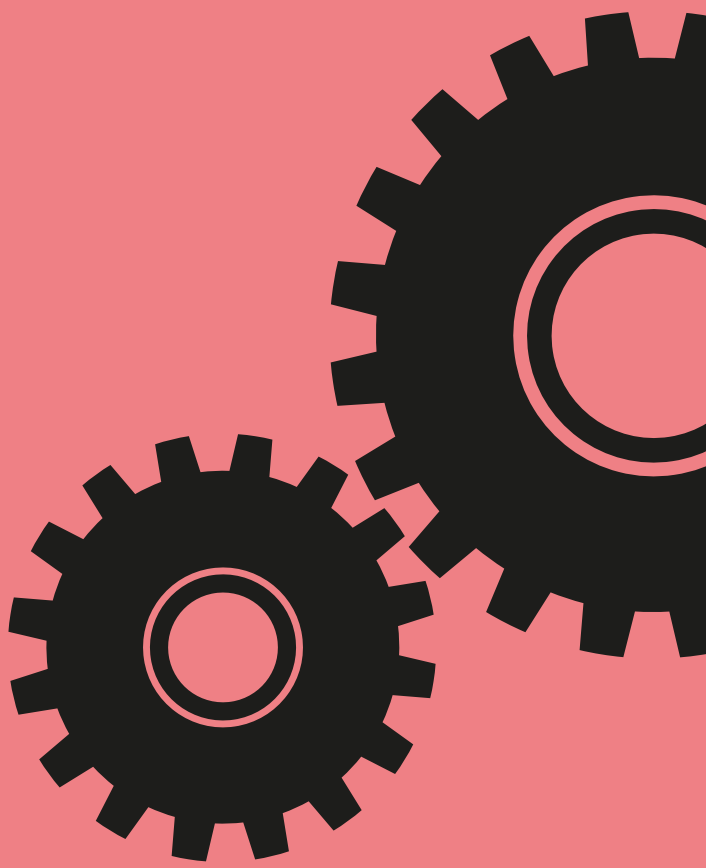
Editoval Tavish Armstrong

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.
ISBN 978-80-88168-14-0
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:

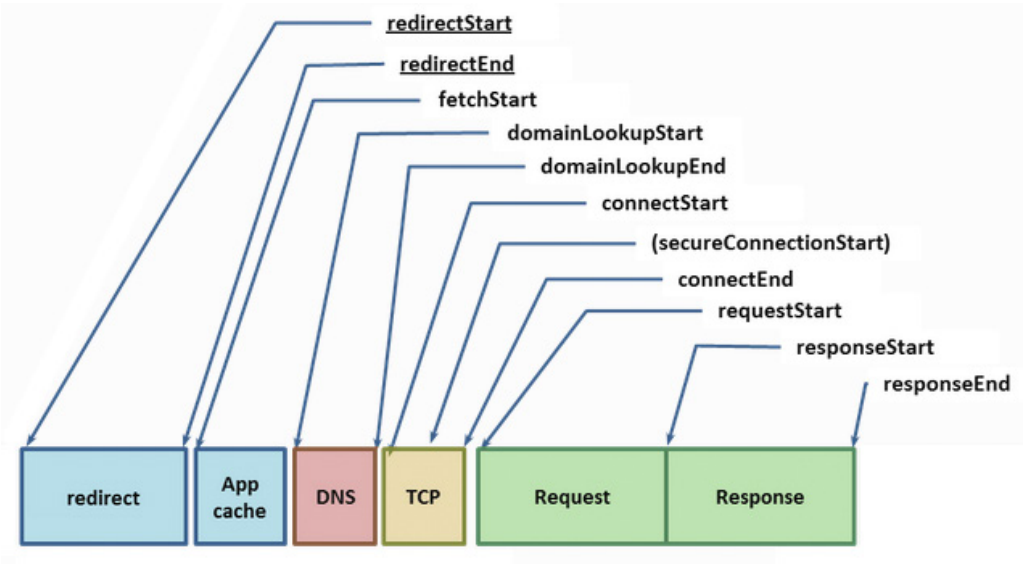
- její velikost je 1 280 KB,
- skládá se z 88 zdrojů,
- připojuje se k více než 15 různým serverům.

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:



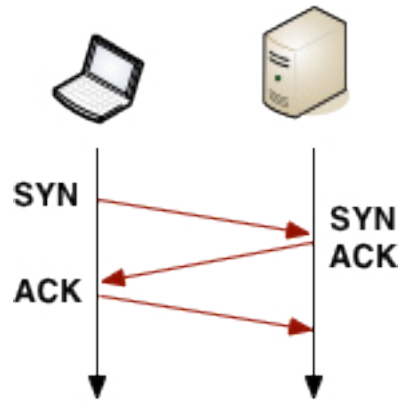
Obrázek 1.1: Časování navigace

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.

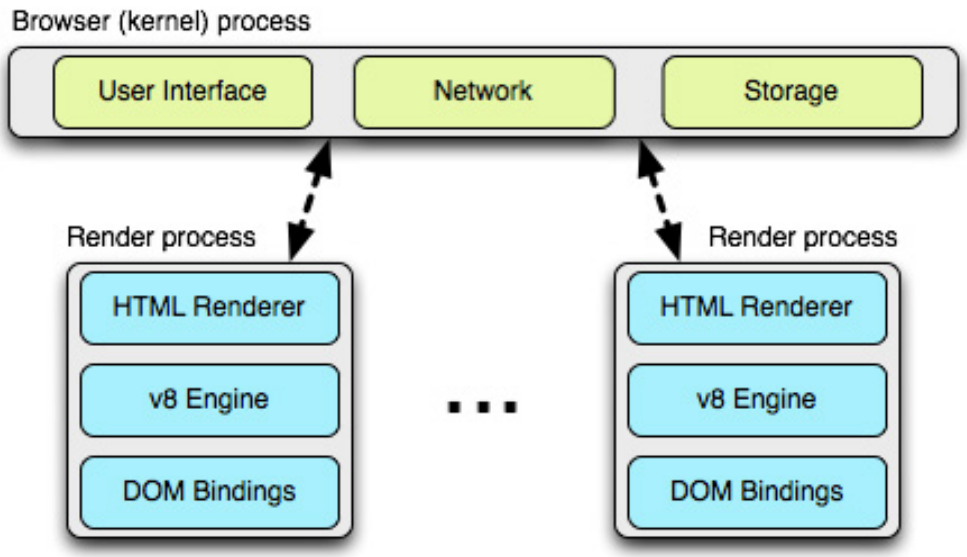
Teď, když jsme identifikovali problém, pojďme se podívat podrobněji na implementace.

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

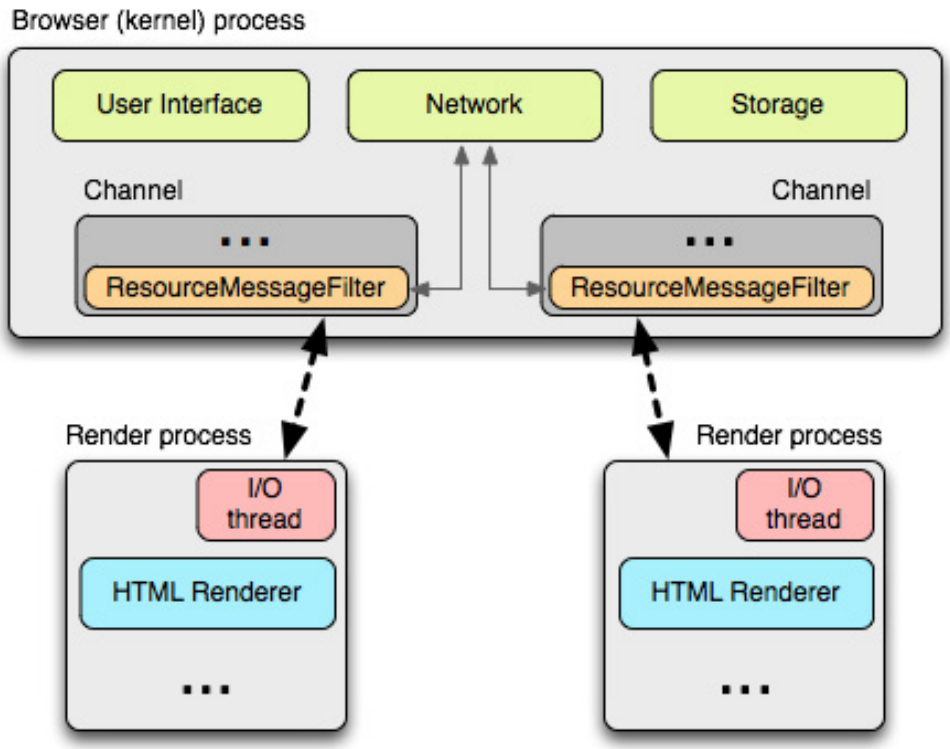
Architektura vyhrazuje každé záložce jeden vykreslovací proces. Ten obsahuje instance vykreslovacího jádra prohlížeče Blink a jádro V8 JavaScriptu, společně s vazebným kódem, který propojuje tyto a několik málo dalších komponent[2].

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

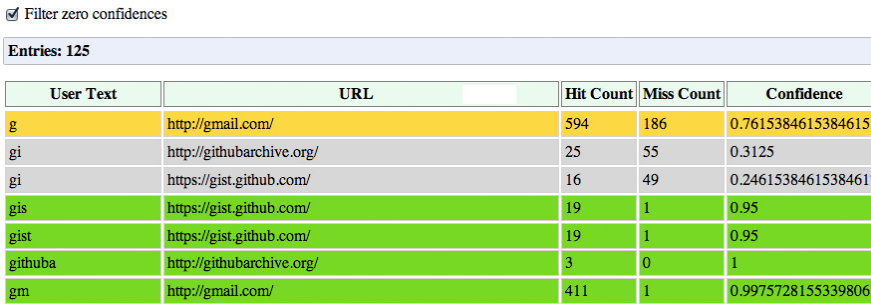
Host name	How long ago (HH:MM:SS)	Motivation
http://www.google-analytics.com/	15:31:33	n/a
https://a248.e.akamai.net/	15:31:30	n/a
https://csi.gstatic.com/	15:31:16	n/a
https://docs.google.com/	15:31:18	n/a
https://gist.github.com/	15:31:34	n/a
https://lh6.googleusercontent.com/	15:31:16	n/a
https://secure.gravatar.com/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

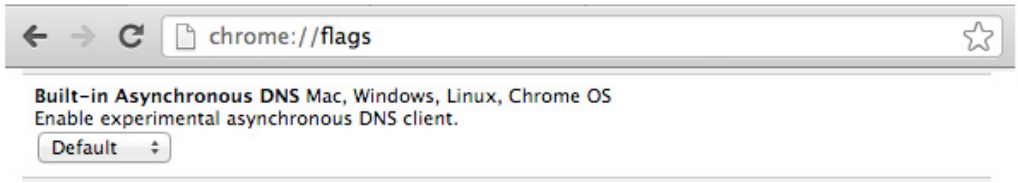
Proč se jednoduše nespolehnout na automatiku prohlížeče? V některých případech můžete chtít předběžně rozpoznat jméno serveru, který není nikde na stránce zmíněn. Učebnicovým příkladem je přesměrování: odkaz může odkazovat na server – jako je například analytická sledovací služba –, který následně přesměruje 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.

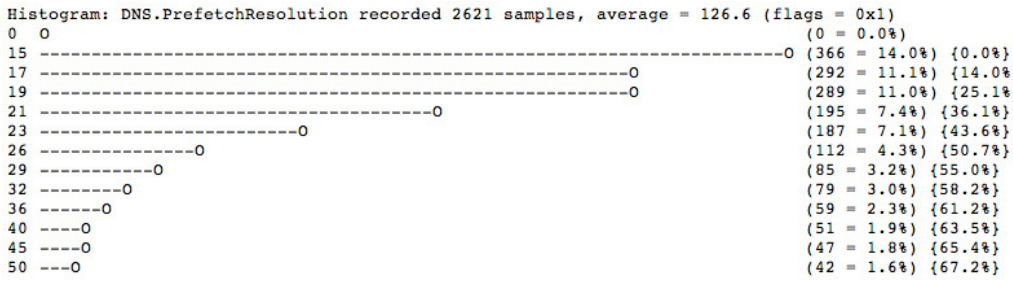


Obrázek 1.9: Povolení asynchronního DNS překladače

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.

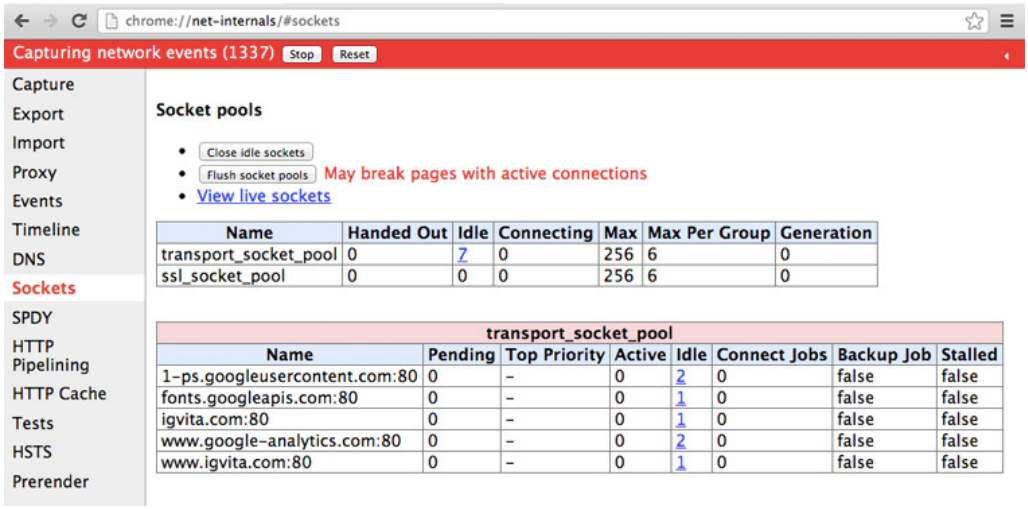
Chcete-li vidět hostitele, u kterých bylo spuštěno předběžné připojení TCP, otevřete novou záložku a navštivte chrome://dns.

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

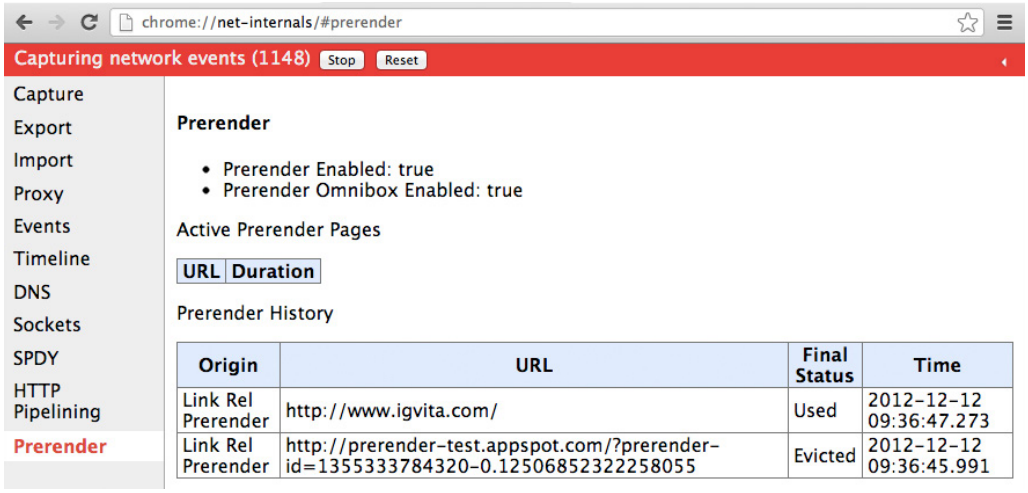
Každá optimalizace, kterou jsme až dosud probrali, pomáhá snižovat zpoždění mezi přímým požadavkem uživatele na navigaci a výsledným vykreslením stránky v jeho záložce. Nicméně, co je potřeba pro opravdu okamžitý dojem? Na základě dat z UX (user experience - dojem uživatele), které jsme viděli dříve, 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.