Mendelova univerzita v Brně
Provozně ekonomická fakulta

Webová architektura v prostředí vysoké zátěže

Diplomová práce

Vedoucí práce:
Ing. Michael Štencl, Ph.D.

Bc. Jakub Škrha

Brno 2012

Text poděkování

Text prohlášení

místo a datum prohlášení ....

Abstract

Citace práce v anglickém jazyce

abstrakt práce v anglickém jazyce

Abstrakt

Citace práce v českém jazyce

abstrakt práce v českém jazyce

Obsah

1 Úvod

Něco na téma jak vzniká vysoká zátěž na iternetu, jaký je vliv internetu, roustoucí zájem o internet a stoupající zátěž. Uvidíme, napíšu to jako poslední.

2 Nežádoucí vlivy a důsledky vysoké zátěže

Úspěch internetových a webových projektů je přímo úměrný výši návštěvnosti, používání a registracím a samozřejmě výdělku z aplikace. Obecně se dá předpokládat, že čím je větší návštěvnost projektu, tím jsou větší i zisky. A to už díky reklamní činnosti či placenných služeb. Ovšem zde se dá velice jasně konstatovat, že tyto výdělky nejsou až tak lehce získané. Nejenom, že si musí aplikace získat své uživatele, ale musí řešit problémy s obrovským počtem uživatelů, čili problémy s vysokou zátěží.

Vysoká zátěž může mít ve své podstatě několik nežádoucích vlivů, které můžou mít až katastrofický scénář. Může docházet k takovému zatížení aplikace, že odpovědi na jednotlivé požadavky mohou trvat velice dlouhou dobu. Tím pádem si uživatel může rozmýšlet, zda-li příště navštíví tuto webovou aplikaci, či zkusí některou z jiných možných konkurenčních alternativ. Takto velice nepříznivý scénář může být ještě horším. A to tak, že díky velkému zatížení dojde dokonce k výpadku celé aplikace, a tím pádem už se uživateli nedostane vůbec žádné odpovědi na jeho požadavek. Při takovém scénáři existuje vysoká pravděpodobnost ztráty a poklesu uživatelů, což může znamenat velký pokles zisků pro firmu či společnost.

Po technické stránce se při velké zátěži vytvoří pro každý požadavek celý samostatný proces či vlákno procesu, které si klade nároky na procesor CPU a operační paměť RAM serveru. Vznikají tak i procesy, které musí čekat na přidělení takových prostředků a tím pádem zatížení, které roste a čeká na vykonání může server přetížit až už nebude provozuschopný. Takovéto zatížení je možné pozorovat při výpisu právě běžících procesů a jejich vytížení na RAM či CPU (například příkazem top). Dále je možno pozorovat tzv. Load Average (například příkazem uptime), které představuje počet procesů čekajích na přidělení prostředků během jedné, pěti či patnácti minut. Takto je možné sledovat jaké jsou nežádoucí vlivy vysoké zátěže na technické úrovni.(Kyle Rankin, 2010)

Je nutné podotknout, že existuje i jeden skrytý a ne tak viditelný důsledek vysoké zátěže. Tím je fakt, že jakmile se začne zvedat návštěvnost uživatelů, a tím pádem i zátěž aplikace, projektové vedení si klade požadavek, aby tento nárůst uživatelů již zůstal, a naopak se dokonce i zvětšoval. A to vše z toho důvodu, že velký počet uživatelů, znamená velkou zátěž pro aplikaci, ovšem velký finanční přínos pro firmu.

3 Zneužití vysoké zátěže

Z nežádoucích vlivů a důsledků uvedených v předchozí kapitole vyplývá, že rostocí zátěž může způsobit katastrofické scénáře, čili může negativně působit na celou aplikaci. Tento poznatek představuje obrovské riziko při jeho zneužití. Zpravidla může být způsobeno úmyslným či neúmyslným chováním nějaké organizace či jedince. Takovéto zneužití má svoji oporu i v zákonech, kde hrozí až odmětí svobody několika let.

U webových projektů je možné z těch neúmyslných zmínit například nějaké klientské chyby programů, či větší míru indexace robotů jednotlivých vyhledáváčů. Roboti vyhledávačů pravidelně prochází webovou aplikaci a indexují její jednotlivé části, a tyto výsledky zohledňují následně ve svých výsledcích ve vyhledávání. Tito roboti mohou představovat nežádoucí zátěž.

Tím druhým a daleko nebezpečnějším úmyslným způsobem lze mnohdy způsobit daleko větší škody. Toto úmyslné chování už je klasifikováno jako útok na webovou aplikaci za který hrozí postih podle zákona. Tyto útoky jsou nazývány DoS, neboli Denial of Service. Jejich cílem je ochromit infrastrukturu celé webové architektury přehlcením požadavků na které aplikace bude vytvářet odpovědi. Tyty útoky využívají chyb, nedostatků či nedokonalostí protokolů ICMP, TCP, UDP a jiných protokolů či samotných webových aplikací. Například útok s názvem Tcp Syn Flood, kdy útočník vytíží aplikaci SYN pakety pro navázání spojení využívá nedokonalosti TCP protokolu. Dalšími útoky využívajích nedokonalostí mohou být ICMP Flood, Ping of Death, Smurf Attack, IP Spoofing, Fraggle Attack, Teardrop, Application level aj. Ovšem proti většině těchto útoků už dnes existuje ochrana ve formě servisních instalací jednotlivých systémů či aktualizací programů síťových zařízení a nebo lehkou konfigurací. (Faisal Khan, 2009)

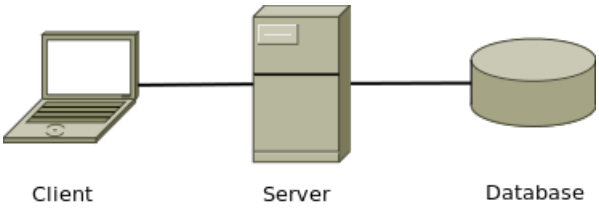
Ovšem co v dnešní době představuje daleko větší nebezpečí, jsou útoky typu DDoS, neboli Distributed Denial of Service. V tomto případě jako princip obdobný jako u DoS, ovšem s tím, že je úkol distribuovaný. Tedy je spuštěn z několika stanic, několika uživateli a pomocí různých nástrojů. Tento způsob je tedy daleko více organizovaný a daleko více nebezpečný a učinnější. (Faisal Khan, 2009)

Právě v dnešních dnech se stává symbolem boje za svobodu internetu skupina s názvem Anonymous, která využívá útoků DDoS. Při svých útocích využívají například útoky typu Slowloris, kdy útočník využívá protokolu HTTP na aplikační úrovni a chce celou odpověď na svůj požadavek. Při navázaném spojení ovšem odesílá HTTP hlavičky co nejpomaleji, aby tak co nejvíce prodloužili dobu spojení a získal prostor pro vytvoření dalšího spojení, čili další zátěže. Tato skupina napadá webové aplikace veřejnosti neoblíbených politických stran, vládních organizací, protipirátských asociací a jiných subjektů. Získávají si tím obrovskou podporu ve společnosti i médiích, která s jejich kroky souhlasí. Dokonce pro své útoky využívají i příznivců z řad veřejnosti, kteří nemusí odborníky informačních technologií. Stačí jim si pouze stáhnout upravený program, v určený čas ho spustit a připojit. Útoky probíhají hlášeně či neohlášeně, organizovaně a ditribuovaně. Otázkou zůstává, kdy

jejich konání přeroste z útoků pro dobro společnosti, a stanou se útoky pro vydírání, posílení moci, či za účelem finančního obohacení. V ten moment i společnost, která je v tyto dny podporuje, může pocítit, jak jsou pro ně nebezpeční. I v historii Země nalezneme spoustu skupin, které byly lidmi podporovány a nakonec se z nich stal symbol krutosti, tyranie, úzkosti a neštěstí. Proto je důležité jejich útoky nepodceňovat a umět se bránit. Má práce se nezabývá konkrétním řešením nějakého z typů útoků, ale zabývá se obecně vysokou zátěží, a jak ustát narůst obrovské zátěže a tedy i nějaký útok.(Pavel CČepskyý, 2012)

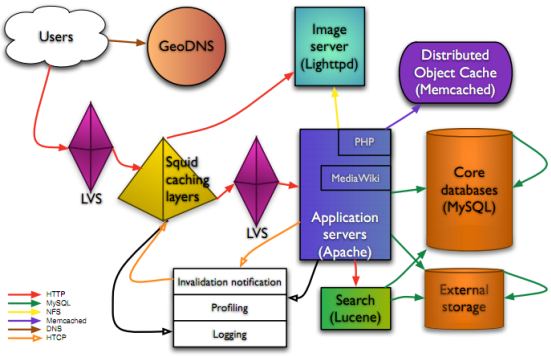
4 Tří a vícevrstvá architektura webových systémů

Webová architektura je ve svém základu třívrstvá. První vrstvu tvoří klient, neboli uživatel se svým Hardware a Software, a svými aplikačními požadavky. Druhou vrstvu tvoří aplikační server, který zpracovává požadavky aplikace, tedy požadavky klienta, zpracuje tento požadavek, vytvoří odpověď a zašle zpět klientovi. Ovšem k tomu, aby mohl tuto odpověď vytvořit, potřebuje i data aplikace, která jsou uloženy v perzistentní databázi, která tvoří třetí a poslední vrstvu třívrstvé architektury. Každá z vrstev, tedy prezentační, aplikační i datová má své místo a svou správu v aplikaci.(Jaroslav Zendulka, 2005)



Obrázek 1: Tří vrstvá architektura webových aplikací

V architektuře webových aplikací s vysokou zátěží už je potřeba jiného přístupu. V tomto případě se dá říci, že je třívrstvá architektura nedostačující. Je potřeba počítat se síťovými prvky pro load balancing, s více aplikačními stroji, s databázovými replikacemi, s DNS řešením pro geografické rozdělení zátěže, s CDN pro rozdělení zátěže přidělování obsahu, s vrstvami pro cache aplikace a s dalšími vrstvami pro backendové či frontendové aplikace a služby. V tomto případě neexstuje žádné jasně dané a pevné řešení, každá aplikace si s sebou nese své individuální a charakteristické řešení a strategii, i když některé osvědčené postupy se opakují. Tyto strategie už nesou název vícevrstvá architektura.



Obrázek 2: Webová architektura společnosti Wikimedia provozující Wikipedia.org

Nutkno podoknout, že webové architektury využívají nejčastěji ke své komunikaci mezi klientem a architekturou protokol HTTP, který využívá portu číslo 80 a protokolu TCP pro komunikaci. Proto je celá má odborná studie založena na práci s tímto protokolem.

5 Typy webového obsahu

Obsah webových aplikací je rozdělen na dva typy. Obsah může být buď statického nebo dynamického charakteru. Jak už vyplývá z obecné teorie systémů, statický obsah je konečný, transaprentní a měněn podle předepsaných pravidel, zatím co dynamický obsah se mění v závislosti na čase a jiných nepředvídatelných pravidlech.

Statickým obsah tedy můžeme chápat videa, obrázky, statické HTML stránky. Jedná se tak o obsah u které známe jeho přesnou životnost, nebo dokážeme určit kdy přesně se má měnit. Například invalidace u statického obsahu proběhne pouze jednou za deset minut. A i když je obsah změněn uprostřed tohoto intervalu, změna se projeví až v přelomu těchto pravidelných desetiminutových intervalů, tedy až po invalidaci. Obsah je tak celých deset minut statický a neměnný.

Naopak dynamický obsah se může změnit kdykoli, nepředvídatelně, typicky na nějakou uživatelovu činnost a nebo v závislosti na čase. Příkladem může být nákupní košík v internetovém obchodě, který se mění v závislosti na uživatelově činnosti. Nebo například při vložení komentáře pod určitý článek se tento komentář ihned zobrazí v seznamu komentářů. Takový obsah se pak těžko ukládá do cache pamětí, dokonce je i náročný i pro rozložení zátěže, poněvadž musí být zachováno uživatelovo sezení aplikace (session).

Z uvedených vlastností vyplývá, že správa statického obsahu je jednodušší, kdežto správa dynamického přístupu je náročnější. Dynamický přístup často předpokládá propuštění požadavku až na aplikační server, který určí výsledek. Statický obsah lze jednoduše ukládat do cache paměti, protože známe jeho přesnou dobu platnosti. Ve své práci ukazuji, analyzuji a testuji oba dva přístupy a srovnávám jejich přínos jak z pohledu aplikace, tak náročnosti na vybudování.

6 Aplikační vrstva

Aplikační vrstva představuje jádro webové architektury. Jejím účelem je příjmout a zpracovat klientův požadavek, vytvořit odpověď a tuto odpověď zaslat nazpět klientovi. Na aplikační vrstvu jsou tak kladeny úkoly celé režie procesu tvorby odpovědi, a tím pádem má velkou zoodpovědnost a mnohdy i největší zátěž.

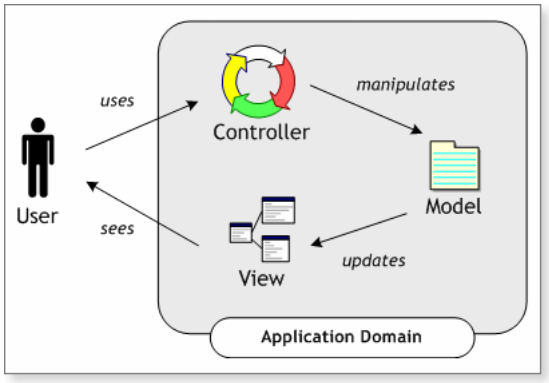
6.1 Webová aplikační architektura MVC

V dnešní moderní aplikační vrstvě se používá aplikační architektura návrhového vzoru MVC pro přehlednější a rychlejší způsob tvorby aplikace. Tato zkratka vychází z tří slov Model, View a Controller, které představují tři základní vrstvy aplikační architektury. Často bývá označovám i jako MVC framework, který rozděluje aplikaci do tří modulů. (Rudolf Pecinovskyý, 2007)

Controller je prvotní inicializační vrstva každého požadavku. Zpracovává příchozí data, parametry a atributy dané akce od uživatele, provádí jejich kontrolu a formátování. Stará se i o zabezpečení dané konkrétní akce vrstvy Controller. Často spolupracuje s vrstvou Model, které předává požadavky na data aplikace, a tyto data dále zpracovává pro předání do vrstvy View. (Rudolf Pecinovskyý, 2007)

Model má za úkol přistupovat k datovým úložištím, a to ať už k perzistentní databázi nebo souborovému systému, cache či jiným typům úložišť. Zapouzdřuje tak datovou logiku frameworku. Často se jedná o soubor dalších návrhových vzorů, kde se může vyskytnout přepravka (Crate) či jejich kolekce pro přenášení dat, zástupce (Proxy) pro přístup k implemtacím nad přepravkami, příkaz (Command) pro vykonání nějaké akce či příkazu, strategie (Strategy) pro určení nějaké konkrétního algoritmu ze skupiny algoritmů nad určitou úlohou, a mnohé další z návrhových vzorů. Modelová vrstva bývá označována za nejsložitější vrstvu, a právě proto je potřeba dodržovat techniky OOP včetně návrhových vzorů pro další možnou rozšiřitelnost a pro přehlednost. (Rudolf Pecinovskyý, 2007)

View klade důraz na presentační úroveň, tedy na grafickou a jinou interakci s uživatelem. Zpracovává tak výsledek práce vrstvy Controller nad vrstvou Model a zobrazuje výsledek určitých operací. Často využívá nějakých šablonovacích přístupů. (Rudolf Pecinovskyý, 2007)



Obrázek 3: Návrhový vzor MVC a jeho životní cyklus

6.2 Optimalizace aplikační vrstvy

Optimalizace na úrovni aplikační vrstvy může mít několik způsobů a přístupů. Tato činnost se týká převážně programátorů a softwarových inženýrů, kteří mají za úkol vývoj a údržbu aplikační vrstvy. K tomu, aby se dali identifikovat problematické části pro optimalizaci slouží tzv. profilery. Ty mají obecně za úkol vyprofilovat jednotlivé funkce, metody, procedury, dotazy a příkazy, které se na dané vrstvě, již je profiler určen, vyskytují, a určit jejich dobu trvání, počet volání, čas spuštění, závislosti a další parametry. Profilování, neboli určení kandidátů pro optimalizaci, je prvním a nejdůležitějším krokem pro optimalizaci aplikační vrstvy webové architektury. Další kroky se týkají především těchto oblastí:

- Výběr nejoptimálnějšího algoritmu pro danou úlohu
- Výběr nejrychlejšího MVC frameworku
- Vytváření cache souborů aplikace
- Způsob překladu a vykonání zdrojových souborů
- Přidání další vrstvy architektury - aplikační cache

6.3 Druhy aplikačních vrstev

Existuje celá škála různých programovacích jazyků a webových serverů pro implementaci aplikace. Každý z nich má své výhody a nevýhody, specifická řešení a přístupy. Uvádím zde krátký seznam těch v praxi nejběžněji se vyskytujících:

- Webový server Apache2 s programovacím jazykem PHP
- Java Servlets, Java Spring Source
- C# s technologií .NET

- Ruby on Rails
- Python a Django
- a mnohé další

6.4 Ajax a webové služby

Ajax, neboli Asynchronous JavaScript and XML, se dnes stává nedílnou součástí při vývoji webových aplikací. Aplikace tak dostávájí interaktivnější charakter bez nutnosti znovuzaslání celého požadavku webové aplikaci. Celý tento přístup probíhá nejvíce na straně klienta. Je použito javascriptu pro programovou implementaci, který má přístup ke stromu objektů dokumentu zvaného DOM, neboli Document Object Model. Do aplikační vrstvy jsem se rozhodl přidat AJAX z toho důvodu, že používá objekt XMLHttpRequest pro komunikaci s aplikačním serverem. Tyto aplikační požadavky jsou nazývány webovými službami pro Ajax. Tyto požadavky jsou vykonávány na aplikační vrstvě a představují potenciální zátěž, která musí být i v některých případech optimalizována.(Brett McLaughlin, 2005)

7 Dabázová vrstva

Úkolem databázové vrstvy ve webové architektuře je zajišťovat datové služby a uchovávat tak aplikační data perzistentní. V oblasti webových architektur se nejčastěji vyskytují relační databázové systémy, a proto i má práce je soustředěna na tento typ databázových systémů. Databázový systém obecně tvoří databáze, jakožto skupina strukturovaných homogenních souborů, a SŘBD, neboli Systém řízení báze dat, jakožto integrovaný softwarový prostředek řídící bázi dat.

7.1 Optimalizace SQL dotazů

Optimalizace dotazů SQL je nedílnou součásti procesu práce s databázovým systém v prostředí vysoké zátěže. Je totiž důležité nejenom si umět získat potřebná data, ale je potřeba zvážit i za jakou cenu tyto data prostřednictvím databázavého systému získáváme. Hovoříme-li o webových architekturách s vysokou zátěží, je tento proces optimalizace velice důležitý. Každá operace, každý dotaz, každá akce potřebuje ke své realizaci určité hardwarové a systémové prostředky, a v prostředí vysoké zátěže je důležité ušetřit co nejvíce těchto prostředků.

K tomu, abychom mohli vůbec přistoupit k optimalizaci SQL dotazů, je potřeba určit a identifikovat, které tyto dotazy jsou opravdu náročné na prostředky a čas, neboli mají vysokou cenu. K tomu slouží tzv profilery (viz. kapitola 6.2). Profilery mohou být určeny pro aplikační vrstvu, kde profilují nejenom zdrojové kódy aplikace, ale samozřejmě i databázové dotazy, které jsou z této aplikační úrovně spuštěny. Tímto způsobem je možné získat přehled všech operací, které probíhají na aplikační i databázové vrstvě, poněvadž tyto vrstvy spolu neúzce souvisí a spolupracují. Další možností je použít profiler určený přímo k databázové vrstvě. Takový profiler pak profiluje pouze databázovou vrstvu, jednotlivé databázové dotazy, jejich cenu, dobu trvání, a jiné další statistiky.

Každý SQL dotaz má nějaký svůj exukuční plán. Databázový systém po obdržení SQL dotazu vybírá z několika možných exekučních plánů ten nejoptimálnější, který je po té v databázi proveden. Při výběru exekučního plánu je brán v potaz výběr indexu a způsob skenu tabulek, vybraná spojení, aj. Exukuční plán je možné zobrazovat v mnoha databázových systémech pomocí EXPLAIN a identifikovat tak místa exekučního plánu, která mohou být kandidátem pro optimalizaci.(Bohdan Blaha, 2007)

Pro optimalizaci SQL dotazů je možné určit několik základních oblastí, na které je možné se zaměřit při konkrétní optimalizace určitého SQL dotazu:

- Normalizovaný databázový návrh
- Vnořené SQL dotazy
- Indexace, výběr indexu a způsob prohledávání
- Výběr druhu a pořadí spojení

- Způsob používání podmínek, klauzulí a operátorů

7.2 Indexace

Indexace je důležitá a nejefektivnější optimalizace dotazů SQL. Při průchodu dat tabulkou má databáze na výběr několik možností prohledání. První možností je prohledat všechny řádky tabulky podle sql podmínek. To je nazýváno obecně Full Table Scan, nebo také Sequence Scan, neboli sekvenční prohledávání. Další možností je pužití některého z indexů pro přístup k hodnotám namapovaných na jejich ROWID, které ukazuje na fyzické uložení. Toto prohledání bývá nazýváno jako Index Range Scan, nebo jen obecně Index Scan, neboli indexační prohledávání. Samozřejmě prohledávání tabulek pomocí indexace je výrazně rychlejší a tím pádem důležité pro opimalizaci SQL. (Bohdan Blaha, 2007)

Indexy jsou fyzicky i logicky uloženy v asociativních tabulkách, a díky tomu tak i odděleny od datových tabulek. Čili při smazání indexů se nesmiží ani nijak neovlivňí datové tabulky. Pouze se může zpomolit přístup k datům, který byl rychlejší pomocí těchto indexů. Tabulky s indexy jsou samozřejmě uloženy na disku, poněvadž jejich velikost je obrovská a nevešly by se do operační paměti RAM. Operační paměť a přístup k ní je daleko rychlejší než přístup k datům uložených na disku, a proto je potřeba volit nějaký vhodný algoritmus prohledání a přístupu k indexům, a od toho se odvyjí i název a druh používaných indexů. V každém databázovém systému samozřejmě naleznete některé typické a některé atypické druhy indexů. (Bohdan Blaha, 2007) Zde je krátký výběr možných indexů:

- B-tree - pro přístup pomocí Root-Node-List
- Bitmap - pro výčtové sloupce
- R-tree - typ indexu optimalizovaný pro geometrická data.
- GiST - zobecněný vyhledávací strom
- a další

7.3 Partitioning

Partitioning, který je občas do češtiny překládán jako segmentace, občas jako škálování, slouží v relačních databázových systémech k rozdělování tabulek a indexů do menších částí a komponent. Díky tomu je pak činnost databáze rychlejší a snadnější. Při této segmentaci tak může dojít k rozdělení tabulek i na více pevných disků či serverů. Tyto segmenty jsou na sobě nezávisle, ale přitom je k nim možné přistupovat přes tabulku, pro kterou byla segmentace vytvořena. Databázová tabulka a její vlastnosti, jako například referenční integrita nebo žádná redundance, jsou stále zachovány a fungují přes všechny její segmenty. Dokonce i když dojde k selhání či výpadku jednoho ze segmentů, ostatní jsou stále přístupné a je možné s

nimi pracovat. Partitioning je možné provádět na několika úrovních a podle různých klíčů. U segmentovaných tabulek je tak důležité si rozmyslet jakou strategii si zvolit.

Při vertikální segmentaci dojde k segmentaci podle definovaných sloupců databázové tabulky. Klíčem při tomto rozdělení je určení sloupečků, které se nepoužívají ve where klauzuli, nebo jsou prázdné či zřídka používané.

Častěji používáným přístupem je horizontální segmentace tabulek, čili segmentace podle řádků. Zde se segmentují řádky, podle určité hodnoty databázového sloupce. To do jakého segmentu bude řádek tabulky vložen rozhoduje nějaký interval, či hodnota výčtu a nebo nějaká funkce.

Další možností je aplikační úroveň segmentace, která se ne často objevuje v souvislosti s Partitioning. Nejedná se totiž o segmentaci určité databázové tabulky, ale o segmentaci databáze. Část tabulek je umístěna na jednom serveru, část na dalším serveru, a tak dále.

Partitioning je důležitým nástrojem při optimalizaci databázové vrstvy v architektuře webových systémů v prostředí vysoké zátěže. Dá se totiž předpokládat, že se zvětšující se zátěží roste i počet záznamů tabulek, a tak se doba přístupu zvětšuje a prostředky zatěžují ještě víc. Tyto problémy dokáže vyřešit partitiong. (Eli White, 2009)

7.4 Replikace

Replikací rozumíme technologii, kdy je možné nasadit více databázových serverů v rámci jedné databáze. Jedná se tak o sdílení dat mezi více hardwarovými, softwarovými a jinými prostředky a jejich přenositelnost. Účelem replikace je tak dosáhnout vysoké dostupnosti databázového systému a škálování výkonu pro optimalizaci v prostředí vysoké zátěže. Obecně existují dvě základní varianty databázových replikací od kterých se odvyjí jejich další využití. Samozřejmě v závislosti na konkrétním databázovém systému pak existují další členění a nastavení.

Replikace varianty master-slave je podporována ve většině databázových systémech. Jedná se o jednodušší tzv. jednosměrnou replikaci. V této variantě je určen autonomní prvek, jedna replikace, která akceptuje a zpracovává požadavky na změny. Takováto replikace nese název master. Prvek s názvem slave je věrnou kopií autonomního prvku master. Slouží pouze ke čtení a může jich být více pro jeden master. Jakmile master obdrží a zpracuje požadavek na změnu, tak jej po dokončení přenese na ostatní slave replikace.

Replikace typu master-master bývá označována jako obousměrná. To znamená, že jsou v rámci jednoho databázového systému minimálně dvě replikace typu master, které akceptují všechny druhy požadavků na změny i čtení a přenáší je vzájemně mezi sebou. Z této vlastnosti vyplývá, že může dojít ke kolizím, kdy například dvě replikace master zapisují do stejné tabulky. Takovéto kolize jsou nevyhnutelné, a je potřeba je řešit.

Způsob přenosu mezi jednotlivými replikacemi může být synchronní či asynchronní. U synchronního přenosu se čeká až se změny provedou na všechny ostatní

repliky. Takovýto proces je časově náročný, ovšem na druhou stranu je celý databázový systém konzistentní jako celek. U asynchronního přenosu se nečeká na dokončení přenosu mezi ostatními replikacemi. Díky tomu je celý databázový systém rychlejší, ovšem může dojít k nekonzistenci, kdy na ostatní replikace ještě nejsou přenesena všecha data.

Administrace, nástroje a konfigurace replikací jsou zabudované v téměř každém databázovém systému. Je důležité ale poznamenat, že tyto nástroje nejsou mnohdy dostačujícími řešeními pro architektury v prostředí vysoké zátěže a je proto nutné používání jiných doplňkových nástrojů. Také je více než důležité říct, že v prostředí vysoké zátěže se webová architektura bez databázových replikací jen těžko obejde.(Tomaášš Vondra, 2011)

7.5 Druhy relačních databází

V dnešní době existuje několik druhů relačních databázových systémů. Každý z nich má své klady a zápory, ovšem princip a způsob práce těchto databází je v základu podobný. Uvádím zde přehled těch v praxi se běžně vyskytujících:

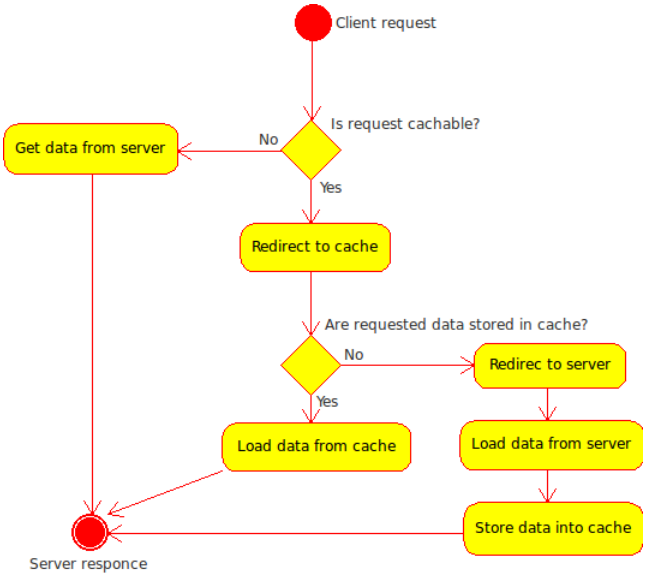
- Oracle
- MySQL
- PostgreSQL
- MSSQL
- Firebird
- a mnoho dalších

8 Webové cache

Webové cache jsou důležitou vrstvou pro rozšíření výchozí třívrstvé architektury. Jak už napovídá z názvu cache, jedná se o vyrovnávací paměť jejíž hlavní účel je zrychlit odpověď webové aplikace na požadavek klienta. Zároveň je důležité poznamenat, že nefunkčnost, zánik či pád cache vrstvy nesmí nijak ovlivnit chod apikace, která musí být funkční i nadále. Účel této vrstvy je pouze zrychlit přístup k datům či již jednou interpretovaných odpovědí webové aplikace.

Webová cache se tak nachází mezi klientem a aplikační vrstvou. Základní komunikace probíhá tak, že klient pošle HTTP požadavek, aplikační server ho příjme a navrátí HTTP odpověď. Při existenci cache vrstvy probíhá komunikace jiným způsobem. Klient pošle HTTP požadavek a webová architektura zjistí, zda-li je tento HTTP požadavek určen pro ukládání do cache paměti. Pokud ano, tak webová architektura zkusí získat odpověď z cache paměti. Když tuto odpověď v paměti nalezne, zašle jej přímo klientovi. V opačném případě je požadavek poslán do aplikační vrstvy architektury, která jej zpracuje, vytvoří odpověď kterou uloží do cache paměti a pošle klientovi. Při příštím stejném požadavku bude odpověď vrácena z paměti cache. Dojde-li ke změně dat, která jsou uložena v cache paměti, dojde k invalidaci uložených dat v cache paměti a celý proces začne od začátku s novým HTTP požadavkem.

Pro webové architektury v prostředí vysoké zátěže je tato vrstva nevyhnutelná. Odpovědi z cache vrstvy jsou daleko rychlejší, dokonce až mnohonásobně, než odpovědi z aplikační vrstvy. Ve své práci se zabývám právě různými druhy cache a z výsledků profilování a testování vyplývá, že jsou tato tvrzení pravdivá.



Obrázek 4: Diagram aktivit cache webových systémů

8.1 HTTP hlavičky pro ovládání cache

Komunikace ve webových aplikacích probíhá ve valné většině na úrovni protokolu HTTP. Každý HTTP požadavek obsahuje HTTP hlavičky, kterými určuje požadavek, klienta a obsahuje časové razítko. Odpověď na tento požadavek navrací v hlavičkách návratový kód, informace o serveru, časové razítko, typ obsahu odpovědi, délku odpovědi a informace pro případné ukládání do cache paměti. Právě jednotlivé cache hlavičky jsou důležité pro nastavení ukládání a invalidace do proxy cache paměti nebo do reverzní proxy cache paměti.

Existují dva základní modely pro určení práce s cache vrstvou. Prvním je expirační model. Tento model určuje, do kdy je platná HTTP odpověď, neboli do kdy může tuto odpověď cache vrstva ukládat a označovat ji jako platnou a stále čerstvou. Tento model může být realizován dvěma způsoby. Prvním je hlavička Expires udávající časové razítko do kdy je odpověď čerstvá. Druhým, novějším, daleko flexibilnější a konfigurovatelnější způsobem je použití Cache-control. Díky tomuto způsobu je možné určit další parametry a to nejenom do kdy je daná HTTP odpověď platná. Je možné tím i určit, zda-li je určena jen pro sdílené cache (proxy cache), nebo jen pro uživatelské prohlížečové cache, nebo zda-li je ukládat či nikoli, atd.(RFC:2616, 1999)

Druhým modelem je model validační. Ten určuje způsob komunikace pro zjištění, zda-li je odpověď uložená v cache paměti stále validní. Základem je, že cache, která má uloženu odpověď se zeptá je-li odpověď stále validní a server odpoví jestli ano či vrátí novou čerstvou odpověď. Realizováno to může být opět dvěma způsoby. Tím prvním je určení tzv. Last-Modified. Při prvním požadavku cache získá odpověď s touto hlavičkoku. Odpověď si uloží a při dalším požadavku se cache vrstva dotáže serveru pomocí hlavičky If-Modified-Since s časovým razítkem, zda-li je odpověď stále validní. Server odpoví návratovým kódem 304, který říká že nedošlo k žádné změně od daného časového razítka, a nebo vrátí novou čerstvou odpověď pokud došlo ke změně. Podobný princip je založen i na způsobu pomocí hlavičky Etag neboli Entity tag. Etag je nějaký unikátní identifikátor vygenerovaný serverem pro danou HTTP odpověď. Cache vrstva si při prvním požadavku uloží odpověď s tímto Etag. Při dalším požadavku pošle na server HTTP požadavek s hlavičkou If-None-Match s tímto Etag. Pokud se Etag shodují, vrátí aplikačnín server odpověď s návratovým kódem 304, a nebo celou čerstvou odpověď.(RFC:2616, 1999)

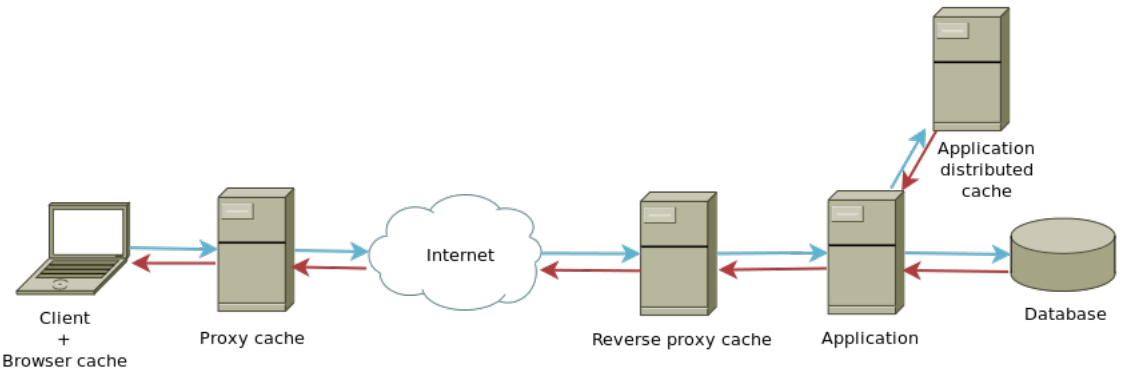
Ovládání pomocí HTTP hlaviček slouží hlavně k statiskému obsahu. Pro tento účel je pak dobré zvážit jestli je lepsí odpovědi měnit v nějakých časových intervalech a používat tak expirační model, či jestli se nevyplatí nasazovat model validační, který nese složitější administraci.

Příklad HTTP hlaviček pro HTTP odpověď:

```
HTTP/1.1 200 OK
Date: Mon, 23 May 2005 22:38:34 GMT
Server: Apache/1.3.3.7 (Unix) (Red-Hat/Linux)
Last-Modified: Wed, 08 Jan 2003 23:11:55 GMT
Etag: "3f80f-1b6-3e1cb03b"
Accept-Ranges:  none
Content-Length: 438
Connection: close
Content-Type: text/html; charset=UTF-8
```

8.2 Druhy cache

Ve webových architekturách exstuje několik druhů cache vrstev. Jejich členění se odvyjí od jejich účelu, pozici v architektuře a typu ukládaných dat. Tyto atributy mezi sebou neúzce souvisí. Základním členěním může být jejich pozice z pohledu internetu, a to na klientskou a serverovou část. Za klientskou část považujeme cache systém uživatelova prohlížeče, či systémovější proxy cache. Na straně serveru se vyskytují reverse proxy cache či distribuované cache.



Obrázek 5: Druhy cache ve webové architektuře

8.2.1 Proxy cache a cache prohlížeče

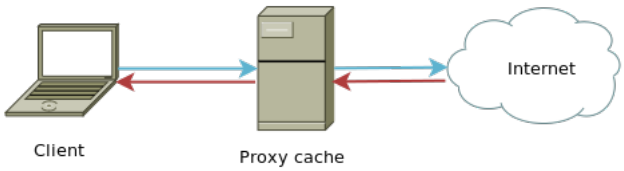
Tyto druhy cache se týkají klientské části webové architektury, neboli části před internetem. Znamená to tedy, že odpověď na klientův požadavek je vrácena z cache paměti lokální sítě, a tento požadavek vůbec nevstoupí do internetu na vzdálený dotazovaný server. Tyto cache vrstvy zde nejsou jen pro to, aby zrychlily odevzvu na požadavek, ale také aby zmenšily odchozí komunikaci z lokální sítě. A to z toho důvodu, aby byla šířka pásma přístupná i jiným službám, a ušetřili se prostředky za odchozí komunikaci.

V dnešních moderních prohlížečích je práce s cache pamětí podporována. Cache prohlížeče může nastavovat, konfigurovat, či vymazat pouze uživatel, neboli klient sám. Tato paměť ukládá hlavně statický obsah, například obrázky, CSS soubory či JS soubory. Cache paměť najde své uplatnění například když jsou uživatelovi

požadavky směřovány jedné webové aplikaci se stejným vzhledem, rozhranním a nebo částí funkcionalit. Takováto data mohou být ukládána do cache paměti na dlouhou dobu, a proto musí aplikace dbát na to, aby byla tato data z cache promazávána tak jak je opravdu potřeba. Jednou z možností je určit jinou url adresu pro stejný soubor, pomocí tzv. Query string, například takto:

```
<link rel="stylesheet" type="text/css" href="style.css?QUERY_STRING" media="all" />
```

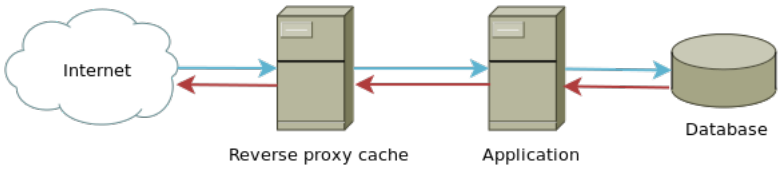
Proxy cache bývají budovány a instalovány providery a poskytovali připojení k internetu. Jejich záměr je jednoduchý. Jde o to snížit míru odchozí komunikace, nezatěžovat tolik šířku pásma a zmenšit tak náklady spojené s touto komunikací. Nejběžnějším nastavením těchto proxy cache pamětí bývá ukládání a invalidace obsahu v závislosti na HTTP hlavičkách přijaté odpovědi. Nainstalujeme-li například takovouto cache ve firmě či nějakém sídlišti s několika desítky, sty či dokonce až tisíci uživateli, je možné, že část z nich má každé ráno stejný požadavek na stejný zpravodajský server, tudíž mají i stejnou odpověď. A tak může být první odpověď uložena do cache paměti a těm dalším uživatelům může být odpověď servírována z proxy cache paměti.(Mark Nottingham, 2012)



Obrázek 6: Proxy cache ve webové architektuře

8.2.2 Reverzní proxy cache

Reverzní proxy cache, někdy označovány jako tzv. Gateway cache, mají podobný systém činnosti jako proxy cache. Mají za úkol ukládat statický obsah ve formě HTTP odpovědí a tyto odpovědi vracet nazpět odkud přišel jejich požadavek. Úkolem je tak pokud možno nepropustit požadavek dále na aplikační vrstvu, stejně jako úkolem proxy cache je pokud možno nepropouštět požadavek dále do internetu. Rozdíl mezi reverzní proxy cache a proxy cache spočívá v jejich pozici ve webové architektuře, a také v tom, kým jsou instalovány. Pozice reverzní proxy cache je na straně serveru, neboli blíže aplikační vrstvě architektury, tedy na druhé straně internetu, než odkud přichází požadavky klientů a kde jsou instalovány proxy cache. Reverzní proxy cache jsou instalovány samotnými administrátory a programátory webové aplikace. Jejich účel je zajistit rychlejší odezvu webové aplikace na HTTP požadavky klientů. Reverzní proxy cache jsou nedílnou součástí webové architektury v prostředí vysoké zátěže a bez jejich existence by jen těžko mohla být webová aplikace spolehlivá a rychlá.(Mark Nottingham, 2012)



Obrázek 7: Reverzní proxy cache ve webové architektuře

8.2.3 Aplikační distribuovaná cache

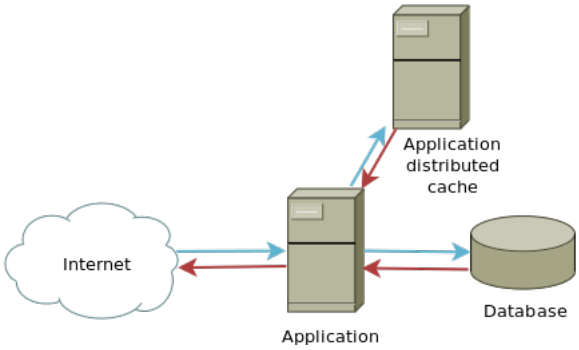
Doposud byly popisovány cache paměti pro ukládání statického obsahu. Ovšem v praxi se od projektového vedení setkáme s požadavky na dynamické chování aplikace. Příkladem může posloužit opět situace s komentáři, kdy jeden uživatel vloží komentář k některému z článků. Tento komentář se musí okamžitě objevit v seznamu komentářů k článku. V případě statického obsahu, který se mění pouze v nějakých intervalech, může uživatel nabýt dojmu, že tento komentář se nepodařilo vložit a zkusí ho vložit znovu. Takovéto chování aplikace určitě není v souladu s dobrým chováním interakce mezi uživatelem a aplikací. K tomu, aby se dala data ukládat do cache a aplikace byla stále dynamická, slouží tzv. aplikační distribuovaná cache. HTTP požadavek je tak rozdistribuován mezi menší požadavky a mezi cache paměti.

Jak už vyplývá z názvu druhu této cache, o manipulaci s touto cache pamětí se stará aplikační vrstva. Standartně se ukládají do cache paměti ta data, k nimž je dlouhá doba přístupu, obvykle data z databáze. Účelem je tedy ušetřit spojení a dotazy nad databází. Jeden HTTP požadavek s konkrétní url je často rozdělen na několik požadavků do databáze, v závislosti na tom, o která data se jedná. Například v rámci jedné HTTP stránky může být databáze dotázána o požadovaný článek a seznam komentářů pod článkem. Databáze tak obdrží dva SQL dotazy, na které vrátí příslušná data z databáze. Aby se příště aplikace nemusela na tato data dotazovat, uloží si je do aplikační cache paměti.

Při ukládání takových dat do cache paměti aplikační vrstva určuje jak a kam mají být data uložena, a kdy a za jakých podmínek mají být invalidována. Při aplikaci této cache vrsty dochází k propuštění požadavku na aplikační servery, což znamená větší zátěž aplikační vrstvy. Reverzní proxy cache jsou v tomto směru nepoužitelné, slouží pouze pro statický obsah a všechny požadavky propouští dále na aplikační vrstvu.

Při práci s aplikační distribuovanou cache je důležité si uvědomit několik základních pravidel. Prvním pravidlem je ukládat data do cache paměti co nejmenší. Účelem je tak distribuovat jednotlivá data v různých kontextech. Dále je důležité ukládat data znovuzkonstruovatelná. Je důležité umět z cache paměti přečíst to, co do ní bylo uloženo. Nejlepší je ukládat data serializovaná. Dále je důležité dodržet jednoznačný a jasně definovaný klíč či jmenný prostor pro tato data. Je totiž důležité vědět kde a jak se ptát na data. Největším problémem je invalidace dat v cache

paměti. Stejná data mohou být interpretována v různém kontextu. Například při vložení nového článku se změní data pro seznam všech nejnovějších článků, a také data pro seznam nejnovějších článků v dané rubrice. Aplikace má za požadavek dynamické chování, a tak musí být tento článek vidět ihned po vložení ve všech zmíněných kategorií. Čili musí dojít k invalidaci všech závislostí. Tento problém je označován jako invalidační kaskády, a proto je důležité konstruovat aplikaci tak, aby se těmto kaskádám vyhýbalo co nejvíce či s nimi počítat už pří návrhu aplikační cache vrstvy.



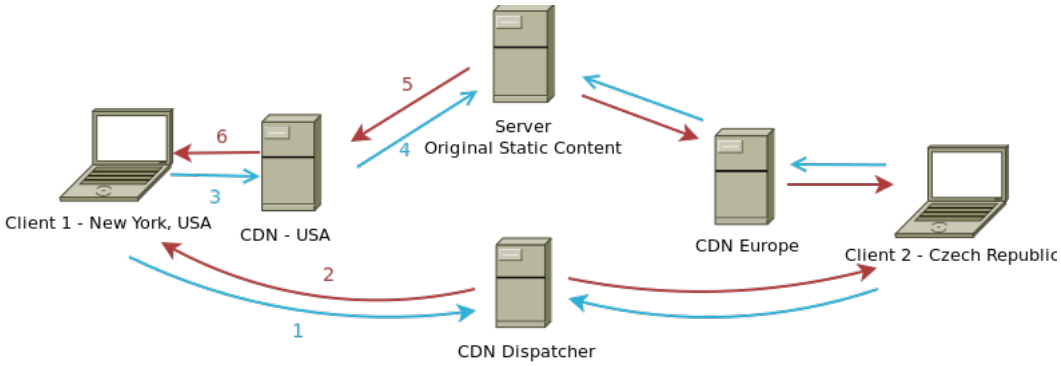
Obrázek 8: Aplikační distribuovaná cache ve webové architektuře

9 CDN

Vrstva CDN neboli Content Delivery Network je důležitou součástí webové architektury v prostředí vysoké zátěže. Jejím úkolem je rozdělit zátěž webové architektury mezi více uzlů. Jedná se o několik serverů, které mají za úkol klientům předávat vesměs statický obsah ve formě multimédií, stream videa či softwarových aktualizací.

CDN mají obvykle jeden centrální prvek, který rozhoduje o tom na který z CDN uzlů se požadavek přesměruje. Rozhodovat se může v závislosti na spoustě ukazatelů, a záleží už na konkrétním využití CDN. Rozhodovat se mohou například na základě geografického rozdělení. Centrálním prvkem je tak DNS server, který má přehled o kompletní topologii sítě CDN. Má tak přehled o tom, který ze serverů se kde nachází. Dalším ukazatelem pro centrální prvek může být stav serverů, jejich vytížení operační paměti RAM a procesoru CPU, stav odesílaných paketů na síťových kartách, stav cache pamětí apod.

Existují komerční i open source CDN sítě, větší společnosti pak mývají vlastní řešení. Mezi CDN se objevují i možnosti pro servírování dynamického obsahu, ovšem primární účel by měl být obsah statický. CDN najdou obrovské uplatnění tam, kde dochází k přenosům velkých souborů, které se už nijak nemění, a nebo se mění ve velkých intervalech.(Pavel Mikulka, 2008)



Obrázek 9: Ukázka komunikace klienta s CDN sítí

10 Virtualizace

Virtualizace je moderní pojem, který umožňuje lepší škálovatelnost aplikace a možnost práce s jednotlivými vrstvami webové aplikace a jejími servery. V dnešní době se nejvíce hovoří o tzv. virtualizaci platforem, což znamená provozování více virtuálních počítačů s vlastním operačním systémem na jednom fyzickém počítači. Takto virtualizované počítače i včetně operačních systémů a aplikací jsou od sebe navzájem izolovány.

Příklady profesionálních nástrojů pro virtualizaci počítačů jsou VMWare, Hyper-V a nebo třeba Citrix Xen Server. Základem virtualizace je fyzický server, který je pod správou vrstvy hypervizor, která řídí přístup k fyzickým zdrojům a rozhraním. Takto přípravený fyzický server se nazývá host. Několik hostů může být spravováno v jedné skupině nazývané cluster, která má své zdroje. To vše bývá spravováno centralním serverem pomocí desktopového klienta či webového rozhraní.

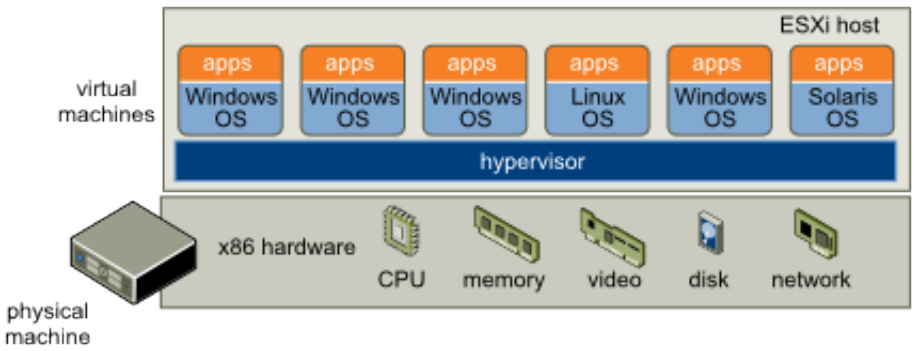
Virtualizace přináší spoustu výhod. Jednou z nich je elastičnost zdrojů. Jakmile se totiž některý z virtualizovaných serverů jeví jako velice vytížený, je možné mu během pár vteřin přidat chybějící prostředky. Stejně tak je možné tyto prostředky ubrat u serverů, kteří jej nevyužijí.

Pro webové architektury tou největší výhodou je konsolidace serverů, neboli provozování většího počtu virtualizovaných serverů na menším počtu serveru fyzických. Tohoto se hodně využívá ve webových architektůrách. Vzhledem k náročnosti jednotlivých vrstev webové architektury se v dnešních dnech používá přístupu s více virtualizovanými servery, které mají mnohdy jediný účel. Například jeden server pro datábázi, další pro webový server, mailový server, cache servery, apod.

Další výhodou může být redukce hardware, el. energie, prostor a dalších zdrojů. Bez virtualizace by přidání dalšího fyzického serveru znamenalo nakoupení HW komponent, vytvoření dalšího místa v serverovně a přivést další zdroj elektrické energie. Pro virtualizaci nového serveru stačí vyhodnotit stávající využití fyzických prostředků jinými virtuálními servery, dále zhodnotit rezervy na fyzických serverech, popřípadě dokoupit už jen dílčí komponenty jako například operační paměť RAM.

Obrovskou výhodou virtualizace je možnost sjednocení vývojového, testovacího a produkčního prostředí. Díky virtualizaci získáme během pár okamžiků věrnou kopii některého ze serverů a stejně tak je možné ho velice rychle přenést na jiný fyzický server. Dokonce se může tento přenos obejít i bez výpadku serveru, pokud jsou ve stejném clusteru.

Virtualizace dnes znamená obrovské ulehčení pro administraci jednotlivých serverů, jejich konfiguraci, rozšiřitelnost, apod. Dokonce právě vznikající trend s názvem Cloud Computing vychází z virtualizace a jeho modernější oblastí. Proto je důležité pro webovou architekturu v prostředí vysoké zátěže virtualizovat její jednotlivé vrstvy se servery.(Petr Hruůsša, 2011)



Obrázek 10: Ukázka virtualizovaných systémů v prostředí na jednom fyzickém serveru

11 Rozložení zátěže

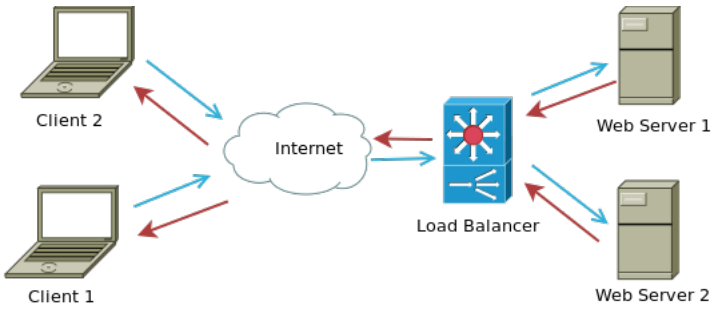
Rozložení zátěže, neboli load balancing, se používá v momentě, kdy zátěž webové architektury přeroste možnosti a výkonnost nějakého ze serverů. Tím pádem je nutné přidat další server a zajistit tak rovnoměrné rozložení zátěže mezi tyto servery.

Nejznámějším způsobem rozložení zátěže je tzv. Round Robin DNS. Při dotazu klienta na IP adresu cílové serveru, DNS server vrátí náhodný záznam jednoho z několika cílových serverů. Tímto způsobem je zátěž rozdělena náhodně, čili zhruba rovnoměrně mezi všechny servery. Zde nastává problém se sezením mezi aplikací a klientem, tzv. session. Pokud se klient připojuje k různým serverům zcela náhodně, tak není zaručeno že bude mít vždy stejné sezení s aplikací. Je totiž důležité, aby například když se uživatel přihlásí do aplikace, tak se uloží informace o přihlášení do jeho sezení a při zobrazení příští stránky se po něm nevyžadovalo opětovné přihlášení. V tomto případě náhodného přidělování koncových serverů je potřeba ukládat tato sezení do nějakého společného úložiště pro všechny koncové uzly.

Další možností je řešit rozložení zátěže na úrovní transportní vrstvy. K tomu slouží HW load balancery, které rozhodují na který ze serverů se požadavek přepošle dál. Tato profesionální zařízení už mohou rozhodnout v závislosti na jiných parametrech jako aktuální počet dotazů, přenášených dat, vytíženosti šířky pásma, zdrojové IP adresy, její geografické polohy, apod.

Existují i aplikační load balancery. Ty najdou své uplatnění ve webových architekturách. Jedná se tak o nějaký server s běžícím softwarovým load balancerem, který přijme a zpracuje požadavek a rozhodne, kterému z koncových serverů ho předá dál. Aplikační load balancer by proto neměl mít problém zajistit klientovi vždy stejný koncový server.

Ve webových architektůrách v prostředí vysoké zátěže je zcela nevyhnutelné použití více uzlů, například více aplikačních serverů. Vrstva pro rozložení zátěže je tak nedílnou součástí webové architektury v prostředí vysoké zátěže.



Obrázek 11: Ukázka jednoduchého rozložení zátěže

12 Další vrstvy aplikace

V prostředí vysoké zátěže je velice důležitá škálovatelnost. Ta vzniká i přidáváním dalších samostatných vrstev a ubírání tím tak práce jiným vrstvám. Proto je dobré se zaměřit i na další vrstvy, další aplikace, služby a možnosti, které mohou běžet samostatně bez nutnosti režie a zásahu jiných vrstev, a přítom mohou hodně odlehčit dalším vrstvám.

Jednu z možností pro přidání další vrstvy aplikace a odlehčení tak jiné jsou No SQL databáze. Jak už bylo nastíněho v kapitole 7, jedním z největších zdrojů zpoždění ve webových architektrách v prostředí vysoké zátěže mohou být databáze. Proto je vhodné kombinovat toto primární úložiště i s jinými možnostmi. NoSQL databáze jsou od klasických relačních databázi odlišné tím, že nemají žádnou referenční integritu, pevnou strukturu záznamů, nezabraňují redundanci, používají jiný jazyk pro dotazování a jsou volnější a flexibilnější. Jejich uplatnění se nalezne pro data, ve kterých není potřeba uchovávat nějakou větší logiku, a se kterými není potřeba provádět složitější operace. Mohou se hodit i k uchovávání dat k některým statistikám pro analýzy v business inteligence.

Většina webových aplikací podporuje funkci pro vyhledávání. Ať už se jedná o kompletní prohledávání všech dat aplikace či pouze o našeptávač, tak je asi zbytečné těmito akcemi zahlcovat databázový čas a prostředky. Existují totiž i jiné možnosti, jiné aplikace, které zastávají funkčnost vyhledávání. Těmto aplikacemím či službám se předávají data k prohledávání, které si samy prohledávají, indexují a vytváří další statistiky pro vyhledávání a nezatěžují tím aplikační či databázovou vrstvu.

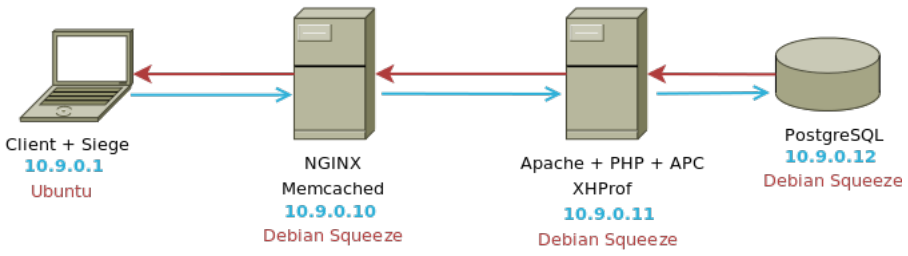
Určitě existuje celá řada dalších aplikací a služeb, které pomůžou většímu rozdělení a škálování aplikace na jiné další části, které dokáží fungovat nezávisle na sobě. Důležité je totiž co nejvíce odlehčovat vrstvám webové architektury v prostředí vysoké zátěže.

13 Praktická část s experimenty a výsledky

V praktické části se zaměřuji na testy, optimalizaci jednotlivých vrstev webové architektury v prostředí vysoké zátěže. Ve své práci rozebírám a experimentuji s jednotlivými vrstvami architektury, tedy aplikační vrstvou, databázovou vrstvou, reverzní proxy cache vrstvou a aplikační distribuoavanou cache vrstvou. Součástí práce nejsou pouze popis a charakteristika jendotlivých kroků optimalizace, ale i jejich výsledné statistiky a zátěžové testy. Hlavní částí praktické části je ovšem návrh a implementace aplikační úrovně plně podporující práci s aplikační distribuovanou cache. Dále je popsána i výsledná konfigurace a nastavení jednotlivých vrstev architektury. Výsledkem je tedy kompletní použitelná studie řešící problematiku webové architektury v prostředí vysoké zátěže.

13.1 Vrstvy webové architektury

Webová architektura sestává z klientské vrstvy, reverzní proxy cache vrsrtvy, aplikační vrstvy, aplikační distribuované cache a databázové vrstvy. Klientská část je tvořena jedním uživatelem s operačním systémem Linux Ubuntu a zátěžovým testovacím nástrojem Siege. Reverzní proxy cache vrstva je realizována programem Nginx. Aplikační vrstva se skládá z webového serveru Apache2 s programovacím jazykem PHP a s profilovacím nástrojem XHProf. Aplikační distribuovaná cache vrstva je tvořena programem Memcached. Databází je PostgreSQL. Všechny tyto vrstvy serverové části jsou vytvořeny na zvláštních serverech. Pro reverzní proxy cache a aplikační distribuovanou cache byl zvolen jeden server. Každý ze serverů je virtualiziován za pomoci virtualizačního nástroje VirtualBox. Celá webová architektura je konfigurována v jedné virtuální lokální síti 10.9.0.0/24 se statickým přidělováním IP adres. Rozhraní pro přípojení těchto virtualizovaných serverů do virtuální sítě je realizováno pomocí síťových mostů. Vzhledem k tomu, abych mohl ukázat velké poměrové rozdíly v jednotlivých krocích optimalizace v jednotlivých vrstvách architektury, rozhodl jsem se o velice slabé hardwarové vybavení jednotlivých serverů, která mají 512MB operační paměti RAM a jeden jednojádrový procesor s frekvencí 2GHz.



Obrázek 12: Vrstvy implementované webové architektury

13.2 Testovací a profilovací nástroje

Proto, abych mohl všecha svá tvrzení a závěry řádně podložit a abych mohl identifikovat místa nutná pro optimalizaci, jsou potřeba určité testovací a profilovací nátroje. Pomocí profilovacího nástroje XHProf jsem vyprofiloval statistiky pro aplikační vrstvu. Udělal jsem rozbory exekučních plánů pro SQL dotazy pro databázovou vrstvu, aby byla vidět míra jejich optimalizace. Také bylo potřeba vytvořit prostředí vysoké zátěže.

13.2.1 XHProf

K tomu, aby bylo možné profilovat programovací jazyk PHP, jsem zvolil profilovací nástroj XHProf. Tento nástroj byl vyvinut v programovacím jazyce C. Uvolněn byl pod open-source licencí Apache 2.0. Autorem tohoto profilovacího nástroje je společnost Facebook. Tento profilovací nástroj dokáže vyprofilovat nejenom dobu trvání jednotlivých metod a funkcí, ale i jejich procesorové či paměťové nároky. XHProf je možné provozovat na operačních systémech Linux, FreeBSD a Mac OS X.

Jeho instalace a konfigurace je jednoduchá a bezproblémové. Stačilo mi si stáhnout instalační balíček s rozšířením, zkomipilovat a nainstalovat. Po instalaci jej bylo nutné zaregistrovat jako rozšíření pro PHP v konfiguračním souboru php.ini. Samotné použití je pak velice snadné. Na začátek PHP skriptu jsem přidal zdrojový kód s jeho aktivací a na konci PHP skriptu kód pro jeho vypnutí.

Výsledkem profilování je přehledná statistika jednotlivých metod a funkcí ve formátu HTML. Všechno v přehledné tabulce, ve které je možné měnit pohled výsledků profilování podle specifické metody či funkce. V tabulce je možné pozorovat čas strávený vykonáváním pouze dané funkce, čas strávený vykonáváním i funkcí z ní volané, počet volání funkce, doba trvání, procesorový čas a paměťové nároky. Výsledkem může být i graf vygenerovaný z těchto statistik a ze závislostí mezi voláním jednotlivých funkcí.

Tento program je nenáročný a může být provozován i v ostrém provozu. Výsledky takového profilování jsou důležité k identifikaci problémových míst k optimalizaci, čili pro webovou architekturu v prostředí vysoké zátěže jsou nevyhnutelné.(Jakub Onderka, 2012)

Příklad použití XHProf profilování:

```
<?php
// Enable profiling
if (extension_loaded('xhprof')) {
    include_once '/usr/local/lib/php/xhprof_lib/utils/xhprof_lib.php';
    include_once '/usr/local/lib/php/xhprof_lib/utils/xhprof_runs.php';
    xhprof_enable(XHPROF_FLAGS_CPU + XHPROF_FLAGS_MEMORY);
}

// Php code for profiling
...

// Disable profiling
if (extension_loaded('xhprof')) {
    $profiler_namespace = 'myapp';  // namespace for your application
    $xhprof_data = xhprof_disable();
    $xhprof_runs = new XHProfRuns_Default();
    $run_id = $xhprof_runs->save_run($xhprof_data, $profiler_namespace);
}
?>
```

13.2.2 Siege

Vzhledem k tomu, abych mohl nasimulovat prostředí vysoké zátěže jsem zvolil testovací nástroj Siege. Siege je tedy nástrojem pro testování webových aplikací a jejich doby odezvy v prostředí vysoké zátěže. Tento nástroj má opět lehkou a přívětivou instalaci, nachází se totiž v základních instalačních balíčcích pro operační systém Linux Debian. Siege je programem spouštěným z příkazové řádky. Jako každý program, tak i siege má samozřejmě svůj konfigurační soubor, který je důležitý hlavně pro nastavení maximální prováděcí doby jednotlivých testů.

Volby parametrů tohoto příkazového programu jsou velice bohaté, ale přitom jednoduché pro pochopení a použití. Základními parametry jsou:

- c - počet simulovaných konkurenčních uživatelů
- d - interval zpoždění mezi jednotlivými uživatelskými požadavky
- r - počet repetic pro zátěžové testy

Ve své práci jsem prováděl zátěžové testy simulováním deseti uživatelů, v intervalu od nuly do jedné sekundy ve třech repeticích. Spuštění vypadá takto:

```
siege -d1 -c10 -r3 -v http://dp-xskrha.local/hello/Mendelu
```

13.2.3 PostgreSQL Explain

Pro konkrétní analýzu problémových SQL dotazů je potřeba identifikovat a analyzovat jejich exekuční plán. V databázi PostgreSQL a ve většině databázích k tomu slouží příkaz explain, který takto zobrazí exekuční plán. Ve své práci jsem analyzoval problémová místa pomocí XHProf profilování, ze kterého jsem zjistil, že všechna ploblémová místa souvisí s dotazy pro databázi. Po této fázi identifikace následovala fáze vysvětlení pomoc SQL dotazu explain.

Pomocí explain je možné vysvětlit všechny kroky exekučního plánu. Zjistíme tak jaký způsob prohledání byl zvolen, zda-li sekvenční nebo indexový. Dále je možné zjistit druh spojení, jestli merge join nebo hash join, apod. U jednotlivých kroků exekučního plánu je zobrazena jejich cena, předpokládaný počet řádků a předpokládaný počet sloupců. Takto je možné identifikovat slabá místa exekučního plánu a provést tak možnou úspěšnou optimalizaci.

Ukázka analýzy exekučního plánu:

```
explain analyze select * from film f where film_id in (select film_id from film_actor);
                                    QUERY PLAN
-------------------------------------------------------------------------------------------
Hash Join  (cost=117.26..195.78 rows=977 width=390) (actual time=21.067..26.570 rows=997 loops=1)
  Hash Cond: (f.film_id = film_actor.film_id)
  -> Seq Scan on film f  (cost=0.00..65.00 rows=1000 width=390)
        (actual time=0.016..1.699 rows=1000 loops=1)
  -> Hash (cost=105.05..105.05 rows=977 width=2) (actual time=21.029..21.029 rows=997 loops=1)
    -> HashAggregate  (cost=95.28..105.05 rows=977 width=2)
          (actual time=17.598..19.298 rows=997 loops=1)
       -> Seq Scan on film_actor  (cost=0.00..81.62 rows=5462 width=2)
             (actual time=0.012..8.019 rows=5462 loops=1)
Total runtime: 28.020 ms
```

13.3 Aplikační vrstva PHP

Ve své aplikační vrstvě jsem zvolil pro implementaci programovací jazyk PHP 5.3 běžící na webovém serveru Apache 2.0. Webový server Apache je jedním z nejrozšířenějších a nejpopulárnějších webových serverů na internetu. Byl implementován v roce 1996 v jazyce C++. Jeho instalace, konfigurace a administrace není nikterak složitá. Na mnoha webových hostinzích je dostupný v základní konfiguraci. Je to volně použitelný produkt, který obsahuje spoustu různých přídavných módů. Z těchto důvodů jsem ho vybral pro praktickou část své diplomové práce.

Programovací jazyk PHP se stal jedním z nejpoužívanějších programovacích jazyků pro svoji srozumitelnost, přenositelnost a jednoduchost. Je to dynamicky typovaný programovací jazyk, čili i z těchto důvodů je hodně ohebný. Plně podporuje OOP přístup, čili je možné vyžívat těchto technik včetně návrhových vzorů, které jsou pro složité webové aplikace v prostředí vysoké zátěže velice důležité. Aplikační vrstva samozřejmě podporuje plně koncept návrhového vzoru MVC využíváním MVC frameworku Symfony 2.

13.3.1 Optimalizace pomocí APC

Programovací jazyk PHP je sice jednoduchý na vývoj, čitelnost a dynamičnosti, ovšem jeho daň je časová náročnost pro překlad. Tento jazyk je interepretovaný jazyk, čili při jeho interpretaci dochází k překladu do opcode PHP kompilátorem. Vzniká tak mezikód, který je následně po skončení překladu vykonán.

K tomu, aby se nemusel pokaždé překládat stejný script PHP kompilátorem, je možné ho uložit do cache pro mezikód, neboli opcode cache. K tomuto účelu slouží APC, neboli Alternative Opcode Cache. APC je rozšířením pro PHP vy-

tvořené přímo samotnými tvůrci programovacího jazyka PHP. Toto rozšíření není nijak zvláště náročné na intalaci ani konfiguraci. Dokonce je k němu možné zprovoznit i webové rozhraní ukazující počet úspěšných či neúspěšných dotazů do cache a jiné další statistiky. Prostřednictvím APC programového rozhraní se dají do cache paměti ukládat i jiné další hodnoty, a to právě přímo ze zdrojových kódů. Úložištěm pro APC je operační paměť RAM. Z toho důvodu je tedy důležité, aby webserver Apache 2.0 stále běžel zapnutý. Při restartu webového serveru dojde k invalidaci celé APC cache paměti.(Vito Chin, 2010)

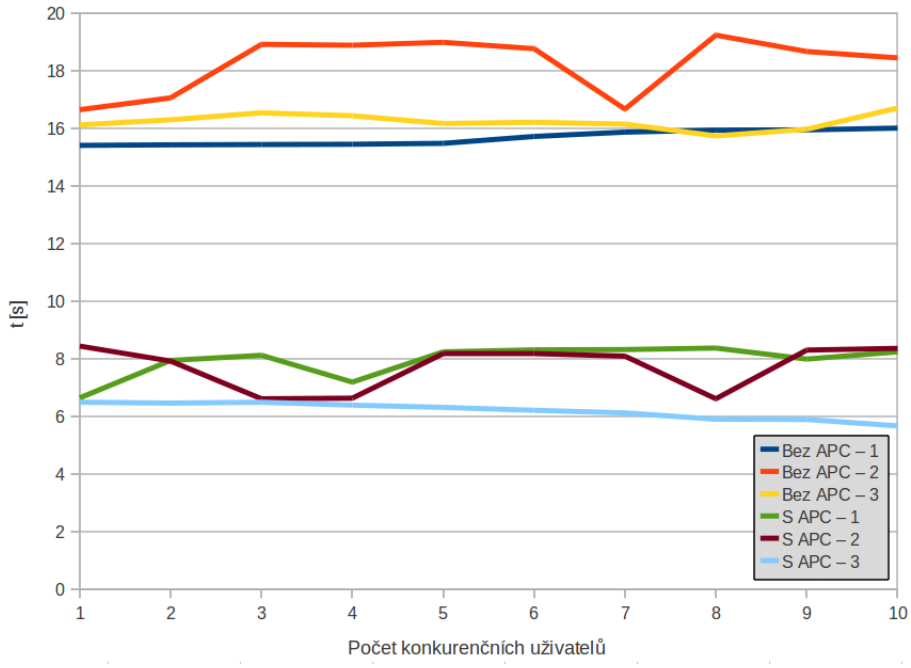
Většina interpretovaných jazyků je překládána do nějakého mezikódu. Proto se při žádné změně zdrojových kódů dá využít už jednou zkompilovaných výsledků. Tato forma recyklace zkompilovaného mezikódu dokáže přinést velké výsledky při optimalizaci. U APC jsou tyto výsledky více než viditelné.

13.3.2 Dosažené výsledky

Pro dosažení výsledků optimalizace aplikační vrstvy jsem vygeneroval prostředí vysoké zátěže deseti konkurenčních uživatelů s intervalem požadavků do jedné sekundy ve třech repeticích. Na aplikačním serveru tak vzniklo deset vláken Apache 2.0 s PHP 5.3 pro každého z uživatelů a celý procesorový výkon byl tak rozdělen rovnoměrně mezi jednotlivé vlákna po deseti procentech. Vytížení procesoru aplikačního serveru tak dosahovalo sta procent, čili maximálního vytížení. Toto prostředí jsem nasimuloval jak pro neoptimalizované prostředí bez APC, tak i pro optimalizovné prostředí s APC. Pro každý požadavek docházelo ke zpracování celého MVC frameworku, čili ke zpracování mnoho funkcí a metod. Z XHProf profilování jsem vyprofiloval výsledky pro jeden požadavek jak pro situaci s použitím APC, tak i bez použití APC. Výsledky tohoto profilování jsou takovéto:

- neoptimalované - bez APC - 1,3s
- optimalizované - s APC - 0,4s

Z výsledku vyplývá, že při optimalizaci aplikační vrstvy pomocí APC došlo k zrychlení překladu a provedení PHP skriptů o více než padesát procent, a to jak při jediném požadavku, tak i v prostředí vysoké zátěže. Takovýto fakt se dá určitě označit úspěšnou optimalizací. Alternativou může být použití přístupu s webovým serverem Lighttpd a s cache pamětí pro mezikód eAccelerator. Výsledky by měli být hodně podobné. V dnešních dnech se objevují a publikují zprávy o uvolnění HIP-HOP pro PHP od společnosti Facebook. HIP-HOP používá úplně jiného přístupu pro zrychlení běhu PHP scriptů. Tato varianta je dnes ještě hodně čerstvá a ne tak prověřená, ovšem představuje další možnosti pro optimalizaci aplikační vrstvy v prostředí vysoké zátěže v budoucích dnech.

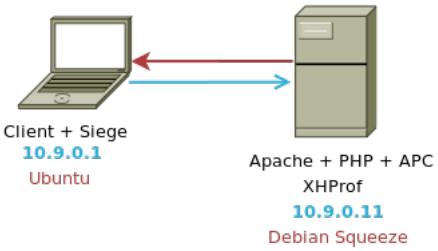


Obrázek 13: Graf s výsledky porovnání doby trvání jednotlivých požadavků pro repetice s použitím a bez použití APC

```
top - 17:14:24 up 6 min,  3 users,  load average: 1.37, 0.43, 0.18
Tasks: 137 total,   1 running, 136 sleeping,   0 stopped,   0 zombie
Cpu(s): 12.6%us, 47.0%sy,  0.0%ni, 10.3%id,  0.0%wa, 30.1%hi,  0.0%si,  0.0%st
Mem:    514700k total,   361288k used,   153412k free,    11304k buffers
Swap:   392184k total,        0k used,   392184k free,   196388k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
 1085 www-data  20   0 92012  19m  12m S 11.7  3.9   0:01.78 apache2
 1088 www-data  20   0 80288  21m  15m S 11.4  4.2   0:00.81 apache2
 1091 www-data  20   0 80780  17m  11m S 11.1  3.5   0:02.12 apache2
 1846 www-data  20   0 82652  15m 7600 S 11.1  3.1   0:01.27 apache2
 1086 www-data  20   0 94976  37m  17m S 10.4  7.4   0:07.75 apache2
 1847 www-data  20   0 84100  19m  10m S 10.4  3.9   0:01.12 apache2
  977 root      20   0 44240  18m 6460 S  6.5  3.7   0:05.02 Xorg
 1855 www-data  20   0 82652  15m 7604 S  3.9  3.1   0:00.86 apache2
 1089 www-data  20   0 80352  13m 7700 S  3.6  2.6   0:01.30 apache2
 1856 www-data  20   0 80688  15m 9360 S  3.6  3.0   0:00.79 apache2
 1857 www-data  20   0 82820  17m 9988 S  3.3  3.6   0:00.92 apache2
 1890 serga01   20   0 83304  11m 9216 S  1.0  2.3   0:00.63 gnome-terminal
 1786 serga01   20   0 19940 9836 8076 S  0.7  1.9   0:00.60 metacity
 1143 root      20   0 17408 3048 2236 S  0.3  0.6   0:00.14 console-kit-dae
 1784 serga01   20   0 21928 9404 6728 S  0.3  1.8   0:00.46 gnome-settings-
 1788 serga01   20   0 87592  18m  13m S  0.3  3.7   0:01.37 gnome-panel
 1930 root      20   0  2468 1192  900 R  0.3  0.2   0:00.14 top
    1 root      20   0  2036  716  624 S  0.0  0.1   0:00.94 init
    2 root      20   0     0    0    0 S  0.0  0.0   0:00.00 kthreadd
    3 root      RT   0     0    0    0 S  0.0  0.0   0:00.00 migration/0
    4 root      20   0     0    0    0 S  0.0  0.0   0:00.01 ksoftirqd/0
    5 root      RT   0     0    0    0 S  0.0  0.0   0:00.00 watchdog/0
```

Obrázek 14: Výpis systémových procesů aplikačního serveru s APC v prostředí vysoké zátěže



Obrázek 15: Výsledné zapojení webové architektury s aplikační vrstvou a APC

13.4 Databázová vrstva

Databázová vrstva je důležitou a nezbytnou součástí webové architektury. Její úloha je perzistence dat pomocí databázového systému. Ovšem přístup k datům může být někdy velice zdlouhavý, až v řádech několik minut a hodin. V prostředí vysoké zátěže dochází obvykle k obrovskému růstu dat a je potřeba nastavit optimalizovat databázovou vrstvu tak, aby jejich doba přístupu byla co nejkratší.

Ve své práci jsem vytvořil databázi nad databázovým systémem PostgreSQL. Tento systém je volně dostupný a najde velké uplatnění právě ve webových architekturách. Podporuje všechny standartní databázové operace a přístupy a je oblíben pro svoji spolehlivost.

Databáze modelované aplikace obsahuje záznamy v řádech miliónů řádků. To z toho důvodu, aby bylo nasimulováno prostředí vysoké zátěže i s velkou databází. Datové záznamy byly pro potřeby mé práce vygenerovány a nepředstavují žádnou paralelu s reálnými daty.

13.4.1 Optimalizace databáze

Před samotnou optimalizací databázové vrstvy nebyla databáze žádným způsobem optimalizována, ať už za pomocí indexace, optimalizace SQL dotazů, segmentování či vytvoření replikací. Přístup k datům v databázi také nebyl žádným způsobem efektivní. Jednalo se o znovu se opakující dotazy a dokonce místo jednoho SQL dotazu se objevovalo více dotazů pro zjištění stejné informační hodnoty. Rozhodl jsem se tedy pro aplikování kroku optimalizace databázové vrstvy pomocí optimalizace dotazů SQL.

Pří procesu optimalizace jsem si napřed určil problematické sql dotazy pomocí XHProf profilování. Z tohoto procesu jsem zjistil které sql dotazy je potřeba zoptimalizovat. Dalším krokem následoval rozbor jejich exekučního plánu, vyhledání řešení a jeho aplikace. Řešením optimalizace některých z exukučních plánů byla indexace příslušných sloupců pro nahrazení sekvenčního prohledání za indexové prohledání. Zde je ukázka rozboru neoptimalizovaného exekučního plánu se sekvenčním prohledáním:

```
SELECT * FROM dp_comment WHERE article_id = 15 ORDER BY create_time DESC;
------------------------------------------------------------------------------
Sort  (cost=121201.07..121201.08 rows=3 width=540) (actual time=16300.440..16300.444 rows=3 loops=1)
  Sort Key: create_time
  Sort Method:  quicksort  Memory: 20kB
  ->  Seq Scan on dp_comment  (cost=0.00..121201.05 rows=3 width=540)
         (actual time=13.531..16300.393 rows=3 loops=1)
        Filter: (article_id = 15)
Total runtime: 16300.526 ms
```

Po optimalizaci je vidět, že sekvenční prohledání bylo odstraněno a nahrazeno indexovým prohledáním, které je daleko efektivnější:

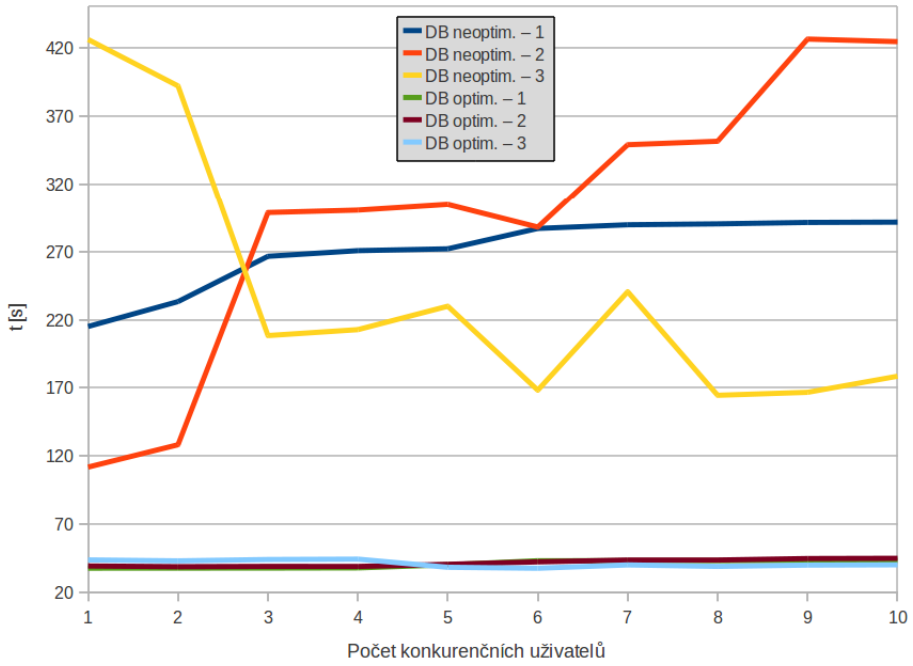
```
SELECT * FROM dp_comment WHERE article_id = 15 ORDER BY create_time DESC;
------------------------------------------------------------------------------
Sort  (cost=8.47..8.48 rows=3 width=540) (actual time=0.119..0.134 rows=4 loops=1)
  Sort Key: create_time
  Sort Method:  quicksort  Memory: 21kB
  ->  Index Scan using dp_comment_article_id_fk_i on dp_comment  (cost=0.00..8.45 rows=3 width=540)
         (actual time=0.043..0.068 rows=4 loops=1)
        Index Cond: (article_id = 1441)
Total runtime: 0.303 ms
```

13.4.2 Dosažené výsledky

Při optimalizaci dotazů jsem důkladně prozkoumával exekuční plány jednotlivých dlouhotrvajících SQL dotazů. Na spoustě místech byly řešitelné pomocí indexace. Abych tyto kroky potvrdil i pro prostředí vysoké zátěže, rozhodl jsem se nasimulovat prostředí vysoké zátěže opět pomocí programu siege, který vytvořil deset konkurenčních uživatelů s intervalem požadavku do jedné sekundy a ve třech repeticích. Test proběhl samozřejmě před i po kroku optimalizace.

Z výsledků experimentu vyplývá, že SQL dotazy, které trvaly několik řádů sekund se zoptimalizovali až pod dobu jedné sekundy. Z grafu v prostředí vysoké zátěže je vidět, že před optimalizací byla maximální doba jednoho požadavku až 420 sekund. Celý proces byl hodně nestabilní a kolísavý. Po optimalizaci je viditelné ustálení činnosti databázového systému a zmenšení doby odezvy. Zrychlení je vidět až o více než padesát procent. Dalšími budoucími řešeními, která se by se dala aplikovat mohou být segmentace či replikace. Samotná optimalizace SQL dotazů řeší určité problémy, ovšem databáze může přerůst až do takových rozměrů, kdy je potřeba sáhnout k těmto dalsím řešením.

Databázová vrstva bývá velice často největším zdrojem zpomalení. Dotazy na tuto vrstvu jsou nejnáročnější ze všech procesů skrze celou webovou architekturu. Proto je velice důležité tento proces nepodceňovat a věnovat tomu patřičné prostředky.



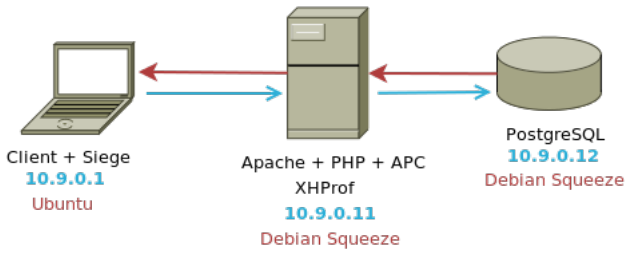
Obrázek 16: Graf s výsledky porovnání doby trvání jednotlivých požadavků pro repetice před a po optimalizaci SQL dotazů

Function Name	Calls	Calls%	Incl. Wall Time (microsec)	IWall%
Current Function				
call_user_func_array@1	209	49.4%	35,558,194	97.1%
Exclusive Metrics for Current Function			1,839	0.0%
Parent functions				
Symfony\Component\DependencyInjection\ContainerBuilder::createService	5	2.4%	35,555,786	100.0%
Twig_Template::getAttribute	204	97.6%	2,408	0.0%
Child functions				
Dpxskrha\FrontBundle\Model\Models\Comment\CommentService::getCommentsCount	1	0.5%	15,898,330	44.7%
Dpxskrha\FrontBundle\Model\Models\Article\ArticleService::getArticlesNew	1	0.5%	14,046,220	39.5%
Dpxskrha\FrontBundle\Model\Models\Article\ArticleService::getArticlesCount	1	0.5%	5,397,372	15.2%
Dpxskrha\FrontBundle\Model\Models\Member\MemberService::getMembersCount	1	0.5%	186,680	0.5%
Dpxskrha\FrontBundle\Model\Models\Category\CategoryService::getAllCategories	1	0.5%	27,077	0.1%
Dpxskrha\FrontBundle\Model\Models\Category\Entity\Category::getTitle	31	14.8%	96	0.0%
Dpxskrha\FrontBundle\Model\Models\Article\Entity\Article::getCategory	30	14.4%	94	0.0%
Dpxskrha\FrontBundle\Model\Models\Article\Entity\Article::getMember	30	14.4%	91	0.0%
Dpxskrha\FrontBundle\Model\Models\Article\Entity\Article::getImageUrl	15	7.2%	81	0.0%
Dpxskrha\FrontBundle\Model\Models\Category\Entity\Category::getUrl	23	11.0%	72	0.0%
Dpxskrha\FrontBundle\Model\Models\Article\Entity\Article::getContent	15	7.2%	52	0.0%
Dpxskrha\FrontBundle\Model\Models\Article\Entity\Article::getUrl	15	7.2%	50	0.0%
Dpxskrha\FrontBundle\Model\Models\Article\Entity\Article::getTitle	15	7.2%	50	0.0%
Dpxskrha\FrontBundle\Model\Models\Member\Entity\Member::getFirstName	15	7.2%	46	0.0%
Dpxskrha\FrontBundle\Model\Models\Member\Entity\Member::getLastName	15	7.2%	44	0.0%

Obrázek 17: XHProf profilování aplikačního modelu databázových dotazů před optimalizací SQL dotazů

Function Name	Calls	Calls%	Incl. Wall Time (microsec)	IWall%
Current Function				
call_user_func_array@1	209	49.4%	24,632,393	84.0%
Exclusive Metrics for Current Function			1,985	0.0%
Parent functions				
Symfony\Component\DependencyInjection\ContainerBuilder::createService	5	2.4%	24,629,750	100.0%
Twig_Template::getAttribute	204	97.6%	2,643	0.0%
Child functions				
Dpxskrha\FrontBundle\Model\Models\Comment\CommentService::getCommentsCount	1	0.5%	18,444,038	74.9%
Dpxskrha\FrontBundle\Model\Models\Article\ArticleService::getArticlesCount	1	0.5%	5,708,528	23.2%
Dpxskrha\FrontBundle\Model\Models\Article\ArticleService::getArticlesNew	1	0.5%	262,783	1.1%
Dpxskrha\FrontBundle\Model\Models\Member\MemberService::getMembersCount	1	0.5%	194,205	0.8%
Dpxskrha\FrontBundle\Model\Models\Category\CategoryService::getAllCategories	1	0.5%	20,080	0.1%
Dpxskrha\FrontBundle\Model\Models\Article\Entity\Article::getTitle	15	7.2%	116	0.0%
Dpxskrha\FrontBundle\Model\Models\Article\Entity\Article::getMember	30	14.4%	113	0.0%
Dpxskrha\FrontBundle\Model\Models\Article\Entity\Article::getCategory	30	14.4%	101	0.0%
Dpxskrha\FrontBundle\Model\Models\Category\Entity\Category::getTitle	31	14.8%	92	0.0%
Dpxskrha\FrontBundle\Model\Models\Article\Entity\Article::getImageUrl	15	7.2%	85	0.0%
Dpxskrha\FrontBundle\Model\Models\Category\Entity\Category::getUrl	23	11.0%	74	0.0%
Dpxskrha\FrontBundle\Model\Models\Article\Entity\Article::getUrl	15	7.2%	56	0.0%
Dpxskrha\FrontBundle\Model\Models\Article\Entity\Article::getContent	15	7.2%	48	0.0%
Dpxskrha\FrontBundle\Model\Models\Member\Entity\Member::getFirstName	15	7.2%	47	0.0%
Dpxskrha\FrontBundle\Model\Models\Member\Entity\Member::getLastName	15	7.2%	42	0.0%

Obrázek 18: XHProf profilování aplikačního modelu databázových dotazů po optimalizaci SQL dotazů



Obrázek 19: Výsledné zapojení webové architektury s optimalizovanou databázovou vrstvou

13.5 Aplikační distribuovaná cache

Úkolem aplikační distribuované cache je ukládat jednou dotazovaná data do cache paměti a ušetřit tak SQL dotazy do databáze, které mohou být velice náročné. Zároveň ponechává webovou aplikaci dynamickou. Celý tento proces je řízen aplikací a její logikou, a tak je potřeba už v návrhu počítat s existencí dalšího typu úložiště, s přístupem k tomuto úložišti a s invalidací neplatných dat. Při použití aplikační distribuované cache vrstvy se kladou velké systémové a výpočetní nároky na aplikační vrstvu. Jejím úkolem už tak není jen vykonávat zdrojový kód PHP skriptů a práce s databází, ale navíc i řízení celé logiky aplikační distribuované cache. Ve své práci uvádím výsledky, které tato tvrzení dokládají.

13.5.1 Optimalizace aplikace pomocí Memcached

Ve své práci jsem zvolil aplikační distribuovanou cache s názvem Memcached. Memcached obsahuje rozhraní pro práci v různých programovacích jazycích, například v Java, Ruby On Rails, .NET, apod. Somozřejmě obsahuje i rozhraní pro programovací jazyk PHP, použitého v mé studii. PHP rozhraní pro Memcached je součástí dopňkových rozšíření pro PHP, čili instalace tohoto rozhraní je bezproblémová. Vzhledem k bohaté podpoře různých programovacích jazyků, je možné použít návrh, přístup a výsledky mé studie aplikační distribuované cache Memcached i pro webové architektury implementované v jiných programovacích jazycích než jen PHP.

Memcached je volně šířitelným software s vysokou rychlostí a stabilitou. Své uplatnění najde v prostředích vysoké zátěže, kde dokáže rapidně snížit dobu přístupu k datům a zachovává aplikaci dynamickou. Memcached byla vyvinuta a implentována v jazyce C pro Live Journal v roce 2003, kdy měla tato webová aplikace přístup několika miliónů dynamického zobrazení stránek během jednoho dne. V takovémto prostředí vysoké zátěže obstála a stává se dnes běžnou součástí vrstev webových arhitektur v prostředí vysoké zátěže.

Fyzickým úložištěm pro Memcached je operační paměť RAM. Do paměti ukládá hodnoty na principu key-value, čili každá hodnota je ukládána a čtena podle konkrétního klíče. Připojení aplikační vrstvy k jednotlivým serverům s memcached není nikterak složité. Těchto cache serverů, neboli uzlů, může být i více a je možné určit v jakém poměru se budou hodnoty do jednotlivých uzlů ukládat. Ne všechny uzly totiž mohou mít stejné hardwarové vybavení, zejména stejnou velikost operační paměti RAM. Aplikační vrstva ani nemusí znát informace o tom, na jakém konkrétním uzlu se uložila jaká hodnota. Pouze zašle Memcached požadavek pro hodnotu podle klíče a Memcached už sama rozhoduje na kterém z uzlů se tato hodnota nachází. Všechny hodnoty jsou ukládány serializované, nebo v textové formě, a nebo v nějakém jiném standardizovaném formátu, který může vybrat aplikační vrstva.(Brad Fitzpatrick, 2004)

Systém Memcached představuje optimálního kandidáta pro tvorbu aplikační distribuované cache pro dynamické webové aplikace. Ovšem vzhledem k náročnosti požadavků pro tuto vrstvu webové architektury je důležitý správný návrh a implementace aplikace. To z toho důvodu, aby byla ukládána a zobrazována čerstvá, aktuální a správná data pro zobrazení korektních informací pro uživatele webové aplikace.

13.5.2 Specifikace aplikace

Pro aplikační vrstvu byl použit programovací jazy PHP s MVC frameworkem Symfony 2. Každý MVC framework by měl pracovat s daty a jejich úložištěm ve vrstvě Model. Vyjímkou by neměli být ani data ukládána pomocí distribuované aplikační cache Memcached.

Model aplikační vrstvy je rozdělen na další čtyři vrstvy, kde každá z nich má svůj jasně definovaný účel. První vrstvou je vrstva Service, neboli vrstva, která

je přístupna z jiných vrstev MVC frameworku. Této vrstvě mohou být předány i nějaké parametry. Servisní vrsta požadavek zpracuje, zpracuje předané parametry a předá řízení další vrstvě modelu kterou je DAO, neboli Data Access Object. Účelem této vrstvy je vybrat způsob přistupu ke konkrétním typům úložišť. Jednotlivé typy úložišť jsou přístupné pomocí vrstvy s názvem Mapper. Pro každý typ úložiště existuje Mapper, který má za úkol realizovat daný požadavek na konkrétní úložiště, například databázi či aplikační distribuovanou cache. Úložiště tak tvoří poslední vrstvu pro Model.

Data, neboli dotazy se kterými aplikace pracuje, mohou být tří druhů z pohledu čtení z úložiště a ukládání do úložiště. Dotazy mohou být buď skalárem, čili jednou hodnotou, nebo entitou, neboli objektem s více atributy, či kolekcí entit. Tyto typy dotazů jsou buď předány databázi a nebo distribuované aplikační cache, podle toho jak rozhodne aplikační vrstva. Každý z těchto dotazů definuje svůj SQL dotaz pro načtení z databáze a klíč pod kterým je uložen do Memcached. Do memcached jsou data ukládána serializovaná, a to z toho důvodu, aby mohla být opět zrekonstruovatelná.

Největší problém obecně pro cache vrstvy bývá jejich invalidace. V distribuovaných aplikačních cache vrstvách mohou vznikat tzv. cache kaskády. Vyjímkou tomu nebylo ani v mém návrhu aplikace. Tyto kaskády vznikají tím, že stejná data jsou interpretována v jiných kontextech, čili jakmile dojde ke změně těchto dat, musí se změnit i celá tato kaskáda, neboli se celá kaskáda musí invalidovat. Jedná se tak o závislosti mezi jinými výsledky dotazů na stejných datech. Jako příklad můžu uvést, že jeden stejný článek se může nacházet v kolekci nejnovějších článků a zároveň v kolekci článků dané kategorie. Jakmile dojde ke změně článku v databázi, je potřeba invalidovat i obě dvě tyto kolekce v cache paměti Memcached. Ve své práci tuto skutečnost řeším pomocí Interface Dependency Injection, která určuje způsob předávání závislostí mezi jednotlivými objekty. Vytvoří se tedy nějaká konkrétní kaskáda, která určuje závislosti mezi konkrétními daty a jednotlivými dotazy. Jakmile dojde ke změně určitých dat, tak kaskáda invaliduje všechny závislosti dotazů na těchto datech.

Takto jsou splněny všchny podmínky pro správnou práci a organizaci aplikační distribuované cache. Data jsou do cache paměti ukládána v nejmenších a jasně definovaných formátech, podle jednoznačného a konkrétního klíče a jsou ukládána znovuzrekonstruovatelná. Invalidace dat uloženách v cache paměti je řešena systémem invalidačních kaskád.

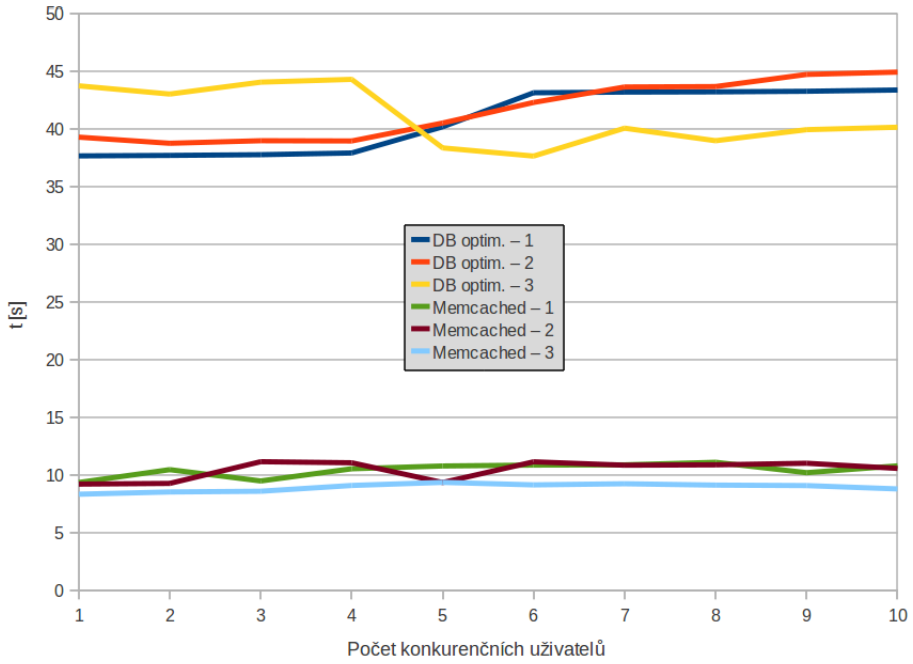
13.5.3 Dosažené výsledky

Po zhodnocení dosažených výsledků webové architektury s aplikační distribuovanou cache pomocí Memcached bylo potvrzeno další zrychlení odezvy odpovědí na požadavky klientů v prostředí vysoké zátěže. Opět jsem nasimuloval prostředí vysoké zátěže pomocí zátěžových testů programem siege, kdy odpověď požadovalo deset konkurenčních uživatelů ve stejný moment ve třech repeticích. Z XHProf pro-

filování jsem vyprofiloval výsledky pro jeden požadavek jak pro situaci s použitím aplikační distribuované cache, tak i pro situaci pouze s optimalizovanou databází bez použití aplikační ditribuované cache. Výsledky tohoto profilování jsou takovéto:

- neoptimalované - bez Memcached - 25,9s
- optimalizované - s Memcached - 0.95s

Výsledky potvrdili že došlo k velkému zrychlení odezvy. Zároveň bylo otestováno, že aplikace stále zůstává dynamickou a všechna data jsou invalidována správně a ve správný čas. Ovšem stále zůstává obrovské zatížení aplikační vrstvy. Z toho tedy vyplývá, že při použití aplikační distribuované cache je potřeba počítat s tímto zatížením, které už je nutno řešit více aplikačními servery a rozdělováním zátěže mezi tyto servery. Server určen pro aplikační distribuovanou cache neměl žádné velké vytížení a tudíž se zde nachází potenciál, který se stále jestě může využít.



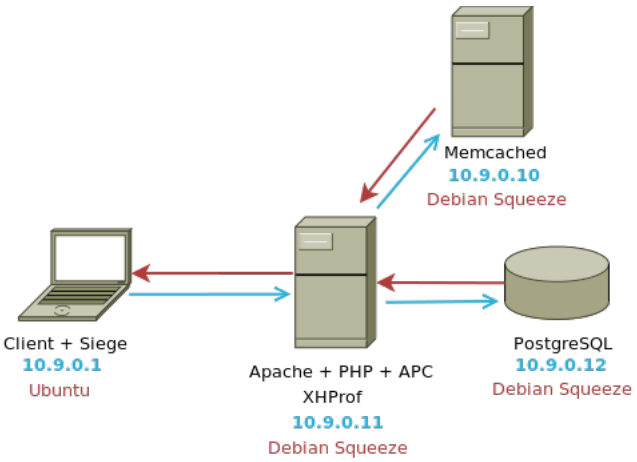
Obrázek 20: Graf s výsledky porovnání doby trvání jednotlivých požadavků pro repetice v prostředí s Memcached a v prostředí s optimalizovanou databází PostgreSQL

Function Name	Calls	Calls%	Incl. Wall Time (microsec)	IWall%
Current Function				
call_user_func_array@1	209	49.4%	24,632,393	84.0%
Exclusive Metrics for Current Function			1,985	0.0%
Parent functions				
Symfony\Component\DependencyInjection\ContainerBuilder::createService	5	2.4%	24,629,750	100.0%
Twig_Template::getAttribute	204	97.6%	2,643	0.0%
Child functions				
Dpxskrha\FrontBundle\Model\Models\Comment\CommentService::getCommentsCount	1	0.5%	18,444,038	74.9%
Dpxskrha\FrontBundle\Model\Models\Article\ArticleService::getArticlesCount	1	0.5%	5,708,528	23.2%
Dpxskrha\FrontBundle\Model\Models\Article\ArticleService::getArticlesNew	1	0.5%	262,783	1.1%
Dpxskrha\FrontBundle\Model\Models\Member\MemberService::getMembersCount	1	0.5%	194,205	0.8%
Dpxskrha\FrontBundle\Model\Models\Category\CategoryService::getAllCategories	1	0.5%	20,080	0.1%
Dpxskrha\FrontBundle\Model\Models\Article\Entity\Article::getTitle	15	7.2%	116	0.0%
Dpxskrha\FrontBundle\Model\Models\Article\Entity\Article::getMember	30	14.4%	113	0.0%
Dpxskrha\FrontBundle\Model\Models\Article\Entity\Article::getCategory	30	14.4%	101	0.0%
Dpxskrha\FrontBundle\Model\Models\Category\Entity\Category::getTitle	31	14.8%	92	0.0%
Dpxskrha\FrontBundle\Model\Models\Article\Entity\Article::getImageUrl	15	7.2%	85	0.0%
Dpxskrha\FrontBundle\Model\Models\Category\Entity\Category::getUrl	23	11.0%	74	0.0%
Dpxskrha\FrontBundle\Model\Models\Article\Entity\Article::getUrl	15	7.2%	56	0.0%
Dpxskrha\FrontBundle\Model\Models\Article\Entity\Article::getContent	15	7.2%	48	0.0%
Dpxskrha\FrontBundle\Model\Models\Member\Entity\Member::getFirstName	15	7.2%	47	0.0%
Dpxskrha\FrontBundle\Model\Models\Member\Entity\Member::getLastName	15	7.2%	42	0.0%

Obrázek 21: XHProf profilování aplikačního modelu dotazů před optimalizací pomocí Memcached

Function Name	Calls	Calls%	Incl. Wall Time (microsec)	IWall%
Current Function				
call_user_func_array@1	209	49.4%	87,007	9.2%
Exclusive Metrics for Current Function			2,077	2.4%
Parent functions				
Symfony\Component\DependencyInjection\ContainerBuilder::createService	5	2.4%	84,173	96.7%
Twig_Template::getAttribute	204	97.6%	2,834	3.3%
Child functions				
Dpxskrha\FrontBundle\Model\Models\Article\ArticleService::getArticlesNew	1	0.5%	56,214	64.6%
Dpxskrha\FrontBundle\Model\Models\Member\MemberService::getMembersCount	1	0.5%	10,114	11.6%
Dpxskrha\FrontBundle\Model\Models\Category\CategoryService::getAllCategories	1	0.5%	6,436	7.4%
Dpxskrha\FrontBundle\Model\Models\Article\ArticleService::getArticlesCount	1	0.5%	6,181	7.1%
Dpxskrha\FrontBundle\Model\Models\Comment\CommentService::getCommentsCount	1	0.5%	5,143	5.9%
Dpxskrha\FrontBundle\Model\Models\Article\Entity\Article::getImageUrl	15	7.2%	127	0.1%
Dpxskrha\FrontBundle\Model\Models\Article\Entity\Article::getCategory	30	14.4%	122	0.1%
Dpxskrha\FrontBundle\Model\Models\Article\Entity\Article::getMember	30	14.4%	109	0.1%
Dpxskrha\FrontBundle\Model\Models\Category\Entity\Category::getTitle	31	14.8%	103	0.1%
Dpxskrha\FrontBundle\Model\Models\Category\Entity\Category::getUrl	23	11.0%	86	0.1%
Dpxskrha\FrontBundle\Model\Models\Article\Entity\Article::getUrl	15	7.2%	69	0.1%
Dpxskrha\FrontBundle\Model\Models\Member\Entity\Member::getFirstName	15	7.2%	59	0.1%
Dpxskrha\FrontBundle\Model\Models\Article\Entity\Article::getContent	15	7.2%	58	0.1%
Dpxskrha\FrontBundle\Model\Models\Article\Entity\Article::getTitle	15	7.2%	56	0.1%
Dpxskrha\FrontBundle\Model\Models\Member\Entity\Member::getLastName	15	7.2%	53	0.1%

Obrázek 22: XHProf profilování aplikačního modelu dotazů po optimalizaci pomocí Memcached



Obrázek 23: Výsledné zapojení webové architektury s aplikační distribuovanou cache vrstvou Memcached

13.6 Reverzní proxy cache

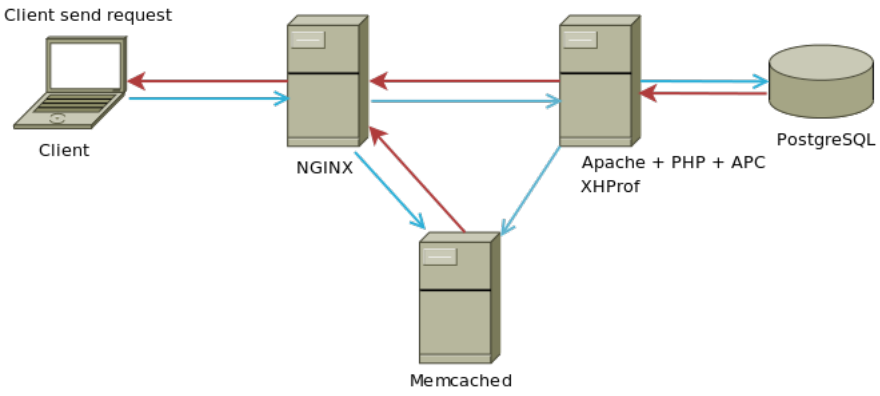
Nejrychlejším a nejefektivnější způsobem zůstává stále použití reverse proxy cache. Ovšem ale aplikace tak přichází o dynamičnost. Ale i tak se dají najít situace, kdy je možné použití statického cachevání pomocí reversní proxy cache. Například archiv článků nějakého určitého období může být statického charakteru, poněvadž takovýto obsah se dynamicky nemění. Existuje spousta programů pro řešení klasické reverzní proxy cache (viz. kapitola 8.2.2). V této části své práce popíšu netradiční způsob pro řešení reverzní proxy cache, ale tento způsob má své uplatnění i v praxi a navazuje na předchozí kapitolu 13.5.

13.6.1 Optimalizace pomocí reverzní proxy cache Nginx s Memcached

Jedním z moderních nástrojů pro reverzní proxy cache a se nazývá Nginx. Nginx má spoustu dalších funkcionalit a modulů. Obsahuje modul pro webový server, pro vyvažování zátěže (load balancing), pro geografické vyvažování zátěže, pro servírování obrázků, pro podporu memcached, a spoustu dalších modulů. Tento opravdu silný nástroj byl vytvořen programátorem Igorem Sysoevem v roce 2004 pro druhou největší ruskou webovou aplikaci Rambler.ru, která servírovala přes 500 milionů HTTP odpovědí během jednoho dne. Nginx jako reverzní proxy cache je řízen HTTP hlavičkami jako klasická reverzní proxy cache (viz. kapitola 8.1). HTTP odpovědi ukládá do přesně definovaného adresáře souborového systému. Tento klasický způsob se hodí pro skutečný statický obsah, kterým jsou obrázky, CSS soubory, JS soubory apod.(Will Reese, 2008)

Ve své práci jsem použil Nginx jako reverzní proxy cache s modulem pro Memcached. Tento netradiční způsob řešení reverzní proxy cache s sebou přináší určité výhody. Jednou z výhod je, že řízení Memcached je plně pod kontrolou aplikační

vrstvy. Aplikační vrstva ukládá a invaliduje výsledné HTTP odpovědi na HTTP požadavky klientů do Memcached, kterou používá reverzní proxy cache Nginx. Jakmile Nginx obdrží HTTP požadavek a z konfigurace zjistí, že HTTP odpověď je možné hledat pod URL adresou v paměti Memcached, zkusí ji tam najít. Pokud ji nenalezne propustí tento požadavek dále na aplikační vrstvu. Aplikační vrstva sestaví HTTP odpověď, uloží ji pod URL adresou HTTP požadavku do cache paměti pro Memcached, a pošle klientovi. Při příštím HTTP požadavku Nginx nalezne HTTP odpvěď v paměti Memcached a pošle klientovi, bez toho, aniž by požadavek propustila na aplikační vrstvu. Takovéto chování je obdobné jako při klasickém nastavení reverzní proxy cache, s tím rozdílem, že ukládání a invalidace nejsou řízeny HTTP hlavičkami. Další výhodou může být využití Ajaxu a metodiky aplikační distribuované cache pro vytvoření dynamického přístupu ukládání a invalidace dat do cache paměti pro Memcached a Nginx.(Ilya Grigorik, 2008)



Obrázek 24: Schéma webové architektury při použití Nginx s modulem pro Memcached

13.6.2 Využití Ajax a webových služeb pro NGINX s Memcached

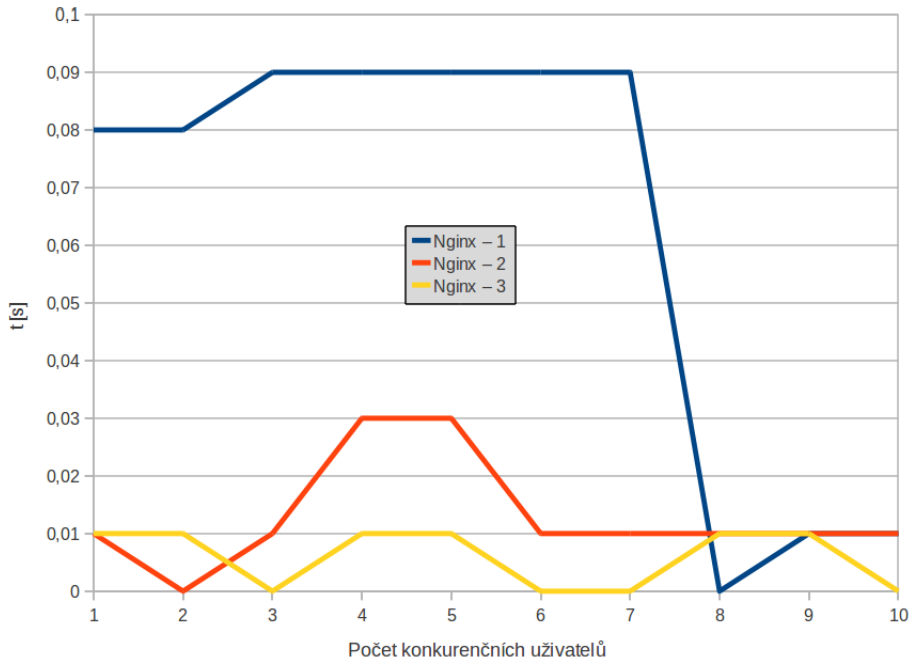
Aplikační vrstva nemusí ukládat do cache paměti Memcached pro Nginx pouze celé kompletní HTML stránky, jakožto odpovědi HTTP požadavků. Může ukládat i další typy odpovědí jako například Ajaxem standardizovaný JSON, což je druh přenosu dat mezi Aplikační vrstvou a technologií Ajax klientské vrstvy. Díky Ajax tak může dojít k rozdělení jednoho HTTP požadavku na více různých požadavků. Klient zašle HTTP požadavek a Nginx navrátí HTTP odpověď ve formě statické HTML stránky, která obsahuje několik dalších požadavků pomocí Ajax. Odpovědi na tyto Ajax požadavky mohou být dynamického či statického charakteru. Tímto způsobem je webová aplikace rozdělena mezi více webových služeb, které mohou pracovat samostatněji a nezávisle na sobě, a aplikace se tak stává více škálovatelnou.

Vzhledem k tomu, že aplikace a její logika řídí ukládání do cache paměti Memcached, může tak řídit i jejich invalidaci podobným způsobem jako v aplikační distribuované cache (viz. kapitola 13.5.2). Hlavní rozdíl je v tom, že v případě aplikační distribuované cache je práce s cache řízena plně aplikační vrstvou, jak při změně

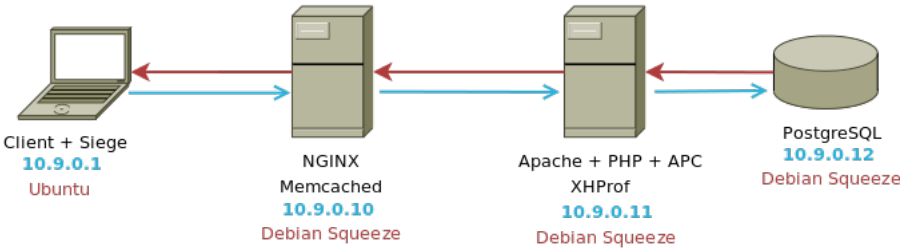
tak čtení. Zatím co systém webové architektury s Nginx a Memcached je řízen aplikační vrstvou pro změnu a invalidaci dat a reverzní proxy cache vrstvou pro čtení a získávání dat z cache paměti. Dochází tak k hybridnímu způsobu práce s reverzní proxy cache a aplikační distribuovanou cache. Toho je možné využít pro webové služby, které tak mohou být dynamického nebo statického charakteru.

13.6.3 Dosažené výsledky

Z dosažených výsledků vyplývá absulutně nejrychlejší doba pro HTTP odpovědi ze všech mých experimentů. Opět jsem nasimuloval prostředí vysoké zátěže pro deset konkurenčních uživatelů v jeden samý okamžik a ve třech repeticích. Ovšem tento experiment má tu nevýhodu, že obsah takovéto webové architektury se stává statickým. Ale při aplikaci myšlenky z kapitoly 13.6.2 jsou HTTP odpovědi servírovány uživateli velice rychle a jakmile jsou data změněna, může dojít k ivalidaci dat z aplikační vrstvy (viz. kapitola 13.6.2). Takto se obsah webové aplikace může stát dynamickým. Tento způsob dynamického chování jsem implementoval v aplikační vrstvě i pro Nginx.



Obrázek 25: Graf s výsledky porovnání doby trvání jednotlivých požadavků pro repetice s reverzní proxy cache Nginx s pamětí Memcached



Obrázek 26: Výsledné zapojení webové architektury s reverzní proxy cache vrstvou Nginx s pamětí Memcached

14 Diskuze

Ve své práci jsem se nejvíce zaměřil na jednotlivé vrstvy webové architektury obzvláště na vrstvy serverové části. Celkem jsem provedl čtyři větší experimenty, jeden experiment pro aplikační vrstvu, jeden pro databázovou vrstvu, jeden pro aplikační distribuovanou cache vrstvu a jeden pro reverzní proxy cache vrstvu. Pro kazdy z testu jsem navrhnul, implementoval, testoval a vyhodnocoval jejich webovou architekturu, konfiguraci, funkcionalitu, zátěžové testy a schopnost obstát v prostředí vysoké zátěže.

První experiment se týkal aplikační vrstvy a možnosti využití opcode cache pro programovací jazyk PHP (viz. kapitola 13.3.1). Konfigurace, implementace a realizace tohoto expermentu nebyla až tak náročná a pracná. Vzhledem k nastudovaným teoretickým základům, kdy jsem věděl jako cestou se vydávat to nebylo tak složité a výsledky byly více než dobré, čili takovýto krok mohu jen doporučit pro ostré nasazení. V opravdu vysokých zátěžích je pak důležité nasadit více aplikačních serverů pro vyvažování zátěže, neboli load balancing.

Druhý experiment pracoval se standartní třívrstvou architekturou, tj. klientem, aplikačním serverem a databázovým systémem (viz. kapitola 13.4.1). V tomto experimentu jsem se zaměřil na databázový systém, jeho přístupnot a časovou odezvu v prostředí vysoké zátěže. V tomto směru jsem zjistil, že optimalizace databáze je velice důležitá, protože se přístup a odpovědi na dotazy mohou zrychlit v řádech sekund, minut i hodin. Samozřejmě jakmile se prostředí vysoké zátěže začne zvětšovat a začne ještě více růst zatížení databáze, je potřeba sáhnout k dalším krokům jako například partitioning nebo replikace. Samozřejmě v architektuře webových systémů bývá databázová vrstva tou nejvytíženější a nejpomalejší vrstvou. Proto je dobré zaměřit pozornosti i na to, jak zmenšit dotazování do databáze.

Ve třetím experimentu jsem rozšířil třívrstvou architekturu o vrstvu aplikační distribuované cache (viz. kapitola 13.5). Právě diký této vrstvě je možné získat rychlejší přístup k již jednou načtených datům z databáze. Zároveň zůstává obsah webové aplikace dynamický. Ovšem zároveň je nutné zmínit, že zde ovšem zůstává velké zatížení aplikační vrstvy. Proto může v prostředí vysoké zátěže nastat situace, kdy je potřeba mít více aplikačních serverů pro vyvažování zátěže. Výsledky testů simulovaného prostředí vysoké zátěže byly rozhodně lepší s využitím aplikační distribuované cache Memcached než výsledky pouze s databází. Ovšem v tomto případě je potřeba přihlédnout k nutnosti zajistit tuto funkcionalitu v jádru aplikace a její logice. Je potřeba zajistit aplikaci přístup ke cache, zajistit ukládání dat ve spravném, dohodnutém a znovuzkonstruovatelném formátu a hlavně zajistit jejich správnou a včasnou invalidaci. S požadavkem aplikační distribuované cache je tak potřeba počítat při jejím návrhu, protože jinak může docházet k častým chybám a náklady, ať už časové či finanční, na zavedení takovéto vrstvy mohou být vysoké.

V posledním experimentu jsem zkombinoval reverzní proxy cache Nginx s aplikační distribuovanou cache Memcached (viz. kapitola 13.6). První HTTP

požadavky zpracovala aplikační vrstva, vytvořila HTTP odpověď a uložila ji do cache paměti Memcached pro reverzní proxy cache Nginx. Při příštím dotazu našla reverzní proxy cache odpověď v cache paměti a okamžitě ji vrátila klientovi. Invalidace těchto odpovědí probíhá automaticy pomocí Memcached a časového razítka. Tento způsob práce s obsahem webové aplikace se stává tak statický. Ovšem vzhledem k tomu, že odpovědi ukládá aplikační vrstva, může tyto odpovědi stejně tak invalidovat podobně jako v případě aplikační distribuované cache. Jeden HTTP požadavek se tak dá rozložit na více požadavků pomocí technologie Ajax. Odpovědi na jednotlivé požadavky Ajax jsou také ukládány do cache paměti a jejich invalidace může proběhnout ihned. Dochází tak k hibridnímu využití reverzní proxy cache a aplikační distribuované cache. Výsledky pro tento způsob práce byly více než uspokojivé. Obsah webové aplikace je dynamický a rychlý, a to díky režii ukládání a invalidace na straně aplikační distriboavané cache a režii čtení na straně reverzní proxy cache.

15 Závěr

Závěr ve smyslu nákladů a přínosů, kdy je lepší co. Uvidíme, napíšu jako poslední.

16 Reference

Kyle Rankin *Linux Journal: Hack and / - Linux Troubleshooting, Part I: High Load* [online]. Dostupné z: http://www.linuxjournal.com/magazine/hack-and-linux-troubleshooting-part-i-high-load.

Faisal Khan *DOS ATTACKS: Dos Attacks Overview - What are DoS attacks* [online]. Dostupné z: http://dos-attacks.com/what-are-dos-attacks/.

Pavel Čepský *Lupa.cz: Útoky jménem Anonymous: Jak se rodí hackeři?* [online]. Dostupné z: http://www.lupa.cz/clanky/utoky-jmenem-anonymous-jak-se-rodi-hackeri/.

Doc.Ing.Jaroslav Zendulka,CSc. *VUT-FIT: 10. Architektura klient/server a třívrstvá architektura* [online]. Dostupné z: http://www.fit.vutbr.cz/study/courses/DSI/public/pdf/nove/10_clsrv.pdf.

Brett McLaughlin *Ibm develper works: Mastering Ajax* [online]. Dostupné z: http://www.ibm.com/developerworks/web/library/wa-ajaxintro1/index.html.

Rudolf Pecinovský *Návrhové vzory : 33 vzorových postupů pro objektové programování* 1. vyd. Brno: Computer Press, 2007. 527 s. ISBN 978-80-251-1582-4.

Bohdan Blaha *SQL Optimalizace v Oracle* Praha: Unicorn College, 2010. 47 s. Dostupné z: http://www.unicorncollege.cz/katalog-bakalarskych-praci/bohdan-blaha/attachments/Blaha_Bohdan_-_Optimalizace_SQL_dotaz%C5%AF_v_datab%C3%A1zi_Oracle.pdf.

Eli White *Habits of Highly Scalable Web Applications* DCPHP Conference 2009.

Tomáš Vondra *Replikace v PostgreSQL* CSPUG Konference 2011.

R. Fielding, UC Irvine, J. Gettys, J. Mogul, Compaq, H. Frystyk, L. Masinter, Xerox, P. Leach, Microsoft, T. Berners-Lee, W3C/MIT *RFC:2616 - Hypertext Transfer Protocol – HTTP/1.1.*

Mark Nottingham *CACHING TUTORIAL for Web Authors and Webmasters* [online]. Dostupné z: http://www.mnot.net/cache_docs/.

Jakub Onderka *Zdrojak.cz: Profilování PHP skriptů pomocí XHProf* [online]. Dostupné z: http://www.zdrojak.cz/clanky/profilovani-php-skriptu-pomoci-xhprof/.

Vito Chin *Techportal Ibuildings: Understanding APC* [online]. Dostupné z: http://techportal.ibuildings.com/2010/10/07/understanding-apc/.

Brad Fitzpatrick *Linux Journal: Distributed Caching with Memcached* [online]. Dostupné z: http://www.linuxjournal.com/article/7451.

Will Reese *Linux Journal: Nginx: the High-Performance Web Server and Reverse Proxy* [online]. Dostupné z: http://www.linuxjournal.com/magazine/nginx-high-performance-web-server-and-reverse-proxy.

Ilya Grigorik *Igvita.com: Nginx and Memcached, a 400% boost!* [online]. Dostupné z: http://www.igvita.com/2008/02/11/nginx-and-memcached-a-400-boost/.

Bc. Pavel Mikulka *Implementace CDN a clusteringu v prostředí GNU/LINUX s testy výkonnosti*, Brno: VUT, 2008. 65 s. Dostupné z: http://www.vutbr.cz/www$_b$ase/zav$_p$race$_s$oubor$_v$erejne.php?file$_i$d = 7793.

Petr Hrůša *Výhody a nevýhody virtualizace platforem a aktuální nabídka* [online]. Dostupné z: http://www.aquasoft.eu/blog/nazor$_o$dbornika.php?nazor = 450.

17 Přílohy