třídách specializovaných ze STL šablon).

Původní verze XSC data bindingu byla vysoce neefektivní. Dokonce i relativně skromné nasazení řádově třeba jen několika set až několika tisíc komponent se zpracovávalo téměř půl hodiny. Po analýze vykonávání tohoto procesu pomocí nástrojů, jako je Rational Quantify, byl odhalen velmi jednoduchý problém: generovaný XSC kód jednotlivě vkládal prvky do jeho vnitřních datových struktur (v tomto případě, std :: vector) naivním způsobem. V důsledku toho strávil přemrštěné množství času přerozdělením a kopírováním dat uvnitř těchto kontejnerů, když další vložený prvek způsobil alokaci paměti.

Níže představujeme specifické pokyny, kterých si vývojáři musí být vědomi:

- Buďte si vědomi nákladů na své abstrakce. Abstrakce na vysoké úrovni, jako jsou například kontejnerové třídy, které jsou k dispozici v C ++ STL, můžou značně zjednodušit programy snížením potřeby reprodukovat (z velké části standardizovaný) složitý kód náchylný k chybám na nižší úrovni. Je důležité charakterizovat, dokumentovat (při psaní abstrakce) a porozumět (při jejich použití), jaké skryté náklady mohou vzniknout pomocí operací vyšší úrovně, poskytovaných vaší abstrakcí.

- Používejte vhodné abstrakce pro váš případ užití. Často existuje možnost volby abstrakcí, které poskytují podobné funkce. Příkladem může být volba mezi std::vector a std::list; každá má své výhody. V XSC byl zpočátku používán std :: vector, protože jsme žádali náhodný přístup k prvkům v data bindingu; cenou byl extrémně špatný výkon při analýze XML dokumentu kvůli špatnému výkonu vložení. Náš případ užití však vyžaduje pouze sekvenční přístup, takže nakonec byl více žádoucí mnohem lepší výkon vložení std::list.

Pochopením specifických požadavků konkrétního případu užití našich z XML generovaných tříd, zejména to, že většina uzlů je navštívena jednou a je možné je navštívit v řadě, jsme schopni aplikovat vzor Očekávaný případ užití aplikováním dvou dalších optimalizačních vzorů. V tomto případě je použitelný vzor Vyhýbání se obecnosti, protože se vědomě vyhýbáme obecnosti generování tříd pro přístup k datům bez náhodného přístupu ke kontejnerům. Dále jsme se rozhodli použít nejefektivnější datovou strukturu (vzor Efektivní datové struktury), abychom se vypořádali s nedostatkem obecnosti.

2. Předběžné zpracování XML souborů pro nasazení, která jsou citlivá na dobu zpracování. Optimalizace procesu konverze XML do IDL přinesla přijatelnější časy konverze, ale tento krok v procesu nasazení stále spotřebovával velkou část z celkového času potřebného pro nasazení.

Této dosud nevyřešené režii se lze vyhnout použitím dalšího optimalizačního vzoru: Pokud je to možné, provádějte nákladné výpočty mimo kritickou cestu. V mnoha případech může být výsledek nákladných postupů a výpočtů předem vypočten a uložen pro pozdější použití. To platí zejména v případech, jako je například XML plán nasazení, u kterého je vysoce nepravděpodobná změna mezi okamžikem, kdy je vygenerován, a kdy je požádáno o nasazení aplikace.

Tento optimalizační přístup aplikuje optimalizační vzor Přesouvání v čase přesunutím nákladné konverze plánu nasazení do efektivnějšího binárního formátu mimo kritickou cestu nasazení aplikace. Při použití tohoto vzoru jsme nejprve převedli plán nasazení do jeho IDL formy pro běhové prostředí. Pak jsme serializovali výsledek na disk pomocí Common Data Representation (CDR) [OMG08] binárního formátu, definovaného podle CORBA specifikace. SEAMONSTER online plánovač mohl využít této optimalizace tím, že vyprodukoval za významného snížení zpoždění místo XML plánů binární plány nasazení.

Tento CDR binární formát nezávislý na platformě a používaný pro ukládání plánu nasazení na disku, má stejný formát, jaký je používán i pro přenos plánu přes síť za běhu. Výhodou tohoto přístupu je to, že využívá silně optimalizované deserializační manipulační programy, poskytované na základě CORBA implementace. Tyto manipulační programy vytvoří z binárního proudu na disku reprezentaci datové struktury plánu nasazení v paměti.

Optimalizace plánu analýzy

Kontext

Po načtení plánu nasazení komponent do paměti, před provedením jakékoli následné aktivity procesu nasazení, musí být analyzována cílová middleware infrastruktura. Tato analýza se vyskytuje ve fázi přípravy plánu a je popsána v části 6.2. Cílem této analýzy je zjistit (1) počet dílčích problémů nasazení, které jsou součástí plánu nasazení, a (2) které instance komponent patří ke kterému dílčímu problému.

Jak již bylo zmíněno v části 6.2, výstupem tohoto procesu analýzy je sada „místně omezených" dílčích plánů. Místně omezený dílčí plán obsahuje všechna potřebná metadata k provedení úspěšného nasazení. Z tohoto důvodu obsahuje kopie informací obsažených v původním plánu (popsáno v části 6.2).

Analýza runtime plánu je vlastně prováděna v průběhu fáze přípravy plánu nasazení dvakrát: jednou na globální úrovni a znovu v každém uzlu. Plány globálního nasazení jsou rozděleny podle uzlu, ke kterému jsou přiřazeny jednotlivé instance. Výsledkem této dvoudílné analýzy je nový dílčí plán pro každý uzel, který obsahuje jen instance, připojení a jiná metadata komponent, která jsou potřebná pro tento uzel.

Algoritmus pro dělení plánů používaný naší DAnCE implementací D&C specifikace je jednoduchý. Pro každou instanci, která má být nasazena v plánu, algoritmus určuje, který dílčí plán by ji měl obsahovat a načíst příslušnou (nebo vytvořit novou) datovou strukturu dílčího plánu. Stejně jako je určen tento vztah, zkopírují se všechna metadata nutná pro tuto instanci komponenty do dílčího plánu, a to včetně připojení, metadat popisujících spustitelné soubory, sdílené knihovny závislostí atd.

Problém
I když je tento přístup koncepčně jednoduchý, je plný náhodných složitostí, které v praxi přinášejí následující neefektivity:

1. *Referenční zastoupení v IDL.* Plány rozmístění jsou obvykle přenášeny v sítích, takže musí dodržovat pravidla mapování jazyka CORBA IDL. Vzhledem k tomu, že IDL nemá žádný koncept referencí nebo ukazatelů, musí se k popisu vztahů mezi elementy plánu použít nějaký alternativní mechanismus. Plán nasazení ukládá všechny hlavní elementy v sekvencích, takže odkazy na jiné entity mohou být do těchto sekvencí reprezentovány pomocí jednoduchých indexů. I když tato implementace může následovat odkazy v konstantním čase, znamená to také, že se stanou tyto odkazy neplatné, pokud jsou entity plánu zkopírovány do dílčích plánů, jak se jejich postavení v sekvencích plánu nasazení bude s největší pravděpodobností lišit. Je také nemožné určit, zda byl již zkopírován cíl odkazu bez prohledávání dílčího plánu, což je časově náročné.

2. *Alokace paměti v sekvencích plánu nasazení* - mapování CORBA IDL požaduje, aby byla sekvence uložena na po sobě následujících paměťových adresách. Je-li změněna velikost sekvence, její obsah bude s největší pravděpodobností kopírován na jiné místo v paměti, aby se přizpůsobil větší velikosti sekvence. S výše shrnutým přístupem, jak roste velikost plánu, dojde k podstatnému zvýšení režie na kopírování. Tato režie je problematická zejména v systémech s omezenými zdroji (jako je naše případová studie SEAMONSTER), jehož omezená paměť musí zůstat využitelná pro komponenty aplikace. V případě, že je cílová infrastruktura neefektivní ve využívání tohoto zdroje, vyčerpá buď dostupnou paměť, nebo způsobí významné zahlcení dostupné virtuální paměti (obě mají dopad na zpoždění nasazení a dobu použitelnosti flash paměti).

3. *Neefektivní paralelizace analýzy plánu.* Algoritmus popsaný výše by měl významně těžit z paralelizace, protože proces analýzy jediné komponenty a určení, které elementy musí být zkopírovány do dílčího plánu, je nezávislý na všech ostatních komponentách. Provedení algoritmu ve více vláknech by však pravděpodobně nebylo efektivní, protože musí být serializován přístup k dílčím plánům ke kopírování metadat instance, aby nedošlo k poškození dat. V praxi jsou instance komponent v prováděcím plánu obvykle seskupeny podle uzlu a/nebo procesu, protože jsou plány nasazení často generovány z modelovacích nástrojů. V důsledku toho by pravděpodobně soutěžilo více vláken o zamčení stejného dílčího plánu, který by způsobil, že by „paralelizovaný" algoritmus běžel do značné míry sekvenčně. Zatímco paralelizace byla historicky viděna jako neaplikovatelná na DRE systémech s omezenými zdroji (např. SEAMONSTER), nástup vícejádrových procesorů v jediné základové desce počítačů motivuje v těchto prostředích k použití paralelismu.

Vzory optimalizačních principů v analýze plánu nasazení

Tento výkonový problém by mohl být vyřešen použitím vzoru Specifikace vs. Implementace, a využitím některých z výše popsaných principů optimalizace pro nástroj XSC, a to zejména být si vědom nákladů na abstrakci, a použití vhodných kontejnerů pro případ užití. Například můžou být použity ukazatele/reference místo sekvence indexů k odkazování na související datové struktury, což potenciálně odstraňuje nutnost pečlivě přepsat odkazy, když jsou entity plánu kopírovány mezi plány. Stejně tak by mohl asociativní kontejner (například mapa STL) uložit místo sekvence plán objektů, čímž se zvyšuje efektivita vkládání plánu entit do dílčích plánů.

I když jsou tyto a další podobné možnosti lákavé, existují některé vnitřní složitosti v požadavcích standardu D&C, které činí tyto optimalizace méně atraktivními. Protože musí být data předána jiným entitám jako součást procesu nasazení, zavedlo by použití efektivnějších reprezentací pro analýzu ještě další konverzní krok do procesu nasazení. Tato konverze by potenciálně převážila veškeré přínosy dosažené touto novou reprezentací.

Mnohem atraktivnější výsledek je aplikovat na tento problém jinou sadu principů optimalizace, jak je uvedeno níže:

- *Uložte do mezipaměti dříve vypočítané výsledky pro pozdější použití.* To je příklad vzorů Posouvání v čase a Využívání stavu. Je možné provádět kroky jednoduché předběžné analýzy k předběžnému výpočtu hodnoty, který bude časově náročnější provést později. V tomto případě nejprve iterujeme plán pro stanovení velikosti finální velikosti nezbytné pro obsažení vypočítaných dílčích plánů a uložení tohoto stavu do mezipaměti pro pozdější použití.

- *Tam, kde je to možné, předběžně alokujte paměť pro vaše datové struktury.* V důsledku dodatečného stavu vzniklého v kroku předběžné analýzy popsané výše, můžeme použít *Zamezení plýtvání* a vyhnout se bezdůvodnému plýtvání předběžným alokováním sekvencí, které předtím byly znovu alokovány pokaždé, když se objevil nový element plánu.

- *Vytvořte si algoritmy využitím paralelizace.* I když to může být chápáno jako aplikace *Přidávání hardwaru*, tento vzor mluví spíše o využití paralelizačních schopností hardwaru. Navíc tento vzor mluví o přidání speciálního hardwaru k provádění specializovaných výpočtů.

  Využití více univerzálních procesorů je jeden důležitý nově vznikající princip. Vzhledem k tomu, že vícejádrové počítače pronikají do domén stolních počítačů a serverů a jsou čím dál tím běžnější i v oblasti vestavěných systémů, je stále důležitější navrhovat pro tuto důležitou vlastnost hardwaru. Proto navrhujeme další vzor, který budeme nazývat *Design pro paralelizaci*, kdy optimalizujeme návrh algoritmů a rozhraní pro paralelizaci, uvedenou v tabulce 6.2.

- *Strukturujte sdílený přístup k datům, aby se zabránilo zbytečnému používání synchronizace.* Synchronizace, například za použití vzájemného vyloučení k ochraně přístupu ke sdíleným

datům, je zdlouhavá a náchylná k chybám. Kromě toho může příliš horlivé využívání synchronizace často zcela negovat paralelizaci vašich algoritmů. Mnohem lepším přístupem je strukturovat vaše algoritmy, aby se úplně eliminovala nutnost použití synchronizace; vyžadujte pouze sdílený přístup k datům pro čtení místo sdíleného přístupu pro zápis.

Tento princip optimalizace je nejen důležitým společníkem vzoru *Design pro paralelizaci* navrhovaného výše, ale také obecně moudrá programovací praxe: uvíznutí a neošetřené souběhy vláken (race-condition) způsobené nesprávnou synchronizací jsou zhoubné a obtížně diagnostikovatelné chyby. Naše nedávná práce v softwarových frameworcích určených pro frakční kosmické lodě skutečně navrhla komponentní model, který úplně odstraňuje z kódu aplikace synchronizaci [DEG+12]. Za tímto účelem navrhujeme další optimalizační vzor, který nazýváme Zamezení synchronizace, v rámci kterého bychom se měli vyhnout příliš horlivé synchronizaci a zamykání, uvedené v tabulce 6.2 níže.

Tyto principy mohou být aplikovány na výše popsaný algoritmus a vytvořit tak verzi, která je mnohem vhodnější pro optimalizaci; nový algoritmus (spolu s tím, jak výše uvedené zásady ovlivňují design) je popsán níže.

1. *Fáze 1: Určení počtu vytvářených dílčích plánů.* V této fázi jedno vlákno iteruje přes všechny instance komponent obsažené v plánu nasazení pro určení počtu potřebných dílčích plánů. Je-li tato operace provedena na globální úrovni, vyžaduje konstantní dobu operace na instanci. Když se tato operace provádí na místní úrovni, vyžaduje, aby byla vyhodnocena místní omezení (popsáno v části 6.2). Vzhledem k tomu, že tato fáze je potenciálně časově náročná, výsledky jsou ukládány do mezipaměti pro pozdější použití. To je příklad *Posouvání v čase* a *Využívání stavu*.

2. *Fáze 2: Předběžně alokujte datové struktury pro dílčí plány.* Využitím informací sesbíraných ve fázi 1 předběžně alokujeme paměť pro datové struktury potřebné k sestavení dílčích plánů. Jako součást této předběžné alokace je možné rezervovat paměť pro každou sekvenci v datové struktuře dílčího plánu, aby se zabránilo opakované změně velikosti a kopírování. Aby byl odhad těchto délek efektivní, jsou ve fázi 1 shromažďovány statistiky. Toto je příklad *Zamezení plýtvání*.

3. *Fáze 3: Sestavte uzlově specifické dílčí plány.* Tato fáze nového procesu analýzy je podobná algoritmu popsanému na začátku této části. Hlavním rozdílem je, že výsledky fáze předběžné analýzy v mezipaměti jsou použity jako vodítko pro tvorbu dílčích plánů. Místo toho, aby se po pořadí uvažovala každá instance (jako původní implementace DAnCE dělala), LE-DAnCE plně sestavuje jeden dílčí plán v čase zpracování instance na základě jednotlivých uzlů. Tento přístup zjednodušuje paralelizaci této fáze vyčleněním jednoho vlákna na dílčí plán a vylučuje jakýkoliv sdílený stav mezi vlákny, s výjimkou přístupu typu jen pro čtení k původnímu plánu. Proto není nutné chránit přístup do dílčích plánů a používat jakékoliv zamykací mechanismus. Toto je příklad *Design pro paralelizaci* a *Zamezení synchronizace*.

Revidovaný algoritmus výše je mnohem efektivnější implementace analýzy plánu a může ukázat zlepšení i na jedno jádrových vestavěných procesorech typických pro případ užití SEAMONSTER: postup výše je podstatně paměťově efektivnější, a to jak z hlediska použitého prostoru, tak množství nezbytné opakované alokace. Použití vícejádrových vestavěných procesorů by výrazně zlepšilo výkon za běhu ve srovnání se starým algoritmem.

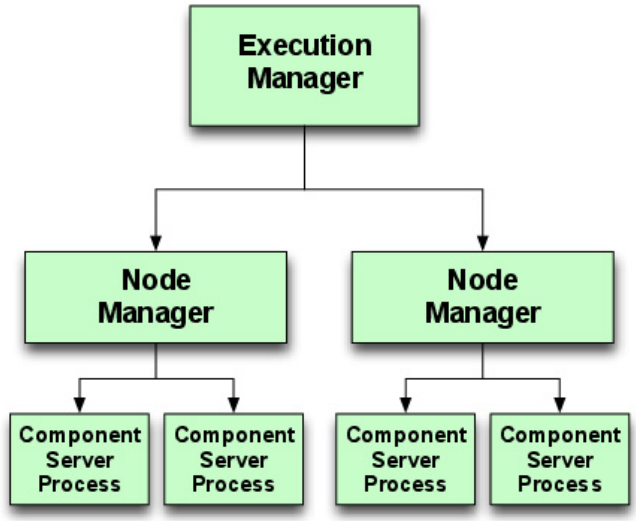
Optimalizace prostřednictvím snížení serializace vykonávání úkolů nasazení

Kontext
Komplexnosti uvedené níže prezentují sériové (neparalelní) vykonávání úkolů nasazení. Související zdroje zpoždění v DAnCE existují jak na globální úrovni, tak na úrovni uzlu. Na globální úrovni je tento nedostatek paralelizace výsledkem transportu pomocí CORBA, použitém v DAnCE. Nedostatek paralelismu na místní úrovni nicméně vyplývá z nedostatku specifičnosti týkající se rozhraní implementace D&C s cílovým komponentním modelem, který je obsažen v D&C specifikaci.

Proces nasazení D&C uvedený v části 6.2 umožňuje globálním entitám rozdělit proces nasazení do několika uzlově specifických dílčích úkolů. Každý dílčí úkol je odeslán do individuálních uzlů pomocí jediného vzdáleného volání, se všemi údaji získanými uzly předán zpět globálním entitám prostřednictvím výstupních parametrů, které jsou součástí podpisu operace popsané v IDL. Vzhledem k synchronní (požadavek / odezva) povaze CORBA protokolu zpráv použitého k implementaci DAnCE, je konvenčním přístupem zasílat tyto dílčí úkoly sériově každému uzlu. Tento přístup je snadno proveditelný na rozdíl od složitosti použití mechanismu CORBA asynchronní metody volání (AMI) [AOS +00].

Problém
Aby se minimalizovala počáteční složitost implementace, použili jsme synchronní vyvolání (nepochybně krátkozrace) jako volbu pro design úvodní implementace DAnCE. Tato globální synchronicita pracovala správně pro relativně malé nasazení s méně než 100 komponentami. Se zvyšujícím se počtem uzlů a instancí přiřazených k těmto uzlům však tato globální/lokální serializace přinášela výrazné zpoždění při nasazení.

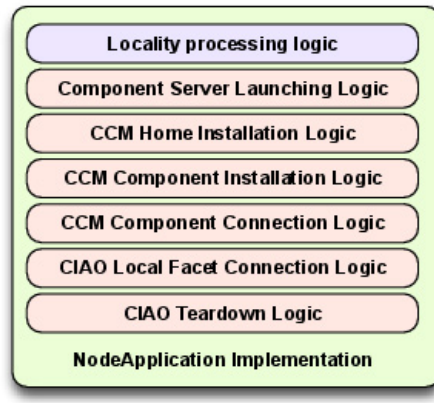
Toto serializované vykonávání přineslo nejproblematičtější snížení výkonu v naší SEAMONSTER případové studii, tj. omezené výpočetní zdroje dostupné na hardwaru v terénu způsobily, že to často trvalo několik minut. Takové zpoždění na úrovni uzlu se rychle může stát katastrofálním. Zejména i relativně skromné nasazení zahrnující desítky uzlů rychle stupňuje zpoždění při nasazení systému na půl hodiny nebo více.



Obrázek 6.2: Zjednodušená, serializovaná DAnCE architektura

Tento problém serializace však není omezen pouze na globální / místní odesílání úkolů; stejně existuje i v uzlově specifické části infrastruktury. D&C specifikace neposkytuje žádné vodítko, jak by měla UzlováAplikace komunikovat s cílovým komponentním modelem jako je CORBA Component Model (CCM), a místo toho považuje takovéto rozhraní za implementační detail.

D&C architektura byla v DAnCE implementovaná pomocí tří procesů, jak je zobrazeno na Obrázku 6.2. Procesy VýkonnýManažer a UzlovýManažer konkretizují jejich přidružené instance AplikačníManažer a Aplikace v jejich adresních prostorech. Když UzlováAplikace instaluje instance konkrétních komponent, založí podle potřeby jeden (nebo více) oddělených aplikačních procesů. Tyto aplikační procesy používají rozhraní odvozené ze starší verze CCM specifikace, která umožňuje UzlovéAplikaci vytvořit instance kontejnerů a komponent individuálně. Tento přístup je podobný přístupu používanému implementací CARDAMOM [Obj06] (což je další open source implementace CCM), která je přizpůsobena pro podnikové DRE systémy, jako jsou systémy řízení letového provozu.



Obrázek 6.3: Předchozí DAnCE implementace UzlovéAplikace

DAnCE architektura znázorněná na Obrázku 6.2 byla problematická s ohledem na paralelizaci, protože implementace UzlovéAplikace integrovala přímo veškerou logiku potřebnou pro instalaci, konfiguraci a připojení instancí (jak je znázorněno na Obrázku 6.3), než aby vykonávala pouze některé procesy a delegovala zbývající část konkrétní logiky nasazení na aplikační proces. Tato těsná integrace stěžovala paralelizaci instalačních procedur na úrovni uzlu z následujících důvodů:

- Množství dat sdílených logikou generického nasazení (část implementace UzlovéAplikace, která interpretuje plán) a logiky zaváděcího specifického nasazení (část, která má specifickou znalost o způsobu manipulace s komponenty) ztěžovala paralelizaci jejich instalace v rámci jednoho serveru komponent, protože tato data musí být upravena v průběhu instalace.

- Skupiny komponent instalovaných do oddělených aplikačních procesů byly považovány za samostatné nasazení dílčích úkolů, takže tato seskupení bylo zpracováno postupně jedno za druhým.

Vzory optimalizačních principů při redukci fází serializace

Podobně jako u problému analýzy popsaného výše, jedná se o problém, kde má nadměrná serializace dopad na výkon. V tomto případě však místo přehodnocení algoritmu procesu nasazení budeme znovu posuzovat architektonické řešení systému. V zájmu řešení výkonového problému jsme v tomto případě aplikovali tyto zásady pro optimalizaci DAnCE:

1. *Nedovolte, aby specifikace příliš omezovala váš design.* Při implementaci systémového nebo softwarového frameworku podle specifikace je často přirozené modelovat váš návrh dle omezení a implicitních předpokladů specifikace. Často můžete architektonicky navrhnout vaši implementaci s cílem zavést architektonické prvky nebo chování tak, aby zůstaly v mezích specifikace. Toto je příklad jak vzoru *Specifikace vs. Implementace*, tak vzoru *Stupně volnosti*.

2. *Udržujte přísné oddělení zájmů.* Ujistěte se, že váš systém pracuje ve vrstvách nebo modulech, které interagují prostřednictvím dobře definovaných rozhraní. To pomáhá zajistit, že stav každé vrstvy nebo modulu je dobře popsaný, což zjednodušuje interakce mezi logicky oddělenými částmi vašich aplikací a usnadňuje aplikovat vzor *Design pro paralelizaci*. Kromě toho pomáhá zajištění soběstačnosti stavu každé vrstvy aplikovat vzor Zamezení synchronizace.

   Kromě toho lze modularizací designu vašeho softwaru často odhalit způsoby, jak mohou být použity jiné vzory optimalizačních principů. Jako takový navrhujeme další vzor optimalizačního principu, Rozdělení zájmů, využitím oddělení zodpovědností modularizací architektury (shrnuté v tabulce 6.2). Ačkoli běžně může být úroveň nepřímosti odsuzována, protože by mohla vést k výkonovým penalizacím, někdy může odhalit nové příležitosti nebo pomůže použít jiné optimalizace.

3. *Zajistěte, aby tyto vrstvy nebo moduly mohly komunikovat asynchronně.* Pokud moduly nebo vrstvy ve vaší architektuře mají rozhraní, které předpokládají synchronní provoz, ztěžuje to využití paralelního provozu ke zlepšení výkonu. I v případě, že rozhraní je samo o sobě synchronní, je často možné použít i jiné techniky, jako je například využití abstrakce, které umožňuje interakci se synchronním rozhraním asynchronním způsobem. Zamezení vzájemné synchronní interakce je další důležitou aplikací vzoru *Design pro paralelizaci*.

Uplatňování těchto zásad na globální úrovni (například VýkonnýManažer) je popsáno v části 6.2, oddělení zájmů je udržováno na základě skutečnosti, že i zdroje na úrovni uzlu jsou v oddělených procesech, a pravděpodobně i v odlišných fyzických uzlech. Asynchronnosti je v tomto kontextu také snadné dosáhnout, protože jsme byli schopni využít CORBA asynchronní metod volání (AMI), které umožňují klientovi (v tomto případě globální infrastruktura) komunikovat asynchronně se synchronním rozhraním serveru (v tomto případě infrastruktura na úrovni uzlu) a vysílá více požadavků na jednotlivé uzly paralelně. Toto je příklad vzoru Stupně volnosti v tom, že specifikace neodmítá koncept asynchronní interakce mezi těmito subjekty.

Uplatňování těchto zásad na infrastrukturu na úrovni uzlu bylo nicméně náročnější. Jak je popsáno výše, naše počáteční implementace měla špatné oddělení zájmů, což extrémně stěžovalo exekuci ve více vláknech, abychom paralelizovali aktivitu nasazení na úrovni uzlu. Na podporu toho jsme vytvořili novou abstrakci na úrovni uzlu, kterou jsme nazvali ManažerLokality, který byl výsledkem použití výše uvedených zásad optimalizace.

LE-DAnCE architektura na úrovni uzlu (např. UzlovýManažer, UzlovýAplikačníManažer a UzlováAplikace) nyní funguje jako uzlově omezená verze globální části architektury OMG D&C. Spíše než přímo spouštěním instalací instancí konkrétních komponent UzlovouAplikací, je tato odpovědnost nyní přenesena na instanci ManažeraLokality. Infrastruktura na úrovni uzlu provádí druhé „rozdělení“ plánu obdrženého z globální úrovně seskupením instancí komponent do jednoho nebo více aplikačních procesů. UzlováAplikace pak plodí řadu procesů

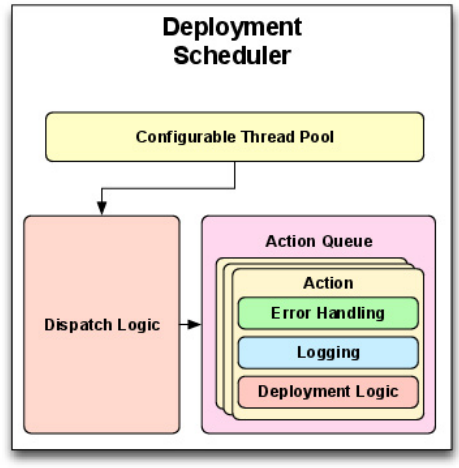
ManažeraLokality a deleguje tyto „procesně omezené“ (tj. obsahující pouze komponenty a připojení patřící do jediného procesu) plány paralelně do každého aplikačního procesu.

ManažerLokality je příkladem vzoru *Specifikace vs. Implementace*. Specifikace naznačovala, že UzlováAplikace je konečná entita, která komunikuje s middleware komponenty; zjištěním, že naše implementace by mohla zavést další vrstvu abstrakce, jsme byli schopni aplikovat celou řadu dalších modelů optimalizace.

Na rozdíl od předchozí implementace DAnCE UzlovéAplikace, LE-DAnCE ManažerLokality funguje jako obecný aplikační proces, která striktně odděluje zájmy mezi obecnou logikou nasazení potřebnou k analýze plánu a specifickou logiku nasazení potřebnou k instalaci a řízení životního cyklu v konkrétních instancích komponent middlewaru. Toto oddělení je dosaženo použitím entit zvaných *Obsluha instalace instance*, které poskytují dobře definované rozhraní pro řízení životního cyklu instance komponenty, včetně instalace, odstranění, připojení, odpojení a aktivace. *Obsluhy instalací* jsou také použity v souvislosti s UzlovýmiAplikacemi pro správu životního cyklu procesů ManažeraLokality.

Vznik těchto instalačních manipulátorů je příkladem vzoru *Stupňů volnosti*; na základě specifikace explicitní interakce s middleware komponentou nám ponechala možnost navrhnout svou vlastní interakci. Při návrhu jsme aplikovali vzor Rozdělení zájmů.

Snížení serializovaného provedení kroků nasazení použitím ManažeraLokality. Nový ManažerLokality a instalační manipulátor v LE- DAnCE podstatně ulehčily paralelizaci DAnCE. Paralelismu, jak v ManažeruLokality, tak v UzlovéAplikaci, je dosaženo použitím entity nazývané Plánovač nasazení, která je znázorněna na Obrázku 6.4.



Obrázek 6.4: DAnCE Plánovač nasazení

Plánovač nasazení kombinuje vzor Příkaz [GHJV95] a vzor Objekt činnosti [SSRB00]. Individuální činnosti nasazení (například instalace instance, připojení instance atd.) jsou zapouzdřeny uvnitř Objektu činnosti spolu s požadovanými metadaty. Každá jednotlivá činnost nasazení je volaním metody Instalační manipulátor, takže tyto činnosti nemusí být přepsány pro každý potenciální cíl nasazení. Zpracování chyb a protokolování logiky je také plně obsaženo v rámci individuálních činností, což je další zjednodušení ManažeraLokality.

Jednotlivé činnosti (např. instalace komponenty nebo vytvořit připojení) jsou naplánovány k vykonání konfigurovatelným zásobníkem vláken. Tento zásobník může poskytnout na základě volby uživatele buď jednovláknové nebo vícevláknové chování v závislosti na požadavcích aplikace. Tento zásobník vláken může být také použit k implementaci sofistikovanějšího chování plánování, např. plánovací algoritmus na bázi priority, který dynamicky mění pořadí instalace instancí komponent na základě metadat v plánu.

ManažerLokality určuje, které činnosti jsou provedeny během každé jednotlivé fáze nasazení a vytvoří jeden Objekt činnosti pro každou instrukci. Tyto činnosti jsou pak předávány do plánovače nasazení pro jejich provedení, zatímco hlavní řídící vlákno čeká na signál o dokončení z Plánovače nasazení. Po dokončení získá ManažerLokality buď návratové hodnoty, popř. chybové kódy z realizovaných činností a dokončí fázi nasazení.

Pro zajištění paralelismu mezi instancemi ManažeraLokality na stejném uzlu je v implementaci UzlovéAplikace také použit LE-DAnCE.

Plánovač nasazení spolu s Instalačním manipulátorem pro procesy ManažeraLokality.

Použití Plánovače nasazení na této úrovni pomáhá překonat významný zdroj zpoždění nasazování na úrovni uzlu. Vytváření instancí ManažeraLokality může trvat významně dlouho ve srovnání s dobou potřebnou k nasazení instance komponent, takže paralelizací tohoto procesu můžeme dosáhnout významných úspor zpoždění v případě, že aplikační nasazení mají mnoho procesů ManažeraLokality na jeden uzel.

Celkově vzato, dynamické přeuspořádání událostí nasazení a paralelní instalace instancí ManažeraLokality je slibným přístupem ke snížení zpoždění nasazení v případě SEAMONSTER. Přiřazení vysoké priority kritickým událostem nasazení, jako je aktivace nebo změna konfigurace senzoru pozorujícího přítomné přírodní jevy, DAnCE může pomoci zajistit, aby kritické potřeby byly včas uspokojeny. Kromě toho může paralelismus umožněný tímto designem snížit zpoždění tím, že umožňuje spustit i další instance ManažeraLokality v případě, že je jeden blokován na I/O načítáním nových implementací komponent, nebo využitím novějších vícejádrových vestavěných procesorů.

6.4 Závěrečné poznámky

Tato kapitola poskytuje přehled o *Deployment And Configuration Engine* (DAnCE), který je implementací specifikace *OMG Deployment and Configuration*. Jako výzkumný nástroj byl DAnCE použit k demonstraci nových technik pro nasazení a konfiguraci (D&C) aplikací na bázi komponent v DRE systémech. Zatímco jeho výkon byl uspokojivý pro úzce zaměřené ukázky vyžadované pro publikace a ukázky, jeho výkon nebyl uspokojivý při aplikaci na rozsáhlejších produkčních DRE systémech. Řada faktorů, včetně měnícího se architektonického vlastnictví a na demo zaměřeného charakteru vývoje DAnCE, způsobily řadu špatných designových rozhodnutí, brzy se staly zakořeněnými v architektuře a designu a vážně narušovaly výkon.

Typický případ užití DAnCE, v tomto případě platforma *South East Alaska Monitoring Network for Science, Telecommunications, Education, and Research* (SEAMONSTER), byl popsán pro zdůraznění mnohých optimalizačních příležitostí v DAnCE. Motivován tímto případem užití, popisuje tento dokument, jak jsme aplikovali katalog principů optimalizace z oblasti sítí, abychom přehodnotili a přebudovali design a implementaci DAnCE a napravili nedostatky uvedené výše. Kromě toho jsme popsali další tři principy optimalizace: řešení paralelizace, synchronizace a oddělení zájmů. Tyto dodatečné vzory, ve spojení se vzory popsanými v úvodním katalogu, byly použity k vývoji LE-DAnCE, a podstatně zlepšily výkon a spolehlivost DAnCE. Souhrn původního katalogu vzorů, spolu s našimi dodatky, je uveden v tabulce 6.2. Stejně tak důkladná kvantitativní diskuse o výsledcích výkonnostních rozšíření je popsána v [OGST13].

Na základě našich zkušeností při aplikaci optimalizace na LE-DAnCE popsané v této kapitole a pozorování výsledků jsme dospěli k následujícím poučením:

- *Využití paralelizace je kritickou optimalizační příležitostí.* Jak se vícejádrové procesory staly standardní vlastností dokonce i vestavěných zařízení, je kriticky důležité, aby algoritmy a procesy byly navrženy tak, aby využívaly této schopnosti. Při optimalizaci algoritmů a procesů paralelizací buďte rozumní při uplatňování synchronizace, protože nesprávné používání zámků může způsobit to, že paralelní systémy budou pracovat sériovým způsobem, nebo v horším případě nějakým způsobem špatně.

- *Pokud je to možné, posuňte časově náročné operace mimo kritickou cestu.* Zatímco naše optimalizace plánu analytické části procesu D&C je (popsáno v kapitole 6.3) byla efektivní při snižování celkového zpoždění u rozsáhlých nasazení, další zlepšení je možné dosáhnout použitím vzoru Posunutí v čase. Stejně jako u problému analýzy XML, popsaného v části 6.3, výsledek této operace je pravděpodobně zpomalený v bodě, kdy je generován XML plán. Tento proces by mohl být podobně předběžně vypočten a poskytnut D&C infrastruktuře pro další úsporu zpoždění. Předávání těchto předem vypočtených plánů (jak pro globální dělení, tak pro místní dělení) by bylo příkladem aplikování optimalizačního vzoru *Předávání nápověd*.

- *Serializované vykonávání procesů je hlavním zdrojem problémů s výkonem v systémech DRE.* Vykonávání úkolů pro sériové zpracování při navrhování distribuovaných systémů umožňuje významné koncepční a implementační zjednodušení. Tato jednoduchost je však často spojena s významnou výkonnostní penalizací. Dodatečná složitost asynchronní interakce se ale vyplatí.

- *Nedostatek jasného architektonického a technického vedení poškozuje open source projekty.* Vývojáři často přispívají k open source projektům řešením úzkého problému a krátce poté odcházejí. Bez jasného vedení nakonec přerostou špatná architektonická a technická rozhodnutí učiněná jednotlivými přispěvateli do téměř nepoužitelného projektu.

TAO, CIAO a LE-DAnCE jsou dostupné v open-source formě na download.dre.vanderbilt.edu.

Vzor	**Vysvětlení**	**Příklad v DaNCE**
Zamezení plýtvání	Vyhnout se zjevnému plýtvání	Předem alokovat paměť, když se analyzují plány nasazení
Přesouvání v čase	Přesouvání výpočtů v čase (předběžný výpočet, líné vyhodnocování, sdílení výdajů, dávkování)	Předem konvertovat plán nasazení do binárního formátu, potenciálně předem vypočíst plán rozdělení.
Specifikace uvolnění	Specifikace uvolnění (kompromis- určitost za čas, kompromis - přesnost za čas, a posunutí výpočtu v čase)	*Potenciálně předem vypočíst plán rozdělení.*
Využití jiných komponent	Využití jiných systémových komponent (využití lokality, kompromis paměť za rychlost, využití hardware)	(n/a)
Přidání Hardware	Přidejte hardware pro zlepšení výkonu	(n/a)
Efektivní postupy	Vytvořte efektivní postupy	XML-IDL Data Binding
Vyhýbání se obecnosti	Vyhněte se zbytečným obecnostem	Optimalizovat plán analýzy
Specifikace vs. Implementace	Nepleťte si specifikaci a implementaci	Manažér lokality

Vzor	Vysvětlení	Příklad v DaNCE
Předávání nápověd	Předávání informací jako nápovědy v rozhraních	*Potenciálně použít pro výpočet plánu rozdělení předem*
Předávání informací	Předávání informací v hlavičkách protokolu	(n/a)
Očekávaný případ užití	Optimalizujte očekávaný případ	XML-IDL Data Binding
Využívání stavu	Přidejte nebo využijte stav pro získání rychlosti	Předem alokujte plány potomka v průběhu plánu analýzy
Stupně volnosti	Optimalizujte stupně volnosti	Instalační manipulátory ManažeruLokality
Využijte konečné vesmíry	Použijte speciální techniky pro konečné vesmíry	(n/a)
Efektivní datové struktury	Použijte efektivní datové struktury	Optimalizace XML-IDL data binding
Design pro paralelizaci	Optimalizujte design pro paralelizaci	Paralelní zpracování plánu potomků
Zamezení synchronizace	Zamezení synchronizace a zamykání	Nesynchronizovaný přístup k rodičovským plánům v průběhu analýzy plánu
Oddělení zájmů	Použijte striktní oddělení zájmů pro modularizaci architektury	ManažerLokality

Tabulka 6.2: Katalog optimalizačních principů a známých případů užití v LE-DAnCE

7 Infinispan

(Manik Surtani)

7 Infinispan

7.1 Úvod

Infinispan[1] je open source platforma datového gridu. Je to distribuované NoSQL úložiště typu klíč-hodnota, které ukládá data v paměti. Software architekti obvykle používají datové grudy jako Infinispan buď jako výkonnost zvyšující distribuované vyrovnávací paměti místo drahých a pomalých úložišť dat, jako jsou relační databáze, nebo jako distribuované NoSQL datové úložiště k nahrazení relační databáze. V obou případech je hlavním důvodem pro zvážení datového gridu v softwarové architektuře výkon. Potřeba rychlého přístupu k datům s nízkou latencí je stále častější.

Jako takový je výkon Infinispanu jediným důvodem jeho existence. Na druhou stranu je základ kódu Infinispanu extrémně výkonnostně citlivý.

7.2 Přehled

Před nahlédnutím do hlubin Infinispanu zauvažujme, jak se Infinispan obvykle používá. Infinispan spadá do kategorie softwaru nazývaného middleware. Podle Wikipedie, middleware „lze popsat jako softwarová lepidlo" - komponenty, které sedí na serverech mezi aplikacemi, jako jsou webové stránky a operační systém nebo databáze. Middleware se často používá ke zvýšení produktivity a efektivity vývojáře aplikace a urychlení výroby aplikací, které jsou také více udržovatelné a testovatelné. To vše je dosaženo modularizací a opětovným použitím komponent. Infinispan je specificky často umístěn mezi jakýmkoliv aplikačním zpracováním nebo byznys logikou a vrstvou datového úložiště. Ukládání dat (a načítání) jsou často největší úzká místa a umístění paměťového datového gridu před databází často věci výrazně urychlí. Navíc je ukládání dat také často místem soupeření a možného selhání. Opět platí, že využitím Infinispanu před (nebo dokonce místo) tradičnějšími úložišti dat můžou aplikace dosáhnout větší pružnosti a škálovatelnosti.

Kdo Infinispan používá?

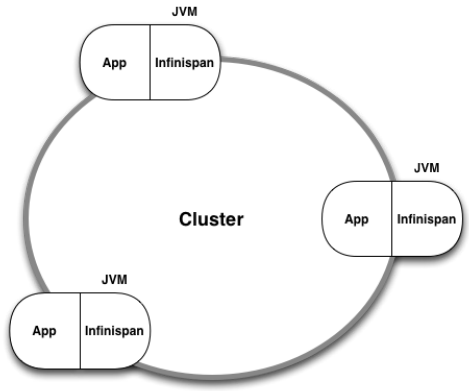
Infinispan byl použit v několika průmyslových odvětvích, od telekomunikací po finanční služby, od špičkových e-commerce až po výrobní systémy, hry a mobilní platformy. Datové gridy obecně byly vždy populární v odvětvích finanční služeb, vzhledem ke svým přísným požadavkům na extrémně rychlý přístup k velkému objemu dat takovým způsobem, který je chrání před individuálním selháním stroje. Tyto požadavky se od té doby rozšířily do dalších odvětví, což přispívá k oblíbenosti Infinispanu v tak širokém spektru aplikací.

1: http://www.infinispan.org

Jako knihovna nebo jako server

Infinispan je implementovaný v Java (a část v Scala) a může být použit dvěma různými způsoby.

Za prvé, může být tato platforma použita jako knihovna vložená do Java aplikace zahrnutím Infinispan JAR souborů, odkazováním a vytvořením instancí Infinispan komponent programově. Tímto způsobem běží Infinispan komponenty ve stejném JVM jako aplikace a část paměti halda aplikace je přidělena pro uzel datové sítě.



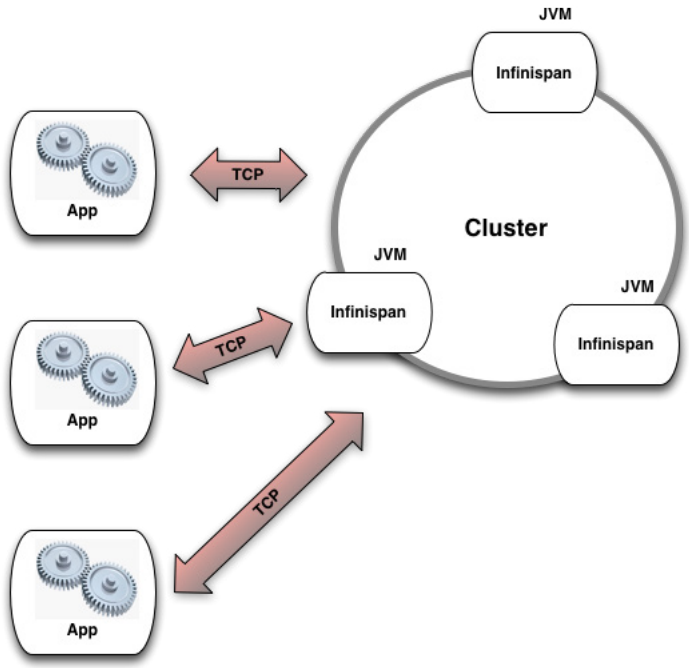
Obrázek 7.1: Infinispan jako knihovna

Za druhé může být použita jako vzdálený datový grid spuštěním instancí Infinispan a umožněním jejich zformování do klastru. Klient se pak může připojit k tomuto klastru přes některou z mnoha dostupných klientských knihoven. Tímto způsobem běží každý Infinispan uzel ve svém vlastním izolovaném JVM a má k dispozici celou JVM halda paměť.

Architektura Peer-to-Peer

Infinispan instance se v obou případech navzájem detekují prostřednictvím sítě, tvoří klastr a začnou sdílení dat, aby poskytly aplikacím datovou strukturu v paměti transparentně pokrývající všechny servery v klastru. Přidávání uzlů do klastru umožňuje aplikacím teoreticky adresovat neomezené množství úložného prostoru v paměti, což zvyšuje celkovou kapacitu.

Infinispan je technologie peer-to-peer, kde je každá instance v klastru rovná každé další instanci v klastru. To znamená, že neexistuje jediný bod selhání a jediné úzké místo. A co je nejdůležitější, poskytuje aplikace s elastickou datovou strukturou, které lze škálovat horizontálně přidáním dalších instancí. A můžou být také škálovány zpátky, a to vypnutím některých instancí, přičemž aplikace může pokračovat v činnosti bez ztráty celkové funkčnosti.



Obrázek 7.2: Infinispan jako vzdálená datová mřížka

7.3 Referenční srovnání Infinispanu

Největším problémem při srovnávání distribuované datová struktury jako je Infinispan, je sada nástrojů. Existuje žalostně málo nástrojů, které vám umožní měřit výkon ukládání a načítání dat, zatímco škálujete tam a zpátky. Neexistuje také nic, co by umožnilo srovnávací analýzu, aby měřilo a porovnávalo výkonnost různé konfigurace, velikostí klastru, atd. Radar Gun byl vytvořen, aby nám s tím pomohl.

Radar Gun je podrobněji popsán v části 7.4. Další nástroje, které jsou zde zmíněné – Yahoo Cloud Serving Benchmark, Grinder a Apache JMeter - nejsou popsány tak do hloubky, i když jsou velmi důležité pro srovnání Infinispan. O těchto nástrojích již existuje mnoho online literatury.

Radar Gun

Radar Gun[2] je open source srovnávací framework, který byl navržen tak, aby prováděl referenční (stejně jako konkurenční) srovnání, a to měřením škálovatelnosti a generováním sestav z nasbíraných údajů. Radar Gun je specificky zaměřen na distribuované datové struktury, jako je Infinispan, a byl značně využíván v průběhu vývoje Infinispanu pro identifikaci a opravu slabých míst. Viz část 7.4 pro více informací o frameworku Radar Gun.

Yahoo Cloud Serving Benchmark

Yahoo Cloud Serving Benchmark[3] (YCSB) je open source nástroj vytvořený pro testování zpoždění při komunikaci s dálkovým úložištěm dat při čtení nebo zápisu dat různých velikostí. YCSB zachází se všemi datovými úložišťmi jako s jediným vzdáleným koncovým bodem, takže se nepokouší měřit škálovatelnost při přidávání nebo odebírání uzlů klastru. Protože YCSB nemá žádnou představu o distribuované datové struktuře, je užitečný pouze pro srovnávání Infinispanu v režimu klient/server.

Grinder a Apache JMeter

Grinder[4] a Apache JMeter[5] jsou dva jednoduché open source generátory zatížení, které mohou být použity k testování libovolných serverů naslouchání na socketu. Jsou vysoce skriptovatelné a stejně jako YCSB užitečné při srovnávání Infinispanu při použití v režimu klient/server.

7.4 Radar Gun

Začátky

Vytvořen hlavním vývojovým týmem Infinispan, Radar Gun začínal jako projekt na Sourceforge, nazývaný Cache Benchmarking Framework[6] a původně byl navržen tak, aby porovnával vložené Java vyrovnávací paměti, běžící v různých režimech a v různých konfiguracích. Byl navržen tak, aby byl srovnávací, takže automaticky spustil stejné srovnání vůči různým knihovnám mezipaměti, či různým verzím stejné knihovny, nebo pro testování výkonové regrese.

Od svého vzniku již získal nový název (Radar Gun), nový domov na GitHub[7] a řadu nových funkcí.

2: https://github.com/radargun/radargun/wiki

3: https://github.com/brianfrankcooper/YCSB/wiki

4: http://grinder.sourceforge.net/

5: http://jmeter.apache.org/

6: http://sourceforge.net/projects/cachebenchfwk/

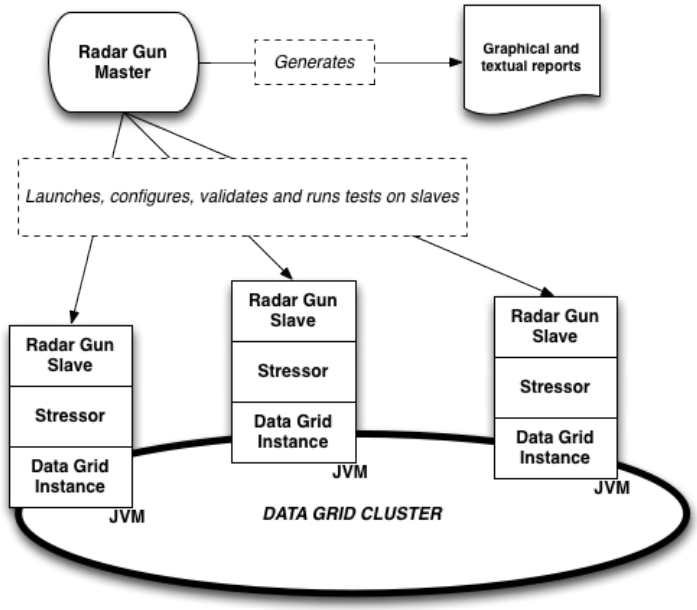
7: http://github.com/radargun

Distribuované vlastnosti

Radar Gun byl brzy rozšířen i na distribuované datové struktury. Stále zaměřen na vložené knihovny, Radar Gun je schopen spustit více instancí frameworku na různých serverech, což následně spustí instance distribuované knihovny mezipaměti. Srovnání pak běží paralelně v každém uzlu klastru. Výsledky se kompletují a řídícím modulem Radar Gunu jsou vygenerovány reporty. Je nepraktické a neproveditelné spouštět ručně a znovu srovnání na klastrech různých velikostí, od dvou uzlů až po stovky nebo dokonce tisíce uzlů, a proto je schopnost automaticky vytvářet a vypínat uzly zásadní pro testování škálovatelnosti.

Rychlý a chybný je k ničemu!

Radar Gun pak získal schopnost provádět kontrolu stavu před a po spuštění každé fáze referenčního srovnání, aby bylo zajištěno, že je klastr stále v správném/bezchybném stavu. To umožnilo včasné odhalení chybných výsledků a opětovné spuštění srovnání, aniž by se čekalo na ruční zásah na konci běhu, který může trvat mnoho hodin.



Obrázek 7.3: Radar Gun

Profilování

Radar Gun je také schopen spustit a připojit profilovač instancí do každého uzlu datového gridu a zachytit snímek profilovače pro více vhledu do dění v každém uzlu při zatížení.

Výkonnost paměti

Radar Gun má také schopnost měřit stav spotřeby paměti každého uzlu pro měření výkonu paměti. V úložišti dat v paměti není výkon jen o tom, jak rychle budete číst nebo zapisovat data, ale také, jak dobře struktura funguje s ohledem na spotřebu paměti. To je obzvláště důležité v systémech založených na Java, kde garbage collection může nepříznivě ovlivnit reakční schopnost systému. Garbage collection je diskutováno podrobněji dále.

Metriky

Radar Gun měří výkonnost z hlediska transakcí za sekundu. To je zachyceno pro každý uzel a agregováno v řídícím modulu. Jak čtení, tak zápis se měří a zobrazují samostatně, i když jsou prováděny souběžně (pro zajištění realistického testu, kde jsou tyto operace prokládány). Radar Gun také zachycuje prostředky, mediány, směrodatné odchylky, maximální a minimální hodnoty pro transakce čtení a zápisu, a i ty jsou zaznamenány, i když nemusí být zobrazeny. Výkon paměti je také zachycen, a to formou stopy pro danou iteraci.

Rozšiřitelnost

Radar Gun je rozšiřitelný framework. To vám umožní připojit své vlastní vzory přístupu k datům, datovým typům a velikostem. Dále také umožňuje přidávat adaptéry pro jakékoliv datové struktury, knihovny mezipaměti nebo NoSQL databázi, které byste chtěli vyzkoušet.

Koncoví uživatelé jsou často také povzbuzeni k používání Radar Gunu při pokusech o porovnání výkonu různých konfigurací datového gridu.

7.5 Hlavní podezřelí

Existuje několik subsystémů v Infinispan, které jsou hlavními podezřelými na výkonově úzká místa, a jako tací jsou kandidáti na pečlivé prozkoumání a potenciální optimalizaci. Podívejme se na každý z nich po pořadí.

Síť

Síťová komunikace je nejdražší části Infinispanu, ať již používána pro komunikaci mezi vrstevníky (peer) nebo mezi klienty a samotnou mřížkou.

Síť peer

Infinispan se využívá pro komunikaci mezi uzly JGroups[8], což je open-source knihovna pro peer--to-peer skupinovou komunikaci. JGroups může využívat buď TCP nebo UDP síťové protokoly, včetně UDP multicast, a poskytuje vysokoúrovňové funkce, jako jsou záruky o doručení zprávy, opakovaný přenos a řazení zpráv, i přes nespolehlivé protokoly jako UDP.

Je kriticky důležité správně vyladit vrstvu JGroups, aby odpovídala vlastnostem vaší sítě a aplikace, například velikosti doby platnosti (TTL), velikostí paměťových zásobníků, velikosti zásobníku vláken. Je také důležité počítat se způsobem, jímž JGroups sestavuje pakety - kombinování několika malých zpráv do jednotlivých síťových paketů – nebo fragmentaci, opačná činnost, kde jsou velké zprávy rozděleny do několika menších síťových paketů.

Síťová architektura na vašem operačním systému a vaše síťové zařízení (přepínače a směrovače), by měly být rovněž nakonfigurovány tak, aby odpovídaly této konfiguraci. Parametry protokolů IP, UDP i TCP hrají roli pro zajišťování optimálního výkonu nejdražší komponenty ve vašem datovém gridu.

Nástroje, jako je netstat a wireshark, můžou pomoci analyzovat pakety a Radar Gun může pomoci řídit zátěž přes grid. Radar Gun může být také použit k profilaci vrstvy JGroups Infinispanu a pomoci najít úzká místa.

Serverové sokety

Infinispan využívá oblíbený Framework Netty[9] pro vytváření a řízení serverových soketů. Netty je wrapper asynchronního frameworku Java NIO, který využívá I/O asynchronní síťové funkce poskytované operačním systémem. To umožňuje efektivní využití zdrojů na úkor přepínání kontextu. Obecně platí, že to při zatížení funguje velmi dobře.

Netty umožňuje několik úrovní ladění pro zajištění optimálního výkonu. Patří mezi ně velikosti mezipaměti, počet pracovních vláken, a také by se měly shodovat s parametry pro odesílací a přijímací mezipaměti operačního systému.

Serializace dat

Před odesláním dat po síti je třeba serializovat objekty aplikace do bajtů tak, aby mohly být odeslány přes síť do gridu, a pak znovu do uzlů mřížky. Na cílovém uzlu je bajty třeba deserializovat zpět do objektů aplikace. Při zpracování je ve většině běžných konfigurací asi 20 % času spotřebováno na serializaci a deserializaci.

8: http://www.jgroups.org

9: http://www.netty.io

Výchozí Java serializace (a deserializace) je notoricky pomalá, a to jak v CPU cyklech, tak v produkovaných bajtech, které jsou často zbytečně velké, což znamená větší objem dat k přenosu po síti.

Infinispan využívá vlastní serializační schéma, kde nejsou do proudu zapsány úplné definice tříd. Místo toho se používají pro známé typy magická čísla, kde je každý známý typ reprezentován jedním bajtem. Tím se výrazně zlepšuje nejenom rychlost serializace a deserializace, ale také se produkuje mnohem kompaktnější bajtový proud pro přenos po síti. Externalizér je registrován u každého známého datového typu registrovaného magickým číslem. Tento externalizér obsahuje logiku pro převod objektu do bajtů a naopak.

Tato technika funguje dobře u známých typů, jako jsou interní Infinispan objekty, které jsou vyměňovány mezi uzly. Vnitřní objekty, jako jsou příkazy, obálky, apod. mají externalizéry a odpovídající jedinečná magická čísla. Ale co objekty aplikace? Pokud Infinispan narazí na neznámý typ objektu, ve výchozím nastavení se vrátí k Java serializaci pro daný objekt. To umožňuje Infinispanu okamžitě pracovat, i když, pokud se jedná o neznámé typy objektů aplikace, méně efektivním způsobem.

Chcete-li tento postup obejít, Infinispan umožňuje vývojářům aplikací zaregistrovat také externalizéry pro datové typy aplikace. Vývojář aplikace může také zapisovat a zaregistrovat externalizér implementací pro každý typ objektu aplikace, což umožňuje výkonnou, rychlou a efektivní serializaci i aplikačních objektů.

Kód externalizéru byl vydán jako samostatná, opakovaně použitelná knihovna, tzv. JBoss Marshalling[10]. Je dodávána s Infinispanem, zahrnuta v Infinispan distribucích a také používána v různých dalších open source projektech ke zlepšení výkonnosti serializace.

Zapisování na disk

Kromě udržování dat v paměti je může Infinispan také volitelně zapsat na disk. To může být buď z toho důvodu, aby např. data zůstala na uzlu po jeho restartu, přičemž vše v paměti existuje také na disku. Nebo to může být konfigurováno jako odložení dat z paměti na disk, když Infinispan vyčerpá fyzickou paměť, přičemž v tomto případě funguje podobným způsobem, jako stránkování virtuální paměti operačním systémem na disk. V druhém případě jsou data zapsána na disk pouze v případě potřeby odložení dat z paměti pro uvolnění místa.

Persistence z důvodu uchování dat na uzlu může být buď online, kdy je blokováno aplikační vlákno, než jsou data bezpečně zapsána na disk, nebo offline, kdy jsou data zapisovaná na disk pravidelně a asynchronně. V druhém případě není vlákno aplikace blokováno čekáním na proces perzistence, avšak výměnou za nejistotu, zda vůbec data na disku úspěšně přetrvala.

10: http://www.jboss.org/jbossmarshalling

Infinispan podporuje několik připojitelných vyrovnávacích pamětí - adaptérů, které mohou být použity pro perzistenci dat na disku nebo jinou formou sekundárního úložiště. Současná výchozí implementace je zjednodušená implementace hash bucketu a spojeného seznamu, kde je každý hash bucket reprezentován souborem v souborovém systému. I když je tato implementace snadno použitelná a konfigurovatelná, není ale nejvýkonnější.

Dvě vysoce výkonné implementace, založené na nativní mezipaměti v souborovém systému, jsou v současné době v plánu. Obě budou napsány v C, s možností systémových volání a využívání přímého I/O, pokud jsou dostupné (například v systémech Unix), aby se vyhnuly vyrovnávacím mezipamětím jádra.

Jedna z implementací bude optimalizována tak, aby byla použita jako stránkovací systém, a proto je potřeba, aby měla náhodný přístup, případně b-stromovou strukturu.

Druhá bude optimalizována jako trvalé úložiště a zrcadlo, které se uloží do paměti. Jako taková to bude konstrukce pouze při přidávání dat za existující data, určená pro rychlé psaní, ale ne nezbytně pro rychlé čtení/hledání.

Synchronizace, zamykání a souběh

Stejně jako u většiny middleware podnikové třídy, je i Infinispan silně zaměřen na moderní, vícejádrové systémy. Ve vícejádrových a SMP systémech máme k dispozici pro využití paralelismu velké množství hardwarových vláken, stejně jako neblokujících, asynchronních I/O při komunikaci se sítí a diskem. Jádrové datové struktury Infinispanu využívají softwarové techniky transakční paměti pro souběžný přístup ke sdíleným datům. Tím se minimalizuje nutnost explicitních zámků, vzájemných vyloučení a jiných forem synchronizace, a preferují se techniky, jako operace typu porovnej-a-nastav ve smyčce, k dosažení správnosti při aktualizaci sdílených datových struktur. Tyto techniky prokazatelně zlepšují využití CPU v multijádrových a SMP systémech, a i přes značnou složitost kódu přispívají k celkovému výkonu při zatížení.

Kromě výhod používání softwarových transakčních paměťových přístupů to do budoucna také umožňuje Infinispanu využít sílu instrukcí pro podporu synchronizace v hardwarem implementované transakční paměti, pokud se taková CPU stanou samozřejmostí, s minimální změnou designu Infinispanu.

Několik datových struktur používaných v Infinispanu je jako vystřižených z akademických výzkumných prací. Ve skutečnosti byl neblokující, obousměrný spojový seznam nepoužívající zámky[11] použité v Infinispanu první Java implementací takovéto struktury. Jiné příklady zahrnují nové návrhy pro amortizaci[12] použití zámku a adaptivní politiky nahrazení.[13]

11: http://www.md.chalmers.se/~tsigas/papers/Lock-Free-Deques-Doubly-Lists-JPDC.pdf

12: http://dl.acm.org/citation.cfm?id=1546683.1547428

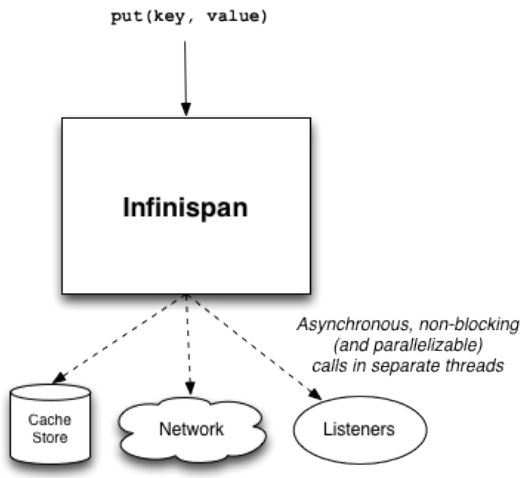
13: http://dl.acm.org/citation.cfm?id=511334.51134

Přepínání vláken a kontextu

Různé subsystémy Infinispanu využívají asynchronní operace, které probíhají v samostatných vláknech. Například JGroups přiděluje vlákna pro monitorování síťového soketu, která pak dekódují zprávy a předávají je vláknu pro doručení zprávy. To se může zase pokoušet uložit data ve vyrovnávací paměti na disku, která může být též asynchronní a používat samostatné vlákno. Posluchači mohou být obeznámeni i o změně a může to být konfigurovatelné a asynchronní.

Při práci se zásobníkem vláken na zpracování takového asynchronního úkolu existuje vždy režie na přepínání kontextu. Za povšimnutí stojí i to, že vlákna nejsou právě levné zdroje. Alokování odpovídajícího počtu a konfigurace vláken jsou důležité pro každé použití instalace využívající jakoukoliv z asynchronních funkcí Infinispanu.

Specifickými oblastmi, kam se podívat, jsou zásobníky asynchronních transportních vláken (v případě použití asynchronní komunikace) a zajištění, že tento zásobník vláken je alespoň tak velký, jako očekávaný počet souběžných aktualizací, které každý uzel očekává zpracovat. Podobně by tak při ladění JGroups, OOB[14] a příchozí mezipaměti vláken měly být zásobníky vláken alespoň tak velké, jako je očekávaný počet souběžných aktualizací.



Obrázek 7.4: Vláknování v Infinispanu

14: http://www.jgroups.org/manual/html/user-advanced.html#d0e3284

Garbage Collection

Obecně osvědčené postupy s ohledem na práci s JVM garbage collectors jsou důležitým aspektem pro jakýkoli software založený na jazyku Java, a Infinispan není výjimkou. Ještě důležitější je to pak pro datový grid, protože kontejnerové objekty mohou přežívat po dlouhou dobu, kdy je také vytvořeno mnoho přechodných objektů souvisejících se specifickou operací nebo transakcí. Navíc pozastavení garbage collectoru může mít nepříznivý vliv na distribuované datové struktury, protože může způsobit, že uzel přestane reagovat a bude označen jako chybný.

Toto bylo vzato v úvahu při navrhování a vyvíjení Infinispanu, ale zároveň je třeba toho hodně zvážit při konfiguraci JVM ke spuštění Infinispanu. Každý JVM je jiný. Každopádně byly provedeny různé analýzy[15] pro optimální nastavení určitých JVM při spuštěném Infinispanu. Optimální konfigurací by například mohlo být použití OpenJDK[16] nebo Oracle HotSpot JVM[17], za využití Concurrent Mark a Sweep collectoru[18] souběžně s velkými stránkami[19] pro JVM o velikosti 12 GB haldy na každou.

Garbage collectory bez pozastavení - jako C4[20], používané v Azul Zing JVM[21], pak stojí za úvahu v případě, kdy se pozastavení garbage collection stává znatelným problémem.

7.6 Závěr

Výkonově zaměřený middleware jako Infinispan musí být architektonicky navržen a vyvinut s ohledem na výkon v každém kroku. Od použití nejlepších neblokujících a nezamykajících algoritmů, přes pochopení vlastností garbage collectoru, vyvíjení s ohledem na režii pro přepínání JVM kontextu, až po schopnost udělat v případě potřeby krok mimo JVM (například psaní nativních perzistentních komponent). Toto jsou všechno důležité součásti myšlení, potřebné pro vývoj v Infinispanu. Správné nástroje pro srovnávání a profilování, stejně jako srovnávací kritéria v kontinuálním integračním stylu, pomáhají zajistit, aby nebyl přidáváním funkcí obětován výkon.

15: http://howtojboss.com/2013/01/08/data-grid-performance-tuning/

16: http://openjdk.java.net/

17: http://www.oracle.com/technetwork/java/javase/downloads/index.html

18: http://www.oracle.com/technetwork/java/javase/gc-tuning-6-140523.html#cms

19: http://www.oracle.com/technetwork/java/javase/tech/largememory-jsp-137182.html

20: http://www.azulsystems.com/technology/c4-garbage-collector

21: http://www.azulsystems.com/products/zing/virtual-machine

8 Talos

(Clint Talbert a Joel Maher)

8 Talos

Jedním z našich úplně prvních automatizačních systémů v Mozille byl framework pro testování výkonu a nazývali jsme ho Talos. Talos byl věrně udržován bez podstatných modifikací od svého vzniku v roce 2007, i když mnoho z původních předpokladů a designových rozhodnutí bylo ztraceno se změnou vlastnictví nástroje.

V létě roku 2011 jsme se konečně začali tázavě dívat na šum a kolísání čísel Talosu, a začali jsme přemýšlet, jak bychom mohli udělat nějakou malou úpravu do systému a započali tak jeho zlepšení. Netušili jsme, že jsme se chystali otevřít Pandořinu skříňku.

V této kapitole budeme detailizovat, co jsme našli, když jsme tento software loupali vrstvu po vrstvě, jaké problémy jsme odkryli, a jaké kroky jsme udělali na jejich řešení v naději, že byste se mohli poučit jak z našich chyb, tak z našich úspěchů.

8.1 Přehled

Pojďme rozkrýt různé části Talosu. Ve svém jádru je Talos jednoduchý testovací systém, který vytváří nový Firefox profil, inicializuje profil, kalibruje prohlížeč, vykonává specifikovaný test a nakonec reportuje souhrn výsledků testu. Testy žijí uvnitř Talos repositáře a jsou dvojího typu: jedna stránka, která reportuje jediné číslo (např. dobu spuštění přes onload manipulátor webové stránky), nebo soubor stránek, které se cyklicky střídají pro měření časů načítání stránky. Rozšíření pro Firefox se používá k přecházení stránky a shromažďování informací, jako jsou doby načítání paměti a stránky, vynucení garbage collection, a testování různých režimů prohlížeče. Původním cílem bylo vytvořit co nejvíce generický testovací systém, aby vykonával všechny druhy zkoušek a měřil nějakou sadu výkonových atributů definovaných samotným testem.

Pro reportování dat může Talos odesílat JSON do Graph Server, což je interní grafová webová aplikace, která přijímá data z TALOSu, splňující specifický, předem definovaný formát pro každý test, hodnoty, platformu a konfiguraci. Graph Server také slouží jako rozhraní pro zkoumání trendů a výkonnostní regrese. Lokální instance standardního Apache webového serveru poskytuje stránky během zkušebního provozu.

Poslední komponentou Talosu je nástroj pro reportování regresí. Po každém uložení změn kódu do Firefox úložiště proběhne několik testů v Talosu, tyto testy nahrají svá data do Graph Serveru a další skript zpracuje data z Graph Serveru a zjistí, zda došlo k regresi. Pokud je nalezena regrese (tj. analýza skriptu indikuje, že vrácený kód způsobil v tomto testu významné snížení výkonu), skript zašle e-mailovou zprávu na e-mailový seznam, jakož i jednotlivci, který uložil změny dotčeného kódu.

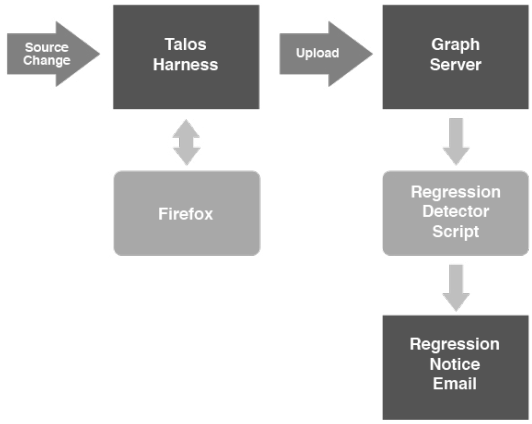
Zatímco tato architektura, jak je shrnuto na Obrázku 8.1, se zdá být poměrně jednoduchá, v průběhu let se každý kus Talosu proměnil tím, jak Mozilla přidávala nové platformy, produkty a testy. S trochou nadhledu nutně potřeboval celý systém Talos jako end-to-end řešení seriózní práci:

- Šum - skript sledující příchozí data označoval tolik vrcholků v testovacím šumu, jako u skutečných regresí, a nedalo se mu věřit.

- Pro zjištění regresí skript porovnával každé vrácení kódu do Firefoxu s hodnotami z předchozích tří vrácení a z třech následujících. To znamenalo, že výsledky z Talosu pro vaše vrácení kódu nemusely být k dispozici po dobu několika hodin.

- Graph Server měl tvrdý požadavek, aby všechna příchozí data byla vázaná na předem definovanou platformu, větev, typ testu a konfigurace. To znamenalo, že přidávání nových testů bylo těžké, protože zahrnovalo spuštění SQL příkazů oproti databázi pro každý nový test.

- Talos sám o sobě bylo těžké spustit, protože svůj požadavek, aby byl generický, bral trochu příliš vážně - měl „konfigurační“ krok k vytvoření konfiguračního skriptu, který ve svém dalším kroku použil ke spuštění testu.



Obrázek 8.1: Talos architektura

Zatímco jsme v létě roku 2011 dřeli na Talosu, abychom přidali podporu pro nové platformy a testy, narazili jsme na výsledky magisterské práce Jana Larrese, ve které zkoumal velké množství šumu objevujícího se v Talos testech. Analyzoval různé faktory, včetně hardwaru, operačního

systému, souborového systému, ovladačů a Firefoxu, které by mohly ovlivnit výsledky Talos testů. V návaznosti na tuto práci věnoval Stephen Lewchuk svou stáž snaze o statistické snížení šumu, který jsme viděli v těchto testech.

Na základě jejich práce a zájmu jsme začali tvořit plán na odstranění nebo snížení šumu při testech Talosu. Dali jsme dohromady hackery testovacího systému, aby na něm samotném pracovali, webové vývojáře na aktualizaci Graph Serveru a statistiky, aby určili optimální způsob, jak provést každý test tak, aby vyprodukoval předvídatelné výsledky s minimálním šumem.

8.2 Pochopení toho, co měříte

Při provádění testování výkonu je důležité mít užitečné testy, které poskytují hodnotu vývojářům produktu a pomáhají zákazníkům vidět, jak bude tento produkt fungovat za určitých podmínek. Je také důležité mít opakovatelné prostředí, takže si můžete reprodukovat výsledky podle potřeby. Ale to, co je nejdůležitější, je pochopení toho, jaké testy máte a v těchto testech měříte.

Po několika týdnech našeho projektu jsme se všichni naučili více o celém systému a začali jsme experimentovat s různými parametry pro odlišné spuštění testů. Jedna opakující se otázka byla: „Co ta čísla znamenají?“ Na to nebylo snadné odpovědět. Mnohé z těchto testů měly už léta malou nebo žádnou dokumentaci.

Ještě horší ale byla nemožnost produkovat stejné výsledky lokálně reportované z automatického testovacího běhu. Vyšlo najevo, že Talos sám provedl výpočty (vynechal nejvyšší hodnotu na stránku a pak reportoval průměrné hodnoty pro zbytek cyklů) a Graph Server udělal výpočet také (vynechal nejvyšší hodnotu stránky, pak průměroval stránky dohromady). Konečným výsledkem bylo, že neexistovala žádná historická data, která by měla velkou hodnotu, a ani nikdo nechápal testy, které jsme spouštěli.

Měli jsme nějaké znalosti o jednom konkrétním testu. Věděli jsme, že tento test udělal v čase snímky 100 nejčastějších webových stránek a načetl každou stránku jednu po druhé, a opakoval to desetkrát. Talos načetl stránku, počkal na událost mozAfterPaint (standardní událost spuštěná, když Firefox vykreslí webovou stránku) a poté zaznamenal čas od načítání stránky po přijetí této události. Při pohledu na 1 000 datových bodů vytvořených z jediného testu neexistoval zřejmý vzor. Představte si, že zredukujeme těchto 10 000 bodů do jediného čísla a to budeme sledovat v průběhu času. Co když jsme udělali parsování CSS rychleji, ale obrázek jsme načetli pomaleji? Jak bychom to zjistili? Bylo by možné vidět stránku 17 zpomalit, pokud všech ostatních 99 stránek zůstalo stejných? Pro ukázku toho, jak byly hodnoty vypočteny v původní verzi Talosu, zvažte následující čísla.

Pro následující stránku načte hodnoty:

- Stránka 1: 570, 572, 600, 503, 560
- Stránka 2: 780, 650, 620, 700, 750
- Stránka 3: 1220, 980, 1000, 1100, 1200

Nejdříve sám Talos vynechal nejvyšší hodnotu a kalkuloval medián:

- Stránka 1: 565.5
- Stránka 2: 675
- Stránka 3: 1050

Tyto hodnoty byly předloženy Graph Serveru. Graph Server vynechal nejvyšší hodnotu, vypočetl průměr pomocí těchto hodnot na jednu stránku a reportoval tuto jednu hodnotu:

$$\frac{565.5 + 675}{2} = 620.25$$

Z této konečné hodnoty bude vytvořen v průběhu času graf, a jak můžete vidět, generuje přibližnou hodnotu, která není víc k ničemu, než k hrubému vyhodnocení výkonnosti. Kromě toho, pokud byla regrese detekována pomocí hodnoty jako tato, bylo mimořádně těžké pracovat zpětně a zjistit, které stránky způsobily regresi, aby vývojář mohl být nasměrován k opravě specifického problému.

Byli jsme odhodláni dokázat, že můžeme snížit šum v datech z tohoto testu 100 stránek. Vzhledem k tomu, že test měřil čas k načtení stránky, nejprve jsme potřebovali izolovat test od jiných vlivů v systému, jako je ukládání do vyrovnávací paměti. Změnili jsme test raději na opakované načítání stejné stránky než cyklení mezi stránkami, takže čas pro načtení byl měřen na stránku, která byla většinou načítaná ve vyrovnávací paměti. I když tento přístup není směrodatný tomu, jak koncoví uživatelé skutečně procházejí web, snížilo to šum v zaznamenaných údajích. Bohužel ale vzorek s pouhými 10 datovými body pro danou stránku nebyl použitelný.

Změnou velikosti našeho vzorku a měřením standardní odchylky hodnot načtení stránky z mnoha testovacích běhů jsme zjistili, že šum byl snížen v případě, že se stránka načte alespoň 20krát. Po dlouhém experimentování tato metoda našla optimum při 25 načteních a ignorování prvních 5. Jinými slovy, přezkoumáním standardní odchylky hodnot vícenásobného načítání stránek jsme zjistili, že 95 % našich výsledků s šumem nastalo v rámci prvních pěti načtení. I přesto, že jsme

nepoužili těchto prvních pět datových bodů, ukládali jsme je, takže pokud do budoucna budeme chtít, můžeme změnit naše statistické výpočty.

Všechno toto experimentování nás přivedlo k některým novým požadavkům pro sběr dat, která Talos vykonával:

- Všechny shromážděné údaje je třeba uložit do databáze, nejen průměry průměrů.
- Test musí získat nejméně 20 použitelných datových bodů na jeden test (v tomto případě na jednu stránku).
- Aby nedošlo k maskování regrese na jedné stránce zlepšením na jiné stránce, musí být každá stránka počítaná samostatně. Žádné další zprůměrování hodnot mezi stránkami.
- Každý test, který je spuštěn, musí mít vývojáře, který test vlastní a dokumentaci o tom, co se sbírá a proč.
- Na konci testu musíme být v čase reportování výsledků schopni detekovat regrese pro danou stránku.

Uplatňování těchto nových požadavků na celý systém Talos bylo správnou věcí, ale v ekosystému, který vyrostl kolem Talosu, bylo přepnutí do tohoto nového modelu příliš velkým úkolem. Museli jsme se rozhodnout, zda systém refaktorujeme nebo přepíšeme.

8.3 Přepsání vs. refaktorování

Vzhledem k našemu výzkumu změn Talosu jsme věděli, že budeme dělat drastické změny. Nicméně všechny historické změny Talosu v Mozille vždycky trpěly strachem z „prolamování čísel". Mnohé kusy Talosu byly postaveny v průběhu let dobře mínícími přispěvateli, jejichž příspěvky dávali smysl v té době, ale bez dokumentace nebo dohledu nad směrováním nástrojového řetězce se Talos stal jakousi směsicí kódu, který nebylo snadné testovat, upravovat nebo mu rozumět.

Vzhledem k našemu strachu z nedokumentované temné hmoty v kódu, v kombinaci s problémem, že budeme muset ověřit nová měření proti starým měřením, jsme místo toho zahájili refaktorové úpravy Talosu a Graph Serveru. Nicméně rychle bylo zřejmé, že bez masivních změn architektury databázového schématu nebude systém Graph Server nikdy schopen zpracovat celý soubor surových dat z výkonových testů. Navíc jsme neměli čistý způsob, jak aplikovat naše nově vyzkoumané statistické metody do backendu Graph Serveru. Proto jsme se rozhodli přepsat Graph Server od nuly, vytvořením projektu s názvem Datazilla. Toto rozhodnutí nebylo lehké, protože jiné open source projekty používaly kód Graph Serveru pro vlastní automatizaci výkonu.

Na straně systému Talosu jsme také udělali prototyp od nuly. Dokonce jsme měli funkční prototyp, který provedl jednoduchý test a byl asi o 2000 řádků kódu kratší.

I když jsme přepsali Graph Server od začátku, měli jsme obavy pokračovat s naším novým Talos zkušebním prototypem. Naše obava byla, že bychom mohli ztratit možnost spouštět čísla „postaru", abychom mohli porovnat nový přístup se starým. Tak jsme opustili náš prototyp a upravili samotný Talos postupnými transformacemi do generátoru dat a přitom ponechali stávající části, které prováděly načítání průměrů do starého systému Graph Serveru. Jednalo se o mimořádně špatné rozhodnutí. Měli jsme vybudovat nový testovací systém a pak porovnávat nový systém se starým.

Snaha podporovat původní tok dat a nové metody pro měření dat každé stránky se ukázala být obtížná. Na druhou stranu nás to donutilo restrukturalizovat většinu interního kódu frameworku a zefektivnit poměrně dost věcí. Ale museli jsme tohle všechno dělat po částech na běžícím kusu automatizace, což nám způsobilo několik problémů v našich pokračujících integracích.

Bylo by mnohem lepší vyvíjet úplně od nuly, vynecháním starého kódu, jak framework Talosu, tak reportovací systém Datazilly. Zvláště co se týče fázování, bylo mnohem jednodušší fázovat nový systém bez pokusu připojit se k běžícímu automatu pro generování vývojových dat pro připravovaný Datazilla systém. Mysleli jsme si, že to bylo nevyhnutné proto, abychom mohli generovat testovací data v reálném sestavení a za skutečného zatížení, abychom zajistili, že náš design je správně škálován. Tato stavební data nestála za složitost modifikace produkčního systému. Kdybychom to věděli v době, kdy jsme zahájili jeden rok dlouhý projekt místo plánovaného šestiměsíčního projektu, přepsali bychom Talos a výsledný framework úplně od nuly.

8.4 Vytváření výkonnostní kultury

Být open source projektem znamená přijmout myšlenky a kritiku od jiných osob a projektů. Neexistuje žádný ředitel vývoje říkající, jak to bude fungovat. S cílem získat co nejvíce možných informací a učinit správné rozhodnutí, bylo nevyhnutné vtáhnout do projektu mnoho lidí z mnoha různých týmů. Projekt byl zahájen se dvěma vývojáři frameworku Talosu, dvěma na Datazillu/Graph Server a dvěma statistiky zapůjčenými z našeho týmu metrik. Od začátku jsme tento projekt otevřeli našim dobrovolníkům a vtáhli jsme do Mozilly mnoho nových tváří, stejně jako další, kteří používali Graph Server a některé TALOS testy na vlastních projektech. Jak jsme společně pracovali, pomalu jsme pochopili, které permutace zkušebních testům nám dají výsledky s menším šumem, povedlo se nám zapojit do projektu několik vývojářů Mozilly. Naše první schůzky s nimi byly pochopitelně obtížné, vzhledem k velkým navrhovaným změnám. Mystérium „Talosu" dělalo těžkým prodat to mnohým vývojářům, kteří se hodně starali o výkon.

Důležitá zpráva, které chvíli trvalo, než se usadila, byla, proč přepisování velkých komponent systému byl dobrý nápad, a proč bychom to nemohli jednoduše „na místě opravit". Nejběžnější

zpětnou vazbou bylo učinit několik malých změn do stávajícího systému, ale všichni, kteří tyto návrhy dělali, netušili, jak tento systém fungoval. Dělali jsme mnoho prezentací, mnoho speciálních schůzek, blogovali jsme, psali jsme, tweetovali atd. Udělali jsme všechno, co jsme mohli, abychom dostali názory. Protože jediná věc by byla horší, než dělat všechnu tuto práci k vytvoření lepšího systému, a to dělat všechnu práci a nemít nikoho, kdo by ji používal.

Uběhl rok od našeho prvního přezkoumání problematiky šumu Talosu. Vývojáři se těší na to, co uvolníme. Framework Talos byl refaktorován tak, že má jasnou vnitřní strukturu, a tak, že může současně reportovat do Datazilly a starého Graph Serveru. Ověřili jsme, že Datazilla může zpracovat rozsah dat, který jsme na ni hodili (1 TB dat za šest měsíců) a prověřili jsme naše metriky pro výsledky výpočtů. Nejvíce vzrušující bylo, že jsme našli způsob, jak poskytovat v reálném čase analýzy regrese/zlepšení na bázi jednotlivých změn do Mozilla stromů, což je velká výhra pro vývojáře.

Tak, teď když někdo tlačí změnu ve Firefox, zde je to, co dělá Talos:

- Talos sbírá 25 datových bodů pro každou stránku.
- Všechna tato čísla jsou odeslána do Datazilly.
- Datazilla provádí statistickou analýzu po vynechání prvních pěti datových bodů. (95 % šumu se nachází v prvních pěti datových bodech).
- Následně se použije Welchův T-test pro analýzu čísel a zjištění, zda existují nějaké extrémní hodnoty v datech na stránku ve srovnání s předchozími trendy z předchozích testů.[1]
- Všechny výsledky analýzy T-testu se pak pošlou do False Discovery Rate filtru, který zajišťuje, že Datazilla dokáže detekovat případné falešně pozitivní výsledky vznikající kvůli šumu.[2]
- Nakonec, pokud jsou výsledky v naší toleranci, Datazilla spustí na výsledky exponenciální vyhlazovací algoritmus ke generování nového trendu.[3] V případě, že výsledky nejsou v naší toleranci, nevytvoří novou čáru trendu a stránka je označena jako chybná.
- Určíme celkovou hranici přijetí/zamítnutí, založenou na procentu přijatých stránek. 95 % přijatých případů znamená „přijato“.

1: https://github.com/mozilla/datazilla/blob/2c39a3/vendor/dzmetrics/ttest.py

2: https://github.com/mozilla/datazilla/blob/2c369a/vendor/dzmetrics/fdr.py

3: https://github.com/mozilla/datazilla/blob/2c369a/vendor/dzmetrics/data_smoothing.py

Výsledky se vrací do systému Talos v reálném čase a Talos pak může hlásit sestavovacímu skriptu, zda je či není výkonnostní regrese. To vše se odehrává s 10–20 Talos běhy dokončenými každou minutu (tj. 1 TB dat) při současné aktualizaci výpočtů a uložení statistik.

Pokud to vezmeme z pohledu funkčního řešení pro plnou verzi Firefoxu, nahrazení stávajícího řešení vyžaduje spuštění obou systémů bok po boku. Tento proces zajistí, že vidíme všechny regrese hlášené původním Graph Serverem a ujistíme se, že jsou reálné a hlásí je také Datazilla. Vzhledem k tomu, že Datazilla reportuje na úrovni jednotlivých stránek místo testovací sady, bude potřebná aklimatizace na nové UI a způsob reportování regresí.

Při pohledu zpět by bylo rychlejší, kdybychom od začátku nahradili starý Talos. Refaktorováním však Mozilla přinesla mnoho nových přispěvatelů do projektu Talos. Refaktorování nás také přinutilo lépe porozumět testům, což se projevilo v opravě mnohých chybných testů a vypnutím testů s malou nebo žádnou hodnotu. Takže při zvažování, zda přepsat nebo refaktorovat, není jedinou metriku ke zvážení celkově vynaložený čas.

8.5 Závěr

V posledním roce jsme sáhli do každé části automatizovaného testování výkonnosti v Mozille. Analyzovali jsme testovací systém, nástroje pro reportování a statistickou spolehlivost výsledků, které byly generovány. V průběhu tohoto roku jsme využili toho, co jsme se naučili, aby se framework Talosu zjednodušil na údržbu, snadněji běžel, jednodušeji nastavil, snadněji testoval experimentální záplaty a byl méně náchylný k chybám. Vytvořili jsme Datazilla jako rozšiřitelný systém pro ukládání a načítání všech našich výkonnostních metrik z Talosu a jakýchkoliv budoucích automatizací výkonu. Restartovali jsme naši statistickou analýzu výkonu a vytvořili statisticky životaschopnou detekci regrese/zlepšení na jednu stránku. Učinili jsme všechny tyto systémy snadněji použitelné a otevřenější tak, aby se mohl každý přispěvatel kdekoli podívat do našeho kódu a dokonce experimentovat s novými metodami statistické analýzy na našich výkonnostních datech. Náš stálý závazek znovu a znovu revidovat data v každém milníku projektu a naše ochota vyhodit data, která se ukázala jako neprůkazná nebo neplatná, nám pomohl udržet naši pozornost, jak jsme posouvali tento gigantický projekt vpřed. Vtažení lidí z různých týmů v Mozille, stejně jako mnoho nových dobrovolníků, pomáhalo potvrdit platnost úsilí a také pomáhalo zavést opětovné vzkříšení monitorování výkonnosti a analýzu dat napříč několika oblastmi aktivit Mozilly, což má za následek ještě více daty řízenou, na výkon zaměřenou kulturu.

9 Zotonic

(Arjan Scherpenisse a Marc Worrell)

9 Zotonic

9.1 Úvod do Zotonicu

Zotonic je open source framework pro vývoj webových aplikací v plném rozsahu, od frontendu po backend. Skládá se z malého souboru základních funkcí, implementuje navíc odlehčený, ale rozšiřitelný Content Management systém. Hlavním cílem Zotonicu je snadno vytvořit dobře fungující webové stránky „z krabice“, takže aby byly webové stránky od začátku dobře škálovatelné.

Zatímco sdílí mnoho funkcí a funkcionalit s webovými vývojovými frameworky, jako je Django, Drupal, Ruby on Rails a Wordpress, jeho hlavní konkurenční výhodou je jazyk, kterým je Zotonic vyvinut: Erlang. Tento jazyk, původně vyvinutý pro budování telefonních přepínačů, umožňuje Zotonicu být odolný vůči poruchám a mít skvělé výkonové charakteristiky.

Jak název napovídá, tato kapitola se zaměřuje na výkon Zotonicu. Podíváme se na důvody, proč byl vybrán Erlang jako programovací platforma, následně prozkoumáme HTTP požadavky, pak se vrhneme na strategie ukládání do vyrovnávací paměti, které Zotonic využívá. Nakonec popíšeme optimalizace, které jsme aplikovali do dílčích modulů Zotonicu a databáze.

9.2 Proč Zotonic? Proč Erlang?

První práce na Zotonicu byla zahájena v roce 2008 a stejně jako mnoho projektů, pocházel z „škrabání svědícího“. Marc Worrell, hlavní architekt Zotonicu, pracoval sedm let v Mediamatic Labu v Amsterdamu na CMS podobném Drupalu, napsaném v PHP / MySQL s názvem Anymeta. Hlavní paradigma Anymety bylo, že implementovala „pragmatický přístup k vytváření sémantického webu“ modelováním všeho v systému jako generických „věcí“. I když byla úspěšná, její implementace trpěla problémy se škálovatelností.

Poté, co Marc opustil Mediamatic, strávil několik měsíců navrhováním správného, Anymetě podobného CMS, od nuly. Hlavními cíli návrhu Zotonicu bylo, že musí být snadno použitelný pro frontend vývojáře; musí podporovat snadný vývoj webových rozhraní s rychlou odezvou, současně umožňovat dlouhodobé připojení a mnoho krátkých požadavků; a musel mít dobře definované výkonové charakteristiky. Ještě důležitější je, že musel řešit nejběžnější problémy limitující výkonnost v předchozích přístupech vývoje webu, například musel odolat „ Shashdot efektu“ (náhlý příval návštěvníků).

Problémy s klasickým přístupem PHP+Apache

Klasické nastavení PHP běží jako modul uvnitř kontejneru webového serveru jako je Apache. Pro každou žádost se Apache rozhoduje, jak ji zpracuje. Když se jedná o požadavek na PHP, předá ho mod_php5 a PHP interpret spustí interpretaci skriptu. Toto je spojeno se latencí spouštění: typicky trvá až 5 ms, a PHP kód pak stále potřebuje být spuštěn. Tento problém může být částečně zmírněn pomocí PHP akcelerátorů, které předem kompilují PHP skript obcházením interpretu. Režii spouštění PHP lze také zmírnit pomocí správce procesů, jako je PHP-FPM.

Nicméně systémy jako tento ještě trpí problémem architektury, která nic nesdílí. Když skript potřebuje databázové připojení, je třeba ho vytvořit. Totéž platí i pro jakékoliv jiné I/O zdroje, které by jinak mohly být sdíleny mezi požadavky. Různé moduly mají k překonání tohoto problému trvalé připojení, ale neexistují žádná obecná řešení tohoto problému v PHP.

Manipulace s dlouho existujícími klientskými připojeními je také těžká, protože tato připojení potřebují pro každý požadavek samostatné vlákno nebo proces webového serveru. Případ Apache a PHP-FPM není škálován pro mnoho souběžných dlouho existujících připojení.

Požadavky na moderní webový framework

Moderní webové frameworky pracují typicky se třemi třídami HTTP požadavků. První jsou dynamicky generované stránky: dynamicky obsloužené, obvykle vytvořené pomocí procesoru šablon. Druhou je statický obsah: malé i velké soubory, které se nemění (např. JavaScript, CSS a soubory médií). Třetí jsou dlouhodobá připojení: WebSockets a žádosti s dlouhým dotazováním pro přidání interaktivity a obousměrné komunikace do stránek.

Před vytvořením Zotonicu jsme hledali softwarový framework a programovací jazyk, který by nám umožnil splnit naše návrhové cíle (vysoce výkonné, přátelské pro vývojáře), a u kterého bychom se vyhnuli úzkým místům spojeným s tradičními web serverovými systémy. V ideálním případě bude software splňovat následující požadavky.

- Souběžný: je třeba podporovat mnoho souběžných připojení, která nejsou omezena počtem unixových procesů nebo OS vláken.

- Sdílené zdroje: je třeba mít mechanismus pro levné sdílení zdrojů (například ukládání do vyrovnávací paměti, DB připojení) mezi požadavky.

- Aktualizace kódu za běhu: pro snazší vývoj a umožnění aktualizace produkčních systémů (udržení prostojů na minimu) by bylo dobré, kdyby mohly být změny v kódu nasazeny v běžícím systému, aniž by bylo nutné ho restartovat.

- Podpora vícejádrového CPU: moderní systém potřebuje škálovat přes více jader, protože současné procesory mají tendenci škálovat počet jader, na rozdíl od zvyšování rychlosti taktovací frekvence.

- Odolný proti chybám: systém musí být schopen zvládnout mimořádné situace, „špatně se chovající“ kód, anomálie nebo nedostatek zdrojů. V ideálním případě by toho systém dosáhl tím, že by měl nějaký mechanismus dohledu pro restartování vadných částí.

- Distribuovaný: v ideálním případě má systém vestavěnou a snadno nastavitelnou podporu pro distribuci přes více uzlů, což umožní lepší výkon a ochranou proti poruše hardwaru.

Erlang k záchraně

Pokud je nám známo, Erlang byl jediný jazyk, který tyto požadavky splnil „rozbalením z krabice“. Erlang VM, v kombinaci s jeho Open Telecom Platform (OTP), poskytoval systém, který nám dal a nadále dává všechny potřebné vlastnosti.

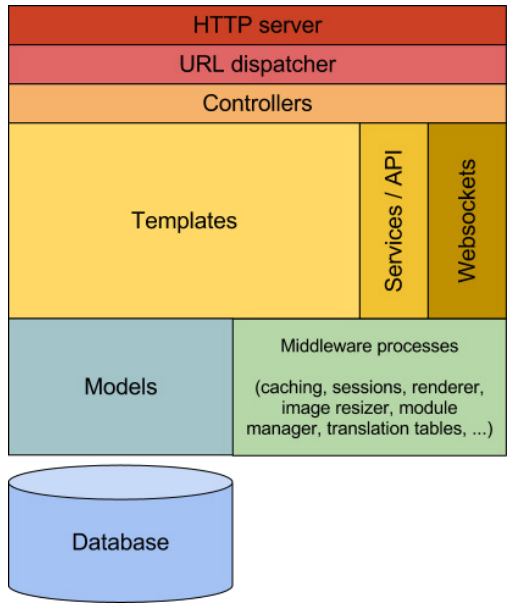
Erlang je (většinou) funkční programovací jazyk a běhový systém. Erlang/OTP aplikace byly původně vyvinuty pro telefonní přepínače, a jsou známé pro jejich odolnost vůči chybám a pro jejich souběžný charakter. Erlang využívá souběžnosti modelu na bázi aktéra: každý aktér je lehký „proces“, a jediný způsob, jak sdílet stav mezi procesy, je zaslat zprávu. Open Telecom Platform je sada standardních Erlang knihoven, které mimo jiné umožňují odolnost proti chybám a procesní dohled.

Odolnost vůči chybám je jádrem jeho paradigmatu programování: hlavní filozofií systému je nechte ho havarovat . Procesy nesdílejí žádný stav (ke sdílení stavu si musí posílat zprávy navzájem), jejich stav je izolován od ostatních procesů. Jako takový jeden havarující proces nikdy nesloží celý systém. Když dojde k chybě procesu, jeho nadřízený proces může rozhodnout o restartu.

Nechte ho havarovat vám také umožňuje programovat pro šťastný případ. Použití porovnávání vzorů a ochrany funkce pro zajištění normálního stavu znamená, že je potřeba méně kódu pro zpracování chyb, což obvykle vede k čistému, stručnému a čitelnému kódu.

9.3 Architektura Zetonicu

Než budeme diskutovat optimalizace výkonu Zotonicu, pojďme se podívat na jeho architekturu. Obrázek 9.1 popisuje nejdůležitější komponenty Zotonicu.



Obrázek 9.1: Architektura Zotonicu

Diagram ukazuje vrstvy Zotonicu, kterými prochází HTTP požadavek. Pro diskusi o výkonnostních problémech budeme muset vědět, co tyto vrstvy jsou a jak ovlivňují výkon.

Za prvé, Zotonic přichází s vestavěným webovým serverem Mochiweb (další projekt Erlangu). Nevyžaduje externí webový server, což udržuje závislosti nasazení na minimu.[1]

Stejně jako mnoho jiných webových frameworků, směrovací systém URL je používán pro předávání požadavků na řadič. Řadiče zpracují každou žádost způsobem RESTful prostřednictvím knihovny Webmachine.

Řadiče jsou záměrně „hloupé“, bez velké specifické aplikační logiky. Zotonic poskytuje řadu standardních řadičů, které jsou pro rozvoj základních webových aplikací často postačující.

1: Nicméně je možné přidat dopředu další webový server, například když ostatní internetové systémy běží na stejném serveru. Ale pro běžné případy to není nutné. Je zajímavé, že typické optimalizace používané jinými frameworky, jsou přidávány před svůj aplikační vyrovnávací webový server, jako je například Varnish, pro poskytování statických souborů, ale u Zotonicu by to významně neurychlilo požadavky, protože Zotonic také ukládá statické soubory v mezipaměti.

Například jediným cílem controller_template je odpovídat na HTTP GET žádosti o sestavení stránky pomocí dané šablony.

Šablonovým jazykem je Erlang implementace známého Django Template Language, nazývaného ErlyDTL. Obecnou zásadou v Zotonic je, že šablony řídí požadavky na data. Šablony rozhodnou, které údaje potřebují, a získají je z modelů.

Modely vystaví funkce pro načtení dat z různých datových zdrojů, např. databáze. Modely vystaví API pro šablony a diktují, jak mají být použity. Modely jsou také zodpovědné za ukládání výsledků do vyrovnávací paměti; rozhodnou, kdy a co je v mezipaměti a na jak dlouho. Když šablony potřebují data, volají model, jako kdyby se jednalo o globálně dostupnou proměnnou.

Model obaluje modul Erlangu, který je zodpovědný za určité údaje. Obsahuje všechny funkce nutné pro načtení a ukládání dat potřebným způsobem. Například centrální model Zotonicu se nazývá m.rsc aposkytuje přístup ke generickým zdrojům („stránka“) datového modelu. Vzhledem k tomu, že zdroje používají databázi, používá m_rsc.erl databázové připojení pro načtení svých dat a jejich zaslání do šablony, a ukládá je do mezipaměti, kdykoli je to možné.

Přístup „šablony řídí data“ se odlišuje od jiných webových frameworků, jako jsou Rails a Django, které obvykle následují více klasický MVC přístup, kdy řadič přiřadí data do šablony. Zotonic používá méně „řadičově centrický“ přístup, takže typické webové stránky mohou být postaveny jen psaním šablon.

Zotonic používá PostgreSQL pro datové persistence. Část 9.8 vysvětluje důvody této volby.

Další koncepty Zotonicu

I když jsou hlavním bodem této kapitoly výkonové charakteristiky webových požadavků, je dobré znát i některé z dalších konceptů, které jsou v jádru Zotonicu.

Virtuální hosting

Jedna instance Zotonicu typicky obslouží více, než jeden web. Zetonic je navržen pro virtuální hosting, včetně doménových aliasů a SSL podpory. A vzhledem k izolaci procesů Erlangu neovlivní havárie webových stránek jiné weby spuštěné na stejném VM.

Moduly

Moduly jsou způsob, jak Zotonic seskupuje funkcionality dohromady. Každý modul je ve svém vlastním adresáři, který obsahuje Erlang soubory, šablony atd. Mohou být povoleny na jeden web. Moduly lze připojit do administrátorského systému: například mod_backup modul přidává správu verzí do editoru stránek a také provádí denní plnou zálohu databáze. Další modul mod_github odhaluje webhook, který vytáhne, rekonstruuje a znovu načte Zotonic web z github, což umožňuje nepřetržité nasazení.

Notifikace
Chcete-li povolit rozšiřitelnost kódu, provádí se komunikace mezi moduly a komponenty jádra pomocí notifikačního mechanismu, který se chová buď jako mapa, anebo agreguje pozorovatele určité pojmenované notifikace. Posloucháním notifikací se pro modul stává snadným přepsání nebo rozšíření určitého chování. Volající funkce rozhoduje, zda se používá mapování nebo agregace. Například notifikace admin_menu je agregace nad moduly, která umožní modulům přidávat nebo odebírat položky nabídky v administrátorském menu.

Datový model
Hlavní datový model, který Zotonic používá, lze přirovnat k Drupal modulu Node; „každá věc je věc". Datový model se skládá z hierarchicky kategorizovaných zdrojů, které se připojují k jiným zdrojům použitím označených okrajů. Stejně jako zdroj inspirace Anymeta CMS, je i tento datový model volně založený na principech sémantického webu.

Zotonic je rozšiřitelný systém, všechny části systému se sčítají, vezmete-li do úvahy výkon. Můžete například přidat modul, který zachycuje webové požadavky, a dělá něco pro každou žádost. Takovýto modul může mít vliv na výkon celého systému. V této kapitole jej vynecháme, a místo toho se zaměříme na problémy výkonu jádra.

9.4 Řešení problému: Boj se Slashdot efektem

Většina webových stránek žije běžný život na malém místě někde na webu. Až dokud jedna z jeho stránek nenarazí na titulní stránku populární webové stránky, jako je CNN, BBC nebo Yahoo. V takovém případě se provoz na webových stránkách pravděpodobně během okamžiku zvýší na desítky, stovky nebo dokonce tisíce stránkových žádostí za sekundu.

Takový náhlý nárůst přetíží tradiční webový server a znepřístupní ho. Pojem „Slashdot efekt" byl pojmenován po webu, který začal tento druh drtivého doporučování. Ještě horší je, že přetížený server je někdy velmi těžké restartovat vzhledem k tomu, že nově nastartovaný server má prázdné vyrovnávací paměti, žádné připojení k databázi, často nezkompilované šablony atd.

Mnoho anonymních návštěvníků žádajících přesně stejnou stránku zhruba ve stejné chvíli by nemělo přetížit server. Tento problém je snadno řešitelný pomocí vyrovnávací proxy, jako je Varnish, která ukládá statické kopie stránky a pouze jednou za čas kontroluje aktualizace stránky.

Nával návštěvnosti se stává náročnější, pokud se vytváří dynamické stránky pro každého jednotlivého návštěvníka; ty nemůžou být ukládány v mezipaměti. Stanovili jsme si, že tento problém v Zetonicu vyřešíme.

Uvědomili jsme si, že většina webových stránek:

- má pouze omezený počet velmi oblíbených stránek,
- má dlouhou řadu mnohem méně populárních stránek, a
- mnoho společných částí na všech stránkách (menu, nejvíce čtené položky, zprávy atd.)

a rozhodli se:

- ukládat často používaná data v mezipaměti, takže není potřebná žádná komunikace k jejich zpřístupnění,
- sdílet sestavení stránek pomocí šablon a dílčích šablon mezi požadavky a na stránkách webu, a
- explicitně navrhnout systém, který zabrání přetížení při startu a restartu serveru.

Ukládání často používaných dat v mezipaměti

Proč načítat data z externího zdroje (databáze, memcached), když jiný požadavek data načetl několik milisekund předtím? Vždy jsme ukládali jednoduché datové požadavky. V další části je podrobně diskutován mechanismus ukládání v mezipaměti.

Sdílení vykreslené šablony a dílčí šablony mezi stránkami

Při vykreslování stránky nebo zahrnuté šablony může vývojář přidat nepovinné direktivy pro ukládání v mezipaměti. To uloží vykreslený výsledek na určitou dobu.

Uložení začíná funkcionalitou, kterou jsme nazvali *memo*: zatímco je vykreslena šablona a jeden nebo více procesů požádá o stejné vykreslování, budou pozdější procesy pozastaveny. Když je sestavení stránky provedeno, všem čekajícím procesům bude zaslán výsledek sestavení.

Samotná memorizace, bez jakéhokoliv dalšího ukládání do vyrovnávací paměti, poskytuje velké zvýšení výkonu tím, že výrazně snižuje množství paralelního zpracování šablon.

Zabránění přetížení serveru při startu nebo restartu

Zotonic záměrně zavádí několik úzkých míst. Tato úzká místa omezují přístup k procesům, které využívají omezené zdroje, nebo pokud je drahé (jde-li o procesor nebo paměť) je provést. Úzká místa jsou v současné době nastavena pro kompilátor šablony, proces změny velikosti obrázku a zásobník databázových připojení.

Úzká místa jsou implementována tím, že mají omezenou pracovní kapacitu pro provádění požadovaných činností. Pro procesorově nebo diskově intenzivní práci, jako je změna velikosti obráz-

ku, existuje jen jeden proces pro vyřizování požadavků. Požadující procesy pošlou požadavek do Erlang fronty požadavků a čekají, až se zpracuje. Vyprší-li časový limit požadavku, bude shozen. Takto shozený požadavek vrátí stav HTTP 503 *Služba není k dispozici*.

Čekací procesy nepoužívají mnoho zdrojů a úzká místa chrání před přetížením v případě, že je změněna šablona, nebo se nahrazuje obrázek na často požadované stránce a potřebuje oříznout nebo změnit velikost.

Stručně řečeno: zaneprázdněný server může stále dynamicky aktualizovat své šablony, obsah a obrázky, aniž by se přetížil. Zároveň umožňuje selhání jednotlivého požadavku, zatímco samotný systém pokračuje v činnosti.

Zásobník databázových připojení

Ještě pár slov o databázových připojeních. V Zotonicu proces načítá databázové připojení ze zásobníku připojení pro každý jednotlivý dotaz nebo transakci. To umožňuje mnohým souběžným procesům sdílet velmi omezený počet databázových připojení. Srovnejte to s většinou (PHP) systémů, kde každý požadavek drží databázové připojení po celou dobu trvání požadavku.

Zotonic zavře nepoužívaná databázová připojení po určité době nečinnosti. Jedno připojení je vždy ponecháno otevřené, takže systém může vždy rychle zpracovat příchozí požadavek nebo aktivitu na pozadí. Dynamický zásobník připojení drasticky snižuje počet otevřených databázových připojení na jeden nebo dva na většině Zotonic webů.

9.5 Vrstvy ukládání do mezipaměti

Nejtěžší část ukládání je zneplatnění mezipaměti: udržování čerstvých dat v mezipaměti a vyprazdňování starých. Zotonic používá k vyřešení tohoto problému centrální mechanismus s kontrolami závislosti.

Tato část popisuje mechanismus Zetonicu způsobem shora dolů: z prohlížeče přes stack až do databáze.

Ukládání do mezipaměti na straně klienta

Ukládání do mezipaměti na straně klienta je provedeno prohlížečem. Prohlížeč ukládá obrázky, CSS a JavaScript soubory. Zotonic neumožňuje na straně klienta ukládat HTML stránky, vždy všechny stránky dynamicky generuje. Vzhledem k tomu, že je při tom velmi efektivní (jak je popsáno v předchozí části), neukládání HTML stránek zabraňuje zobrazování starých stránek poté, co se uživatel přihlásí, odhlásí nebo jsou umístěny komentáře.

Zotonic zlepšuje výkon na straně klienta dvěma způsoby:

1. Umožňuje ukládání statických souborů (CSS, JavaScript, obrázky atd.) do mezipaměti

2. Zahrnuje více CSS nebo JavaScript souborů do jedné odpovědi

První se provádí přidáním odpovídajících HTTP hlaviček k požadavku[2]:

```
Last-Modified: Tue, 18 Dec 2012 20:32:56 GMT
Expires: Sun, 01 Jan 2023 14:55:37 GMT
Date: Thu, 03 Jan 2013 14:55:37 GMT
Cache-Control: public, max-age=315360000
```

Více CSS nebo JavaScript souborů je zřetězených do jediného souboru, oddělením jednotlivých souborů vlnovkou a uvedením cest pouze v případě, že se mění mezi soubory:

```
http://example.org/lib/bootstrap/css/bootstrap
  ~bootstrap-responsive~bootstrap-base-site~
  /css/jquery.loadmask~z.growl~z.modal~site~63523 81976.css
```

Číslo na konci je časové razítko nejnovějšího souboru v seznamu. Potřebný CSS odkaz nebo JavaScript skriptovací tag je generován pomocí {% lib %} tag šablony.

Ukládání dat do mezipaměti na straně serveru

Zotonic je velký systém a mnohé části v něm ukládají data do mezipaměti nějakým způsobem. Následující odstavce vysvětlují některé zajímavější části.

Statické CSS, JS a soubory obrázků

Manipulace řadiče se statickými soubory je optimalizovaná. Ta může rozložit požadavky kombinovaných souborů do seznamu jednotlivých souborů.

Řadič kontroluje hlavičku If-Modified-Since a v případě potřeby poskytuje HTTP stav 304 *Not Modified*.

2: Všimněte si, že Zotonic nenastaví ETag. Některé prohlížeče zkontrolují ETag pro každé použití souboru požadavkem na server. Což odporuje celé myšlence ukládání do mezipaměti a provádění méně požadavků.

V prvním požadavku zřetězí obsah všech statických souborů do jednoho bajtového pole (Erlang binary[3]). Toto bajtové pole je pak uloženo v mezipaměti centrální depcache (viz část 9.5) ve dvou formách: komprimované (s gzip) a nekomprimované. V závislosti na Accept-Encoding hlavičkách odeslaných prohlížečem poskytne Zotonic buď komprimovanou, nebo nekomprimovanou verzi.

Tento mechanismus je natolik efektivní, že jeho výkon je podobný mnoha vyrovnávacím proxy serverům, zatímco je plně řízen webovým serverem. S dřívější verzí Zotonicu a na jednoduchém hardware (quad core 2,4 GHz Xeon z roku 2008) jsme viděli jeho propustnost ve výši přibližně 6 000 požadavků/sekundu a byli schopni saturovat gigabitové ethernetové připojení požádáním o malý (~ 20 KB) obrázkový soubor.

Vykreslené šablony

Šablony jsou kompilovány do Erlang modulů, po čemž je bajtový kód uložen v paměti. Kompilované šablony jsou nazývány jako běžné funkce Erlang.

Šablonový systém rozpozná změny v šablonách a za běhu znovu překompiluje šablony. Když je kompilace ukončena, použije se Erlang mechanismus pro aktualizace kódu za běhu k načtení nově kompilovaného modulu Erlangu.

Řadiče hlavní stránky a šablony mají možnosti ukládat do mezipaměti výsledky sestavení stránek pomocí šablon. Ukládání může být také povoleno pouze pro anonymní (nepřihlášené) návštěvníky. Stejně jako u většiny ostatních webů, anonymní návštěvníci generují většinou hromadu všech požadavků a tyto stránky nebude možné přizpůsobit a budou (téměř) identické. Všimněte si, že šablona pro sestavení stránky je přechodný výsledek a nikoli konečná HTML. Tento dočasný výsledek obsahuje (mimo jiné) nepřeložené řetězce a fragmenty JavaScriptu. Finální HTML je generován na základě analýzy této dočasné struktury výběrem správného překladu a sesbíráním všech javascriptů.

Zřetězený JavaScript, spolu s jedinečným ID stránky, je umístěn do {% script %} tagu šablony. Mělo by to být těsně nad uzavírací </ body> značkou HTML kódu. Unikátní ID stránky se používá pro propojení vykreslované stránky s procesy zpracování Erlangu a pro interakci Websocket/ Comet na stránce.

Jako u všech šablonových jazyků, mohou šablony zahrnovat i jiné šablony. V Zotonicu jsou šablony obvykle kompilovány v řádku pro vyloučení jakékoliv ztráty výkonu používáním zahrnutých souborů.

3: Bajtové nebo binární pole je nativní Erlang datový typ. V případě, že je menší než 64 bajtů, je zkopírováno mezi procesy, ty větší jsou sdíleny mezi procesy. Erlang také sdílí části bajtových polí mezi procesy s odkazem na tyto části a nekopíruje samotná data, čímž bajtová pole dělá efektivním a snadno ovladatelným datovým typem.

Speciální volby můžou vynutit zapojení běhového prostředí. Jednou z těchto možností je ukládání do mezipaměti. Ukládání lze povolit jen pro anonymní návštěvníky, je možné nastavit dobu ukládání do mezipaměti a mohou být přidány závislosti. Tyto závislosti jsou používány ke zrušení platnosti uložených výsledků sestavení stránky, pokud se změní některý ze zobrazovaných zdrojů.

Jinou metodou pro ukládání částí šablony je použití blokové značky {% cache %} ... {% endcache %}, která ukládá do mezipaměti část šablony za daný čas. Tento tag má stejné možnosti ukládání jako include tag, ale má tu výhodu, že může být snadno přidán do stávajících šablon.

Ukládání v mezipaměti

Veškeré ukládání se provádí v paměti samotné VM Erlangu. Není nutná žádná komunikace mezi počítači nebo procesy operačního systému pro přístup k datům uloženým v mezipaměti. Tím se výrazně zjednodušuje a optimalizuje využití těchto dat.

Pro srovnání, přístup k memcache serveru obvykle trvá 0,5 milisekundy. V porovnání s tím trvá přístup k hlavní paměti v rámci stejného procesu 1 nanosekundu na CPU nalezení v mezipaměti a 100 nanosekund na CPU nenalezení v mezipaměti, nemluvě o velkém rozdílu rychlostí mezi pamětí a sítí.[4]

Zotonic má dva mechanismy pro kešování v paměti[5]:

1. Depcache, centrální mezipaměť pro jeden web

2. Memo Cache slovník procesu

Depcache

Centrálním mechanismem pro ukládání do mezipaměti na každém Zotonic webu je depcache, což je zkratka pro *dependency cache*. Depcache je úložiště v paměti typu klíč-hodnota se seznamem závislostí pro každý uložený klíč.

Pro každý klíč v depcache uložíme:

- hodnotu klíče;

- sériové číslo, globální celé číslo inkrementované při každém požadavku o aktualizaci;

- dobu platnosti klíče (počítaná v sekundách);

4: Viz „Čísla zpoždění, která by měl znát každý programátor"
http://www.eecs.berkeley.edu/~rcs/research/interactive_latency.html.

5: Kromě těchto mechanismů provádí databázový server některé kešování v paměti, ale to není v rámci této kapitoly.

- seznam dalších klíčů, na kterých tento klíč závisí (například ID zdroje zobrazené v šabloně uložené v mezipaměti); a
- pokud se klíč stále ještě vypočítává, seznam procesů čekajících na hodnotu klíče.

Pokud je požadován klíč, pak mezipaměť zjistí, zda je přítomen, zda neexpiroval a zda jsou sériová čísla všech závislých klíčů nižší než sériové číslo klíče uloženého v mezipaměti. V případě, že je klíč stále platný, je vrácena jeho hodnota, jinak se klíč a jeho hodnota odstraní z mezipaměti a vrátí se hodnota undefined.

Případně, pokud byl klíč vypočítáván, byl přidán žádající proces do seznamu čekajících na klíč.

Implementace využívá ETS, Erlang Term Storage, což je standardní implementace rozptylové tabulky, která je součástí distribuce Erlang OTP. Následující tabulky ETS jsou vytvořeny Zotonicem pro depcache:

- Meta tabulka: ETS tabulka drží všechny uložené klíče, expirace a závisející klíče. Záznam v této tabulce je zapsán jako #meta {key, expire, serial, deps}.
- Deps tabulka: ETS tabulka ukládá sériové číslo pro každý klíč.
- Data tabulka: ETS tabulka, která ukládá data každého klíče
- Čekající procesy: ETS tabulka, která ukládá ID všech procesů čekajících na příchod hodnoty klíče.

Tabulky ETS jsou optimalizovány pro paralelní čtení a volající proces k nim obvykle přímo přistupuje. Tím se zabrání jakékoliv komunikaci mezi volajícím procesem a procesem depcache.

Proces depcache je volán pro:

- memorizace, kde procesy čekají na vypočtení hodnoty jiného procesu;
- put (uložení) požadavky, serializující přírůstky sériových čísel; a
- požadavky na vymazání, také serializující depcache přístup.

Depcache může být docela velká. Pro zabránění příliš velkého zvětšování existuje proces garbage collectoru. Garbage collector pomalu iteruje přes celou depcache a vymaže expirované nebo neplatné klíče. Pokud je velikost depcache nad určitou hraniční hodnotu (100 MiB ve výchozím nastavení), pak se garbage collector zrychlí a vymaže 10 % všech položek, na které narazí. V mazání pokračuje do doby, dokud využitá velikost mezipaměti neklesne pod stanovenou mez.

100 MiB se může zdát málo v oblasti databází s více TB. Nicméně mezipaměť většinou obsahuje textová data a bude tedy dostatečně velká, aby obsahovala často používaná data pro většinu webových stránek. V opačném případě může být velikost mezipaměti změněna v konfiguraci.

Memo Cache seznam procesů

Druhé paradigma ukládání v mezipaměti v Zotonicu je memo cache seznam procesů. Jak již bylo uvedeno dříve, přístup k datům je diktován šablonami. Systém mezipaměti používá jednoduché heuristiky pro optimalizaci přístupu k datům.

Důležité v této optimalizaci je ukládání dat procesu zpracujícího požadavek v Erlang ve slovníku procesu. Slovník procesu je jednoduché úložiště klíč-hodnota v haldě procesu. V podstatě přidává stav do funkčního Erlang jazyka. Použití slovníku procesu je obvykle z tohoto důvodu odsuzováno, ale pro ukládání do mezipaměti v procesu je užitečné.

Je-li zdroj přístupný (pamatujte, že zdroj je centrální datovou jednotkou Zotonicu), je kopírován do slovníku procesu. To samé je provedeno s výpočetními výsledky, jako jsou kontroly řízení přístupu, a dalšími daty, jako jsou konfigurační hodnoty.

Každá vlastnost zdroje, např. jeho název, shrnutí nebo tělo textu, musí, pokud je uvedena na stránce, provádět kontrolu řízení přístupu a poté načíst požadovanou vlastnost ze zdroje. Ukládání vlastností celého zdroje do mezipaměti a jeho přístupových kontrol výrazně urychluje využití dat zdrojů a odstraňuje mnoho nevýhod těžko předvídatelných vzorů přístupu šablony k datům.

Stránka nebo proces mohou použít velké množství dat, memo cache má několik tlakových ventilů:

1. při držení více než 10 000 klíčů je celý slovník procesu vyprázdněn. Tím se zabrání tomu, aby slovník procesu držel mnoho nepoužitých položek, jako tomu je v případě procházení dlouhého seznamu zdrojů. Speciální Erlang proměnné, jako např. $ancestors, jsou zachovány.

2. Memo cache musí být programově povolena. To se děje automaticky pro každý příchozí HTTP nebo WebSocket požadavek a šablony.

3. Mezi HTTP/WebSocket požadavky je slovník procesu vyprázdněn, protože vícenásobné sekvenční HTTP/WebSocket požadavky jsou obslouženy ve stejném procesu.

4. Memo cache nesleduje závislosti. Jakékoliv mazání depcache také kompletně vyprázdní slovník procesu provádějícího mazání.

Je-li memo cache zakázaná, je každé vyhledávání zpracováno depcache. To má za následek volání procesu depcache a kopírování dat mezi depcache a žádajícím procesem.

9.6 Erlang virtuální stroj

Erlang virtuální stroj (VM) má několik vlastností, které jsou důležité při pohledu na výkon.

Procesy jsou levné

Erlang VM je specificky navržen tak, aby vykonával mnoho věcí paralelně, a jako takový má vlastní implementaci multiprocesního systému v rámci VM. Erlang procesy jsou naplánovány na základě redukce počtu, kde jedna redukce je zhruba ekvivalentní volání funkce. Proces je ponechán v běhu, dokud se nepozastaví čekáním na vstup (zpráva z jiného procesu), nebo až do provedení pevného počtu redukcí. Plánovač startuje pro každé CPU jádro v jeho vlastní frontě běhu. Pro Erlang aplikace není neobvyklé mít tisíce až miliony živých procesů ve VM v jakémkoliv okamžiku.

Procesy nejsou levné jen startem, ale jsou také levné v paměti, 327x2 byty na jeden proces, což představuje ~ 2,5 KiB na 64 bitovém stroji.[6] To je srovnatelné s ~ 500 KiB pro Javu a výchozí hodnotu 2 MiB pro pthreads.

Vzhledem k tomu, že je použití procesů tak levné, jakékoliv zpracování, které není nutné pro výsledek požadavku, je převedeno do samostatného procesu. Zaslání e-mailu nebo logování jsou příklady úkolů, které by mohly být řešeny samostatným procesem.

Kopírování dat je drahé

Zprávy mezi procesy v Erlang VM jsou relativně drahé, protože zpráva je v procesu kopírovaná. Toto kopírování je nutné kvůli tomu, že Erlang má garbage collector pro každý proces. Prevence kopírování dat je důležitá; což je důvod, proč depcache Zotonicu používá ETS tabulky, ke kterým lze přistupovat z libovolného procesu.

Samostatná halda pro větší bajtová pole

Existuje velká výjimka pro kopírování dat mezi procesy. Bajtová pole větší než 64 bajtů nejsou kopírována mezi procesy. Mají své vlastní haldy a jsou samostatně uvolňována garbage collectorem.

6: Viz http://www.erlang.org/doc/efficiency_guide/advanced.html#id68921

To umožňuje levně poslat velké bajtová pole mezi procesy, protože je kopírován pouze odkaz na pole bajtů. Nicméně to ztěžuje garbage collection, protože všechny odkazy musí být uvolněny garbage collectorem před tím, než může být uvolněno pole bajtů.

Někdy jsou předány odkazy na části velkého pole bajtů: větší pole bajtů nemůže být uvolněno garbage collectorem, dokud se neuvolní garbage collectorem odkaz na menší část. Důsledkem je, že pokud se uvolní větší pole bajtů, je kopírování pole bajtů optimalizací.

Zpracování řetězce je drahé

Zpracování řetězec v jakémkoliv funkčním jazyku může být nákladné, protože řetězce jsou často reprezentovány jako spojené seznamy celých čísel, a vzhledem k funkční povaze Erlangu nemohou být data destruktivně aktualizována.

Pokud je řetězec reprezentován jako seznam, pak je zpracován pomocí rekurzivních funkcí a porovnávání. To ho dělá přirozeným pro funkcionální jazyky. Problém je v tom, že reprezentace dat propojeného seznamu má velkou režii a zasílání seznamu do jiného procesu vždy zahrnuje kopírování celé datové struktury. Tím je seznam neoptimální volbou pro řetězce.

Erlang má svou vlastní kompromisní odpověď pro řetězce: io-seznamy. Io-seznamy jsou vnořené seznamy obsahující seznamy, celá čísla (jedna bajtová hodnota), pole bajtů a odkazy na části jiných polí bajtů. Io-seznamy je velmi snadné používat a připojování, prefixování nebo vkládání dat není drahé, protože potřebují jen změny v relativně krátkých seznamech, a to bez jakéhokoliv datového kopírování.[7]

Io-seznam může být odeslán tak, jak je na „port“ (popisovač souboru), který převede datovou strukturu do proudu bajtů a odešle ji do soketu.

Příklad io-seznamu:

```
[ <<“Hello“>>, 32, [ <<“Wo“>>, [114, 108], <<“d“>>].
```

který se převede do bajtového proudu:

```
<<“Hello World“>>.
```

7: Erlang může také sdílet části bajtového pole s odkazy na tyto části, čímž se obchází nutnost kopírování dat. Vložení do pole bajtů může být reprezentováno io-seznamem o třech částech: odkazy na nezměněné hlavičkové bajty, vložené hodnoty a odkaz na nezměněné koncové bajty.

Je zajímavé, že většinou se zpracování řetězce ve webových aplikacích skládá z:

1. Zřetězení dat (dynamických a statických) do výsledné stránky.

2. Převodu HTML kódu a začištění obsahových hodnot.

Io-seznam Erlangu je perfektní datová struktura pro první bod výše. A druhý bod výše je vyřešen pomocí agresivního začištění veškerého obsahu předtím, než je uložen v databázi.

Kombinace obou znamená, že pro Zotonic je vykreslená stránka jen velké zřetězení polí bajtů a předem očištěných hodnot v jednom io-seznamu.

Důsledky pro Zotonic

Zotonic velmi ztěžuje použití relativně velké datové struktury, kontextu. Jedná se o záznam obsahující veškerá data potřebná pro vyhodnocení požadavku. Obsahuje:

- data požadavku: hlavičky, argumenty požadavku, tělo atd.
- stav Webmachine
- informace o uživateli (např. ID uživatele, informace o řízení přístupu)
- jazykové preference
- třída User-Agent (např. text, telefon, tablet, počítač)
- odkazy na speciální procesy webu (např. oznamovač, depcache, atd.)
- unikátní ID zpracovávaného požadavku (to se stane ID stránky)
- ID proces relace a stránky
- proces připojení k databázi během transakce
- akumulátory pro data odpovědi (např. data, akce, které mají být vykresleny, JavaScript soubory).

Všechna tato data mohou tvořit velkou datovou strukturu. Odeslání tohoto rozsáhlého kontextu do jiných procesů pracujících na požadavku by způsobilo podstatnou režii na kopírování dat.

Proto se snažíme dělat většinu ze zpracování požadavku v jediném procesu Mochiweb, který požadavek přijal. Doplňkové moduly a rozšíření se volají pomocí funkčního volání místo použití meziprocesních zpráv.

Někdy je rozšíření realizováno pomocí samostatného procesu. V tomto případě rozšíření poskytuje funkce akceptující Kontext a ID procesu rozšíření. Tato funkce rozhraní je pak zodpovědná za efektivní zasílání zpráv procesu rozšíření.

Zotonic také potřebuje poslat zprávu při sestavování stránky pomocí šablony, kterou lze uložit v mezipaměti. V tomto případě je Kontext ořezán o všechny průběžné výsledky šablony a některá další nepotřebná data (jako jsou informace o logování) předtím, než je Kkontext zaslán procesu sestavujícímu stránku s dílčí šablonou.

Příliš nás nezajímá zasílání polí bajtů tak, jak jsou, protože ve většině případů jsou větší než 64 bajtů a jako takové nebudou kopírovány mezi procesy.

Pro obsloužení velkých statických souborů existuje možnost využití systémového volání Linux sendfile() pro delegování odeslání souboru do operačního systému.

9.7 Změny v Webmachine knihovně

Webmachine je knihovna implementující abstrakci protokolu HTTP. Je implementovaná nad knihovnou Mochiweb, která provádí nižší úroveň HTTP zpracování, např. procesů přijímače, parsování hlavičky atd.

Řadiče jsou vytvořeny Erlang moduly implementací funkce zpětného volání. Příklady funkcí zpětného volání jsou resource_exists, previously_existed, authorized, allowed_methods, process_post atd. Webmachine rovněž porovnává cesty požadavku oproti seznamu odesílacích pravidel; přiřazením argumentů požadavku a výběrem správného řadiče pro vyřízení HTTP požadavku.

S Webmachine se zpracování HTTP protokolu stává snadným. Z tohoto důvodu jsme se brzo rozhodli postavit Zotonic na Webmachine.

Při budování Zotonicu bylo zjištěno několik problémů s Webmachine.

1. Když jsme začínali, podporoval pouze jeden seznam pravidel odesílání; ne seznam pravidel na jednoho hostitele (tj. web).

2. Odesílací pravidla jsou nastavena v prostředí aplikace a kopírována do procesu zpracování požadavku při spouštění.

3. Některé funkce zpětného volání (jako last_modified) jsou při vyhodnocování požadavku volány vícekrát.

4. Když Webmachine při vyhodnocování požadavku spadne, nevytvoří se žádná položka v logu.

5. Není podpora pro upgrade HTTP, což stěžuje podporu WebSockets.

První problém (neoddělování pravidel odesílání) je jen nepříjemností. Stěžuje interpretaci a snižuje intuitivnost seznamu pravidel pro odesílání.

Druhý problém (kopírování seznamu pro odesílání na každý požadavek) se ukázal být velkým problémem pro Zotonic. Seznamy by se mohly stát tak velké, že jejich kopírování by mohlo zabrat většinu času potřebného pro zpracování žádosti.

Třetí problém (více volání stejných funkcí) přinutil vývojáře řadiče implementovat svůj vlastní mechanismus ukládání do mezipaměti, který je náchylný k chybám.

Čtvrtý problém (žádný log při pádu) ztěžuje identifikaci problémů v produkci.

Pátý problém (bez HTTP upgradu) nám brání použít pěkné abstrakce, které jsou k dispozici v Webmachine pro WebSocket připojení.

Výše uvedené problémy byly natolik závažné, že jsme museli modifikovat Webmachine pro naše vlastní účely.

Byla přidána první nová možnost: dispečer. Dispečer je modul implementující dispatch/3 funkci, která porovnává požadavek se seznamem pro odeslání. Dispečer také zvolí správný web (virtuálního hostitele) pomocí HTTP Host hlavičky. Při testování jednoduchého „Hello World“ řadiče tyto změny trojnásobně zvýšily propustnost. Pozorovali jsme také, že byl mnohem vyšší zisk na systémech s mnoha virtuálními hostiteli a pravidly pro odeslání.

Webmachine udržuje dvě datové struktury, jednu pro data požadavku a jednu pro vnitřní stav zpracování požadavku. Tyto datové struktury se odkazovaly na sebe navzájem a ve skutečnosti se téměř vždy používaly v tandemu, takže jsme je zkombinovali do jediné datové struktury. Což usnadňuje odstranění používání slovníku procesu a přidání jedné nové datové struktury jako argumentu ke všem funkcím uvnitř Webmachine. Výsledkem bylo 20% zkrácení doby zpracování na jeden požadavek.

Optimalizovali jsme Webmachine mnoha jinými způsoby, které zde nebudeme podrobně popisovat, ale nejdůležitější body jsou následující:

- Návratové hodnoty některých zpětných volání řadiče jsou uloženy v mezipaměti (charsets_provided, content_types_provided, encodings_provided, last_modified a generate_etag).

- Větší část využívání slovníku procesu byla odstraněna (méně globální stav, přehlednější kód, snadnější testování).

- Separovali jsme proces logování požadavku; i když požadavek spadne, máme log až do okamžiku pádu.

- Bylo přidáno zpětné volání HTTP Upgrade jako krok po *forbidden* kontrole přístupu pro podporu protokolu WebSocket.

- Původně byl řadič nazýván „zdroj". Změnili jsme ho na „řadič", abychom rozlišovali mezi (datovými) zdroji, které jsou obsluhovány, a kódem, který těmto zdrojům slouží.

- Byly přidány instrumenty k měření rychlosti a velikosti požadavku.

9.8 Datový model: databáze dokumentů v SQL

Z hlediska dat stojí za zmínku, že všechny vlastnosti „zdroje" (hlavní datové jednotky Zotonicu) jsou serializovány do binárního blobu; „skutečné" databázové sloupce se používají pouze pro klíče, dotazování a omezení cizího klíče.

Samostatná „pivotová" pole a tabulky jsou přidány pro vlastnosti, nebo kombinace vlastností, které potřebují indexování, např. full-text sloupce, vlastnosti data atd.

Při aktualizaci zdroje přidává databázový trigger ID zdroje do pivotové fronty. Tato pivotová fronta je používaná samostatným Erlang procesem na pozadí, který indexuje skupinu zdrojů najednou v jediné transakci.

Volba SQL nám umožnila našlápnout: PostgreSQL má dobře známý dotazovací jazyk, velkou stabilitu, známý výkon, vynikající nástroje a jak komerční, tak nekomerční podporu.

Kromě toho není databáze limitujícím faktorem výkonu Zotonicu. Pokud se dotaz stane úzkým místem, pak je úkolem vývojáře optimalizovat konkrétní dotaz pomocí parseru databázových dotazů.

Konečně zlaté pravidlo výkonu pro práci s libovolnou databází je: Nesahejte do databáze; nesahejte na disk; nesahej tedy sítě; sahejte do mezipaměti.

9.9 Srovnávací testy, statistiky a optimalizace

Příliš nevěříme srovnávacím testům, protože často testují pouze minimální části systému a nereprezentují výkon celého systému. Zvlášť když má systém mnoho pohyblivých částí a v Zotonicu je systém mezipaměti a zpracování obecných přístupových vzorů nedílnou součástí designu.

Zjednodušené srovnávací testy

Co by srovnávací test mohl udělat, je ukázat, kde byste mohli optimalizovat systém jako první.

S tímto vědomím jsme testovali Zotonic pomocí TechEmpower JSON srovnávacího testu, který hlavně testuje dispečera požadavků, kodér JSON, zpracování HTTP požadavku a TCP/IP architektura.

Srovnávací test byl vykonán na Intel i7 quad core M620 @ 2.67 GHz. Příkaz byl:

```
rk -c 3000 -t 3000 http://localhost:8080/json
```

Výsledky jsou zobrazeny v tabulce 9.1.

Platforma	x1000 Požadavků/sekundu
Node.js	27
Cowboy (Erlang)	31
Elli (Erlang)	38
Zotonic	5.5
Zotonic w/o přístupový log	7.5
Zotonic w/o přístupový log, s dispečerem zásobníku	8.5

Tabulka 9.1: Výsledky srovnávacího testu

Dynamický dispečer Zotonicu a protokol HTTP abstrakce dává nižší skóre v tomto mikro srovnávacím testu. Ta jsou relativně snadno řešitelná a řešení jsou již v plánu:

- Nahradit standardní webmachine zapisovač logu více efektivním zapisovačem.
- Kompilovat odesílací pravidla v modulu Erlang (místo jediného procesu interpretujícího seznam odesílacích pravidel).
- Vyměnit MochiWeb HTTP obsluhu za Elli HTTP obsluhu.
- Použít pole bajtů ve Webmachine namísto současných seznamů znaků.

Výkonnost v reálném životě

V roce 2013, kdy abdikovala nizozemská královna a následovala inaugurace nového holandského krále, byl národní hlasovací web vytvořen pomocí Zotonicu. Klient požadoval 100% dostupnost a vysoký výkon tak, aby byl schopen zvládnout 100 000 hlasů za hodinu.

Řešením byl systém se čtyřmi virtuálními servery, každý s 2 GB RAM a běžící na svém vlastním nezávislém Zotonic systému. Tři uzly vyřizovaly hlasování, jeden uzel byl pro administraci. Všechny uzly byly nezávislé, ale hlasovací uzly sdílely každý hlas s nejméně dvěma dalšími uzly, takže žádný hlas nebyl ztracen, pokud uzel havaroval.

Jediný hlas dal ~ 30 HTTP požadavků pro dynamické HTML (v několika jazycích), Ajax a statické soubory, jako CSS a JavaScript. Vícenásobné požadavky byly zapotřebí pro výběr tří projektů k hlasování o vyplnění detailů voliče.

Při testování jsme snadno splnili požadavky zákazníka bez tlačení systému na maximum. Simulace hlasování byla zastavena při 500 000 kompletních hlasovacích procedurách za hodinu, za použití přenosové rychlosti kolem 400 Mbps, a 99 % časů zpracování požadavku bylo nižších než 200 milisekund.

Z výše uvedeného je zřejmé, že Zotonic zvládne oblíbené dynamické webové stránky. Na skutečném hardwaru jsme pozorovali mnohem vyšší výkon, zejména pro I/O operace a databázový výkon.

9.10 Závěr

Při budování CMS (systém pro správu obsahu) nebo frameworku je důležité brát v úvahu celou aplikaci, od webového serveru, systému vyřizování požadavků, systému ukládání do mezipaměti, až po databázový systém. Pro dobrý výkon musí společně dobře fungovat všechny části.

Hodně výkonu lze získat předzpracováním dat. Jako příklad předzpracování je předběžný převod HTML kódů a očištění dat před uložením do databáze.

Ukládání často používaných dat do mezipaměti je dobrá strategie pro webové stránky s jasnou sadou oblíbených stránek, následovaných dlouhou řadou méně oblíbených stránek. Umístěním této mezipaměti v stejné paměti s kódem pro vyřizování požadavků poskytuje jasnou výhodu oproti použití oddělených mezipaměti serverů, a to jak v rychlosti, tak v jednoduchosti.

Další optimalizací pro zpracování náhlého výbuchu oblíbenosti je dynamické porovnávání podobných požadavků a jejich zpracování najednou se stejným výsledkem. Když je toto dobře implementováno, lze se vyhnout proxy a všechny HTML stránky lze generovat dynamicky.

Erlang se skvěle hodí pro budování systémů na bázi dynamického webu díky svému lehkému více procesnímu systému, zpracování chyb a řízení paměti.

Použití Erlangu umožňuje Zotonicu vytvořit velmi kompetentní a dobře fungující systém pro správu obsahu a framework bez nutnosti samostatných webových serverů, vyrovnávacích proxy, memcache serverů, nebo obsluh e-mailů. Tím se výrazně zjednodušuje systém řízení úkolů.

Jeden Zotonic server dokáže na současném hardwaru zpracovat tisíce dynamických požadavků stránek za sekundu, čímž lehce poslouží převážné většině webů na Internetu.

Použitím Erlangu je Zotonic připraven na budoucí vícejádrové systémy s desítkami jader a mnoha gigabajty paměti.

9.11 Poděkování

Autoři by rádi poděkovali Michielu Klønhammerovi (Maximonster Interactive Things), Andreasu Steniusovi, Maas-Maartenu Zeemanovi a Atillovi Erdődimu.

10 Tajemství výkonu mobilní sítě

(Bryce Howard)

10 Tajemství výkonu mobilní sítě

10.1 Úvod

Několik posledních let přineslo významný pokrok ve výkonu mobilní sítě. Ale mnoho mobilních aplikací nemůže plně těžit z tohoto pokroku v důsledku zvýšeného síťového zpoždění.

Zpoždění je již dlouho synonymem pro mobilní sítě. Přestože bylo v posledních letech dosaženo pokroku, snížení síťového zpoždění nedrží krok s nárůstem rychlosti. V důsledku tohoto rozdílu je zpoždění, ne propustnost, faktorem limitujícím výkon síťových transakcí.

V této kapitole jsou dvě logické části. První část bude zkoumat specifika mobilní celulární sítě, která přispívají k problému zpoždění. Ve druhé části budou představeny softwarové technologie pro minimalizaci vlivu zvýšeného síťového zpoždění na výkon.

10.2 Na co čekáte?

Zpoždění představuje čas potřebný pro tranzit datového paketu přes síť nebo několik sítí. Mobilní sítě navýšily ve většině internetové komunikace již přítomné zpoždění řadou faktorů, včetně typu sítě (např. HSPA+ vs. LTE), operátora (např. AT&T vs. Verizon) nebo okolností (např. stojící vs. jedoucí, geografie, denní doba atd.). Je těžké uvést přesné hodnoty pro zpoždění mobilní sítě, ale může být zjištěno, že se liší od desítek až stovek milisekund.

Čas cesty tam i zpět (Round-trip time, RTT) je mírou zpoždění datového paketu při jeho pohybu ze zdroje do místa určení a zpět. RTT má zásadní vliv na výkonnost mnoha síťových protokolů. Důvod může být ilustrován na seriózním sportu jako je ping-pong.

V běžném ping-pongu je doba potřebná k cestování míče mezi hráči sotva znatelná. Nicméně jak hráči stojí dále od sebe, začnou trávit více času čekáním na míč, než děláním něčeho jiného. Stejná 5 minutová ping-pongová hra by při regulaci vzdálenosti trvala několik hodin, pokud by hráči stáli tisíce stop od sebe, jakkoliv směšné se to zdá. Nahraďte klienta a server za 2 hráče a RTT za vzdálenost mezi hráči a začnete vidět problém.

Většina síťových protokolů hraje ping-pong jako součást jejich normálního provozu. Jejich voleje jsou obousměrné výměny zpráv potřebných k vytvoření a udržování logické síťové relace (např. TCP), nebo provedení požadavku na službu (například HTTP). V průběhu výměny těchto zpráv jsou odesílána malá nebo žádná data a šířka pásma sítě je do značné míry nepoužívaná. Zpoždění do značné míry odpovídá rozsahu tohoto nedostatečného využívání; každá výměna zpráv způsobuje minimálně RTT zpoždění sítě. Kumulativní dopad na výkon je znatelný.

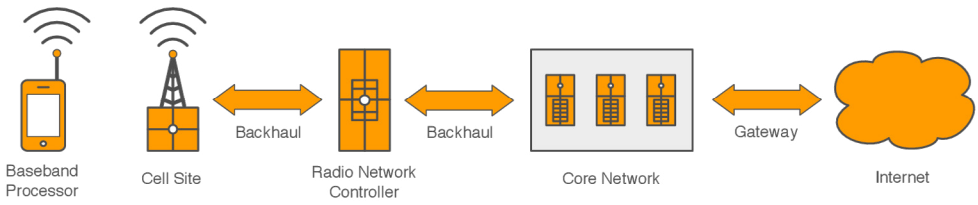
Uvažujme, že HTTP požadavek na stažení 10KiB objektu by zahrnoval 4 výměnné zprávy. Nyní počítejme čas 100 ms na obousměrnou cestu (docela rozumné pro mobilní sítě). Zohledněním obou těchto čísel vypočítáme efektivní propustnost 10KiB za 400 ms, nebo 25KiB za sekundu.

Všimněte si, že šířka pásma je v předchozím příkladu zcela irelevantní, bez ohledu na to, jak rychlá je síť, výsledek zůstává stejný, 25KiB/s. Výkonnost předchozí operace, nebo jakékoliv podobné operace, může být zlepšena pouze jediným, jasnou strategií: vyhnout se obousměrné výměně zpráv mezi síťovým klientem a serverem.

10.3 Mobilní celulární sítě

Níže je uveden zjednodušený úvod do komponent a konvence mobilních celulárních sítí, které zapadají do skládačky se jménem zpoždění.

Mobilní celulární síť je reprezentována řadou vzájemně propojených komponent, které mají vysoce specializované funkce. Každá z těchto složek přispívá k síťovému zpoždění nějakým způsobem, ale v různé míře. Existují specifické konvence pro celulární sítě, jako je řízení rádiových zdrojů, které také ovlivňují rovnici pro zpoždění mobilní sítě.



Obrázek 10.1: Komponenty mobilní celulární sítě

Baseband procesor

Uvnitř většiny mobilních zařízení jsou ve skutečnosti dva velmi sofistikované počítače. Aplikační procesor je zodpovědný za hostování operačního systému a aplikací, a je analogický k počítači nebo notebooku. Baseband procesor je odpovědný za všechny funkce bezdrátové sítě a je analogický k počítačovému modemu, který používá rádiové vlny místo telefonní linky.[1]

1: Ve skutečnosti mnoho mobilních telefonů řídí baseband procesor sadou AT podobných příkazů.
Viz http://www.3gpp.org/ftp/Specs/html-info/ 7 7.htm

Baseband procesor je zdrojem konzistentním, ale obvykle se zanedbatelným zpožděním. Vysokorychlostní bezdrátové sítě jsou děsivě složitá záležitost. Sofistikované zpracování signálu způsobuje fixní zpoždění v rozmezí od mikrosekund po milisekundy pro většinu síťové komunikace.

Buňka

Buňka, synonymum pro vysílač základní stanice nebo buňková věž, slouží jako přístupový bod pro mobilní sítě. Je odpovědností buňky zajistit síťové pokrytí v oblasti známé jako buňka.

Tak jako mobilní zařízení, kterým slouží, je buňka zatížena sofistikovaným zpracováním spojeným s vysokorychlostními sítěmi, a přispívá podobně zanedbatelným zpožděním. Nicméně buňka musí simultánně sloužit stovkám až tisícům mobilních zařízení. Odchylky v zatížení systému přinesou změny propustnosti a zpoždění. Stagnující, nespolehlivý výkon sítě na přeplněných veřejných akcích je často výsledkem buňkou dosažených limitů zpracování.

Poslední generace mobilních sítí rozšířila roli buňky zahrnutím přímého řízení mobilních zařízení. Mnoho funkcí, které byly předtím v odpovědnosti řadiče rádiové sítě, jako je registrace sítě nebo plánování přenosu, jsou teď zpracovány buňkou. Z důvodů vysvětlených později v této kapitole, tato role zodpovídá za redukci zpoždění dosažené poslední generací mobilních celulárních sítí.

Páteřní síť

Páteřní síť je dedikované WAN propojení mezi místem buňky, jeho řadičem a jádrem sítí. Páteřní sítě dlouho byly a jsou notorickými přispěvateli zpoždění.

Zpoždění páteřní sítě klasicky vzniká z okruhově přepínaných nebo rámcových transportních protokolů použitých na starších mobilních sítích (např. GSM, EV-DO). Takovéto protokoly vykazují zpoždění kvůli jejich synchronní povaze, kde jsou logická připojení reprezentována kanálem, který může přijmout nebo odeslat data přes krátký, předem přidělený časový úsek. Pro porovnání, mobilní sítě poslední generace používají páteřní síť založenou na IP paketovém přepínání podporujícím asynchronní datový přenos. Toto přepínání drasticky zredukovalo páteřní zpoždění.

Limitace šíře pásma fyzické infrastruktury jsou pokračujícím úzkým místem. Mnoho páteří nebylo navrženo pro zpracování zátěží v provozní špičce, kterých jsou schopné moderní vysokorychlostní mobilní sítě, a svým zahlcováním často demonstrují velkou rozdílnost v zpoždění a propustnosti. Operátoři se snaží co nejrychleji aktualizovat sítě, ale tato komponenta zůstává slabým místem mnoha síťových infrastruktur.

Řadič rádiové sítě

Řadiče rádiové sítě standardně řídí sousední místa buňky a mobilní zařízení, kterým poskytují služby.

Řadič rádiové sítě přímo koordinuje činnosti mobilních zařízení použitím schématu řízení, založeném na zprávách, známé jako signalizování. V důsledku topologie přenosu zpráv mezi

mobilním zařízením a řadičem musí směrovat přes páteřní síť, čímž dochází k velkému zpoždění. To samotné není ideální, ale zhoršujete se to faktem, že mnoho síťových operací, jako síťová registrace a plánování přenosu, vyžaduje vícenásobné obousměrné výměny zpráv. Řadič rádiové sítě byl z tohoto důvodu klasicky primárním přispěvatelem zpoždění.

Jak již bylo zmíněno, v nejnovější generaci mobilních sítí je řadič zproštěn povinností řízení zařízení, přičemž mnoho z těchto úkolů v současné době zpracovávají přímo samotné buňky. Toto designové rozhodnutí eliminuje faktor zpoždění páteřního zpoždění z mnoha síťových funkcí.

Jádro sítě

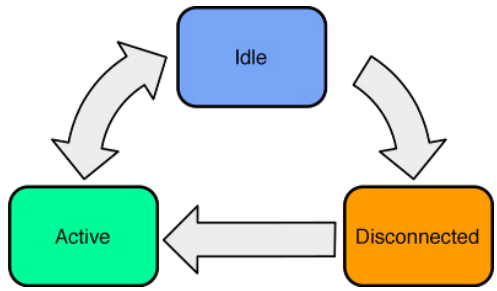
Jádro sítě slouží jako brána mezi privátní sítí operátora a veřejným internetem. Je to právě tady, kde operátoři využívají sériově zapojených síťových zařízení k prosazení jejich politiky kvality služby nebo měření šířky pásma. Pravidlem je, že zachycení síťového provozu znamená zpoždění. V praxi je toto zpoždění obvykle zanedbatelné, ale jeho přítomnost by měla být poznat.

Šetření energií

Jedno z nejvýznamnějších zdrojů zpoždění mobilní sítě je přímo spojeno s omezenou kapacitou baterie mobilních telefonů.

Rádiový přijímač mobilního vysokorychlostního zařízení může za provozu spotřebovat více než 3 Watty energie. Toto číslo je dostatečně velké na to, aby se baterie zařízení iPhone 5 vybila za něco málo více než hodinu. Z tohoto důvodu mobilní zařízení při každé příležitosti vypínají nebo snižují energii do rádiových obvodů. To je ideální pro prodloužení životnosti baterie, ale také to zavádí spouštěcí zpoždění kdykoliv, když jsou znovu zapojena pro odesílání nebo přijímání dat.

Všechny normy pro mobilní celulární sítě formalizovaly schéma řízení rádiových zdrojů (RRM) tak, aby šetřily energii. Většina RRM konvence definuje tři stavy - aktivní, nečinný a odpojený – tak, že každý představuje určitý kompromis mezi spouštěcí latencí a spotřebou energie.



Obrázek 10.2: Přechody stavů řízení rádiových zdrojů

Aktivní
Aktivní představuje stav, kdy lze odesílat a přijímat data při vysoké rychlosti s minimálním zpožděním.

Tento stav spotřebuje velké množství energie i při nečinnosti. Krátké období síťové nečinnosti, často méně než sekunda, spouští přechod do stavu nečinný s nižší energií. Je důležité uvědomit si výkonové důsledky: dostatečně dlouhé pauzy během síťové transakce můžou vyvolat další zpoždění, protože zařízení kolísá mezi aktivními a nečinnými stavy.

Nečinný
Nečinný je kompromisem mezi nižší spotřebou energie a mírným spouštěcím zpožděním.

Zařízení zůstává připojeno k síti, není schopno přenášet nebo přijímat data, ale je schopno přijímat síťové požadavky vyžadující k jejich splnění *aktivní* stav (např. příchozí data). Po přiměřené době nečinnosti sítě, obvykle minutu nebo méně, přístroj přechází do *odpojeného* stavu.

Nečinný přispívá ke zpoždění dvěma způsoby. Za prvé vyžaduje určitý čas na opětovné připojení a synchronizaci analogových obvodů. Za druhé, ve snaze ušetřit ještě více energie, naslouchá rádio jen přerušovaně a mírně zpožďuje odpovědi na příchozí zprávy.

Odpojený
Odpojený má nejnižší spotřebu energie s největším spouštěcím zpožděním.

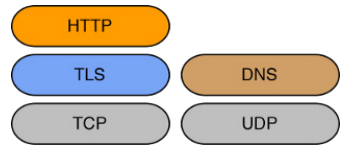
Přístroj je odpojen od mobilní sítě a rádio je deaktivováno. Rádio je aktivováno jen zřídka pro naslouchání síťových požadavků přicházejících přes speciální vysílací kanál.

Odpojený sdílí stejné zdroje zpoždění jako *nečinný* plus další zpoždění na síťové znovupřipojení. Připojení k mobilní síti je složitý proces, který zahrnuje několik kol výměny zpráv (tj. signalizování). Obnovení připojení bude trvat minimálně stovky milisekund, a tak není neobvyklé vidět čas na připojení v sekundách.

10.4 Výkon síťového protokolu

Teď pojďme tam, kde ve skutečnosti máme určitou kontrolu.

Výkonnost síťových transakcí je neúměrně ovlivněna vysokými hodnotami RTT. To je vzhledem k obousměrné výměně zpráv podstatné pro provoz většiny síťových protokolů. Zbývající část této kapitoly se zaměřuje na pochopení, proč se vyskytují tyto výměny zpráv a jak se může snížit nebo dokonce eliminovat jejich četnost.



Obrázek 10.3: Síťové protokoly

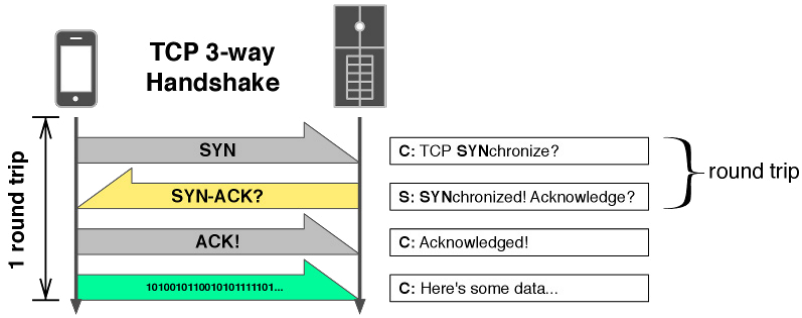
10.5 Protokol pro řízení transportu

TCP představuje velkou část obousměrných zpráv, kterým se snažíme vyhnout. Některé mohou být eliminovány přijetím rozšíření protokolu jako Fast Open. Jiné mohou být minimalizovány parametry ladění systému, jako počáteční kongesční okno. V této části budeme zkoumat oba tyto přístupy a zároveň poskytneme pozadí vnitřností TCP.

TCP Fast Open

Iniciování TCP spojení zahrnuje konvenci pro výměnu zpráv známou jako tříkrokové navázání spojení. TCP Fast Open (TFO) je rozšířením protokolu TCP, který eliminuje obousměrné zpoždění obvykle způsobené procesem navázání spojení.

TCP tříkrokové vyjednávání provozních parametrů mezi klientem a serverem umožní robustní 2 stranou komunikaci. Počáteční SYN (synchronize) zpráva představuje požadavek na připojení klienta. Za předpokladu, že server přijímá pokus o spojení, odpoví zprávami SYN-ACK (synchronize a acknowledge). Na závěr klient rozpozná server zprávou ACK. V tomto bodě bylo vytvořeno logické spojení a klient může zahájit odesílání dat. Pokud jste udrželi pozornost, všimli jste si, že třícestné navázaní spojení zavádí zpoždění odpovídající aktuálnímu RTT.



Obrázek 10.4: 3cestné navázání spojení TCP

Tradičně neexistoval žádný způsob, mimo recyklace připojení, jak zabránit zpoždění TCP tříkrokového navázaní spojení. Nicméně to se nedávno změnilo zavedením specifikace TCP Fast Open IE TF.

TCP Fast Open (TFO) umožňuje klientovi začít posílat data před vytvořením logického připojení. Toto efektivně neguje jakékoliv obousměrné zpoždění z tříkrokového navázání spojení. Kumulativní efekt této optimalizace je impozantní. Podle výzkumu Google TFO může snížit dobu načítání stránky až o 40 %. I když je to stále jen návrh specifikace, TFO je již podporován hlavními prohlížeči (Chrome 22+) a platformami (Linux 3.6+), jiní dodavatelé se zavázali, že ji brzy plně podpoří.

TCP Fast Open je modifikace tříkrokového navázání spojení umožňující přenést malou datovou zprávu (například požadavek HTTP), umístěnou v rámci zprávy SYN. Tato malá zpráva je předána do aplikačního serveru po navázání spojení standardním způsobem.

Dřívější návrhy na rozšíření, jako je TFO, se nakonec nezdařily kvůli bezpečnostním obavám. TFO řeší tento problém nápadem bezpečného tokenu anebo cookie, který je přidělen klientovi v průběhu běžného TCP navázání spojení, a očekává, že budou zahrnuty do SYN zprávy TFO optimalizovaného požadavku.

Existují některé drobné výhrady k používání TFO. Nejdůležitější je absence jakýchkoli záruk idempotence pro data požadavku, dodávaná se zahajovacím SYN zprávy. TCP zajišťuje, že jsou duplicitní pakety (duplikace tává často) ignorovány přijímačem, ale to samé ujištění se nevztahuje na navázání spojení. Pokračují snahy standardizovat řešení v návrhu specifikace, ale TFO může být prozatím bezpečně nasazeno pro idempotentní transakce.

Počáteční kongesční okno

Počáteční kongesční okno (initcwnd) je konfigurovatelné TCP nastavení s velkým potenciálem pro zrychlení menších síťových transakcí.

Nedávná IETF specifikace propaguje zvyšování běžného nastavení počátečního kongesčního okna z 3 segmentů (tj. paketů) na 10. Tento návrh vychází z rozsáhlého průzkumu provedeného společností Google, která prokázala průměrné výkonnostní zvýšení o 10 %. Účel a potenciální dopad tohoto nastavení nelze ocenit bez zavedení TCP kongesčního okna (cwnd).

TCP garantuje provozní spolehlivost klientovi a serveru i přes jinak nespolehlivé sítě. Jedná se o slib toho, že všechny zprávy budou obdrženy v tom pořadí, v jakém byly odeslány. Největší překážkou pro splnění tohoto cíle je ztráta paketů; to vyžaduje detekci, korekci a prevenci.

TCP využívá konvenci pozitivního potvrzení pro detekci chybějících paketů, v rámci které by měl být každý poslaný paket potvrzen jeho zamýšleným příjemcem. Chybějící potvrzení implikuje ztrátu paketu v průběhu přenosu. Zatímco se čeká na potvrzení, vysílané pakety jsou zachovány

ve speciální mezipaměti o velikosti kongesčního okna. Když se mezipaměť zaplní, událost známá jako vyčerpání cwnd zastaví všechny přenosy, až dokud potvrzení od příjemce neumožní mezipaměť vyprázdnit a použít uvolněný prostor pro odeslání více paketů. Tyto události hrají významnou roli ve výkonu TCP.

Kromě limitů šířky pásma sítě jeTCP propustnost omezena hlavně frekvencí událostí vyčerpání cwnd, pravděpodobností, která se vztahuje k velikosti kongesčního okna. Dosažení špičkového výkonu TCP vyžaduje kongesční okno odpovídající stávajícím síťovým podmínkám: je-li příliš velké, existuje riziko zahlcení sítě - přeplňující podmínka poznamenaná rozsáhlou ztrátou paketů; je-li příliš malé, drahocenná šířka pásma zůstane nevyužita. Je logické, že čím více jsou známy síťové podmínky, tím větší je pravděpodobnost výběru optimálního kongesčního okna. Skutečností je, že klíčové atributy sítě, jako je kapacita a zpoždění, jsou těžce měřitelné a jsou neustále v pohybu. Ještě více komplikuje situaci to, že jakékoli TCP spojení na bázi Internetu probíhá po celé řadě sítí.

TCP vyvozuje kapacitu ze stavu přetížení sítě. TCP otevře kongesční okno jen do té míry, než dojde ke ztrátě paketů, což naznačuje, že někde na cestě síť není schopna zvládnout aktuální přenosovou rychlost. Použitím tohoto schématu pro zamezení přetížení TCP minimalizuje událost vyčerpání cwnd do té míry, že spotřebovává veškerou přidělenou kapacitu připojení. A nyní konečně docházíme ke smyslu a významu počátečního nastavení TCP kongesčního okna.

Přetížení sítě nelze zjistit bez detekce ztráty paketů. Nové nebo nečinné připojení postrádá důkazy o ztrátách paketů potřebných pro vytvoření optimální velikost kongesčního okna. TCP přijímá skutečnost, že je lepší začít s kongesčním oknem s nejmenší pravděpodobností vytvoření zahlcení; to původně znamenalo nastavení 1 segmentu (~ 1 480 bajtů) a nějakou dobu to bylo doporučováno. Později experimenty prokázaly, že by mohlo být efektivní nastavení až 4 segmentů. V praxi obvykle najdete nastavení počátečního kogesčního okna na hodnotu 3 segmentů (~ 4 KiB).

Počáteční kongesční okno má negativní vliv na rychlost malých síťových transakcí. Tento efekt je snadno ilustrovatelný. Při standardním nastavení 3 segmentů by vyčerpání cwnd nastalo po zaslání pouhých 3 paketů, nebo 4 KiB. Za předpokladu, že pakety byly vyslány spojitě, příslušné potvrzení nedorazí dřív, než to umožní RTT připojení. Např. v případě, že RTT by byla 100 ms, efektivní rychlost přenosu by byla žalostných 400 bajtů/sekundu. Ačkoli TCP nakonec rozšíří jeho kongesční okno tak, aby plně spotřebovával dostupnou kapacitu, nastane velmi pomalý start. Faktem je, že tato konvence je známá jako pomalý start.

Pomalý start má dopad na výkonnost menších stažení do té míry, že vyžaduje přehodnocení poměru rizika a zisku původního návrhu kongesčního okna. Google právě toto udělal a zjistil, že počáteční kongesční okno o 10 segmentech (~ 14 KiB) vygeneruje největší propustnost při nejmenším přetížení. Reálné výsledky ukazují 10% snížení doby načítání stránky. Připojení se zvýšeným zpožděním obousměrné cesty dosahují ještě lepších výsledků.

Úprava počátečního kongesčního okna z jeho výchozí hodnoty není jednoduchá. Ve většině serverových operačních systémech je nastavení celého systému konfigurovatelné jenom privilegovanými uživateli. Výjimečně může být toto nastavení konfigurovatelné v klientském počítači pomocí neprivilegovaných aplikací, nebo dokonce vůbec. Je důležité si uvědomit, že větší počáteční kongesční okno na straně serveru urychluje stahování, zatímco na straně klienta zrychluje nahrávání. Nemožnost změny tohoto nastavení na straně klienta znamená speciální úsilí, které by se mělo věnovat minimalizaci velikosti požadavku.

10.6 Protokol pro transfer hypertextu

Tato část diskutuje techniky pro snížení efektu latencí obousměrné cesty na výkon Hypertext Transfer Protocol (HTTP).

Keepalive

Keepalive (udrž naživu) je http konvence povolující použít stejné TCP připojení pro několik, sekvenčně odeslaných požadavků. Zamezením minimálně jedné obousměrné cesty, vyžadující TCP tříkrokové navázání spojení, ušetří desítky nebo stovky milisekund na jeden požadavek. Dále má keepalive další, a často málo známou, výkonnostní výhodu. Zachovává stávající TCP kongesční okno mezi požadavky, což má za následek mnohem méně událostí vyčerpání cwnd.

Zasílání zpráv s ohledem na CWND

Při přenášení zprávy mohou HTTP často narazit na rušivá a matoucí zpoždění způsobená TCP událostí vyčerpání cwnd. Významné doby nečinnosti mezi zprávami, obvykle o něco větší než sekunda, přinutí TCP resetovat kongesční okno.

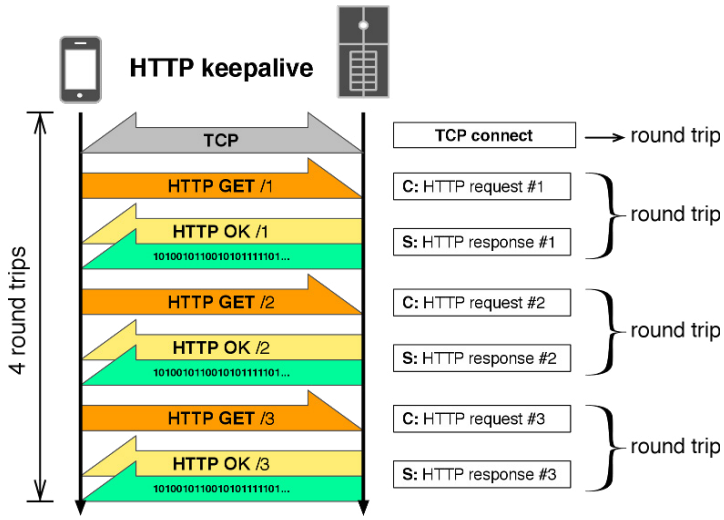
Udržování velikosti zprávy pod počátečním nastavením kongesčního okna - obvykle asi tři segmenty, nebo přibližně 4KiB, může zabránit cwnd vyčerpání. Představíme 2 techniky - redukce záhlaví a delta kódování - které mohou zabránit zprávám překročení tohoto prahu.

Redukce hlavičky

Možná je pro někoho překvapivé, že mnoho typů HTTP požadavků nejsou formálně povinny uvést jakékoliv hlavičku. To může ušetřit hodně místa. Je dobrým pravidlem začít od nulových hlaviček a zahrnout jen to, co je nutné. Mějte se na pozoru u všech hlaviček automaticky připojených k HTTP klienta nebo serveru. Pro některé konfigurace může být nutné toto chování zakázat.

Delta kódování

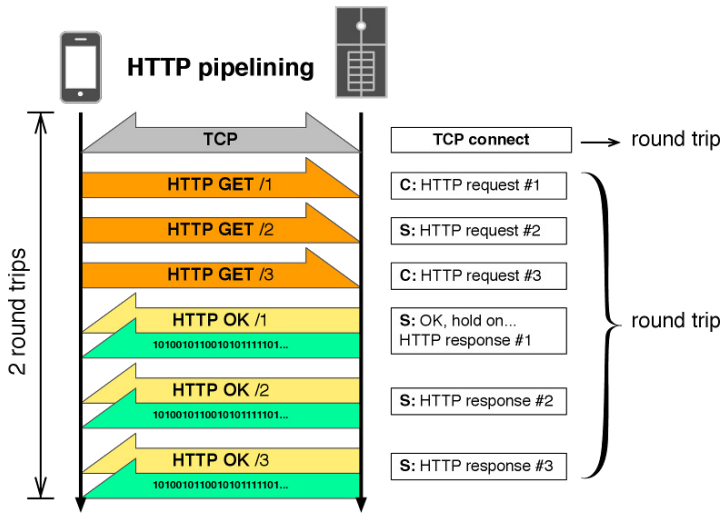
Delta kódování je kompresní technika, která využívá podobnosti po sobě jdoucích zpráv. Delta kódovaná zpráva je reprezentována pouze svými rozdíly od předchozí. Tato technika se obzvláště dobře hodí pro JSON formátovanou zprávu s konzistentním formátováním.



Obrázek 10.5: HTTP keepalive

Pipelining

Pipelining (zřetězení) je HTTP konvence pro přenos více sekvenčních požadavků v rámci jedné transakce. To má výkonnostní výhody HTTP keepalive, a zároveň eliminuje obousměrné cesty typicky vyžadující dodatečné HTTP požadavky.

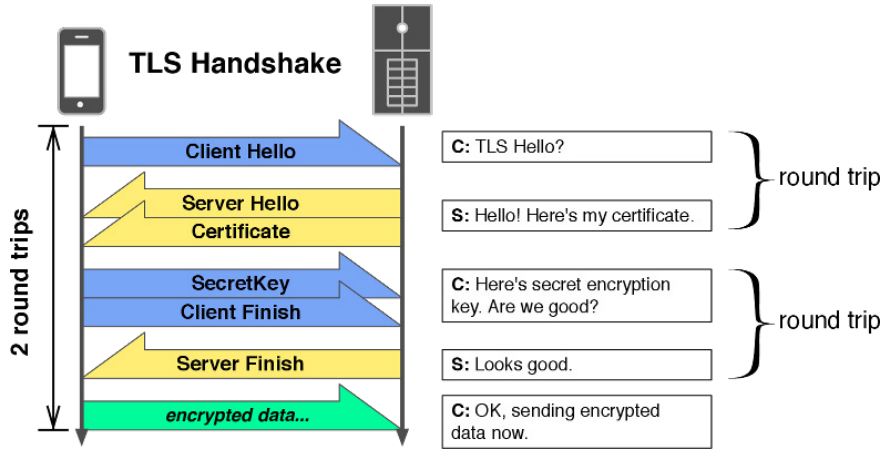


Obrázek 10.6: HTTP pipelining

Pipelining efektivně distribuuje zpoždění obousměrné síťové cesty mezi více HTTP transakcí. Například 5 zřetězených HTTP požadavků přes připojení s RTT 100 ms vyvolá průměrné zpoždění obousměrné cesty 20 ms. Za stejných podmínek sníží 10 zřetězených HTTP požadavků průměrné zpoždění na 10 ms.

10.7 Bezpečnostní transportní vrstva

TLS využívá složité navázání spojení zahrnující dvě výměny klient-server zpráv. TLS-zabezpečené HTTP transakce se můžou jevit z tohoto důvodu znatelně pomalejší. Časté postřehy, že je TLS pomalý, jsou ve skutečnosti stížností na vícenásobná zpoždění obousměrné cesty způsobená protokolem pro navázání spojení.

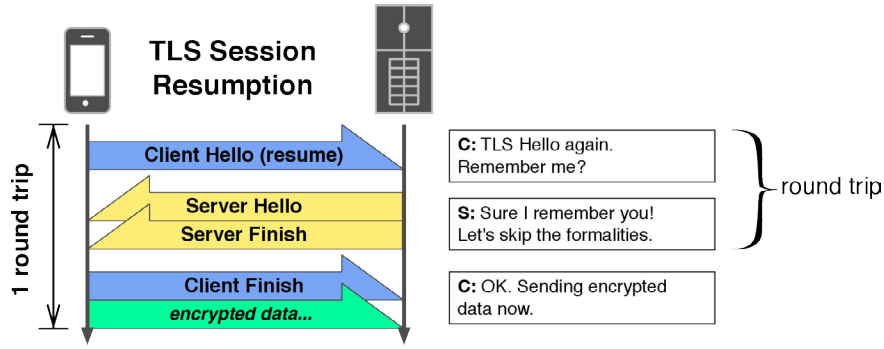


Obrázek 10.7: TLS sekvence navázání spojení

Dobrá zpráva: veškerá technika, která zachovává TCP spojení mezi transakcemi, jako http keepalive konvence, také zachovává relaci TLS. Nicméně, není vždy praktické dlouhodobě udržovat zabezpečené TCP připojení. Nabízejí se zde dvě metody, které urychlují samotné navázání TLS spojení.

Znovuzahájení relace

Funkce TLS *session resumption* (znovuzahájení relace) umožňuje zachování zabezpečené relace mezi TCP spojeními. Znovuzahájení relace odstraňuje počáteční výměnu zpráv pro navázání spojení vyhrazenou pro šifrování veřejným klíčem, který ověřuje identitu serveru a zavádí symetrický šifrovací klíč. Zatímco zamezením výpočetně náročných veřejných kryptografických operací vznikne nějaký výkonový zisk, větší úspora času se získá eliminací zpoždění obousměrné cesty na jednu výměnu zpráv.

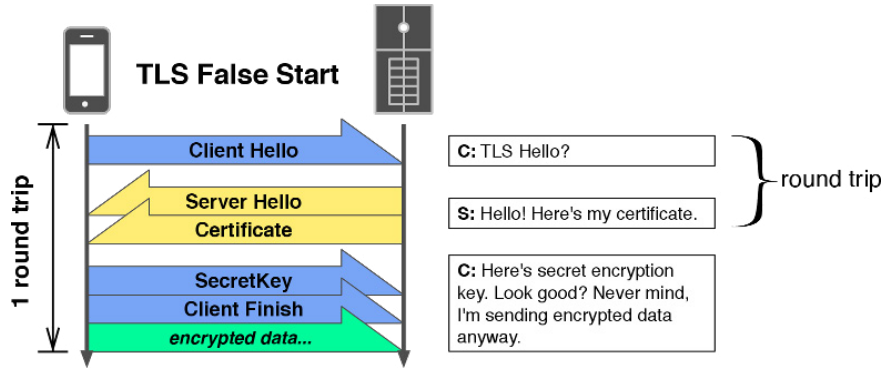


Obrázek 10.8: TLS znovuzahájení relace

Dřívější revize TLS (tj. SSL) závisela na zachování stavu relace serverem, což představovalo skutečnou výzvu pro vysoce distribuované serverové architektury. TLS *session tickets* (vstupenky do relace) nabízí mnohem jednodušší řešení. Toto rozšíření umožňuje klientovi zachování stavu relace v podobě zašifrovaného obsahu (tj. vstupenka do relace), poskytnutého serverem během procesu navázaní spojení. Pokračování relace vyžaduje, aby klient předložil tuto vstupenku na začátku navázání spojení.

Předčasný start

False start (předčasný start) je modifikací protokolu pocházející z chytrého pozorování navazování TLS spojení: technicky může klient začít posílat zašifrované údaje okamžitě po předání jeho konečné zprávy o navázání spojení na server. Na základě tohoto zjištění eliminuje předčasný start zpoždění obousměrné cesty běžně se vyskytující čekáním klienta na konečnou zprávu serveru o navázání spojení.

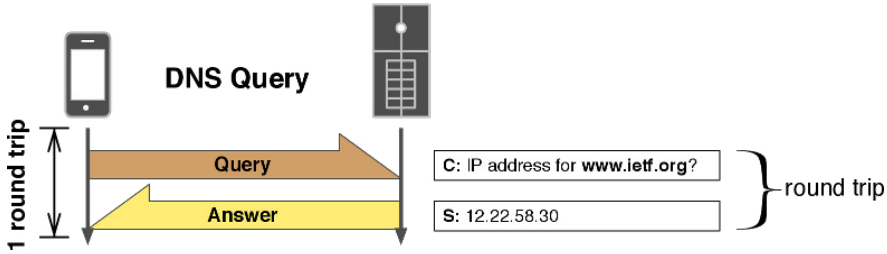


Obrázek 10.9: TLS předčasný start

Předčasný start vykazuje stejný výkonnostní přínos jako znovuzahájení relace s další výhodou, a to být bez stavu - klient a server jsou zbaveni břemena řízení stavu relace. Většina webových klientů podporuje předčasný start s jenom drobnými změnami. A kupodivu, asi v 99 % případů nevyžaduje podpora serveru vůbec žádné změny, co umožňuje okamžitě nasadit tuto optimalizaci na většině infrastruktury.

10.8 DNS

Domain Name System (DNS, systém doménových jmen) poskytuje převod doménového jména na IP adresu potřebnou pro síťové transakce založené na protokolu IP. Protokol DNS je poměrně jednoduchá záležitost, obvykle funguje bez nutnosti spolehlivého přenosového protokolu (pomocí UDP). Bez ohledu na to, DNS dotazy často vykazují velké a extrémně se odlišující doby odezvy (z důvodů přílišné složitosti a početnosti zde nebudeme rozvíjet).



Obrázek 10.10: DNS dotaz

Hostingová platforma obecně nabízí implementaci vyrovnávací mezipaměti pro zabránění častým DNS dotazům. Sémantiky ukládání odpovědí na DNS dotazy do mezipaměti jsou jednoduché. Každá odpověď DNS obsahuje dobu platnosti (TTL) atributu, deklarující, jak dlouho může být výsledek uložen. TTL můžou být v rozmezí od sekund až po dny, ale typicky jsou v řádu několika minut. Velmi nízké hodnoty TTL, obvykle méně než jedna minuta, se používají k ovlivnění rozdělení zátěže nebo minimalizace prostoje na výměnu serveru nebo selhání ISP.

U většiny platforem neberou nativní implementace DNS vyrovnávací mezipaměti v úvahu zvýšené RTT mobilních sítí. Mnoho mobilních aplikací by mohlo těžit ze sofistikovanější implementace mezipaměti. Zde je návrh několika strategií k ukládání do mezipaměti, které pokud se nasadí k aplikačnímu použití, eliminují náhodné a rušivé zpoždění způsobené zbytečnými DNS dotazy.

Obnova po selhání

Vysoce dostupné systémy obvykle spoléhají na redundantní infrastrukturu hostovanou v rámci jejich IP adresního prostoru. Položky Low-TTL DNS přispívají ke zkrácení doby, kdy se síťový klient může odkazovat na adresu hostitele, který selhal, ale zároveň vyvolávají spoustu dalších DNS dotazů. TTL je kompromisem mezi minimalizací prostoje a maximalizací výkonu klienta.

Obecně nemá smysl snižovat výkon klienta, když jsou selhání serveru výjimkou z pravidla. Existuje jednoduché řešení tohoto dilema, než se striktně podřizovat TTL.V mezipaměti uložený DNS záznam je pouze aktualizován, když je zjištěna nevratná chyba na vyšší úrovni protokolu, jako je TCP nebo HTTP. Ve většině scénářů tato technika emuluje chování TTL-konformní DNS mezipaměti, zatímco téměř eliminuje výkonnostní postihy, které jsou za normálních okolností spojené s jakýmkoliv řešením s vysokou dostupností na bázi DNS.

Je třeba poznamenat, že tato technika by pravděpodobně byla nekompatibilní s jakýmkoliv schématem rozdělení zatížení systému na bázi DNS.

Asynchronní obnova

Asynchronní obnova je přístup k DNS ukládání do mezipaměti, který se (většinou) podřizuje poslaným TTL a zároveň do značné míry eliminuje zpoždění častých dotazů DNS. Pro implementaci této techniky je potřebná asynchronní DNS knihovna klienta, jako je c-ares.

Myšlenka je jednoduchá. Požadavek na expirovanou DNS položku v mezipaměti vrátí neaktuální výsledek, zatímco neblokující DNS dotaz je plánován na pozadí k aktualizaci mezipaměti.

Když se implementuje se zálohou pro blokující (tj. synchronní) dotazy pro velmi zastaralé položky, tato technika je téměř imunní vůči zpoždění DNS a zároveň zůstává kompatibilní s mnoha schématy pro selhání a rozložení zátěže na bázi DNS.

10.9 Závěr

Zmírnění dopadů zvýšeného zpoždění mobilních sítí vyžaduje redukování obousměrných cest sítí, které zhoršují efekt zpoždění. Využívaní softwarových optimalizací zaměřených výhradně na minimalizaci nebo eliminaci obousměrného zasílání protokolů má zásadní význam pro překonání tohoto nelehkého výkonového problému.

11 Warp

(Kazu Yamamoto, Michael Snoyman a Andreas Voellmy)

11 Warp

Warp je vysoce výkonná knihovna pro HTTP server napsaná v jazyce Haskell, který je čistě funkcionálním programovacím jazykem. Jak Yesod, webový aplikační framework, tak mighty, HTTP server, jsou implementovány nad Warpem. Podle našich srovnávacích testů propustnosti, mighty poskytuje výkon na stejné úrovni jako nginx. Tento článek vám vysvětlí architekturu Warpu a to, jak jsme dosáhli jeho výkonnosti. Warp může běžet na mnoha platformách, včetně Linuxu, variant BSD, Mac OS a Windows. Pro zbývající část tohoto článku budeme kvůli zjednodušení našeho vysvětlování mluvit jen o Linuxu.

11.1 Síťové programování v Haskellu

Někteří lidé se domnívají, že funkční programovací jazyky jsou pomalé nebo nepraktické. Nicméně, dle našeho nejlepšího vědomí, Haskell poskytuje téměř ideální přístup k síťovému programování.

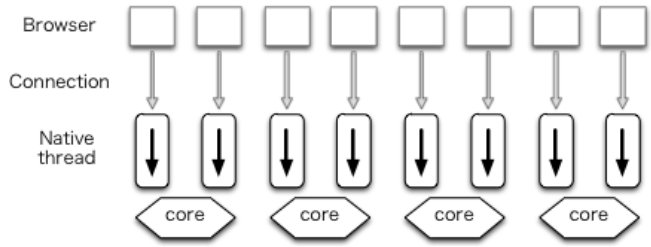
Důvodem je, že Glasgow Haskell Compiler (GHC), vlajková loď mezi Haskell kompilátory, poskytuje lehké a robustní uživatelská vlákna (někdy nazývané zelená vlákna). V této části krátce přezkoumáme některé dobře známé přístupy k síťovému programování na straně serveru a porovnáme je se síťovým programováním v Haskellu. Ukážeme, že Haskell nabízí kombinaci programovatelnosti a výkonnosti, která není k dispozici v jiných přístupech: Vyhovující abstrakce Haskellu umožňují programátorům psát jasný, jednoduchý kód, zatímco sofistikovaný kompilátor Haskell a vícejádrový běhový systém produkují vícejádrové programy, které se vykonávají způsobem velmi podobným nejpokročilejším ručně zhotoveným síťovým programům.

Nativní vlákna

Tradiční servery používají techniku zvanou vláknové programování. V této architektuře je každé spojení zpracováno jedním procesem nebo nativním vláknem (někdy nazývané OS vlákno).

Tato architektura může být dále členěna na základě mechanismu použitého pro vytvoření procesu nebo nativního vlákna. Při použití zásobníku vláken je předem vytvořeno více procesů nebo nativních vláken. Příkladem toho je režim prefork (vytvoření zásobníku vláken/procesů předem) v Apache. Jinak je proces nebo nativní vlákno vytvořeno pokaždé, když se naváže spojení. Toto ukazuje Obrázek 11.1.

Výhodou této architektury je, že umožňuje psát vývojářům jasný kód. Zejména použití vláken umožňuje, aby kód sledoval jednoduchý a známý řídící tok a pouze jednoduché volání procedur pro načtení vstupu nebo odeslání výstupu. Také proto, že jádro operačního systému přiřadí procesům nebo nativním vláknům dostupná jádra, můžeme vyvážit zátížení všech procesorových jader.

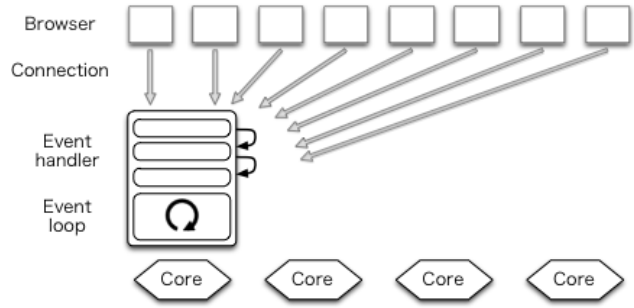


Obrázek 11.1: Nativní vlákna

Jeho nevýhodou je, že nastává velký počet přepínání kontextu mezi jádrem operačního systému a procesy nebo nativními vlákny, což vede k degradaci výkonu.

Událostmi řízená architektura

Ve světě vysoce výkonných serverů bylo programování řízené událostmi nedávným trendem. V této architektuře je více připojení řízeno jediným procesem (Obrázek 11.2). Lighttpd je příkladem webového serveru používajícího tuto architekturu.



Obrázek 11.2: Událostmi řízená architektura

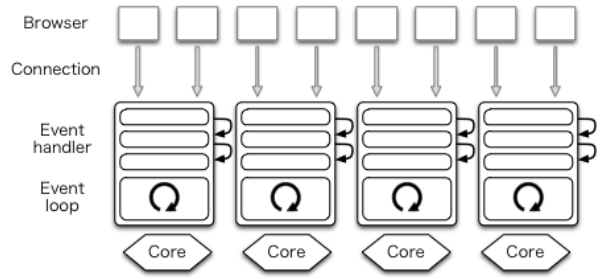
Vzhledem k tomu, že není třeba přepínání procesů, dochází k méně přepínáním kontextu, a výkon je tím vylepšen. To je jeho hlavní výhodou.

Na druhou stranu, tato architektura podstatně komplikuje síťový program. Zejména obrací tok řízení tak, aby událostní smyčka řídila celkové provádění programu. Programátoři proto musí restrukturalizovat svůj program do událostních rutin, z nichž každá provádí pouze neblokující kód. Toto omezení brání programátorům vykonávat I/O operace pomocí volání procedur; místo toho musí být použity složitější asynchronní metody. Podle stejných pravidel, konvenční metody pro zpracování výjimek již nejsou použitelná.

Jeden proces na jedno jádro

Mnozí přišli s nápadem vytvořit n událostmi řízených procesů, které využívají n jader (Obrázek 11.3). Každý proces se nazývá dělník. Mezi dělníky musí být sdílen servisní port. Sdílení portu může být dosaženo pomocí prefork techniky.

V tradičním procesu programování je proces pro nové spojení spuštěn po navázání tohoto spojení. Naproti tomu prefork technika spouští procesy ještě před navázáním nových spojení. I když mají stejný název, tato technika by neměla být zaměňována s režimem prefork serveru Apache.



Obrázek 11.3: Jeden proces na jedno jádro

Webový server, který používá tuto architekturu, je nginx. Node.js používal událostmi řízenou architekturu v minulosti, ale v poslední době také implementoval prefork techniku. Výhodou této architektury je to, že využívá všech jader a zvyšuje výkon. Nicméně, nevyřeší problém programů, které mají špatnou srozumitelnost v důsledku spoléhání se na funkce obsluh a zpětného volání.

Uživatelské vlákna

Problém srozumitelnosti kódu může být vyřešen pomocí uživatelských vláken GHC. Zejména můžeme zpracovat každé HTTP připojení v novém uživatelském vláknu. Toto vlákno je naprogramováno v tradičním stylu použitím logicky blokujících I/O volání. To udržuje program jasný a jednoduchý, zatímco GHC zpracovává složitosti neblokujících I/O a vícejádrovou práci řízení.

Uvnitř GHC multiplexuje uživatelská vlákna přes malý počet nativních vláken. Běhový systém GHC obsahuje vícejádrový plánovač vláken, který může levně přepínat mezi uživatelskými vlákny, protože to dělá bez účasti přepínání kontextu operačním systémem.

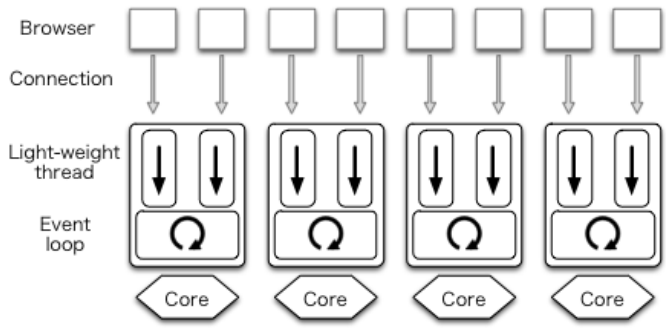
Uživatelská vlákna GHC jsou lehké; na moderních počítačích může běžet 100 000 uživatelských vláken. Jsou robustní; dokonce jsou zachyceny i asynchronní výjimky (tato funkce se používá obsluhou časového limitu a je popsána v části 11.2 a v části 11.7). Kromě toho plánovač obsahuje algoritmus vícejádrového vyrovnávání zátěže pro lepší využití kapacity všech dostupných jader.

Když uživatelské vlákno provádí logicky blokující I/O operaci, jako je příjem nebo odesílání dat do soketu, vlastně se pokouší o neblokující volání. Pokud se to podaří, vlákno pokračuje ihned bez zapojení I/O manažera nebo plánovače vláken. Pokud by volání blokovalo, vlákno místo toho zaregistruje zájem o relevantní událost s I/O manažera běhového systému a poté indikuje plánovači, že čeká. Vlákno I/O manažeru nezávisle sleduje události a notifikuje vlákna, kdy nastane jejich událost, a přiměje je, aby byly přeplánovány pro vykonání. To vše se děje transparentně v uživatelském vlákně, bez jakéhokoliv úsilí ze strany Haskell programátora.

V Haskellu je většina výpočtů nedestruktivních. To znamená, že téměř všechny funkce jsou bezpečné pro vlákna. GHC používá alokaci dat jako bezpečný bod pro přepnutí kontextu uživatelských vláken. Vzhledem k funkčnímu programovacímu stylu jsou často vytvořena nová data a je známo, že tato alokace dat se vyskytuje dostatečně pravidelně pro přepínání kontextu.

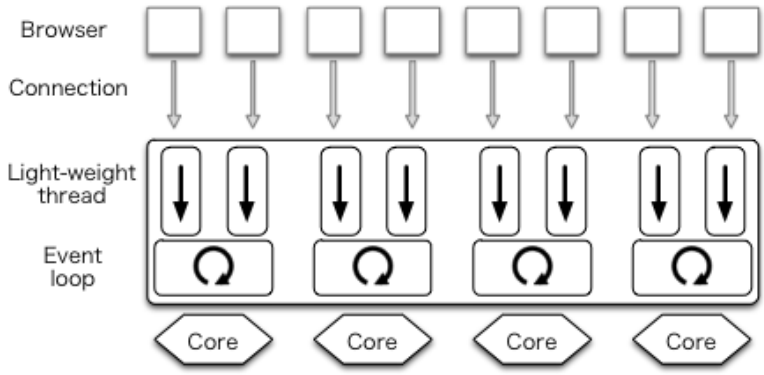
Ačkoli některé jazyky v minulosti poskytovaly uživatelská vlákna, nyní nejsou běžně používané, protože nebyly lehké nebo robustní. Všimněte si, že některé jazyky poskytují korutiny na úrovni knihovny, ale nejsou to preemptivní vlákna. A naopak si všimněte, že Erlang a Go poskytují lehké procesy a lehké korutiny.

V době psaní tohoto textu mighty používal prefork techniku pro větvení procesů za účelem využití více jader. (Warp tuto funkci nemá.) Obrázek 11.4 znázorňuje toto uspořádání v rámci webového serveru s prefork technikou napsáného v jazyce Haskell, kde každé připojení prohlížeče je zpracováno jedním uživatelským vláknem, a jedno nativní vlákno procesu běžící na CPU jádre zpracovává práci z několika připojení.



Obrázek 11.4: Uživatelská vlákna s jedním procesem na jedno jádro

Zjistili jsme, že samotná komponenta I/O manažer, která je součástí GHC běhového systému, má výkonová úzká místa. K vyřešení tohoto problému jsme vyvinuli paralelní I/O manažer, který používá registrační tabulku událostí na jedno jádro a monitory událostí pro výrazné vylepšení vícejádrového škálování. Program Haskell s paralelním I/O manažerem je vykonáván jako jediný proces a více I/O manažerů běží jako nativní vlákna pro využití více jader (Obrázek 11.5). Každé uživatelské vlákno se vykonává na některém z jader.



Obrázek 11.5: Uživatelské vlákna v jednom procesu

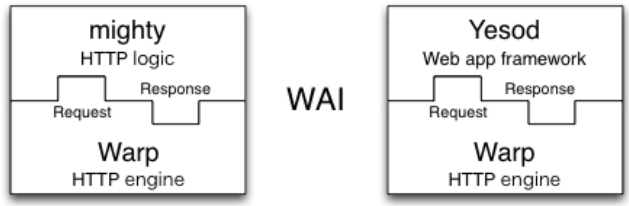
GHC verze 7.8, která obsahuje paralelní I/O Manažer, bude vydána na podzim roku 2013. Samotný Warp s GHC verzí 7.8 bude moci používat tuto architekturu bez jakýchkoliv modifikací a mighty nebude muset používat prefork techniku.

11.2 Architektura Warpu

Warp je HTTP jádrem pro webové aplikační rozhraní (WAI, Web Application Interface). WAI aplikace běží přes HTTP. Jak jsme popsali výše, jak Yesod tak mighty jsou příklady WAI aplikací, jak je znázorněno na Obrázku 11.6.

Typ WAI aplikace je následující:

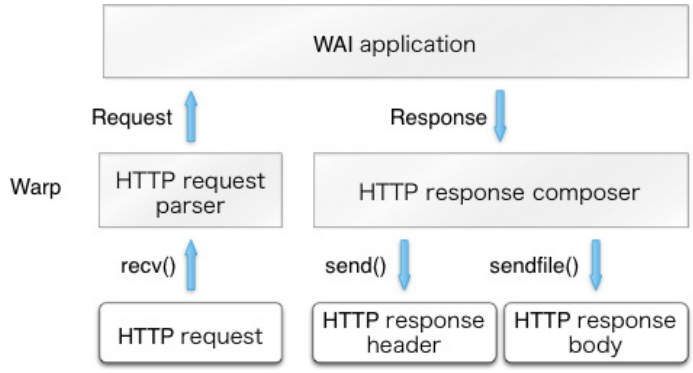
```
type Application = Request -> ResourceT IO Response
```



Obrázek 11.6: Webové aplikační rozhraní (WAI)

V Haskellu jsou typy argumentů funkcí odděleny šipkami vpravo a zcela vpravo je typ návratové hodnoty. Takže můžeme interpretovat definici jako: a WAI Application vezme Request a vrátí Response, který se používá v kontextu, v němž I/O je možný a zdroje jsou dobře řízené.

Po přijetí nového HTTP připojení je pro toto připojení vytvořeno vyhrazené uživatelské vlákno. Nejdříve obdrží HTTP požadavek od klienta a parsuje ho do Request. Poté Warp dá Request WAI aplikaci a obdrží od něj Response. Nakonec Warp vytvoří HTTP odpověď na základě hodnoty Response a pošle ji zpět klientovi. To je znázorněno na Obrázku 11.7.



Obrázek 11.7: Architektura Warpu

Uživatelské vlákno opakuje tento postup podle potřeby a ukončí se, když je připojení uzavřeno protějškem nebo je přijat neplatný požadavek. Vlákno se také ukončí v případě, že není přijato významné množství dat po určité časové období (tj. vypršel časový limit).

11.3 Výkonnost Warpu

Než vysvětlíme, jak zlepšit výkonnost Warpu, rádi bychom ukazali výsledky našeho srovnávacího testu. Měřili jsme propustnost mighty verze 2.8.4 (s Warp verzí 1.3.8.1) a nginx verzí 1.4.0. Naše testovací prostředí je následující:

- Dva „12 jádrové" počítače (Intel Xeon E5645, dva sokety, 6 jader na 1 CPU) propojené 1 Gbps ethernetem.
- Jeden počítač přímo běží na Linux verzi 3.2.0 (Ubuntu 12.04 LTS).
- Druhý přímo běží na FreeBSD 9.1.

V minulosti jsme testovali několik srovnávacích nástrojů a našim oblíbeným byl httperf. Vzhledem k tomu, že používá select () a běží pouze v jednom procesů, dosáhl svých výkonnostních limitů, když jsme se snažili změřit HTTP servery na vícejádrových počítačích. Takže jsme přešli na weighttp, který je založen na libev (rodina epoll) a může používat více nativních vláken. Použili jsme weighttp z FreeBSD následujícím způsobem:

```
weighttp -n 1 -c 1 -t 1 -k http://<ip_address>:<port_number>/
```

To znamená, že je vytvořeno 1000 HTTP připojení, a každé připojení odesílá 100 požadavků. Pro provedení úlohy je vytvořeno 10 nativních vláken.

Cílové webové servery byly kompilovány na Linuxu. Pro všechny požadavky je vrácen stejný soubor index.html. Použili jsme soubor index.html nginx-u o velikosti 151 bajtů.

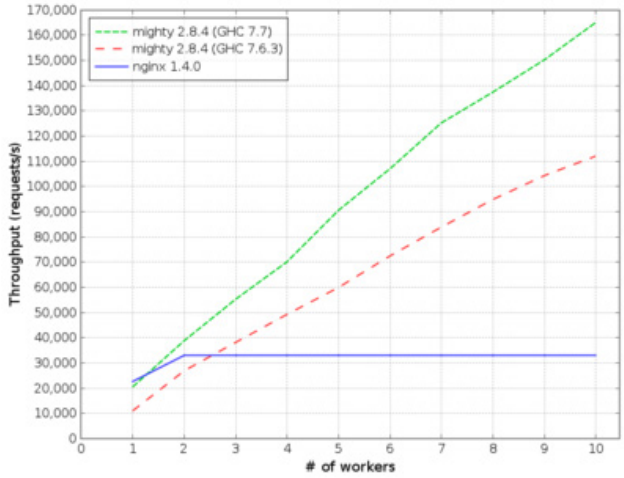
Vzhledem k tomu, Linux/FreeBSD má mnoho řídicích parametrů, musíme pečlivě konfigurovat parametry. Dobrý úvod do ladění parametrů Linuxu můžete najít v ApacheBench a HTTPerf.[1]

Pečlivě jsme nakonfigurovali jak mighty tak nginx takto:

- povolena mezipaměť deskriptoru souboru
- zakázáno logování
- zakázáno omezení rychlosti

1: http://gwan.com/en_apachebench_httperf.html

Tady je výsledek:



Obrázek 11.8: Výkonnost Warpu a nginxu

Osa x je počet dělníků a osa y udává propustnost, měřenou počtem požadavků za sekundu.

- mighty 2.8.4 (GHC 7,7): kompilován GHC verzí 7.7.20130504 (bude GHC verzí 7.8). Využívá paralelního I/O manažera pouze s jedním pracovníkem. GHC runtime varianta, je specifikováno + RTS -qa -A128m -N <x>, kde <x> je počet jader a 128m je velikost alokované oblasti používené garbage collectorem.

- mighty 2.8.4 (GHC 7.6.3): kompilován GHC verzí 7.6.3 (což je poslední stabilní verze).

11.4 Klíčové myšlenky

Při realizaci našeho vysoce výkonného serveru v Haskellu jsme měli na paměti čtyři klíčové myšlenky:

1. Volání co nejmenšího počtu systémových volání

2. Použití speciálních funkčních implementací a vyhnutí se přepočítávání

3. Vyhýbání se zámkům

4. Použití vhodných datových struktur

Vyvolání co nejmenšího počtu systémových volání

I když na většině moderních operačních systémů už nejsou systémová volání tak drahé jako dříve, stále mohou při častém volání přidat významnou výpočetní zátěž. Warp skutečně provádí několik systémových volání při obsluze každé žádosti, včetně recv(), send() a sendfile() (systémové volání, který umožňuje kopírování dat pomocí DMA). Ostatní systémová volání, jako je open(), stat() a close() můžou být často vynechány při zpracování jedné žádosti, a to díky kešovacímu mechanismu, který je popsán v oddíle 11.7.

Abychom viděli, jaká systémová volání jsou aktuálně použita, můžeme použít příkaz strace. Když jsme příkazem strace pozorovali chování nginx, zjistili jsme, že používal accept4(), o kterém jsme v té době nevěděli.

Naslouchající socket je vytvořen s nastavením neblokujícího příznaku za použití standardní síťové knihovny Haskellu. Je-li nové připojení akceptováno z naslouchajícího soketu, je nutné nastavit také odpovídající soket jako neblokující. Síťová knihovna toto implementuje dvojnásobným voláním fcntl(): poprvé pro získání aktuálních příznaků a podruhé pro nastavení příznaků se povoleným neblokujícím příznakem.

V Linuxu je příznak neblokování připojeného soketu vždy vypnutý, i když je poslouchající soket neblokující. Systémové volání accept4() je rozšířená verze Linuxové accept(). Umožňuje nastavit příznak neblokování při akceptaci. Takže pokud budeme používat accept4 (), můžeme se vyhnout dvěma zbytečným voláním fcntl(). Neše záplata pro používání accept4() na Linuxu je již zapracovaná do síťové knihovny.

Speciální funkce a vyhnutí se přepočítávání

GHC poskytuje profilovací mechanismus, ale ten má omezení: správné profilování je možné pouze v případě, že program běží v popředí a nebude spouštět podřízené procesy. Chceme-li profilovat běžící činnosti serverů, musíme tomu věnovat zvláštní péči.

mighty tento mechanismus má. Předpokládejme, že n je počet dělníků v mighty konfiguračním souboru. Když n je větší nebo rovno 2, mighty vytváří n podřízených procesů a nadřazený proces jen doručuje signály. Nicméně pokud n je 1, mighty nevytváří žádný podřízený proces. Místo toho samotný spuštěný proces obsluhuje http požadavky. Pokud je zapnut režim ladění, mighty zůstává na svém terminálu.

Když jsme profilovali mighty-ho, byli jsme překvapeni, že standardní funkce pro formátování datumového řetězce spotřebovává většinu CPU času. Jak mnozí vědí, HTTP server by měl vrátit GMT datumové řetězce v hlavičkových polích jako je Date, Last-Modified, atd.:

```
Date: Mon, 01 Oct 2012 07:38:50 GMT
```

Takže jsme zavedli speciální formátovač pro generování GMT datumových řetězců. Srovnání naši specializované funkce a standardní Haskell implementace s využitím srovnávací knihovny criterion ukázaly, že naše byla mnohem rychlejší. Ale v případě, že HTTP server přijímá více než jeden požadavek za sekundu, server opakuje znovu a znovu stejné formátování. Tak jsme také zavedli mechanismus mezipaměti pro datumové řetězce.

V části 11.5 a 11.6 také vysvětlíme specializaci a vyhýbání se přepočítávání.

Vyhýbání se zámkům

Zbytečné zámky jsou pro programování zlem. Náš kód někdy používá nepotřebné zámky nepozorovaně, protože běhové systémy nebo knihovny používají zámky vnitřně. K implementaci serverů s vysokým výkonem musíme takové zámky identifikovat a pokud možno se jim vyhnout. Stojí za to zdůraznit, že zámky se stanou mnohem kritičtější v případě paralelního I/O Manažera. V části 11.7 a 11.8 budeme mluvit o tom, jak identifikovat zámky a jak se jim vyhnout.

Použití vhodných datových struktur

Standardní Haskell datovou strukturou pro řetězce je String, což je propojený seznam znaků Unicode. Vzhledem k tomu, že programování seznamů je srdcem funkcionálního programování, String je vhodný k mnoha účelům. Ale pro vysoce výkonné servery je seznamová struktura příliš pomalá a Unicode je příliš složitý, protože HTTP protokol je založen na proudech bajtů. Namísto toho používáme pro vyjádření řetězců (nebo zásobníků) ByteString. ByteString je pole bajtů s metadaty. Díky těmto metadatům je možné spojování bez kopírování. To je podrobně popsáno v části 11.5.

Další příklady vhodných datových struktur jsou Builder a double IORef, které jsou popsány v části 11.6 a části 11.7.

11.5 Parser HTTP požadavků

Kromě mnoha otázek, které se zabývají efektivní souběžností a I/O ve vícejádrovém prostředí, Warp také potřebuje mít jistotu, že každé jádro je při plnění svých úkolů efektivní. V tomto ohledu je nejdůležitější komponentou procesor HTTP požadavků. Jeho cílem je vzít proud bajtů pocházející ze vstupního soketu, parsovat řádek požadavku a individuální hlavičky, a nechat tělo požadavku zpracovat aplikací. Je třeba vzít tyto informace a vytvořit datovou strukturu, kterou aplikace (buď aplikace Yesod, mighty, nebo něco jiného) použije k vytvoření své odpovědi.

Samotné tělo požadavku představuje některé zajímavé výzvy. Warp poskytuje plnou podporu pro zřetězení požadavků. V důsledku toho musí Warp sestavit jednotlivé požadavky, jsou-li rozloženy do několika bloků, než je předá do aplikace. Řetězením může být přeneseno více požadavků v jednom připojení. Proto Warp musí zajistit, aby aplikace nespotřebovávala příliš mnoho bajtů,

protože by to odstranilo životně důležité informace z dalšího požadavku. Rovněž musí být zajištěno, aby se zbavil veškerých dat zbývajících z těla požadavku; jinak se tento zbytek bude parsovat jako začátek dalšího požadavku, což způsobí buď neplatný, nebo nepochopený požadavek.

Jako příklad zvažte následující teoretický požadavek od klienta:

```
POST /some/path HTTP/1.1
Transfer-Encoding: chunked
Content-Type: application/x-www-form-urlencoded
0008
message=
000a
helloworld
0000
GET / HTTP/1.1
```

HTTP parser musí extrahovat jméno cesty /some/path a hlavičku Content-Type a předat je do aplikace. Když aplikace začíná čtení těla požadavku, musí vyjmout hlavičky bloků (např. 0008 a 000a), a místo toho poskytnout skutečný obsah, tj. message=helloworld. Je třeba také zajistit, aby nebyly spotřebovány žádné další bajty po ukončení bloku (0000) tak, aby se nezasahovalo do dalšího zřetězeného požadavku.

Psaní parseru

Haskell je známý svými silnými parsovacími schopnostmi. Má tradiční parserové generátory, jakož i kombinátorové knihovny, například Parsec a Attoparsec. Textové moduly Parsec a Attoparsec pracují s plným rozpoznáním Unicode. Nicméně, HTTP hlavičky jsou zaručeně ASCII, takže Unicode rozpoznání je režie, kterou nemusíme vynakládat.

Attoparsec také poskytuje binární rozhraní pro parsování, která by nám umožnilo obejít režii na Unicode. Ale Attoparsec, tak efektivní jak je, stále zavádí režii vzhledem k ručnímu parseru. Takže pro Warp nepoužíváme žádné parsovací knihovny. Místo toho provádíme všechno parsování ručně.

To vede k další otázce: jak můžeme reprezentovat aktuální binární data? Odpovědí je ByteString, což jsou v podstatě tři kusy dat: ukazatel na nějaký kus paměti, offset od začátku této paměti po dotčená data, a velikost našich dat.

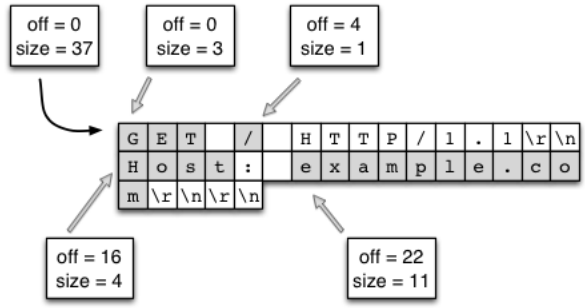
Informace o offsetu se může zdát nadbytečná. Místo toho bychom mohli trvat na tom, že náš paměťový ukazatele poukazuje na začátek našich dat. Avšak zahrnutím offsetu umožňujeme sdílení dat. Více ByteString-ů může směrovat ke stejnému kusu paměti a použít jeho různé části (také známo jako spojování). Nehrozí poškození dat, jelikož ByteString-y (jako většina dat Haskellu) jsou neměnné. Když se poslední ukazatel na kus paměti dále již nepoužívá, pak je paměťový zásobník uvolněn.

Tato kombinace je ideální pro náš případ užití. Když klient odešle požadavek přes soket, bude Warp číst data v poměrně velkých kusech (v současné době 4 096 bajtů). Ve většině případů je to dostatečná velikost pro zahrnutí celého řádku požadavku a všech hlaviček požadavku. Warp pak bude používat jeho ručně napsaný parser k rozdělení tohoto velkého kusu do řádků. To lze efektivně provést z následujících důvodů:

1. Potřebujeme skenovat pouze paměť pro znaky nového řádku. Bytestring knihovna poskytuje takovéto pomocné funkce, které jsou implementovány s C funkcemi nižší úrovně, jako je memchr. (Je to vlastně trochu složitější, vzhledem k víceřádkové hlavičce, ale platí stále stejný základní přístup).

2. K uložení dat není třeba alokovat další paměť. Jen bereme spoje z původního zásobníku. Viz Obrázek 11.9 pro demonstraci spojování jednotlivých komponentů z většího kusu dat. Stojí za to zdůraznit tento bod: vlastně jsme skončili v situaci, která je více efektivní než idiomatické C. V jazyce C jsou řetězce ukončeny null znakem, takže spojování vyžaduje přidělení nového paměťového zásobníku, kopírování dat ze starého zásobníku, a připojení znaku null.

Jakmile je zásobník rozdělen do řádků, provedeme podobný manévr pro konverzi řádků hlavičky do dvojic klíč/hodnota. Předpokládejme, že máme požadavek na:



Obrázek 11.9: Spojování ByteStringů

```
GET /buenos/d%C3%ADas HTTP/1.1
```

V tomto případě bychom potřebovali provést následující kroky:

1. Rozddělit do jednotlivých částí metodu požadavku, cestu a verzi.

2. tokenizovat cestu podél lomítek, s výsledkem ["buenos", "d%C3%ADas"].

3. Procentní dekódování jednotlivých částí, s výsledkem["buenos", "d\195\173as"].

4. UTF8 dekódování každé části, finálně upraveno v Unicode textu: ["buenos", "días"].

Tímto procesem jsme dosáhli několik výkonostních zisků:

1. Stejně jako u kontroly nového řádku, hledání lomítka je velmi efektivní operace.

2. Používáme efektivní vyhledávací tabulku pro převod hexadecimálních znaků na číselné hodnoty. Tento kód je jednoduché paměťové vyhledávání a nezahrnuje žádné větvení.

3. UTF8 dekódování je vysoce optimalizovaná operace v textovém balíčku. Podobně i textový balíček reprezentuje tato data v efektivní, zhuštěné podobě.

4. V Haskellu se tento výpočet až bude třeba jeho výsledek. Pokud dotčená aplikace nepotřebuje textovou verzi cesty, nebude proveden žáden z těchto kroků.

Poslední prováděnou částí parsování je spojování bloků. V mnoha ohledech je to jednodušší formou parsování. Parsujeme jediné Hexadecimální číslo, a pak přečteme uvedený počet bajtů. Tyto bajty jsou předávány do aplikace tak jak jsou (bez kopírování).

Conduit

Tento článek zmínil několikrát koncept přenosu těla požadavku do aplikace. Také narážel na problematiku podání odpovědi aplikací zpět na server a přijímání dat serverem ze soketu a odesílání dat do soketu. Poslední dosud nediskutován související bod je middleware, což jsou komponenty sedící mezi serverem a aplikací, které modifikují jeho požadavek nebo odpověď. Definice middleware je:

```
type Middleware = Application -> Application
```

Intuice za tímto je, že middleware vezme nějakou „vnitřní“ aplikaci, předzpracuje žádost, předá ji do interní aplikace pro získání odpovědi, a pak doupraví odpověď. Pro naše účely je dobrým příkladem gzip middleware, který automaticky komprimuje tělo odpovědi.

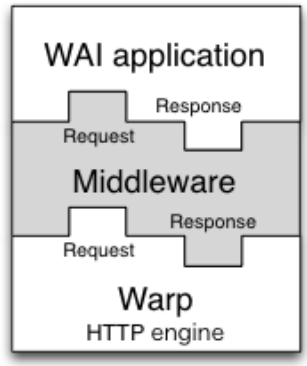
Předpokladem pro vytvoření takových middleware je prostředek k úpravě příchozích i odchozích datových proudů. Historicky standardním postupem ve světě Haskellu je odložené I/O. S odloženým

I/O reprezentujeme proud hodnot jako jednoduchou čistou datovou strukturu. Jak jsou požadována další data z této struktury, jsou prováděny I/O činnosti pro získání dat z jejich zdroje. Odložené I/O poskytuje vysokou úroveň sestavitelnosti. Nicméně pro server s vysokou propustností představuje hlavní překážku: ukončení zdrojů v odloženém I/O je nedeterministické. Použitím odloženého I/O by server pod vysokým zatížením rychle vyčerpal deskriptory souborů.

Bylo by také možné použít abstrakci nižší úrovně, v podstatě zacházet přímo s funkcemi čtení a zápisu. Nicméně, jedna z výhod Haskellu je jeho vysoceúrovňový přístup, což nám umožňuje uvažovat o chování našeho kódu. Co to také není zřejmé je, jak by se takové řešení vypořádalo s některými z běžných problémů, které vznikají při vytváření webových aplikací. Často je například nutné mít mezipaměť, kam čteme určité množství dat v jednom kroku (např. zpracování hlaviček požadavku) a zbytek čteme v jiné části našeho kódu (např. webová aplikace).

Protokol WAI (a tudíž Warp) postaven nad balíčkem Conduit pro řešení tohoto dilematu. Tento balíček provádí abstrakci proudů dat. Zachovává hodně ze sestavitelnosti odloženého I/O, poskytuje mezipaměť, a zajišťuje deterministickou manipulaci zdrojů. Výjimky jsou také zachovány tam, kam patří, v částech vašeho kódu, které se zabývají I/O, místo jejich skrývání v datových strukturách.

Warp reprezentuje proud bajtů přicházející od klienta jako Source, a data pro zaslání klientovi zapisuje do Sink. Aplikace přidá k Source tělo požadavku, a poskytuje odpověď opět jako Source. Middleware jsou schopné zachytit Source s těly požadavků a odpovědí a aplikovat na ně transformace. Obrázek 11.10 ukazuje, jak middleware zapadá mezi Warp a aplikaci. Sestavitelnost balíčku Conduit z toho činí jednoduchý a efektivní operaci.



Obrázek 11.10: Middleware

Vysvětlením na příkladu gzip middleware nám Conduit umožňuje vytvořit middleware, který běží téměř optimálním způsobem. Původní Source poskytnutý aplikací je spojen s gzip pomocí Conduit. Každý nový kus dat produkován počátečním Source se přivádí do knihovny zlib a plní mezipaměť komprimovanými bajty. Když je mezopameť naplněna, je vyslán buď do jiného middleware, nebo do Warpu. Warp pak vezme tuto mezipaměť s komprimovaným obsahem a odešle jej přes soket ke klientovi. V tomto bodě může mezipameť buď znovu použít, nebo je uvolněna. Tímto způsobem máme optimální využití paměti, nevytvářejí se žádné další údaje v případě selhání sítě, a snížíme garbage collection zátěž na běhovém systému.

Condut je sám o sobě je velké téma, a proto nebude probírán do větší hloubky. Je postačující teď říct, že využívání Conduitu je ve Warpu faktorem, který přispívá k vysokému výkonu.

Slowloris ochrana

Máme poslední obavu: útok Slowloris. Jedná se o formu útoku Denial of Service (DoS), kde každý klient odešle velmi malé množství informací. Tímto způsobem je klient schopen udržet vyšší počet připojení na stejném hardware/šíři pásma. Vzhledem k tomu, že web server má konstantní režii pro každé otevřené připojení bez ohledu na přenášené bajty, může to být efektivní útok. Proto musí Warp rozpoznat, že spojení neodesílá dostatek dat po síti a zrušit ho.

Správce časového limitu diskutujeme podrobněji níže, což je opravdové srdce ochrany proti Slowlorisu. Když přijde požádavek na zpracování, naším jediným požadavkem je dotazovat správce časového limitu, aby zjistil, zda byly obdrženy od klienta další data. Ve Warp se to všechno děje na úrovni Conduitu. Jak již bylo zmíněno, příchozí data jsou reprezentována jako Source. Jako součást tohoto Source pokaždé, když je přijat nový kus dat, je dotázán správce časového limitu. Vzhledem k tomu, dotazování správce je levná operace (v podstatě jen paměťový zápis), ochrana Slowloris nebrání významným způsobem výkonnosti jednotlivých obsluh připojení.

11.6 Sestavovač HTTP odpovědi

Tato část popisuje sestavovač HTTP odpovědi ve Warpu. WAI Response má tři konstruktory:

```
ResponseFile Status ResponseHeaders FilePath (maybe FilePart)
ResponseBuilder Status ResponseHeaders Builder
ResponseSource Status ResponseHeaders (Source (ResourceT IO) (Flush Builder))
```

ResponseFile je používán pro zaslání staického souboru zatímco ResponseBuilder a ResponseSource jsou pro zasílání dynamického obsahu vytvořného v paměti. Každý konstruktor zahrnuje jak Status tak ResponseHeaders. ResponseHeaders je definován jako seznam párů klíč/hodnota.

Sestavovač hlavičky HTTP odpovědi

Starý sestavovač sestavil hlavičku HTTP odpovědi pomocí lanu podobné datové struktury Builder. Nejprve konvertoval Status a každý element ResponseHeaders do struktury Builder. Každá konverze běžela v O(1). Pak je žřetězil přidáváním jedné struktury Builder ke druhé. Díky vlastnostem struktury Builder každá operace přidání běžela také v O(1). Nakonec zabalil hlavičku HTTP odpovědi kopírováním dat z Builder do bufferu v O(N).

V mnohých případech je výkonnost struktury Builder dostačující. Ale naše zkušenost je, že není dostatečně rychlá pro vysoce výkonné servery. K eliminaci režie struktury Builder jsme implementovali speciální sestavovač hlaviček HTTP odpovědí přímým použitím memcpy(), vysoce vyladěné funkce pro kopírování bajtů v jazyce C.

Sestavovač těla HTTP odpovědi

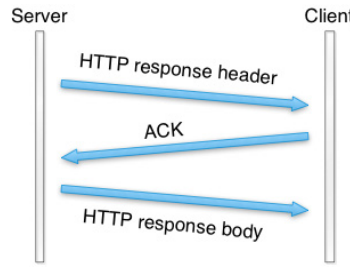
Pro ResponseBuilder a ResponseSource jsou hodnoty struktury Builder poskytovány aplikací zabaleny do seznamu ByteStringů. Sestavená hlavička je vložena před seznam a je použito send() pro zaslání seznamu do fixního bufferu.

Pro ResponseFile k zaslání hlavička a těla HTTP odpovědi používá Warp volání send() a sendfile(). Tento případ ilustruje Obrázek 11.7. Opět open(), stat(), close() a další systémové volání můžou být vynechány díky kešovacímu mechanismu popsanému v části 11.7. Následující dílčí část popisuje další výkonové ladění v případě ResponseFile.

Posílání hlavičky a těla společně

Když jsme měřili výkonnost Warpu posíláním statických souborů, vždycky jsme to dělali s vysokou souběžností (více připojení najednou) a dosáhli jsme dobrých výsledků. Nicméně když jsme si stanovili hodnotu souběžnosti na hodnotu 1, zjistili jsme, že Warp byl opravdu pomalý.

Pozorováním výsledků příkazu tcpdump jsme si uvědomili, že je to proto, že původně Warp používal kombinaci writev() pro hlavičku a sendfile() pro tělo. V tomto případě jsou HTTP hlavička a tělo odesílány v samostatnýchTCP paketech (Obrázek 11.11).



Obrázek 11.11: Sekvence paketů starého Warpu

Pro jejich poslání v jediném TCP paketu (pokud je to možné) přepnul nový Warp volání writev() na volání send(). Využívá send() s MSG_MORE příznakem pro ukládání záhlaví a sendfile() k odeslání jak uložené hlavičky, tak souboru. Podle našeho srovnávacího testu to zvýšilo propustnost nejméně 100 násobně .

11.7 Úklid s časovači

V této části je vysvětleno, jak implementovat časový limit připojení a jak ukládat do mezipaměti deskriptory souboru.

Časovače pro připojení

Aby se zabránilo Slowloris útokům, měla by být zrušena komunikace s klientem, pokud klient neodesílá významné množství dat za určitou dobu. Haskell poskytuje standardní funkci nazvanou timeout, jejíž typ je následující:

```
Int -> IO a -> IO (Maybe a)
```

První argument je délka časového limitu v mikrosekundách. Druhý argument je akce, která zpracovává vstup/výstup (IO). Tato funkce vrací hodnotu Maybe a v IO kontextu.

Maybe je definováno takto:

```
data Maybe a = Nothing | Just a
```

Nothing indikuje chybu (bez specifikovaného důvodu) a Just uzavírá úspěšnou hodnotu a. Takže timeout vrátí Nothing, pokud akce není dokončena ve specifikovaném čase. V opačném případě je vrácena úspěšná hodnota zabalená s Just. Funkce timeout výmluvně ukazuje, jak velká je sestavitelnost Haskellu.

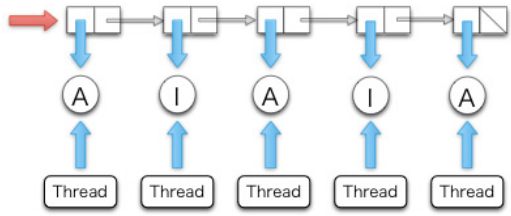
Funkce timeout je vhodná k mnoha účelům, ale její výkonnost je nedostatečná pro implementaci serverů s vysokým výkonem. Problém je v tom, že pro každé vypršení časového limitu tato funkce spustí nové uživatelské vlákno. Zatímco uživatelské vlákna jsou levnější než systémové vlákna, stále zahrnují režijní náklady, které lze sečíst. Musíme se vyhnout vytváření uživatelského vlákna pro zpracování časového limitu každého připojení. Takže jsme zavedli systém pro časový limit, který využívá pouze jedno uživatelské vlákno, nazývané správce časového limitu, k zpracování časových limitů všech připojení. Ve své podstatě jsou to dvě myšlenky:

- double IORef
- bezpečný swapovací a slučovací algoritmus

Předpokládejme, že stav připojení je popisován jako Aktivní a Neaktivní. Abychom vyčistili neaktivní připojení, správce časového limitu opakovaně kontroluje stav každého připojení. Pokud je stav Aktivní, správce ji změní na Neaktivní. Pokud je Neaktivní, správce zruší jeho přidružené uživatelské vlákno.

Každý stav je označován pomocí IORef. IORef je reference, jejíž hodnota může být destruktivně aktualizovaná. Kromě správce časového limitu, každé uživatelské vlákno opakovaně změní svůj stav na Aktivní prostřednictvím vlastního IORef jak jeho připojení aktivně pokračuje.

Správce časového limitu používá k těmto stavům seznam IORef. Uživatelské vlákno pro nové připojení se snaží předřadit svou novou IORef pro Aktivní stav do seznamu. Takže seznam je kritická sekce a potřebujeme atomizaci pro udržení konzistence seznamu.



Obrázek 11.12: A a I indikují Aktivní resp. Neaktivní.

Standardní způsob jak zaručit konsistenci je Mvar v Haskellu. Ale MVar je pomalý, protože je chráněný zámkem. Namísto něj tedy použijme další IORef jako referenci na seznam a atomicModifyIORef k manipulaci s ním. atomicModifyIORef je funkce pro atomickou aktualizaci IORef hodnot. Je implementována pomocí funkce porovnej-a-nastav instrukcí, které jsou podstatně rychlejší než použití zámku.

V následujícím je náčrt bezpečného swapovacího a slučovacího algoritmu:

```
do xs <- atomicModifyIORef ref (\ys -> ([], ys)) -- swap with an empty list, []
   xs' <- manipulates_status xs
   atomicModifyIORef ref (\ys -> (merge xs' ys, ()))
```

Správce časového limitu atomicky odkládá seznam za prázdný seznam. Pak manipuluje se seznamem přepínáním stavu vláken nebo odstraněním zbytečných stavů zrušených uživatelských vláken. Během tohoto procesu můžou být vytvořena nová spojení a jejich stavové hodnoty jsou vloženy přes atomicModifyIORef odpovídajícími uživatelskými vlákny. Poté manažer časového limitu atomicky sloučí ořezaný seznam a nový seznam. Díky odloženému vyhodnocování Haskellu se provádí použití funkce sloučení v O (1) a operace sloučení, což je v O (N), se odkládá, dokud nejsou jeho hodnoty skutečně přečteny.

Časovače pro deskriptory souborů

Uvažujme případ, kdy Warp pošle kompletní soubor voláním sendfile(). Bohužel, musíme zavolat stat() k zjištění velikosti souboru, protože sendfile() na operačním systému Linux vyžaduje od volajícího určit, kolik bajtů má být odesláno (sendfile() na FreeBSD/MacOS má magické číslo 0, která označuje konec souboru).

Pokud WAI aplikace znají velikost souboru, Warp se může vyhnout volání stat(). Pro WAI aplikace je snadné uložit informace o souboru do mezipaměti, jako jsou velikost a čas modifikace. Je-li časový limit mezipaměti dostatečně rychlý (řekněme 10 sekund), riziko nekonzistentnosti mezipaměti není vážné. Vzhledem k tomu, že můžeme bezpečně vyčistit mezipaměť, nemusíme se obávat úniku paměti.

Vzhledem k tomu, že sendfile() vyžaduje deskriptor souboru, naivní sekvence pro zaslání souboru je open(), sendfile(), opakovaně v případě potřeby, a close(). V tomto odstavci zvážíme, jak ukládat do mezipaměti deskriptory souborů, aby se zabránilo volání open() a close() více, než je nutné. Uložení deskriptorů souboru by mělo fungovat takto: v případě, že klient požaduje, aby byl soubor odeslán, je deskriptor souboru otevřen voláním open(). A pokud jiný klient požaduje stejný soubor krátce poté, dříve otevřený deskriptor je znovu použít. Není-li využíván uživatelským vláknem, je později deskriptor uzavřen voláním close().

Typickou taktikou pro tento případ je počítání referencí. Nebyli jsme si jistí, zda můžeme implementovat robustní počítadlo referencí. Co se stane, pokud je uživatelské vlákno zrušeno z neočekávaných důvodů? Pokud se nám nepodaří snížit hodnotu počítadla, deskriptor souboru nebude uzavřen - volání close(). Zjistili jsme, že schéma časového limitu připojení je bezpečné pro znovupoužití jako mechanismus mezipaměti pro deskriptory souborů, protože nepoužívá počítadla referencí. Nicméně nemůžeme jednoduše opětovně použít správce časového limitu z několika důvodů.

Každé uživatelské vlákno má svůj vlastní stav, který není sdílen. Ale rádi bychom ukládali do mezipaměti deskriptory souborů, aby se zabránilo volání open() a close() sdílením. Takže, musíme hledat deskriptor požadovaného souboru ve sbírce v mezipaměti uložených deskriptorů. Vzhledem k tomu, že vyhledávání by mělo být rychlé, neměli bychom použít seznam. Protože požadavky jsou přijímány současně, mohou být otevřeny dva nebo více deskriptory stejného souboru. Proto musíme uložit více deskriptorů pro jeden název souboru. Popisovaná datová struktura se nazývá multimap.

Implementovali jsme strukturu multimap, jejíž vyhledání je O (logN) a prořezávání je O (N) s červeno-černými stromy, jejichž uzly obsahují neprázdné seznamy. Vzhledem k tomu, červeno-černý strom je binární vyhledávací strom, vyhledávání je O (log (N)), kde n je počet uzlů. Můžeme to také převést do uspořádaného seznamu v O (N). V naší implementaci, uzly řezu, které obsahují deskriptor souboru pro uzavření, se během řezu také uzavřou. Použili jsme algoritmus pro konverzi seřazeného seznamu do červeno-černého stromu v O (N).

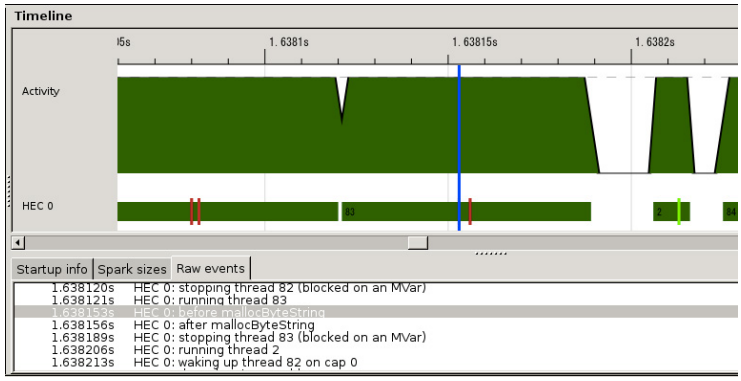
11.8 Budoucí práce

Do budoucna máme několik nápadů na zlepšení Warpu, ale vysvětlíme zde jen dva.

Alokace paměti

Při přijímání a odesílání paketů jsou alokovány mezipaměti. Jsou alokovány jako pole bajtů tak, aby mohly být předány do C procedur jako recv() a send(). Vzhledem k tomu, že je nejlepší přijímat nebo odesílat co nejvíce dat v každém systémovém volání, tyto mezipaměti jsou středně velké. Bohužel, GHC metoda přidělování velkých (větších než 409 bajtů na 64 bitových počítačích) polí bajtůvytvoří globální zámek v běhovém systému. Tento zámek se může stát úzkým místem při škálování nad 16 jader v případě, že každé vlákno často alokuje takovéto mezipaměti.

Provedli jsme počáteční průzkum dopadu na výkon u alokace velkého pole pro generování hlavičky http odpovědi. Za tímto účelem GHC poskytuje eventlog, který může zaznamenat časová razítka každé události. Zapouzdřili jsme funkci pro alokování paměti funkcí pro nahrávání uživatelské události. Potom jsme s ní kompilovali mighty a zaznamenali eventlog. Výsledný eventlog je znázorněn na Obrázku 11.13.



Obrázek 11.13: Eventlog

Malé svislé pruhy v řádku označené „HEC 0“ označují námi vytvořené události. Takže oblast obklopena dvouma pruhama je čas spotřebovaný na alokování paměti. Je to asi 1/10 z HTTP relace. Diskutujeme o tom, jak implementovat alokování paměti bez zámků.

Nové hřmící stádo

Problém hřmícího stáda je „starý, ale nový“ problém. Předpokládejme, že procesy nebo nativní vlákna jsou předem vytvořeny a sdílejí naslouchající soket. Volají accept() na soketu. Po vytvoření

spojení staré implementace Linuxu a FreeBSD probudí všechny procesy nebo vákna. Pouze jeden z nich může spojení přijmout a jiní budou zase spát. Protože to způsobuje mnoho kontextových přepínání, narážíme na problém výkonu. Tento jev se nazývá hřmící stádo. Nedávné implementace Linuxu a FreeBSD probudí pouze jeden proces či nativní vlákno, což tento problém odsunulo do minulosti.

Nedávné síťové servery mají tendenci používat rodinu epoll. Pokud dělníci sdílejí naslouchající soket a manipulují s připojeními funkcí epoll, znovu se objeví hřmící stádo. Důvodem je, že konvencí epoll je upozornit všechny procesy nebo nativní vlákna. nginx a mighty jsou oběťmi tohoto nového hřmícího stáda.

Paralelní I/O manažer je osvobozen od problému nového hřmící stáda. V této architektuře jenom jeden I/O manažer přijímá nová připojení prostřednictvím epoll. A další I/O manažeři pracují s již navázanými spojeními.

11.9 Závěr

Warp je univerzální web serverová knihovna, poskytující efektivní HTTP komunikaci pro širokou škálu případů užití. Za účelem dosažení jeho vysoké výkonnosti byly provedeny optimalizace na mnoha úrovních, včetně síťové komunikace, řízení vláken, a parsování požadavku.

Haskell se ukázal být úžasným jazykem pro psaní takového zdrojového kódu, který usnadňuje psaní vláknově bezpečného kódu a umožňuje vyhnutí se zbytečným kopírováním dat. Vícevláknové běhové prostředí drasticky zjednodušuje proces psaní událostmi řízeného kódu. Vysoká optimalizace GHC znamená, že v mnoha případech můžeme psát kód na vysoké úrovni a stále sklízet výhody vysokého výkonu. Přesto, se vším tímto výkonem, náš zdrojový kód je stále poměrně nepatrný (méně než 1300 SLOC v době psaní). Pokud chcete psát udržovatelný, efektivní, souběžný kód, měl by být Haskell vážně vzat do úvahy.

— 11 Warp (Kazu Yamamoto, Michael Snoyman a Andreas Voellmy)

12 Práce s Big Data v bioinformatice

(Eric McDonald a C. Titus Brown)

12 Práce s Big Data v bioinformatice

12.1 Úvod

Bioinformatika a Big Data

Pole bioinformatiky se snaží poskytnout nástroje a analýzy, které usnadňují porozumění molekulárním mechanismům života na Zemi, a to především na základě analýzy a korelace genomických a proteomických informací. Jak bude k dispozici stále narůstající množství genomových informací, včetně sekvencí genomu a vyjádřených genových sekvencí, stane se více efektivní, citlivá a specifická analýza kritickou.

V sekvenčním zpracování DNA v podstatě „digitalizuje" chemický a mechanický proces informace přítomné v DNA a RNA. Tyto sekvence jsou zaznamenávány s použitím abecedy jednoho písmena na jeden nukleotid. Na tato posloupnost dat jsou prováděny různé analýzy, které mají určit, jak jsou strukturovány do větších stavebních bloků, a jak to souvisí s jinou sekvencí dat. To slouží jako základ pro studium biologické evoluce a vývoje, genetiky a stále více i medicíny.

Data nukleotidových řetězců přichází ze sekvenčního procesu v řetězcích písmen známých jako čtení. (Použití výrazu číst v bio informatickém smyslu je nešťastná kolize s použitím výrazu v informatice a softwarovém inženýrství. To platí zejména proto, že výkon čtení čtení může být laděn, jak budeme dál diskutovat. Abychom zabránili dvojznačnosti u této nešťastné kolize, budeme se odkazovat na sekvence z genomu jako na genomické čtení.) Pro analýzu rozsáhlejších struktur a procesů do sebe musí vícenásobné genomické čtení zapadat. Toto zapadnutí se odlišuje od puzzle v tom, že obraz často není známý a priori a jeho části se mohou překrývat (a často překrývají).

Další komplikaci představuje to, že ne všechna genomová čtení jsou dokonale spolehlivá a můžou obsahovat řadu chyb, jako jsou vložení nebo vymazání písmen nebo substituce nesprávných písmen nukleotidů. Zatímco redundantní čtení může pomoci při skládání nebo doplňování dílků puzzle, kvůli této nedokonalé spolehlivosti je také překážkou ve všech existujících technologiích sekvenčního zpracování. Výskyt chybných genomických čtení roste s objemem dat, a to komplikuje skládání dat.

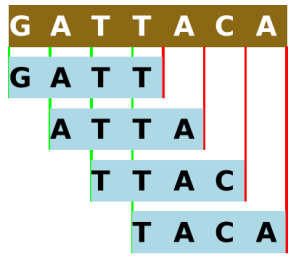
Jak se zlepšila technologie sekvenčního zpracování, objem produkovaných sekvenčních dat začal překračovat možnosti počítačového hardware s použitím obvyklých metod pro analýzu těchto dat. (Velké množství nejmodernějších sekvenčních technologií produkuje obrovské množství genomového čtení, typicky desítky milionů až miliard, z nichž každá má sekvenci 50 až 100 nukleotidů.) Tento trend bude pokračovat a je součástí toho, co je známé v komunitách pro vysoce výkonné výpočty (HPC), analytiku a informatiku jako Big Data [Varc] problém. Jak se hardware stává limitujícím faktorem, obrátila se zvýšená pozornost na softwarová řešení pro zmírnění tohoto problému. V této kapitole představujeme jedno takové softwarové řešení, jak jsme ho ladili a škálovali pro zpracování terabajtů dat.

U výzkumu jsme se soustředili na efektivní předzpracování, v kterém různé filtry a třídící přístupy ořežou, vyřadí a třídí genomová čtení pro zlepšení následné analýzy. Tento přístup má výhodu v omezení změn, které musí být provedeny následnými analýzami, jež obvykle přímo zpracují genomické čtení.

V této kapitole budeme prezentovat naše softwarové řešení a popíšeme, jak jsme ho ladili a škálovali k efektivnímu zpracování stále většího množství dat.

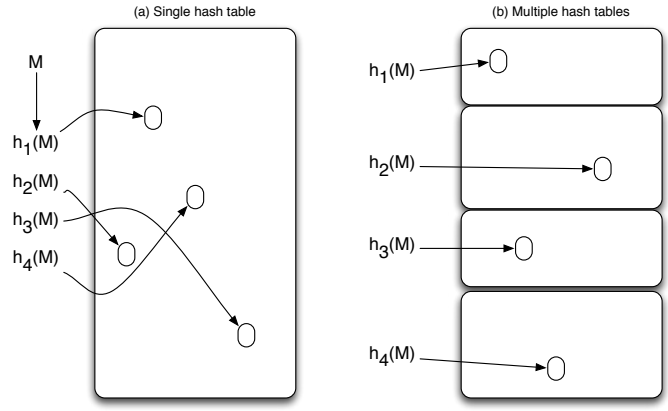
Co je khmer software?

Khmer je naše sada softwarových nástrojů pro předzpracování velkých objemů dat genomových sekvencí před analýzou běžnými bio informatickými nástroji [eab] (bez vztahu k etnické skupině pocházející z jihovýchodní Asie). Tento název pochází z volného spojení s termínem k-mer: jako součást předzpracování, genetické sekvence jsou rozloženy do překrývajících se podřetězců dané délky k. Protože jsou řetězce mnoha molekul často nazývány polymery, řetězce specifického počtu molekul se nazývají k-mery, přičemž každý dílčí řetězec reprezentuje jeden takový řetězec. Všimněte si, že pro každé genomové čtení bude počet k-merů počtem nukleotidů v sekvenci mínus k plus jedna. Takže téměř každé genomické čtení bude rozloženo do mnoha překrývajících se k-merů.



Obrázek 12.1: Rozklad genomové sekvence do 4-merů. V khmer-u jsou dopředná sekvence a reverzní komplement každého k-meru rozsekány na stejnou hodnotu, jako uznání, že DNA je dvouvláknová. Viz část o budoucím směřování.

Protože vám chceme říct o tom, jak jsme měřili a ladili tento kus open source software, přeskočíme hodně z jeho teorie. Postačuje uvést, že počítání k-merů je zásadní pro jeho provoz. Pro kompaktní počítání velkého množství k-merů je použita datová struktura známá jako *Bloomův filtr* [Harvard] (Obrázek 12.2). S vyzbrojením pro počítání k-merů pak můžeme vyloučit vysoce redundantní data z dalšího zpracování, což je proces známý jako „digitální normalizace". V rámci přístupu k ořezávání chyb můžeme sekvence dat s nízkým výskytem ošetřit jako pravděpodobné chyby a vyloučit je z dalšího zpracování. Tato normalizace a procesy ořezávání výrazně sníží množství surových sekvenčních dat potřebných pro další analýzu, přičemž většinou zachovávají informace, které nás zajímají.



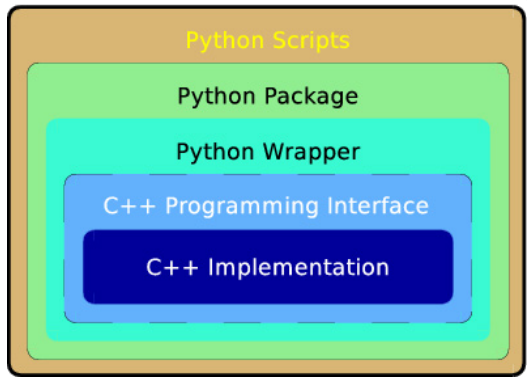
Obrázek 12.2: Bloomův filtr je v podstatě velká hashovací tabulka s pevnou velikostí, do které jsou elementy vloženy nebo dotazovány pomocí vícenásobných ortogonálních hashovacích funkcí, s žádným pokusem o sledování kolize; jsou to proto *pravděpodobnostní* datové struktury. Naše implementace používá několik různých rozptylových tabulek, každá má svou vlastní rozptylovou funkci, ale vlastnosti jsou identické. Obvykle doporučujeme, aby Bloomovy filtry v khmeru byly nakonfigurovány pro používání co nejvíce hlavní paměti, která je dostupná, protože to maximálně snižuje kolize.

Khmer je určen k provozu na velkých datových souborech s miliony až miliardy genomových čtení, obsahující desítky miliard unikátních k-merů. Některé z našich stávajících datových sad vyžadují až terabajty systémové paměti jenom aby držely počty k-merů v paměti, ale to není kvůli neefektivnímu programování: v [PHCK + 12] ukážeme, že khmer je podstatně více paměťově efektivní, než jakékoliv přesné nastavení schématu členství pro velkou řadu zajímavých k-mer problémů. Je nepravděpodobné snadné dosažení významného zlepšení využití paměti.

Náš cíl je tedy jednoduchý: tváří v tvář těchto velkých datových souborů bychom chtěli optimalizovat khmer na čas zpracování, včetně a hlavně zejména na dobu potřebnou k načtení dat z disku a počítání k-merů. Pro zvědavce je zdrojový kód včetně dokumentace ke khmeru dostupný na GitHubu na http://github.com/ged-lab/khmer.git. Khmer je k dispozici již asi čtyři roky, ale pouze zveřejněním několika článků ho jiní začali používat; na základě e-mailové interakce v roce 2012 odhadujeme uživatelskou populaci na asi 100 skupin, i když se zdá, že roste rychle, když se ukazuje, že velká třída problémů skládání se rychleji poddá s khmerem [BHZ + 12].

12.2 Architektura a úvahy o výkonu

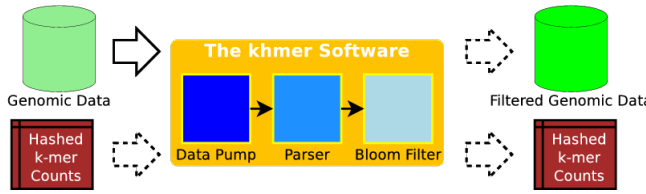
Khmer začínal jako průzkumné programovací cvičení a v průběhu času se vyvinul do zralejšího výzkumného kódu. Od svého založení se soustředil na řešení konkrétních vědeckých problémů s co možná největší přesností nebo „korektností". Postupem času, jak se software začal více využívat po celém světě, se staly více prominentními otázky jako výkon a škálovatelnost. Tyto problémy nebyly v dřívějších dobách nutně zanedbané, ale nyní mají vyšší prioritu, než kdysi. Naše diskuse se točí kolem toho, jak jsme analyzovali a řešili konkrétní výkonové a škálovací problémy. Protože Khmer je výzkumný kód stále ve vývoji, přijímá běžně nové funkce a má kolem sebe vybudovanou rostoucí sbírku dočasných skriptů. Musíme být opatrní, abychom zajistili, že změny provedené s cílem zlepšit výkon nebo škálovatelnost neporuší existující rozhraní nebo nezpůsobí snížení přesnosti a nenaruší správnost výpočtů. Z tohoto důvodu jsme postupovali strategií, která kombinuje automatizované testování s pečlivou inkrementální optimalizací a paralelizací. Ve spojení s dalšími aktivitami týkajícími se softwaru očekáváme, že tento proces bude v podstatě trvalý.



Obrázek 12.3: Vrstvený pohled na khmer software

Jádro software je napsáno v jazyce C ++. Toto jádro se skládá z datové pumpy (složka, která přemísťuje data z online úložiště do fyzické paměti RAM), parserů pro genomické čtení v několika běžných formátech, a několik k-mer počítadel. Aplikační programové rozhraní (API) je postaveno kolem jádra. Toto API lze samozřejmě použít z C ++ programů, jako to děláme u některých našich testů, ale také slouží jako základ pro rozhraní v Pythonu, nad kterým je postaven celý Python balíček. Četné Python skripty jsou distribuovány spolu s balíčkem. To znamená, že Khmer software je ve svém celku kombinací klíčových komponent, napsaných v C ++ kvůli rychlosti, rozhraní vyšší úrovně, vystavených přes Python pro snadnou manipulaci, a sadu nástrojových skriptů, které poskytují pohodlný způsob, jak provádět různé bio informatické úkoly.

Software khmer podporuje dávkové operace v několika fázích, z nichž je každá se samostatnými datovými vstupy a výstupy. Například může vzít sadu genomických čtení, počítat v nich k-mery, a pak popřípadě uložit hashovací tabulky Bloomových filtrů pro pozdější použití. Později může použít uložené hashovací tabulky a provést k-mer filtrování výskytu na nové sadě genomových čtení při ukládání filtrovaných dat. Tato flexibilita v opětovném použití dřívějších výstupů a v rozhodování, co ponechat, umožňuje uživateli přizpůsobit proceduru specificky k jeho /jejím potřebám a omezením při ukládání.



Obrázek 12.4: Datový tok přes khmer software

Spousty a spousty dat (potenciálně terabajtů) musí být přesunuty z disku do paměti pomocí softwaru. Je velmi důležité mít účinnou datovou pumpu, protože vstupní propustnost z disku do CPU může být o tři nebo dokonce o čtyři řády nižší, než je propustnost přenosu dat z fyzické RAM do CPU. Pro některé druhy datových souborů je třeba použít dekompresi. V obou případech musí parser pracovat s výslednými údaji efektivně. Parsovací úkol se točí kolem proměnné délky řádků, ale také musí počítat s neplatným genomickým čtením a zachováním určité části biologické informace, které mohou být využity v průběhu pozdějšího skládání, jako například párování konců sekvence fragmentů. Každé genomické čtení je rozděleno do množiny překrývajících se k-merů a každý k-mer je registrován nebo porovnán s Bloomovým filtrem. Je-li předtím uložený Bloomův filtr aktualizován nebo použit pro srovnání, pak musí být načten z disku. Pokud se Bloomův filtr vytváří pro pozdější použití nebo aktualizaci, pak musí být uložen na disk.

Datová pumpa vždy provádí sekvenční přístup k souborům a potenciálně mohou být žádány k přečtení velké objemy dat najednou. S ohledem na tuto skutečnost následují některé z otázek, které přicházejí na mysl:

- Využíváme plně skutečnosti, že jsou data přístupná sekvenčně?
- Je předem načteno dostatečné množství dat z disku, aby později nedocházelo ke zbytečnému zpoždění?
- Může být použit asynchronní vstup namísto synchronního?

- Můžeme efektivně obejít systémové mezipaměti, abychom omezili kopírování dat v paměti?
- Předkládá datová pumpa parseru data takovým způsobem, že nevytváří žádnou zbytečnou režii na rozhodovací logiku nebo logiku přístupu k datům?

Efektivita parseru je nezbytná, protože data jsou v poměrně volném řetězcovém formátu a musí být před jakýmkoli dalším zpracováním převedena do vnitřní reprezentace. Každý jednotlivý záznam dat je relativně malý (100 až 200 bajtů), existují ale miliony až miliardy záznamů, a proto jsme zaměřili trochu větší úsilí na optimalizaci parseru záznamu. Parser je ve svém jádru smyčkou, která rozděluje datový proud do genomického čtení a ukládá je v záznamech, provádějíc několik počátečních validací.

Některé úvahy o efektivnosti parseru jsou následující:

- Minimalizovali jsme počet, kolikrát parser přistupuje k datům v paměti?
- Minimalizovali jsme kopírování dat v paměti při analýze genomického čtení z datového proudu?
- Minimalizovali jsme režii volání funkcí uvnitř parsovací smyčky?
- Analyzátor se musí vypořádat s chaotickými daty, včetně dvojznačných základů, příliš krátkých genomických čtení a velikostí písmen. Je tato validace DNA sekvence provedena co nejefektivněji?

Pro iteraci přes k-mery v genomickém čtení a jejich hashování bychom se mohli ptát:

- Lze mechanismus k-mer iterace optimalizovat jak na paměť, tak na rychlost?
- Mohou být hashovací funkce Bloomova filtru nějakým způsobem optimalizovány?
- Minimalizovali jsme počet, kolikrát musí rozptylová funkce číst data z paměti?
- Můžeme zvýšit počet rozptylových funkcí v dávkách pro využití dat uložených ve vyrovnávací mezipaměti?

12.3 Profilování a měření

Pouhé přečtení zdrojového kódu se zaměřením na výkon odhalilo řadu oblastí pro zlepšení. Nicméně jsme chtěli systematicky kvantifikovat množství času stráveného v různých částech kódu. K tomu jsme použili několik profilovačů: GNU Profiler (gprof) a Tuning and analysis Utilities (TAU). Vytvořili jsme také nástroje v samotném zdrojovém kódu, což umožnilo zjemnění granularitu pohledu na klíčové výkonnostní metriky.

Přezkoumání kódu

Slepé použití nástroje pro měření systému (softwarového nebo jiného) je zřídka kdy dobrý nápad. Spíše je obecně dobrý nápad získat nějaké pochopení systému před jeho měřením. Za tímto účelem jsme nejprve přezkoumali kód jenom ručně.

Ruční pátrání po rozhodovacím algoritmu u neznámého kódu je dobrým nápadem. (Jeden z autorů, Eric McDonald, neznal khmer software v době, kdy se připojil k projektu, a byl to on, kdo to udělal.) I když je pravda, že profilovače (a jiné nástroje) mohou generovat grafy volání, tyto grafy jsou jen abstraktními souhrny. Ve skutečnosti je procházení kódu a prohlížení funkčních volání mnohem více pohlcující a poučný zážitek. Ladící programy mohou být použity pro tyto účely, ale moc se nehodí k průzkumu méně volaných částí programového kódu. Navíc může být krokování programu docela únavné. Body přerušení mohou být použity pro testování, zda se některé části kódu vykonávají při normálním běhu programu, ale jejich nastavení vyžaduje nějakou apriorní znalost kódu. Alternativně funguje docela dobře použití editoru s více podokny. Čtyři zobrazovací podokna můžou často současně zachytit všechny informace, které člověk potřebuje vědět v daném bodě a je mentálně schopen je zpracovat.

Přezkoumání kódu vykazovalo řadu věcí, z nichž některé, ale ne všechny, byly později potvrzeny profilovacími nástroji. Některé z věcí, kterých jsme si všimli, byly:

- Očekávali jsme, že největší provoz bude v počítání logiky k-merů.
- Nadbytečné volání funkce toupper bylo přítomno v oblastech kódu s největším provozem.
- Vstupní genomické čtení se provádělo řádek po řádku na požádání a bez optimalizace načítání dat předem.
- Data struktury genomického čtení byla předávána hodnotou po každém zparsování platného genomické čtení.

Ačkoli předcházející se může zdát jako poměrně silná sebekritika, chtěli bychom zdůraznit, že až do tohoto bodu byl větší důraz kladen na funkčnost a správnost khmeru. Naším cílem bylo optimalizovat stávající a většinou správný software, ne ho přestavět od nuly.

Nástroje

Profilovací nástroje se zajímají v první řadě o množství času stráveného v určité části kódu. Pro jeho měření vkládají do kódu speciální instrukce při kompilaci. Tyto instrukce změní velikost funkcí, což může ovlivnit umístnění řádků během optimalizace. Tyto instrukce rovněž přímo zvyšují určitou režii na celkovou dobu provedení; zvláště profilování oblastí kódu s velkým provozem může mít za následek poměrně významnou režii. Takže pokud také měříte celkový čas vykonávání vašeho kódu, musíte si být vědomi toho, jak ho ovlivňuje samotné profilování. Chcete-li to posoudit, může být použit jednoduchý mechanismus pro externí sběr dat, jako je /usr/bin/time, k porovnání neprofilového a profilového času vykonávání pro identické nastavení optimalizačních příznaků a provozních parametrů.

Vliv profilování jsme posoudili měřením rozdílu mezi profilovaným a neprofilovaným kódem v celé řadě k- velikostí - menší hodnoty k vedly k většímu počtu k-merů na jedno genomové čtení a zvyšování vlivu profilovače. Pro k = 20 jsme zjistili, že neprofilovaný kód běžel o 19 % rychleji než profilovaný kód a pro k = 30, že neprofilovaný kód běžel o 14 % rychleji než profilovaný kód.

Před jakýmkoliv laděním výkonu ukázala naše profilovací data, že logika počítání k-merů byla část kódu s největším provozem, jak jsme předpovídali. Trochu překvapivé bylo, jak významný podíl to byl (přibližně 83 % z celkové doby), v kontrastu k I/O operacím pro ukládání (přibližně 5 % celkové doby pro střední a nízkou přenosovou rychlost dat).

Vzhledem k tomu, že naše zkušební datové sady byly asi 500 MB a 5 GB, nepředpokládali jsme, že zaznamenáme efekty způsobené ukládáním dat do vyrovnávacích mezipamětí.[1] Opravdu, když jsme kontrolovali efekty ukládání do mezipaměti, zjistili jsme, že nečinily více než nanejvýš pár sekund a nebyly tedy mnohem větší než chybové ukazatele na našich celkových časech provedení. To nás přimělo uvědomit si, že v tomto okamžiku I/O nebylo naším primárním úzkým místem v procesu optimalizace kódu.

Jakmile jsme začali paralelizování khmer softwaru, napsali jsme některé ovladače, které použily OpenMP [mem] pro testování naší paralelizace různých komponent. Zatímco gprof je dobrý v profilování jednovláknového vykonávání, postrádá schopnost sledovat vykonávání na jedno vlákno při použití více vláken a nechápe paralelizační mašinerii jako je OpenMP. Pro C/C ++ je OpenMP paralelizace určena direktivami kompilátoru. GNU C/C ++ kompilátory ve verzi 4.x série rozeznají tyto direktivy, pokud jsou dodány s -fopenmp přepínačem. Jsou-li OpenMP direktivy rozeznány, vloží kompilátory další kód pro vícevláknové zpracování v místech direktiv a kolem základních bloků nebo jiných seskupení, s nimiž jsou spojeny.

1: Pokud je velikost datové mezipaměti větší než data použitá v I/O výkonnostních srovnávacích testech, pak načtení přímo z mezipaměti, než z původního zdroje dat, může zkreslit měření po sobě jdoucích běhů srovnávacích testování. Zajištění zdroje dat většího než datová mezipaměť pomáhá zaručit periodické ukládání dat v mezipaměti, čímž se umožní výskyt kontinuálního proudu neopakujících se dat.

Protože nám gprof nemohl snadno reportovat výsledky pro každé vlákno a nepodporoval OpenMP, jak jsme požadovali, obrátili jsme se na jiný nástroj. Jednalo se o nástroj Tuning and Analysis Utilities (TAU) [eaa], který vzešel ze spolupráce s University of Oregon. Existuje celá řada paralelních profilovacích nástrojů, mnoho z nich se zaměřuje na programy využívající knihovny MPI (Message Passing Interface), které jsou oblíbené pro některé druhy vědeckých výpočetních úloh. TAU také podporuje MPI profilování, ale protože MPI není opravdu volbou pro khmer software v jeho současné podobě, ignorovali jsme tento aspekt TAU. Stejně tak TAU není jediný dostupný nástroj pro profilování jednoho vlákna. Kombinace profilování jednoho vlákna a schopnosti úzké integrace s OpenMP je jedním z důvodů, proč nás zaujal. TAU je také zcela open source a není vázán na jednoho dodavatele.

Zatímco gprof se spoléhá výhradně na instrukce vložené do zdrojového kódu při kompilaci (s některými dalšími připojenými částmi), TAU poskytuje tuto a také další možnosti. Jsou jimi knihovny interpozice (primárně využívány pro MPI profilování) a dynamické vkládání instrukcí do binárních souborů. Pro podporu těchto možností TAU poskytuje vykonávací wrapper s názvem tau_exec. Vkládání instrukcí během kompilace zdrojového kódu je podporováno přes wrapper skript, nazvaný tau_cxx.sh.

TAU potřebuje další konfiguraci pro podporu některých profilovacích aktivit. Například pro získání těsné integrace s OpenMP musí být TAU nakonfigurovaný a zkompilovaný s podporou OPARI. Podobně, pokud chcete použít čítače výkonu vystavené novějšími linuxovými jádry, musí být nakonfigurovány a zkompilovány s podporou PAPI. Jakmile je TAU zkompilován, budete ho pravděpodobně chtít kvůli pohodlí integrovat do svého systému. Například pokud je požadováno TAU profilování, nastavíme náš sestavovací systém tak, aby skript wrapperu tau_cxx.sh byl použit jako C ++ kompilátor. Při pokusu o kompilaci a použití TAU určitě budete chtít přečíst dokumentaci. Vzhledem k tomu, že je mnohem složitější než gprof, není to zdaleka tak snadné a intuitivní.

Manuální profilování

Zkoumání výkonnosti kusu software s nezávislými externími profilovači je rychlý a pohodlný způsob, jak se na první pohled dozvědět něco o době provádění různých částí softwaru. Nicméně profilovače obecně nejsou tak dobré v reportování času stráveného kódem v konkrétní smyčce uvnitř konkrétní funkce, nebo jaká je vstupní rychlost vašich dat. K rozšíření nebo doplnění externích profilovacích možností může být potřebný manuální zásah do kódu. Ten může být méně rušivý než automatické vkládání přídavných, profilovacích instrukcí, protože přímo řídíte, co chcete pozorovat. Za tímto účelem jsme vytvořili rozšiřitelný framework k vnitřnímu měření věcí, jako jsou propustnosti, počty iterací a časování atomických nebo operací s jemnou granularitou v rámci samotného softwaru. Abychom byli sami k sobě upřímní, interně jsme shromažďovali některá čísla, která by mohla být porovnána s měřeními z externích profilovačů.

Pro různé části kódu jsme potřebovali mít různé metriky. Nicméně všechny různé metriky mají určité věci společné. Jedna věc je, že většinou pracují s časovými údaji, a že obecně chcete kumulo-

vat časy naměřené za dobu vykonávání. Další věc je, že je žádoucí mechanismus konzistentního reportování. S ohledem na tyto úvahy jsme poskytli abstraktní základní třídu IPerformanceMetrics pro všechny odlišné metriky. Třída IPerformanceMetrics poskytuje některé konvenční metody: start_timers, stop_timers a timespec_diff_in_nsecs. Metody pro spuštění a zastavení časovače měří jak uplynulý reálný čas, tak uplynulý strojový čas procesoru na jedno vlákno. Třetí metoda vypočítává rozdíly v nanosekundách mezi dvěma standardními timespec objekty knihovny jazyka C, což je docela postačující rozlišení pro naše účely.

Aby bylo zajištěno, že režie manuálně vloženého profilovacího kódu není přítomna v produkčním kódu, pečlivě jsme je ohraničili v podmíněných kompilačních direktivách, takže v rámci sestavování můžeme specifikovat jejich vynechání.

12.4 Ladění

Ladění efektivity software je docela potěšující zkušenost, zejména tváří v tvář bilionů bajtů procházejících přes něj. Náš příběh se nyní obrátí na různá opatření, která jsme přijali pro zlepšení jeho efektivity. Dělíme jej do dvou částí: optimalizaci čtení a parsování vstupních dat a optimalizaci manipulace a psaní obsahu Bloomových filtrů.

12.5 Obecné ladění

Předtím, než se ponoříme do některých specifik toho, jak jsme ladili software khmer, chtěli bychom stručně zmínit několik možností pro obecné ladění výkonnosti. Produkční kód je často sestaven sadou bezpečných a jednoduchých optimalizací, které jsou aktivovány; lze obecně dokázat, že tyto optimalizace nemění sémantiku kódu (tj. nezavádí chyby) a vyžadují pouze jeden kompilační běh. Nicméně kompilátory poskytují další optimalizační volby. Tyto dodatečné možnosti mohou být obecně klasifikovány jako agresivní optimalizace, což je poměrně běžný termín v literatuře kompilátorů, a optimalizace řízeny výstupem z profilování (Profiler guided optimization - PGO) [Varf]. (Přísně vzato nejsou tyto dvě kategorie vzájemně exkluzivní, ale obvykle zahrnují odlišné přístupy.)

Agresivní optimalizace mohou být nebezpečné (mohou způsobit chyby) a v některých případech dokonce snížit výkonnost programu. Nebezpečnými mohou být jejich předpoklady o akceptovatelnosti snížení přesnosti výpočtů s čísly s desetinnou částí, nebo jejich předpoklady o různých argumentech funkcí a operací na různých paměťových adresách. Také mohou být specifické pro konkrétní procesor. Optimalizace řízená výstupem z profilování se navíc ještě rozhoduje podle pozorovaného chování programu. Často používanou technikou je sledování, které funkce programu jsou často volány ve vzájemných závislostech. Pak jsou umístěny v kódu vedle sebe tak, aby se maximalizovala pravděpodobnost, že budou načteny do stejné stránky paměti.

V této fázi našeho projektu jsme se vyhnuli oběma kategoriím dodatečných optimalizací ve prospěch cílených algoritmických vylepšení, která jsou přínosem v mnoha odlišných CPU architekturách. Také z hlediska složitosti sestavení systému může agresivní optimalizace vytvářet problémy s přenositelností a profilově řízené optimalizace zvyšují celkový počet pohyblivých částí, které mohou selhat. Vzhledem k tomu, že nebudeme distribuovat předem kompilované spustitelné programy pro různé architektury a že naše cílové publikum je obvykle ne moc znalé složitostí vývoje softwaru či sestavovacích systémů, je pravděpodobné, že budeme pokračovat ve vyhýbání se těmto optimalizacím, až dokud nebudeme cítit, že přínosy převáží nevýhody. Ve světle těchto úvah je náš hlavní důraz kladen spíše na zlepšení efektivity našich algoritmů, než na jiné druhy ladění.

Datová pumpa a operace parseru

Naše měření ukázala, že čas strávený počítáním k-merů převládá nad časem vykonávání vstupu z úložiště. Vzhledem k tomuto zajímavému faktu se může zdát, že bychom měli věnovat všechno naše úsilí zlepšení výkonnosti Bloomova filtru. Ale stálo za to podívat se na datovou pumpu a parser hned z několika důvodů. Jedním z důvodů bylo, že jsme potřebovali změnit design existující datové pumpy a parseru a přizpůsobit je k využívání více vláken pro dosažení škálovatelnosti. Dalším důvodem bylo, že jsme se zajímali o snižování počtu kopírování dat v paměti, která mohou mít vliv na efektivitu rozhraní Bloomova filtru s parserem. Třetím důvodem je to, že jsme chtěli provádět agresivní před-načítání dat nebo jenom načítání dat v případě, že se nám podaří zlepšit efektivitu logiky počítání k-merů do té míry, že se čas načtení stane konkurenceschopný s časem počítání. Nezávisle na ladění výkonu existovaly také problémy s udržovatelností a rozšiřitelností.

Jak se ukázalo, všechny z výše uvedených důvodů konvergovaly k novému designu. Později budeme podrobněji diskutovat o aspektech bezpečného vícevláknového designu. Nyní se zaměříme na snížení počtu kopírování dat v paměti a schopnosti provádět poměrně agresivní načtení dat předem.

Typicky, když program načte data z blokového úložného zařízení (například pevný disk), určitý počet bloků je uložen operačním systémem do mezipaměti pro případ, že jsou bloky opět potřeba. Existuje nějaký režijní čas spojený s touto činností; navíc množství dat k dopřednímu načtení do mezipaměti nemůže být jemně laděno. Kromě toho nelze k mezipaměti operačního systému přistupovat přímo z uživatelského procesu, a proto musí být zkopírována z mezipaměti do adresního prostoru uživatelského procesu. Jedná se o kopírování z paměti do paměti.

Některé operační systémy, například Linux, umožňují optimalizaci velikosti předem načítaných bloků dat z disku. Například může volat posix_fadvise (2) a readahead (2) pro konkrétní deskriptor souboru. Nicméně to umožňuje poměrně omezenou kontrolu a neobchází ukládání do mezipaměti operačního systému. Zajímáme se o obejití mezipaměti udržované OS. Tuto mezipaměť lze obejít, pokud je soubor otevřen s příznakem O_DIRECT a podporuje to souborový systém.

Použití přímého čtení není úplně jednoduché, protože čtení z úložiště musí být násobkem velikosti bloku v paměťovém médiu, a musí být umístěno do oblasti paměti, která je na adrese, jež je násobkem velikosti tohoto bloku.

To vyžaduje, aby program prováděl další činnosti, které by normálně dělal programový kód pro daný souborový systém. Implementovali jsme přímé čtení, včetně potřebného úklidu. Existují však některé případy, kdy přímé čtení nefunguje nebo je jinak nežádoucí. Pro tyto případy jsme se ještě pokusili vyladit velikost bloku dat načítaných předem.

Náš přístup pro ukládání je sekvenční a poskytnutím nápovědy pomocí posix_fadvise (2) můžeme operačnímu systému říci, aby četl více dat dopředu, než by to dělal za normálních okolností.

Minimalizace kopií z mezipaměti do mezipaměti je žádoucí pro datovou pumpu i parser. V ideálním případě bychom četli jednou z úložiště do naší vlastní mezipaměti a poté skenovali naši mezipaměť jednou za genomické čtení pro určení začátku a délky sekvence v rámci této mezipaměti. Nicméně, logika pro řízení mezipaměti je dost složitá a logika pro parsování (s uvážením našich konkrétních nuancí) je natolik složitá, že dočasné udržování načteného řádku vstupních dat v mezipaměti je postačující pro programátorovo pochopení našeho kódu. Chcete-li snížit dopad udržování jedné řádky v mezipaměti, raději pro kompilátor označíme dotčené funkce direktivou inline. Můžeme zde dále optimalizovat, pokud by se výkon v této části kódu ukázal být nedostačujícím, ale může to být na úkor srozumitelného návrhu softwaru.

Operace Bloomova filtru

Připomínajíc, že pracujeme s posloupností složených z abecedy čtyř písmen: A, C, G a T, se můžete ptát, zda se jedná o velká nebo malá písmena. Vzhledem k tomu, že náš software operuje přímo na uživatelských datech, nemůžeme se spoléhat na to, že data budou konzistentně s malými či velkými písmeny, protože obě platformy sekvenčního řazení a další softwarové balíky mohou měnit velikost písma. Zatímco je snadné opravit je u individuálního genomického čtení, musíme to opakovat pro každý milion či miliardu čtení!

Před laděním výkonu nerozlišoval kód velikost písmen až do míst, kde validoval DNA řetězec a kde vygeneroval rozptylové kódy. V těchto místech prováděl nadbytečné volání funkce toupper z C knihovny pro normalizaci sekvence na velká písmena, a to použitím maker jako je například to následující:

```
#define is_valid_dna(ch) \
    ((toupper(ch)) == 'A' || (toupper(ch)) == 'C' || \
     (toupper(ch)) == 'G' || (toupper(ch)) == 'T')
```

a:

```
#define twobit_repr(ch) \
    ((toupper(ch)) == 'A' ? 0LL : \
     (toupper(ch)) == 'T' ? 1LL : \
     (toupper(ch)) == 'C' ? 2LL : 3LL)
```

Pokud jste si přečetli manuál pro funkci toupper nebo prozkoumáváte hlavičkové soubory knihovny GNU C, můžete zjistit, že to je vlastně lokální funkce a ne jen makro. To tedy znamená, že existuje režie volání potenciálně netriviální funkce, alespoň při použití knihovny GNU C. My ale pracujeme s abecedou čtyř ASCII znaků. Lokální funkce je pro naše účely přehnaná. Takže nejen, že chceme eliminovat nadbytečnost, ale chceme použít něco víc efektivního.

Rozhodli jsme se normalizovat sekvence na velká písmena před jejich validováním. (A samozřejmě, validace se děje před pokusem o jejich konverzi na rozptylové kódy.) I když by mohlo být ideální provést normalizaci v parseru, ukazuje se, že sekvence mohou být zavedeny do Bloomova filtru jinými cestami. Takže prozatím jsme se rozhodli normalizovat sekvence bezprostředně před jejich validací. To nám umožňuje, abychom vynechali veškerá volání toupper jak v sekvenčním validátoru, tak v rozptylových funkcích.

Vzhledem k tomu, že normalizátorem sekvence můžou procházet terabajty genomických dat, je v našem zájmu, abychom jej optimalizovali co nejvíce, jak je to jen možné. Jeden přístup je:

```
#define quick_toupper( c ) (0x60 < (c) ? (c) - 0x20 : (c))
```

Pro každý jeden bajt by výše uvedené mělo provést jedno porovnání, jedno větvení a případně jedno sčítání. Můžeme to udělat lépe, než tohle? Ukázalo se, že ano. Všimněte si, že každé malé písmeno má ASCII kód, který je o 32 (hexadecimálně 20) větší, než jeho velký protějšek a 32 je mocnina 2. To znamená, že velká a malá ASCII písmena se odlišují pouze jedním bitem.

Tento postřeh lze aplikovat bitovou maskou!

```
c &= 0xdf; // quicker toupper
```

Výše uvedené má jednu bitovou operaci, žádné porovnávání a žádné větve. Velká písmena nerušeně projdou; malá písmena se stávají velkými písmeny. Perfektní, jak jsme chtěli. Pro naše potíže jsme získali asi 13% zrychlení běhu celého procesu(!).

Rozptylové tabulky našeho Bloomova filtru jsou… „expanzivní“. Zvýšení počtu pro rozptylový kód konkrétního k-meru znamená zasažení téměř N různých paměťových stránek, kde N je počet rozptylových tabulek alokovaných filtru. V mnoha případech stránky paměti, které musí být

aktualizovány pro další k-mer, jsou zcela odlišné, než pro ten aktuální. To může vést k velkému výpadku stránek z hlavní paměti, aniž by mohly využít výhod ukládání do mezipaměti. Pokud máme genomické čtení se 79 znaků dlouhou sekvencí a skenování k-merů o délce 20, a pokud máme 4 rozptylové tabulky, pak je potenciálně dotčeno až 236 (59*4) různých paměťových stránek. Pokud zpracováváme 50 milionů čtení, pak je snadné vidět, jak je to nákladné. Co s tím máme dělat?

Jedním z řešení je dávková aktualizace rozptylových tabulek. Hromaděním řady rozptylových kódů pro různé k-mery a pak jejich periodické využívání pro zvýšení počtů tabulky po tabulce, můžeme výrazně zlepšit využití mezipaměti. Počáteční práce na tomto poli vypadá docela slibně, a doufejme, že v době, kdy toto čtete, budeme mít plně integrovanou tuto modifikaci do našeho kódu. I když jsme se o tom nezmínili dříve v naší diskusi o měření a profilování, cachegrind, program, který je součástí open-source distribuce Valgrind [eac], je velmi užitečný nástroj pro posouzení efektivity tohoto druhu práce.

12.6 Paralelizace

Růst vícejádrových architektur v dnešním světě láká vyzkoušet využití jejich výhod. Nicméně na rozdíl od mnoha jiných problémových oblastí, jako je výpočetní dynamika kapalin či molekulární dynamika, náš Big Data problém se spoléhá na vysokou průchodnost zpracování dat - za určitým bodem paralelizace se musí stát v podstatě I/O - vázaným. Za tímto bodem přidávání dalších vláken nepomůže, protože rychlost přístupu k úložišti bude dosažena a vlákna se prostě zablokují v čekání dokončení předchozích I/O operací. To znamená, že využití více vláken může být užitečné, zvláště pokud data, která mají být zpracována, se drží ve fyzické paměti RAM, ke které je obecně mnohem rychlejší přístup, než k online úložišti. Jak již bylo uvedeno dříve, implementovali jsme mezipaměť pro načítání dat předem ve spojení s přímým čtením. Tato mezipaměť může používat více vláken; více bude o tom řečeno dále. Rychlost I/O není jediným omezením. Rozptylové tabulky používané pro počítání k-merů jsou dalším omezením. Sdílený přístup k nim bude rovněž diskutován dále.

Bezpečnost vícevláknového zpracování

Před tím, než půjdeme do detailů, bude užitečné objasnit si některé termíny. Lidé si často pletou pojem vícevláknového zpracování s bezpečností vícevláknového zpracování. Je-li něco bezpečné pro více vláken, pak k tomu může být souběžně přistupováno více vlákny bez obav z poškození načítaných nebo ukládaných dat. Je-li něco vícevláknové, pak je to souběžně provozováno více vykonávacími vlákny.

Jako součást naši paralelizační práce jsme přemodelovali část implementace C ++ jádra tak, aby byla vícevláknově bezpečná bez jakýchkoli předpokladů o použité knihovně. Proto může být použit Python threading modul u skriptů, které používají Python wrapper kolem implementace jádra, nebo

u abstrakce vyšší úrovně, jako je OpenMP uveden výše, by mohl být použit okolo jádra C ++ ovladač, nebo například explicitně implementována vlákna s pthready. Dosažení tohoto druhu nezávislosti na použitém modelu práce s vlákny a zajištění bezpečnosti práce s více vlákny při neporušení existujících rozhraní do C ++ knihovny bylo zajímavou výzvou softwarového inženýrství.

Vyřešili jsme to tím, že části API, které byly vystaveny jako vláknově bezpečné, udržovaly vlastní stav objektů jednoho vlákna. Tyto stavové objekty jsou vyhledány v mapě poskytované C ++ Standard Template Library (STL), kde jsou klíči identifikační čísla vláken. Identifikační číslo pro konkrétní vlákno se zjistí tak, že samotně dotčené vlákno získá tuto informaci od jádra operačního systému pomocí systémového volání. Toto řešení zavádí malé množství režie dotazem vlákna na jeho identifikační číslo s každým vstupem do funkce vystavené přes API, ale elegantně se vyhýbá problému porušení stávajících rozhraní, která byla napsána sériovým, tj. jednovláknovým výpočtem.

Datová pumpa a operace parseru

Vícejádrové počítače, se kterými se setkáváme ve světě HPC, můžou mít více řadičů paměti, kde jeden řadič je blíže (pokud jde o vzdálenost přenosu signálu) k jednomu CPU, než k jinému CPU. Jedná se o Non-Uniform Memory Access (NUMA) architektury. Důsledkem práce na počítačích s touto architekturou je, že časy načtení paměti se mohou výrazně lišit v závislosti na fyzické adrese. Protože bioinformatický software často vyžaduje pro běh velké nároky na paměť, často ho najdeme běžet na těchto počítačích. Proto pokud se používá více vláken, která mohou být přiřazena na různé NUMA uzly, musí být vzata do úvahy lokalita fyzické paměti RAM. Za tímto účelem jsme rozdělili naše mezipaměti pro načtení předem do několika segmentů rovnajících se počtu spuštěných vláken. Každé vlákno je odpovědné za alokování paměti pro svou část mezipaměti pro načítání dat předem. Tato část mezipaměti je spravována prostřednictvím stavového objektu, který je udržovaný v rámci jednoho vlákna.

Operace Bloomova filtru

Rozptylové tabulky Bloomova filtru spotřebovávají většinu hlavní paměti (viz Obrázek 12.1), a proto nemůžou být vhodným způsobem rozděleny na oddělené kopie mezi vlákny. Spíše jedna sada tabulek musí být sdílena všemi vlákny. To znamená, že o tyto zdroje dojde k soupeření mezi vlákny. Jsou zapotřebí paměťové bariéry [Vare] nebo nějaké formy zamykání, aby se zabránilo pokusu dvou nebo více vláken o přístup ke stejnému místu v paměti ve stejném okamžiku. Pro zvýšení čítače v rozptylových tabulkách používáme operace atomického sčítání. Tyto atomické operace [Varb] jsou podporovány v řadě platforem několika kompilátory, včetně GNU kompilátorů, a nejsou závislé na nějakém konkrétním modelu použití vláken nebo knihovny. Vytvářejí paměťové bariéry kolem operandů, které se mají aktualizovat, a tím zvyšují bezpečnost vícevláknového zpracování určité operace.

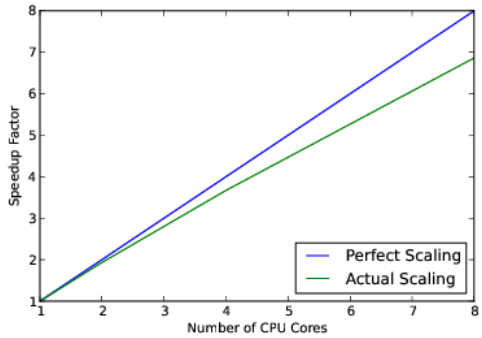
Úzké místo výkonu, které jsme neřešili, je čas na zapsání rozptylové tabulky ven do úložiště po dokončení počítání k-merů. Neměli jsme pocit, že to má tak vysokou prioritu, protože čas pro zápis je konstantní pro danou velikost Bloomova filtru a není závislý na množství vstupních dat.

Pro konkrétní 5 GB sadu dat, kterou jsme použili pro srovnávání, jsme viděli, že počítání k-merů trvalo šestkrát tak dlouho, jak zápis rozptylové tabulky. Pro ještě větší sady dat se poměr stává výraznější. To znamená, že jsme se nakonec zajímali o zlepšení výkonnosti i zde. Jednou z možností je umořit náklady na zápis v průběhu trvání operace počítání k-merů.

Web pro zkracování URL, bit.ly, má implementován Bloomův filtr, nazývaný dablooms [bsd], kterého dosahuje paměťovým mapováním jeho výstupního souboru do rozptylové tabulky paměti. Přijetí jejich nápadu, ve spojení s dávkovou aktualizací rozptylových tabulek, by nám efektivně dalo asynchronní výstup nárazově přes dobu života procesu a oddělilo by zápis dat od jejich výpočtu. Nicméně naším výstupem nejsou jednoduché výpočetní tabulky, které také obsahují hlavičku s metadaty; implementace paměťového mapování ve světle této skutečnosti je snahou, ke které musí být přistoupeno promyšleně a opatrně.

Škálování

Stálo úsilí na vytvoření škálovatelného khmeru za to? Ano. Samozřejmě, že jsme nedosáhli lineárního zrychlení. Ale pro každé zdvojnásobení počtu jader jsme v současné době dosáhli faktoru zrychlení 1,9.



Obrázek 12.5: Faktor zrychlení pro 1 až 8 CPU

V paralelních výpočtech musíme myslet na Amdahlův zákon [Vara] a zákon klesajících výnosů. Obecná formulace Amdahlova zákona v kontextu paralelních výpočtů je $S(N) = \frac{1}{(1-P) + \frac{P}{N}}$, kde S je zrychlení dosažené danými N CPU jádry a P je část kódu, který je paralelizovaný. Pro $\lim_{N \to \infty} S = \frac{1}{(1-P)}$ je konstantní. I/O rychlost přístupu k disku, se kterým čte a na který zapisuje software, je konečná a neškálovatelná; to přispívá k nenulovému $(1 - P)$. Navíc soupeření o sdílené zdroje v paralelizované části znamená, že $\frac{P}{N}$ je ve skutečnosti $\frac{P}{N^l}$, kde $l < 1$ oproti ideálnímu případu I = 1. Proto urychlení nebude růst lineárně s počtem procesorových jader.

Použitím rychlejších systémů pro ukládání dat, jako jsou SSD disky, na rozdíl od hard disků (HDD), zvyšuje rychlost přístupu k disku (a tím snižuje $(1 - P)$), ale to je mimo dosah softwaru.

I když nemůžeme nic dělat s hardware, můžeme se ještě pokusit zlepšit *l*. Myslíme si, že můžeme dále zlepšovat naši práci se sdílenými daty, jako jsou rozptylové tabulky paměti, a že můžeme ještě zefektivnit využití stavových objektů vláken. Práce na těchto dvou věcech nám pravděpodobně umožní zlepšení *l*.

12.7 Závěr

Software khmer je pohyblivý cíl. Pravidelně jsou do něj přidávány nové funkce a pracujeme na jeho začlenění do různých softwarových systémů používaných bioinformatickou komunitou. Stejně jako mnoho jiných softwarů v akademickém prostředí, začal život jako průzkumné programovací cvičení a vyvinul se do výzkumného kódu. Správnost byla a je hlavním cílem projektu. Zatímco výkon a škálovatelnost nemohou být skutečně považovány za dodatečné nápady, dávaly přednost správnosti a použitelnosti. Bylo řečeno, že naše úsilí týkající se škálovatelnosti a výkonu přineslo dobré výsledky, včetně zrychlení v jednovláknovém provedení a schopnosti významně snížit celkovou dobu provedení tím, že použijeme více vláken. Přemýšlení o problémech výkonu a škálovatelnosti vedlo k novému designu datové pumpy a parseru komponent. Do budoucna by měly být tyto složky schopny těžit nejen ze škálovatelnosti, ale i lepší udržovatelnosti a rozšiřitelnosti.

12.8 Budoucí směrování

Při pohledu do budoucna, jakmile jsme vyřešili základní problémy s výkonem, jsme se v první řadě zajímali o růst programátorského API, poskytujícího dobře vyzkoušené případy užití a dokumentaci, a poskytujícího dobře charakterizované komponenty pro integraci do větších zřetězení. Obecněji řečeno, chtěli bychom využít pokroků v teorii nízkopaměťových datových struktur ke zjednodušení některých případů užití, a také se zajímáme o průzkum distribuovaných algoritmů pro některé z více náročných problémů datových sad, které nás čekají v blízké budoucnosti.

Některé další úvahy, kterým čelí vývoj khmeru, zahrnují rozšíření možností rozptylových funkcí, které mají umožnit využití různých rozptylových funkcí pro jednořetězcové DNA a přidání rolovacích rozptylových funkcí pro umožnění k > 32.

Těšíme se na pokračování vývoje tohoto softwaru a doufáme, že má vliv na zpracování Big Data problému, kterému čelí molekulární biologové a bioinformatici. Doufáme, že se vám líbilo čtení o vysoké výkonnosti a použití open source softwaru ve vědě.

12.9 Poděkování

Děkujeme Alexis Black-Pyrkoszovi a Rosangele Canino-Koningové za komentáře a diskusi.

Bibliografie

Bibliografie

[ABB+86] Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tavanian, a Michael Young. Mach: A New Kernel Foundation for UNIX Development. V *Proceedings of the Summer 1986 USENIX Technical Conference and Exhibition*, strany 93–112, Červen 1986.

[AOS+00] Alexander B. Arulanthu, Carlos O'Ryan, Douglas C. Schmidt, Michael Kircher, a Jeff Parsons. The Design and Performance of a Scalable ORB Architecture for CORBA Asynchronous Messaging. V *Proceedings of the Middleware 2000 Conference*. ACM/IFIP, Duben 2000.

[ATK05] Anatoly Akkerman, Alexander Totok, and Vijay Karamcheti. Infrastructure for Automatic Dynamic Deployment of J2EE Applications in Distributed Environments. V *3rd International Working Conference on Component Deployment (CD 2005)*, strany 17–32, Grenoble, Francie, Listopad 2005.

[BHZ+12] CT Brown, A Howe, Q Zhang, A Pyrkosz, a TH Brom. A reference-free algorithm for computational normalization of shotgun sequencing data. In review at PLoS One, Červenec 2012; Předtisk na http://arxiv.org/abs/1203.4802, 2012.

[bsd] bit.ly softwaroví vývojáři. dablooms: a scalable, counting Bloom filter. http://github.com/bitly/dablooms.

[BW11] Amy Brown a Greg Wilson. *The Architecture Of Open Source Applications*. lulu.com, Červen 2011.

[CJRS89] David D. Clark, Van Jacobson, John Romkey, a Howard Salwen. An Analysis of TCP Processing Overhead. *IEEE Communications Magazine*, 27(6):23–29, Červen 1989.

[CT90] David D. Clark a David L. Tennenhouse. Architectural Considerations for a New Generation of Protocols. V *Proceedings of the Symposium on Communications Architectures and Protocols (SIGCOMM)*, strany 200–208. ACM, Září 1990.

[DBCP97] Mikael Degermark, Andrej Brodnik, Svante Carlsson, a Stephen Pink. Small Forwarding Tables for Fast Routing Lookups. V *Proceedings of the ACM SIGCOMM '97 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, strany 3–14. ACM Press, 1997.

[DBO+05] Gan Deng, Jaiganesh Balasubramanian, William Otte, Douglas C. Schmidt, a Aniruddha Gokhale. DAnCE: A QoS-enabled Component Deployment and Configuration Engine. V *Proceedings of the 3rd Working Conference on Component Deployment (CD 2005)*, strany 67–82, Listopad 2005.

[DEG+12] Abhishek Dubey, William Emfinger, Aniruddha Gokhale, Gabor Karsai, William Otte, Jeffrey Parsons, Csanad Czabo, Alessandro Coglio, Eric Smith, a Prasanta Bose. A Software Platform for Fractionated Spacecraft. V *Proceedings of the IEEE Aerospace Conference*, 2012, strany 1–20. IEEE, Březen 2012.

[DP93] Peter Druschel a Larry L. Peterson. Fbufs: A High-Bandwidth Cross-Domain Transfer Facility. V *Proceedings of the 14th Symposium on Operating System Principles (SOSP)*, Prosinec 1993.

[eaa] A. D. Malony et al. TAU: Tuning and Analysis Utilities. http://www.cs.uoregon.edu/Research/tau/home.php.

[eab] C. Titus Brown v al. khmer: genomic data filtering and partitioning software. http://github.com/ged-lab/khmer.

[eac] Julian Seward v al. Valgrind. http://valgrind.org/.

[EK96] Dawson R. Engler a M. Frans Kaashoek. DPF: Fast, Flexible Message Demultiplexing using Dynamic Code Generation. V *Proceedings of ACM SIGCOMM '96 Conference in Computer Communication Review*, strany 53–59. ACM Press, Srpen 1996.

[FHHC07] D. R. Fatland, M. J. Heavner, E. Hood, a C. Connor. The SEAMONSTER Sensor Web: Lessons and Opportunities after One Year. *AGU Fall Meeting Abstracts*, Prosinec 2007.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, a John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[GNS+02] Aniruddha Gokhale, Balachandran Natarajan, Douglas C. Schmidt, Andrey Nechypurenko, Jeff Gray, Nanbor Wang, Sandeep Neema, Ted Bapty, and Jeff Parsons. CoSMIC: An MDA Generative Tool for Distributed Real-time and Embedded Component Middleware and Applications. V *Proceedings of the OOPSLA 2002 Workshop on Generative Techniques in the Context of Model Driven Architecture*. ACM, Listopad 2002.

[HC01] George T. Heineman and Bill T. Councill. *Component-Based Software Engineering: Putting the Pieces Together*. Addison-Wesley, 2001.

[HP88] Norman C. Hutchinson and Larry L. Peterson. Design of the x-Kernel. V *Proceedings of the SIGCOMM '88 Symposium*, strany 65–75, Srpen 1988.

[HV05] Jahangir Hasan and T. N. Vijaykumar. Dynamic pipelining: Making IP-lookup Truly Scalable. V *SIGCOMM '05: Proceedings of the 2005 Conference on Applications, technologies, architectures, and protocols for computer communications*, strany 205–216. ACM Press, 2005.

[Insty] Institute for Software Integrated Systems. Component-Integrated ACE ORB (CIAO). www.dre.vanderbilt.edu/CIAO, Vanderbilt University.

[KOS+08] John S. Kinnebrew, William R. Otte, Nishanth Shankaran, Gautam Biswas, a Douglas C. Schmidt. Intelligent Resource Management and Dynamic Adaptation in a Distributed Real-time and Embedded Sensor Web System. Technical Report ISIS-08-906, Vanderbilt University, 2008.

[mem] OpenMP members. OpenMP. http://openmp.org.

[MJ93] Steven McCanne a Van Jacobson. The BSD Packet Filter: A New Architecture for User-level Packet Capture. V *Proceedings of the Winter USENIX Conference*, strany 259–270, Leden 1993.

[MRA87] Jeffrey C. Mogul, Richard F. Rashid, a Michal J. Accetta. The Packet Filter: an Efficient Mechanism for User-level Network Code. V *Proceedings of the 11th Symposium on Operating System Principles (SOSP)*, Listopad 1987.

[NO88] M. Nelson and J. Ousterhout. Copy-on-Write For Sprite. V *USENIX Summer Conference*, strany 187–201. USENIX Association, Červen 1988.

[Obj06] ObjectWeb Consortium. CARDAMOM - An Enterprise Middleware for Building Mission and Safety Critical Applications. cardamom.objectweb.org, 2006.

[OGS11] William R. Otte, Aniruddha Gokhale, a Douglas C. Schmidt. Predictable Deployment in Component-based Enterprise Distributed Real-time and Embedded Systems. V *Proceedings of the 14th international ACM Sigsoft Symposium on Component Based Software Engineering*, CBSE '11, strany 21–30. ACM, 2011.

[OGST13] William Otte, Aniruddha Gokhale, Douglas Schmidt, a Alan Tackett. Efficient and Deterministic Application Deployment in Component-based, Enterprise Distributed, Real-time, and Embedded Systems. *Elsevier Journal of Information and Software Technology (IST)*, 55(2):475–488, Únor 2013.

[OMG04] Object Management Group. *Lightweight CCM FTF Convenience Document*, ptc/04-06-10 edition, Červen 2004.

[OMG06] OMG. *Deployment and Configuration of Component-based Distributed Applications*, v4.0, Document formal/2006-04-02 edition, Duben 2006.

[OMG08] Object Management Group. *The Common Object Request Broker: Architecture and Specification Version 3.1, Part 2: CORBA Interoperability*, OMG Document formal/2008-01-07 edition, Leden 2008.

[PDZ00] Vivek S. Pai, Peter Druschel, a Willy Zwaenepoel. IO-Lite: A Unified I/O Buffering and Caching System. *ACM Transactions of Computer Systems*, 18(1):37–66, 2000.

[PHCK+12] J Pell, A Hintze, R Canino-Koning, A Howe, JM Tiedje, a CT Brown. Scaling metagenome sequence assembly with probabilistic de bruijn graphs. Accepted at PNAS, Červenec 2012, Předisk na http://arxiv.org/abs/1112.4193, 2012.

[RDR+97] Y. Rekhter, B. Davie, E. Rosen, G. Swallow, D. Farinacci, a D. Katz. Tag Switching Architecture Overview. *Proceedings of the IEEE*, 85(12):1973–1983, Prosinec 1997.

[SHS+06] Dipa Suri, Adam Howell, Nishanth Shankaran, John Kinnebrew, Will Otte, Douglas C. Schmidt, a Gautam Biswas. Onboard Processing using the Adaptive Network Architecture. V *Proceedings of the Sixth Annual NASA Earth Science Technology Conference*, Červen 2006.

[SK03] Sartaj Sahni a Kun Suk Kim. Efficient Construction of Multibit Tries for IP Lookup. *IEEE/ACM Trans. Netw.*, 11(4):650–662, 2003.

[SNG+02] Douglas C. Schmidt, Bala Natarajan, Aniruddha Gokhale, Nanbor Wang, a Christopher Gill. TAO: A Pattern-Oriented Object Request Broker for Distributed Real-time and Embedded Systems. *IEEE Distributed Systems Online*, 3(2), Únor 2002.

[SSRB00] Douglas C. Schmidt, Michael Stal, Hans Rohnert, a Frank Buschmann. *Pattern Oriented Software Architecture: Patterns for Concurrent and Networked Objects, Volume 2*. Wiley & Sons, New York, 2000.

[SV95] M. Shreedhar and George Varghese. Efficient Fair Queueing using Deficit Round Robin. V *SIGCOMM '95: Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communication*, strany 231–242. ACM Press, 1995.

[Vara] Různí. Amdahl's law. https://en.wikipedia.org/w/index.php?title=Amdahl%27s_law&oldid=515929929.

[Varb] Různí. Atomic operations. http://en.wikipedia.org/w/index.php?title=Linearizability&oldid=511650567.

[Varc] Různí. Big data. http://en.wikipedia.org/w/index.php?title=Big_data&oldid=521018481.

[Vard] Různí. Bloom filter. http://en.wikipedia.org/w/index.php?title=Bloom_filter&oldid=520253067.

[Vare] Různí. Memory barrier. http://en.wikipedia.org/w/index.php?title=Memory_barrier&oldid=517642176.

[Varf] Různí. Profile-guided optimization. http://en.wikipedia.org/w/index.php?title=Profile-guided_optimization&oldid=509056192.

[Var05] George Varghese. *Network Algorithmics: An Interdisciplinary Approach to Designing Fast Networked Devices*. Morgan Kaufmann Publishers (Elsevier), San Francisco, CA, 2005.

[VL97] George Varghese a Tony Lauck. Hashed and Hierarchical Timing Wheels: Data Structures for the Efficient Implementation of a Timer Facility. *IEEE Transactions on Networking*, Prosinec 1997.

[WDS+11] Jules White, Brian Dougherty, Richard Schantz, Douglas C. Schmidt, Adam Porter, and Angelo Corsaro. R&D Challenges and Solutions for Highly Complex Distributed Systems: a Middleware Perspective. *the Springer Journal of Internet Services and Applications special issue on the Future of Middleware*, 2(3), Prosinec 2011.

[WKNS05] Jules White, Boris Kolpackov, Balachandran Natarajan, a Douglas C. Schmidt. Reducing Application Code Complexity with Vocabulary-specific XML language Bindings. V *ACM-SE 43: Proceedings of the 43rd annual Southeast regional conference*, 2005.

VÝKONNOST OPEN SOURCE APLIKACÍ
Rychlost, přesnost a trocha štěstí

Editoval Tavish Armstrong

Přeloženo z anglického originálu knihy *The Performance of Open Source Applications*.
1.vydání, 2013
Vydáno nakladatelstvím Lulu Press, Inc (lulu.com), Severní Karolína (USA).

Publikace je součástí série knih věnujících se open source aplikacím: *Architecture of Open Source Applications* (www.aosabook.org).

Vydavatel:
CZ.NIC, z. s. p. o.
Milešovská 5, 130 00 Praha 3
Edice CZ.NIC
www.nic.cz

1. vydání, Praha 2016
Kniha vyšla jako 13. publikace v Edici CZ.NIC.

CZ.NIC | SPRÁVCE DOMÉNY CZ

Edice CZ.NIC

O knize Kniha *Výkonnost open source aplikací* obsahuje dvanáct případových studií, jejichž autoři popisují řadu metod a triků vedoucích ke zlepšení výkonu. Škála je opravdu široká: počínaje spekulativními optimalizacemi, díky nimž nás moderní webové prohlížeče udivují svou předvídavostí, přes sofistikované algoritmy syntaktické analýzy, až po specifika mobilních zařízení. Pro ostříleného softwarového profesionála může být poměrně překvapivá předposlední kapitola, která demonstruje, že i „nepraktické" funkcionální jazyky mohou nabízet elegantní řešení pro zvýšení výkonu. Sekundární, ale rovněž velmi zajímavou problematikou, které se tak či onak dotýká většina kapitol, jsou postupy, jimiž lze změřit reálnou výkonnost softwarových systémů a porovnat ji s alternativními implementacemi.
Publikace, kterou se chystáte číst, je přeložena z anglického originálu knihy *The Performance of Open Source Applications*, která vyšla v roce 2013 a je součástí rozsáhlejší série knih věnujících se open source aplikacím: *Architecture of Open Source Applications* (www.aosabook.org).

O autorovi Na vzniku knihy spolupracovala celá řada autorů a editorem tohoto díla je Tavish Armstrong (tavisharmstrong.com) pocházející z kanadského města Aurora. Tavish během let, kdy žil v Montrealu, vystudoval softwarové inženýrství na Concordia University.

O edici Edice CZ.NIC je jedním z osvětových projektů správce české domény nejvyšší úrovně. Cílem tohoto projektu je vydávat odborné, ale i populární publikace spojené s Internetem a jeho technologiemi. Kromě tištěných verzí vychází v této edici současně i elektronická podoba knih. Ty je možné najít na stránkách knihy.nic.cz

knihy.nic.cz
ISBN 978-80-88168-14-0
9 788088 168140 >