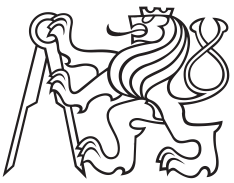
Bakalářská práce



České
vysoké
učení technické
v Praze

F3

Fakulta elektrotechnická
Katedra telekomunikační techniky

Analýza chování serverových aplikací při přetížení

Jaroslav Skřivan

Květen 2015
Vedoucí práce: Ing. Pavel Bezpalec, Ph.D.

Poděkování / Prohlášení

V první řadě bych rád poděkoval Ing. Pavlu Bezpalci, Ph.D. za možnost vyhotovit práci na dané téma, za jeho průběžný dohled nad prací a podnětné komentáře při její tvorbě. Dále bych rád poděkoval RNDr. Petru Olšákovi za kvalitně připravenou šablonu, kterou jsem pro tuto práci použil. V neposlední řadě patří poděkování také mé přítelkyni Andree za jazykovou korekturu textu.

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

V Praze dne

.....

Abstrakt / Abstract

Tato práce se věnuje možnostem optimalizace serverů, na kterých jsou provozovány webové aplikace psané v jazyce PHP a využívající MySQL databázi. Cílem práce je ukázat možnosti zvýšení výkonu celé infrastruktury bez zásahu do kódu aplikace.

Při řešení daného problému jsem testoval HTTP kompresi a HTTP cache. Obě vedly k výraznému snížení vytížení konektivity serveru. Dále jsem zkoumal možnosti optimalizace PHP. Změřil jsem, že PHP 7 a HHVM přinášejí několikanásobné zvýšení výkonu na rozdíl od opcode cache, jež zvýšuje výkon jen o polovinu.

V kapitole o MySQL popisuji query cache, jejíž přínos je v různých prostředích odlišný a obecně doporučuji ji vypnout. Dále jsem při zkoumání storage engine tabulek v MySQL databázi došel k závěru, že nejvhodnější je použití InnoDB. Poslední kapitola je věnována nejdůležitějším parametrům linuxového jádra souvisejícím s TCP spojeními. Zjistil jsem, že výchozí nastavení parametrů není vhodné pro velmi zatížené servery a mělo by být zvýšeno.

Práce dokazuje, že je možné zvýšit výkon webových aplikací úpravou konfigurace serverů.

Klíčová slova: TCP, HTTP, PHP, HHVM, MySQL, linuxové jádro, optimalizace výkonu

This thesis deals with the possibilities of optimising servers hosting web applications which are written in PHP and which store data in the MySQL database. The aim of the thesis is to show the options of increasing the infrastructure's performance with no interference into the application code.

First of all, HTTP compression and HTTP caching were tested. Both resulted in a significant reduction in the server's connectivity utilisation. I then proceeded to the examination of the possibilities of PHP optimisation. My measurements have shown that the deployment of PHP 7 as well as HHVM results in a multiple performance increase, unlike the deployment of opcode cache, which increases the performance only by 50%.

Query cache is described in the chapter on MySQL. Its effect varies in different environments and I generally recommend to turn it off. After examining storage engines in MySQL, I have established InnoDB as the most suitable of all. The last chapter focuses on the most important parameters of the Linux kernel which influence the behaviour of TCP connections. I have determined that the default settings are unsuitable for heavily loaded servers and should be increased.

The thesis proves that it is possible to increase the performance of web applications by adjusting the servers' configuration.

Keywords: TCP, HTTP, PHP, HHVM, MySQL, linux kernel, performance optimisation

Title translation: Analysis of server applications' behaviour during overload

Obsah /

1 Úvod 1
1.1 Motivace 1
1.2 Cíle práce 1
2 Teoretická část 2
2.1 TCP protokol 2
2.1.1 Navázání TCP spojení 3
2.1.2 Ukončení TCP spojení 3
2.1.3 Zobrazení spojení v linuxu 4
2.2 HTTP protokol 4
2.3 Zpracování požadavku na webovém serveru 5
2.3.1 Prefork 5
2.3.2 Vícevláknový přístup 5
2.3.3 Řízený událostmi (Event driven) 6
2.4 PHP 6
2.4.1 PHP jako modul do Apache2 6
2.4.2 PHP-CGI/PHP-FastCGI/PHP-FPM 6
2.4.3 HHVM 7
2.4.4 PHP Sessions 7
3 Popis měření 8
3.1 Měřená aplikace 8
3.2 Měřicí nástroj a postup měření 8
3.3 Měřicí prostředí 9
4 Optimalizace webového serveru 11
4.1 HTTP komprese 11
4.2 Cache na straně klienta 11
4.3 Keep-alive spojení 13
5 Optimalizace PHP 14
5.1 Opcode cache 14
5.2 Srovnání výkonu různých verzí PHP 15
5.3 Srovnání výkonu PHP 7 a HHVM 15
5.4 MySQL komprese 16
5.5 Realpath cache 16
5.6 Výkon open basedir 17
5.7 Zámky na PHP sessions 18
5.8 Plánování zdrojů 18
5.9 Výběr webového serveru 19
6 Optimalizace serveru MySQL 21
6.1 Storage engine 21
6.2 Query cache 22
7 Optimalizace jádra 23
7.1 Počet otevřených souborů 23
7.2 Lokální rozsah portů 23
7.3 Počet opakovaného odeslání nepotvrzeného paketu 23
7.4 Doba ukončení spojení 24
7.5 Fronta příchozích spojení 24
8 Vyhodnocení 25
Literatura 27
A Seznam použitých zkratek 29

Obrázky

2.1. Schéma TCP segmentu 2
2.2. Schéma navázání TCP spojení .. 3
2.3. Schéma ukončení TCP spojení .. 3
2.4. Ukázka výstupu příkazu netstat 4
2.5. Ukázka HTTP požadavku 4
2.6. Ukázka HTTP odpovědi 5
3.1. Ukázka použití programu Siege .. 8
3.2. Schéma zapojení serverů 9
4.1. Vliv HTTP komprese na využití sítě a zatížení procesoru .. 11
4.2. Ukázka HTTP hlavičky klienta s informací o cache 12
4.3. HTTP hlavička serveru s informací o cache 12
4.4. Vliv cachování prohlížeče na využití sítě a na počet požadavků na server 12
4.5. Vliv nastavení Keep-Alive 13
5.1. Vliv opcode cache na výkon PHP 14
5.2. Porovnání výkonu několika verzí PHP 15
5.3. Srovnání výkonu PHP7 a HHVM 16
5.4. Vliv MySQL komprese na výkon webu 16
5.7. Rozdíl výkonu Nginx a Apache2 19
5.8. Závilost počtu odpovědí serveru na době generování odpovědi 20

Kapitola 1
Úvod

Skriptovací programovací jazyk PHP a databázový server MySQL patří dnes mezi velmi populární technologie využívané v mnoha projektech. Není nejspíše nutné zmiňovat, že jedním z největších z nich je momentálně Facebook. Ačkoli se optimalizaci PHP a MySQL věnovalo již nemálo času, díky čemuž jsou tyto technologie obecně považovány za velmi výkonné, mají i svá úskalí, jejichž neznalost může vést k mnoha problémům a výpadkům.

1.1 Motivace

V pracovním životě jsem si již nespočetněkrát ověřil, že každý má odlišné zkušenosti a jednou z nejdůležitějších částí spolupráce je sdílet své znalosti a poznatky. To vede ke zlepšení kvality práce celého kolektivu, jakož i k zajištění stability provozovaných aplikací a předcházení zbytečným problémům. Sdílení poznatků pomocí dokumentů podobných této práci může juniorům pomoci stát se rychleji plnohodnotným členem týmu a seniorům může otevřít prostor pro širší diskuzi. V konečném důsledku vede vše k vylepšení služby a rozšíření znalostní základny.

Práce obsahuje mé znalosti nabyté během několika let správy serverů jednoho z nejnavštěvovanějších portálů na českém internetu. Bohužel byly často získány až řešením (často zbytečné) nedostupnosti webu. Rád bych touto cestou tyto znalosti a zkušenosti předal dál všem, kteří se potýkají s podobnými problémy. V budoucnu mám v plánu dokument nadále rozšiřovat o další poznatky, které díky novým projektům získám ať už já či mí kolegové.

1.2 Cíle práce

Tato bakalářská práce si klade dva cíle.

Cílem měřitelným bude porovnat výkon při použití PHP aplikací na různých technologiích, použití optimalizovat a následně změřit rozdíl ve výkonu, který byl takto dosažen pouze změnou konfigurace. Zajímavé v tomto měření bude porovnání výkonu několika verzí PHP, a tím i zodpovězení otázky, zda se z pohledu výkonu serveru vyplatí udržovat kód kompatibilní s nejnovější verzí či nikoli. Další zajímavé porovnání bude s HHVM, které Facebook vyvíjí právě kvůli optimalizaci výkonu a zrychlení načítání webu, a tím i snížení počtu serverů potřebných k provozu.

Druhým cílem bude sdílení vlastních zkušeností s provozem webů s vysokou návštěvností. Tato práce by měla pomoci nejen správcům, ale i vývojářům vypořádat se s nejčastějšími problémy, které běžně nastávají.

Kapitola 2
Teoretická část

Teoretická část je rozdělena do podkapitol dle jednotlivých vrstev, kterými musí uživatelův požadavek při zpracování projít. Vybrány a popsány jsou pouze ty části vrstev, se kterými se bude pracovat v praktických kapitolách této práce. [1]) Jako první je popsáno TCP, následují komunikační protokol mezi uživatelem a webovým serverem HTTP, samotné zpracování požadavku na úrovni webového serveru a nakonec popis funkčnosti PHP spolu s možnostmi jeho napojení na webové servery.

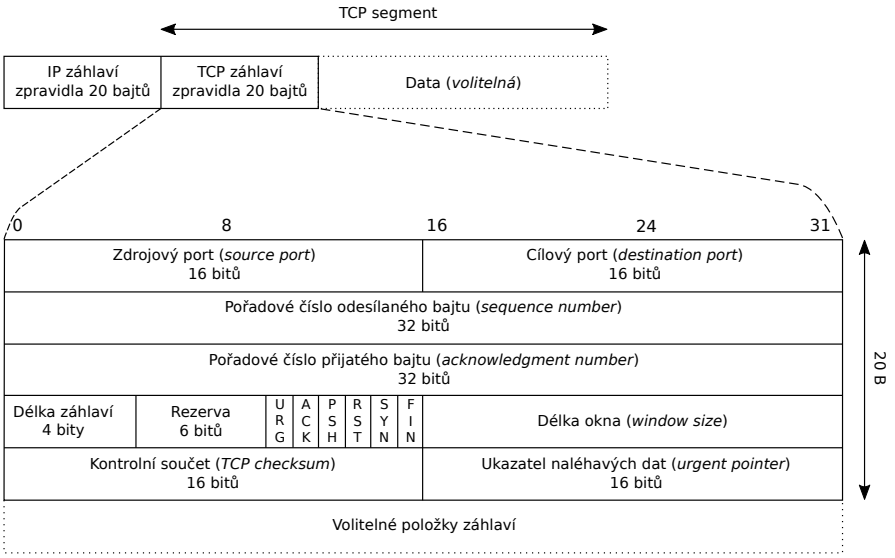
2.1 TCP protokol

Protokol TCP je protokolem 4. vrstvy ISO/OSI modelu a zajišťuje komunikaci mezi dvěma konkrétními aplikacemi běžícími na počítačích, které spolu komunikují. Protokol TCP se stará o správné navázání spojení, zabezpečuje přenos dat a ukončení spojení. Když dojde k problému při komunikaci, zajistí opravu opětovným odesláním dat, případně aplikaci informuje o situaci, ze které se nelze zotavit.

Pro potřeby této práce jsou z TCP protokolu nejdůležitějšími prvky zdrojový port, cílový port a příznak. Ostatní části jsou dobře popsány v 9. kapitole knihy [1].

Zdrojový a cílový port jsou 16bitová čísla, jež mohou nabývat hodnot 0 - 65535. Při komunikaci slouží kombinace zdrojové adresy, zdrojového portu, cílové adresy, cílového portu a protokolu k jednoznačnému určení spojení.

Příznak slouží k identifikaci stavu spojení. Příznaky jsou dále popsány v podkapitolách 2.1.1 a 2.1.2.



Obrázek 2.1. TCP segment [2])

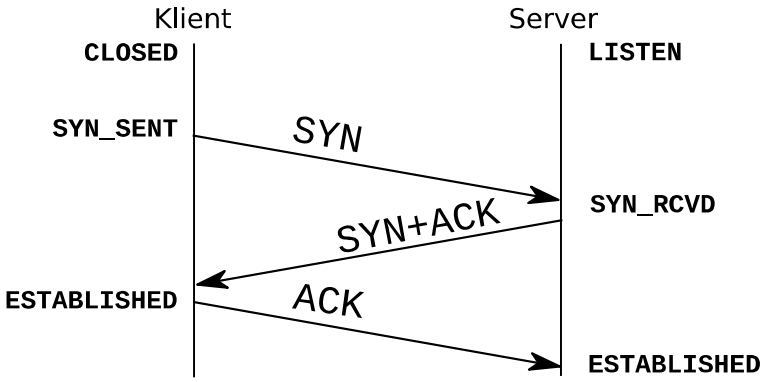
[1]) Všem vrstvám se zevrubně věnují zdroj [1]
[2]) Obrázek je převzat ze zdroje [1]

2.1.1 Navázání TCP spojení

Aby mohlo dojít k přenosu dat mezi dvěma počítači v síti, je nejdříve potřeba navázat spojení. Při navazování spojení se jedna aplikace, nazývaná klient, připojuje k druhé aplikaci, nazývané server, která naslouchá na určitém portu a čeká na příchozí spojení.

Před začátkem komunikace si klient zvolí číslo portu, který bude používat. Pokud nemá žádné preference, může použít číslo 0 a operační systém mu přidělí jedno z volných čísel.

Spojení je navázáno výměnou tří TCP segmentů. První segment jde od klienta k serveru a má příznak SYN, následuje odpověď od serveru s příznakem SYN a ACK. Na závěr ještě klient potvrdí serveru, že tento segment přijal, odesláním segmentu s příznakem ACK. Po této výměně je spojení navázáno. Pro nás jsou důležité stavy, které můžeme vidět v systému v jednotlivých fázích navazování spojení. Tyto stavy jsou zobrazeny v obrázku 2.2.

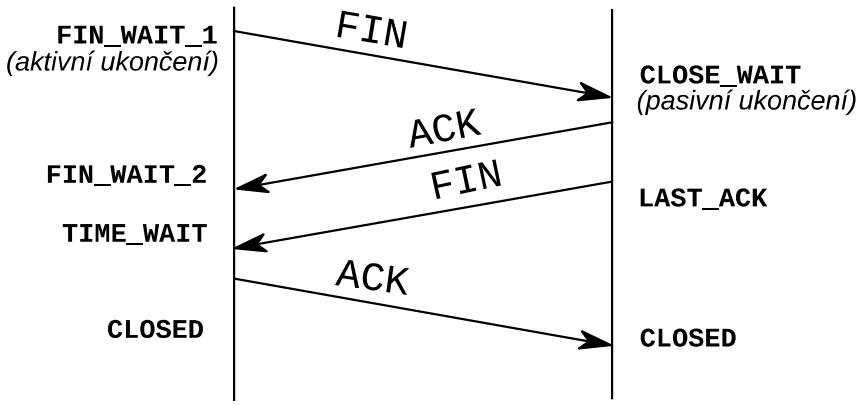


Obrázek 2.2. Třícestné navázání TCP spojení

2.1.2 Ukončení TCP spojení

Dojde-li jedna komunikující strana do stavu, kdy již odeslala všechna data a už žádná další odesílat nebude, může začít proces ukončování spojení zasláním segmentu s příznakem FIN. Tato strana provádí aktivní ukončení spojení a již nesmí odeslat žádná data. Druhá strana ale i nadále může odesílat data a bude provádět pasivní ukončení spojení.

Důležité jsou při ukončování spojení stavy, jejichž jednotlivé fáze lze vidět v systému.



Obrázek 2.3. Schéma ukončení TCP spojení

Jak je zřejmé z diagramu 2.3, jedna ze stran začne aktivní ukončení spojení odesláním segmentu s příznakem FIN a změní svůj stav spojení na FIN_WAIT1, pasivní strana se

po přijetí tohoto segmentu přepne do stavu CLOSE_WAIT a zašle zpět potvrzení, čímž aktivní stranu přepne do stavu FIN_WAIT2. V tomto stavu aktivní strana už nemůže odesílat data, ale stále může data přijímat. V momentě, kdy i pasivní strana dokončí odesílání dat, odešle i ona FIN a přepne se do stavu LAST_ACK, ve kterém již jen čeká na potvrzení. Aktivní strana se po přijetí posledního FIN segmentu přepne do stavu TIME_WAIT, následné odeslání potvrzení pak uzavře spojení na obou stranách.

2.1.3 Zobrazení spojení v linuxu

Pro zobrazení všech aktuálních spojení v linuxu slouží příkaz nestat. Další možností získat seznam spojení je soubor /proc/net/tcp, respektive /proc/net/tcp6 pro IPv6.

```
$ netstat -ant
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address       Foreign Address     State
tcp        0      0 127.0.0.1:53        0.0.0.0:*           LISTEN
tcp        0      0 0.0.0.0:22          0.0.0.0:*           LISTEN
tcp        0    232 109.123.223.232:22  78.45.8.79:49876    ESTABLISHED
```

Obrázek 2.4. Ukázka výstupu příkazu netstat

Další často užívaný příkaz je lsof, který zobrazuje mimo jiné i další informace o běžících procesech.

2.2 HTTP protokol

Při komunikaci na webu se používá protokol 7. vrstvy ISO/OSI modelu nazvaný HTTP (Hypertext Transfer Protocol). Pro požadavek i odpověď se používá stejný formát zpráv, skládající se z povinné hlavičky a volitelné datové části. Tyto dvě části se oddělují jednou prázdnou řádkou. Hlavička obsahuje všechny potřebné informace, jako je dotazovaná adresa, dotazovaný server (host), jazyková sada klienta, komprese a mnoho dalších.

Při komunikaci s webovým serverem vytváří spojení klient (nejčastěji prohlížeč), který sestaví a odešle požadavek na server, kde je zpracován, a klient pak příjme odpověď. V rámci navázaného spojení pak existují další možnosti komunikace, ale to není pro naše účely důležité.

Samotný protokol HTTP byl navržen jako bezstavový. To znamená, že v rámci protokolu není držena žádná souvislost mezi požadavky jednoho uživatele a webový server všechny zpracovává jako by byly první. Řešení této situace je popsáno v kapitole 2.4.4 o PHP sessions.

```
GET /tools.html HTTP/1.1
Host: www.joes-hardware.com
```

Obrázek 2.5. Ukázka HTTP požadavku

```
HTTP/1.1 200 OK
Date: Sun, 01 Oct 2000 23:25:17 GMT
Server: Apache/1.3.11 BSafe-SSL/1.38 (Unix) FrontPage/4.0.4.3
Last-Modified: Tue, 04 Jul 2000 09:46:21 GMT
ETag: "373979-193-3961b26d"
Accept-Ranges: bytes
Content-Length: 403
Connection: close
Content-Type: text/html
```

```
<HTML>
<HEAD><TITLE>Joe's Tools</TITLE></HEAD>
<BODY>
<H1>Tools Page</H1>
<H2>Hammers</H2>
<P>Joe's Hardware Online has the largest selection of hammers on
the earth.</P>
<H2><A NAME=drills></A>Drills</H2>
<P>Joe's Hardware has a complete line of cordless and corded drills, as
well as the latest in plutonium-powered atomic drills, for those big
around the house jobs.</P> ...
</BODY>
</HTML>
```

Obrázek 2.6. Ukázka HTTP odpovědi od serveru

Uvedené příklady pochází ze zdroje [2], který také posloužil pro přípravu této kapitoly.

2.3 Zpracování požadavku na webovém serveru

Většina serverových aplikací, které naslouchají na nějakém portu a zpracovávají příchozí požadavky, se musí vypořádat se situací, kdy požadavky nechodí jeden po druhém za sebou, ale server jich musí obsloužit více v jeden moment. Na webu tomu není jinak, a každý webový server proto musí tuto vlastnost splňovat. Následují nejčastěji používané přístupy, jak se s touto úlohou vypořádat.

2.3.1 Prefork

Tento způsob je nejčastější u webového serveru Apache2. Webový server při spuštění vytvoří několik dalších podprocesů (forků), které budou zpracovávat dotazy. Přijde-li na webový server požadavek, jeden z připravených procesů jej příjme, zpracuje a klientovi odešle odpověď. V průběhu zpracování požadavku je tento proces blokován, nemůže se tedy věnovat ničemu jinému. Chceme-li, aby náš server mohl obsloužit 200 současných spojení, musí mít připraveno 200 procesů.

Ačkoli se tato metoda může zdát neefektivní, má svá pozitiva. Jednou z hlavních výhod je fakt, že procesy mohou vznikat a zanikat bez jakéhokoli dopadu na aplikaci. Pokud tedy máme v aplikaci chybu a víme, že po 1000 požadavcích začne být proces nestabilní například kvůli špatnému uvolňování paměti, můžeme ho ukončit a spustit nový. Výrazným negativem je u preforku zcela jistě paměťová náročnost, která vzniká z nutnosti udržovat velké množství procesů.

2.3.2 Vícevláknový přístup

Dalším důležitým přístupem též dostupným v implementaci Apache2 (worker), který serverové aplikace používají, je přístup vícevláknový. Každý požadavek vytvoří jedno vlákno, ve kterém je obsloužen. Vlákna samozřejmě mohou být připravena z thread poolů, takže se stejně jako u preforku nemusejí vytvářet při každém požadavku. Značnou výhodou je zde mnohem menší paměťová náročnost.

Nevýhodou tohoto přístupu však je to, že všechny části a moduly musí být psány tak, aby podporovaly vícevláknový přístup. To není vždy dodrženo, a proto není v některých případech možné tuto konfiguraci použít. Při vysokém počtu vláken navíc dochází k situacím, kdy aplikace často mezi vlákny přepíná, čímž zvyšuje náročnost na procesor.

2.3.3 Řízený událostmi (Event driven)

Princip tohoto přístupu, používaného ve webserverech lighttpd a nginx, je úplně odlišný od přístupů zmíněných výše. Základní myšlenka je, že existuje jeden proces s jedním vláknem, které v cyklu zpracovává frontu událostí. Každý požadavek, který je vytvořen na takovýto server, je rozdělen na velmi mnoho malých částí, přičemž každá část vrací callback na další. Požadavek je zpracováván postupně. Žádná z částí však nesmí použít blokované volání, na které by musel proces čekat. Jako příklad uveďme čtení požadavku od uživatele. Je zbytečné blokovat proces při čekání na data od uživatele, když můžeme definovat callback, který se má zavolat, až data dostaneme. Mezitím procesor může zpracovávat další požadavky.

Tento přístup je velmi populární na webových serverech, které zpracovávají velké množství požadavků, právě z důvodu, že jsou málo náročný na systémové zdroje. Jeho nevýhodou je, že proces může vytížit jen jedno jádro procesoru. To se však dá řešit vytvořením více procesů. Pro správné zatížení serveru je tak doporučeno vytvářet tolik procesů, kolik má server jader.

2.4 PHP

PHP je víceúčelový skriptovací jazyk spouštěný na straně serveru. Je navržen k vývoji webů, ale používá se také jako obecný programovací jazyk.

Jazyk PHP je navržen jako thread-safe, díky čemuž ho můžeme nasadit i v kombinaci s webovým serverem Apache2, jenž bude požadavky zpracovávat ve vláknech. Bohužel ne všechna rozšíření PHP toto pravidlo dodržují, a proto se tato varianta příliš často nepoužívá. Takovými rozšířeními jsou například setlocale a gettext. Z tohoto důvodu bude v této práci k PHP přistupováno tak, že každý zpracovávaný požadavek používá a blokuje jeden proces, aby požadavek vyřídil.

PHP procesy mezi sebou nesdílejí žádný kontext, proto veškeré sdílení dat musí být provedeno na aplikační úrovni. Chceme-li tedy v aplikaci například udělat cachování dat, musíme tuto vlastnost implementovat na úrovni kódu pomocí sdílené paměti, memcached, souborů či jiných technologií.

2.4.1 PHP jako modul do Apache2

Velmi oblíbenou a nejjednodušší variantou je nasadit PHP jako rozšíření do Apache2. Veškerá komunikace mezi webovým serverem a PHP interpretem probíhá v jednom procesu, PHP skripty jsou proto vykonávány se všemy právy webového serveru.

Značnou nevýhodou tohoto přístupu je složitější plánování zdrojů serveru. Nikdy totiž nevíme, zda proces webového serveru bude zpracovávat PHP skript, či jen statický obsah, který je výrazně méně náročný. Více o plánování zdrojů server v podkapitole 5.8.

2.4.2 PHP-CGI/PHP-FastCGI/PHP-FPM

CGI (Common Gateway Interface) je jednoduché rozhraní navržené za účelem spouštět externí programy informačním serverem nezávisle na platformě. Rozhraní CGI je popsáno v [3].

Celý princip fungování PHP využívajícího CGI je ten, že v momentě, kdy webový server zjistí, že požadavek má být zpracován PHP, vytvoří nový PHP proces, kterému předá všechny potřebné informace, nechá jej požadavek zpracovat a následně jej opět ukončí. Tento způsob má velkou výhodu v tom, že oddělíme zpracování statického obsahu od dynamického. Zdroje se pro tyto obsahy tudíž dají plánovat separátně. Neustálé

vytváření a ukončování PHP procesů bude mít však vysokou režii a server jimi bude zbytečně zatížen.

Za tímto účelem vzniklo FastCGI, které obohacuje stávající CGI o další rozšíření. FastCGI na rozdíl od CGI nechává procesy běžet dlouhodobě a jen jim předává požadavky ke zpracování, když je to potřeba. Po dokončení požadavku se proces uspí a spotřebovává tak minimum prostředků. FastCGI dále přináší možnost zpracovávat požadavky externě na různých serverech, což nám umožňuje úplně oddělit webový server od serveru zpracovávajícího dynamické požadavky.

PHP-FPM je alternativní implementace FastCGI, která obohacuje existující FastCGI o lepší správu procesů. Umožňuje dynamicky vytvářet a ukončovat procesy podle aktuálního vytížení nebo je násilně ukončit v případě, že požadavek trvá déle než je stanovený limit. To se hodí v momentech, kdy přestane fungovat nastavení max_execution_time.

2.4.3 HHVM

HHVM (HipHop virtual machine) je alternativní PHP interpret, za jehož vývojem stojí Facebook. Jeho hlavní předností je JIT (Just In Time) kompilace, známá též pod názvem dynamický překlad. Tato metoda je známá z jazyka Java, kde při spuštění aplikace dochází k překladu kódu podle aktuální platformy. Na rozdíl od Javy provádí HHVM překlad během zpracování požadavku, spolu s dalšími analýzami. Tím může zpracování kódu ještě více optimalizovat. Jako příklad zde poslouží volání funkce, kdy HHVM sleduje datové typy vstupních a výstupních proměnných, což následně použije při optimalizaci. Facebook je ve vývoji HHVM velmi aktivní. Zveřejnil jazyk Hack, vycházející z PHP, ale rozšířený o další funkce. Těmi jsou například type hinting, asynchronní volání, generické třídy a mnoho dalších.

HHVM bohužel používá vícevláknový přístup zpracování požadavku, což může přinést komplikace při jeho nasazení. Týká se to hlavně aplikací využívajících setlocale a další funkce, jež nejsou na vícevláknový přístup připraveny. Původně HHVM obsahoval i webový server, aktuálně je však tato vlastnost již odstraněna a k HHVM se lze připojit již jen pomocí protokolu FastCGI za použití jiného webového serveru.

2.4.4 PHP Sessions

Jak již bylo zmíněno v podkapitole 2.2, HTTP je bezstavový protokol. To přináší komplikace v aplikacích, kde potřebujeme sdílet stav mezi požadavky, například při přihlášení na web a při procházení části určené jen přihlášeným uživatelům. HTTP protokol však podporuje cookies, což je část hlavičky, která může obsahovat různá data o maximální velikosti 4kB. Cookies mohou být nastaveny serverem v hlavičce odpovědi, nebo klientem, například pomocí javascriptu. PHP řeší problém bezstavového HTTP pomocí sessions, tedy serializovaných objektů uložených na serveru. Do jedné cookie je uložen jen náhodně vygenerovaný řetězec, který každou session jednoznačně identifikuje. Většinou se tato cookie jmenuje PHPSESSIONID, to se však dá změnit v nastavení PHP. Alternativou je přenášení tohoto identifikátoru v adrese požadavku, to se dnes již ale příliš nepoužívá.

Kapitola 3
Popis měření

V této kapitole bude představena aplikace, pomocí které budeme měřit, stejně tak jako nástroje, jež k měření použijeme. Dále bude popsána metoda měření. Kapitolu ukončíme popisem serverového vybavení a schématem zapojení jednotlivých serverů.

3.1 Měřená aplikace

Každý kód má jinou rychlost zpracování a je různě optimalizován. Při zpracování požadavku jsou často využívány i další služby (např. databáze), které aplikaci poskytují data nebo rozšiřující funkce. Kód v průběhu zpracování vytváří nová spojení, zasílá požadavky a čeká na odpovědi. Velmi časté je tudíž použití PHP jazyka spolu s MySQL databází. Z výše uvedených důvodů byla pro učely našeho měření vybrána aplikace Wordpress, která se reálné aplikaci velmi podobá a je i velmi oblíbená. Wordpress je nainstalován v aktuální verzi 4.1.2 bez jakýchkoli rozšíření. Záměrně tedy nejsou použita ani rozšíření Wordpressu na cachování, protože se nesnažíme maximalizovat výkon jednoho konkrétního webu, nýbrž celé infrastruktury.

3.2 Měřicí nástroj a postup měření

K samotnému měření byl vybrán nástroj Siege, dostupný v oficiálních repozitářích Debianu. Jednou z významných vlastností Siege je možnost volat více URL adres, které jsou poskytnuty v souboru. URL adresy je možné ze souboru vybírat náhodně, čímž je lépe simulováno chování uživatele. Mezi další vlastnosti Siege, kterých budeme při měření využívat, patří možnost omezit délku měření počtem volání nebo časem.

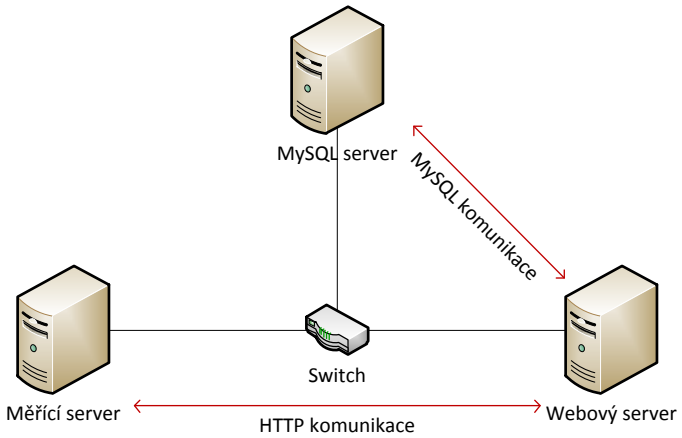
```
$ siege --concurrent=400 --benchmark --time=5M --file=./seznam-url-adres
** SIEGE 2.70
** Preparing 400 concurrent users for battle.
The server is now under siege...
Lifting the server siege...      done.
Transactions:              24232 hits
Availability:             100.00 %
Elapsed time:             299.87 secs
Data transferred:         214.05 MB
Response time:              4.91 secs
Transaction rate:          80.81 trans/sec
Throughput:                 0.71 MB/sec
Concurrency:              396.55
Successful transactions:   24232
Failed transactions:           1
Longest transaction:        9.35
Shortest transaction:       1.49
```

Obrázek 3.1. Ukázka použití programu Siege

Na webovém serveru nás zajímá zátěž procesoru a výkon serveru při přetížení. Z tohoto důvodu bude pro každou měřenou konfiguraci změněn počet procesů nebo vláken zpracovávajících požadavky. Začneme se 16 procesy, neboť 16 je zároveň počet vláken serveru dostupných. Postupně budeme počet procesů zvyšovat na 24, 32 a 64. Experimentálně potom změříme jednu konfiguraci i s větším počtem procesů. V situacích, kdy bude na databázovém serveru zapnuta query cache, budeme měřit i 8 procesů, abychom získali výsledky, kdy server nebude přetížen. Za účelem simulace připojených klientů budeme každou konfiguraci zatěžovat jedním až dvěma sty procesy generujícími požadavky na server. V průběhu měření bude sledován nejen počet správně vygenerovaných odpovědí, ale i průměrná doba odpovědi. Jelikož budeme v průběhu měření webový server restartovat, proběhne před každým měřením zahřívací kolo bez měření výsledků. Tím se naplní cache webového i databázového serveru. Poté proběhne samotné měření. Na server budou zasílány požadavky v daném počtu připojených klientů vždy po dobu jedné minuty.

3.3 Měřicí prostředí

Pro účely měření budou použity tři servery, přičemž jeden bude sloužit ke spouštění měřicího nástroje, druhý bude obsluhovat webové požadavky a třetí bude sloužit pouze jako databázový server. Na všech serverech byla nainstalována aktuálně stabilní verze Debianu – Wheezy. Hardwarové konfigurace byly vybrány tak, aby vždy došlo ke stoprocentnímu vytížení webového serveru a ostatní komponenty měly stále dostupný výkon a neovlivňovaly tak výsledky měření.



Obrázek 3.2. Schména zapojení serverů

Měřicím serverem bude jednoprocesorový server obsahující Intel Core2 Q6600 se čtyřmi jádry, 8GB DDR2 paměti a jeden rotační disk o kapacitě 320GB. Měřicí server bude většinu doby jen čekat na odpovědi od webového serveru, pročež je tato konfigurace dostačující.

Webový server bude obsahovat dva čtyřjádrové procesory Intel Xeon E5620 podporující Hyper-threading, 12GB DDR3 paměti a jedno SSD o kapacitě 240GB.

Databázový server se bude skládat ze dvou šestijádrových procesorů Intel Xeon L5640 podporujících Hyper-threading, 48GB DDR3 paměti a jednoho SSD o kapacitě 240GB. Na databázový server bylo náhodně vygenerováno 10GB dat. Data obsahují celkem 200 217 uživatelů, 200 225 článků a 2 450 134 komentářů, rovnoměrně rozložených mezi

články. Tím se minimalizují situace, že by se při měření jedna stránka načetla vícekrát, zároveň však bude databázový server schopen obsloužit všechny požadavky z paměti, a nebude tak webový server zpomalovat přístupem k disku.

Všechna zařízení budou propojena Mikrotikem 2011iL-IN, který disponuje pěti FastEthernet a pěti Gigabit Ethernet porty. Pro připojení serverů byly použity jen gigabitové porty, aby nedošlo k saturaci linky mezi servery.

Při měření budou použity následující verze PHP:

- PHP 5.4.40
- PHP 5.5.24
- PHP 5.6.8
- PHP 7.0.0 (git commit d880ead8a733202be2f74228339390e81ab824b5)
- HHVM 3.6.1

V některých měřeních bude použita opcode cache. V konfiguraci s PHP 5.4 bude použita xcache, od verze 5.5 bude použita nativní opcache.

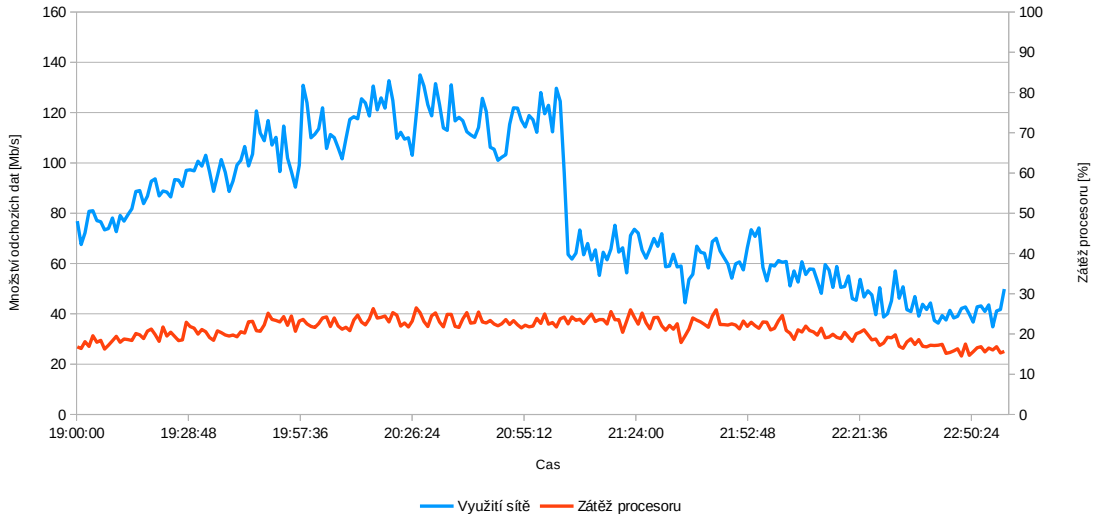
Ze zkupiny webových serverů budou měřeny Apache2 (prefork), kde bude PHP načten jako modul. Tato konfigurace patří mezi nejčastěji používané. Dále byl vybrán Nginx v kombinaci PHP-FPM, který je často skloňován jako řešení problémů s výkonem. A do třetice byl vybrán HHVM s webovým serverem Nginx.

Kapitola 4
Optimalizace webového serveru

Následující kapitoly se budou zabývat možnostmi optimalizace přenosu dat mezi serverem a klientem. V rámci této kapitoly budou všechny technologie testovány na reálném webovém serveru s reálnou zátěží od reálných uživatelů.

4.1 HTTP komprese

Vlastnost HTTP komprese umožňuje přenášet data mezi serverem a klientem v komprimované podobě. V hlavičce požadavku, který klient odesílá na server, může klient pomocí hlavičky `Accept-Encoding` specifikovat, v jakém kódování má server odeslat odpověď. Server následně do své odpovědi přidá hlavičku `Content-Encoding`, která obsahuje informace o zvoleném kódování.



Obrázek 4.1. Vliv HTTP komprese na využití sítě a zatížení procesoru

Přiložený obrázek 4.1 porovnává množství odchozích dat se zátěží procesoru reálného webového serveru, kde byla zapnuta HTTP komprese během večerní špičky. Propad na síťovém provozu je v tomto případě 50%, zatímco zátěž procesoru se nezměnila.

HTTP komprese může výrazně snížit šířku datového pásma, kterou server potřebuje na odbavení požadavků od klientů. Na výkon serveru nemá tato technologie žádný negativní vliv.

4.2 Cache na straně klienta

Snad každý web obsahuje kromě textu i CSS styly, javascript, obrázky nebo jiný podpůrný obsah, který se mění velmi zřídka. Tím pádem vzniká otázka, zda je nutné tyto soubory stahovat při každém načtení webu, či zda je lepší je cachovat na straně klienta.

Webový server může pomocí HTTP hlaviček v odpovědi napovědět prohlížeči, jak dlouho má soubor uchovávat v cache. Prohlížeč si takový soubor uloží na disk a už od webového serveru daný soubor nevyžaduje. Z dlouhodobého hlediska se takto dá ušetřit síťová konektivita a výrazně zrychlit načítání webu. Samotná cache se v rámci HTTP protokolu dá ovládat několika různými způsoby a je na klientské aplikaci, jak s tím naloží.

Úplně nejjednodušším způsobem je hlavička v odpovědi `Expires`, která klientské aplikaci sdělí datum a čas, kdy má vypršet platnost cache.

Webový server může ke každé odpovědi přidat hlavičku `Last-Modified`. Klient se následně při každém dalším požadavku jen zeptá, zda se daný obsah neměnil, a to pomocí hlavičky `If-Modified-Since`. Pokud se nic nezměnilo, je mu navrácena odpověď s kódem 304, v případě změny mu jsou zaslána nová data.

Etag je další HTTP hlavička velmi podobná `Last-Modified`. V tomto případě však nehraje roli čas, ale podpis souboru. Webový server u každého souboru spočítá hash a poskytne ho klientovi v odpovědi. Klient následně při dalším požadavku informuje server hlavičkou `If-None-Match`, kterou verzi souboru má uloženu.

V současné době nabízí nejrozšířenější možnosti nastavení cachování hlavička `Cache-control`, která mimo klasického omezení na čas myslí i na situace, kdy lze mít mezi serverem a klientem proxy server, který sám o sobě může také provádět cachování.

V rámci HTTP protokolu lze jednotlivé možnosti kombinovat. Následující ukázka obsahuje části HTTP hlavičky od klienta a serveru.

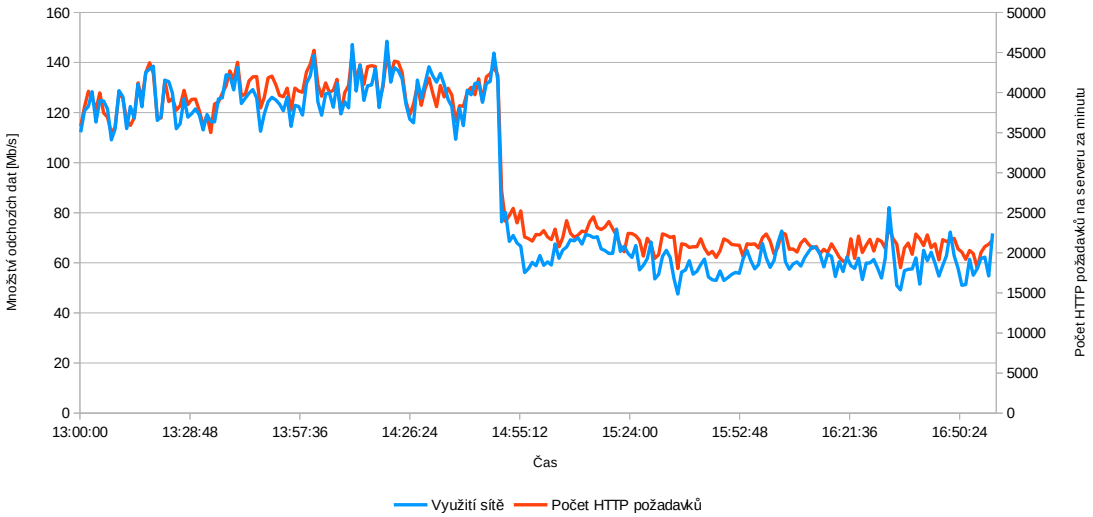
```
If-Modified-Since:Tue, 21 Apr 2015 14:17:29 GMT
If-None-Match:"55365bf9-1dc"
```

Obrázek 4.2. Ukázka části HTTP hlavičky klienta s informací o cache

```
Cache-Control:max-age=43200
ETag:"55365bf9-1dc"
Expires:Fri, 01 May 2015 22:11:45 GMT
Last-Modified:Tue, 21 Apr 2015 14:17:29 GMT
```

Obrázek 4.3. Ukázka části HTTP hlavičky serveru s informací o cache

Webové servery mají konfiguraci pro cachování již připravenu. Programátor se tudíž nemusí starat o správnost hlaviček, většinou stačí změnit několik málo nastavení a hlavičky se správně nastaví.



Obrázek 4.4. Vliv cachování prohlížeče na využití sítě a na počet požadavků na server

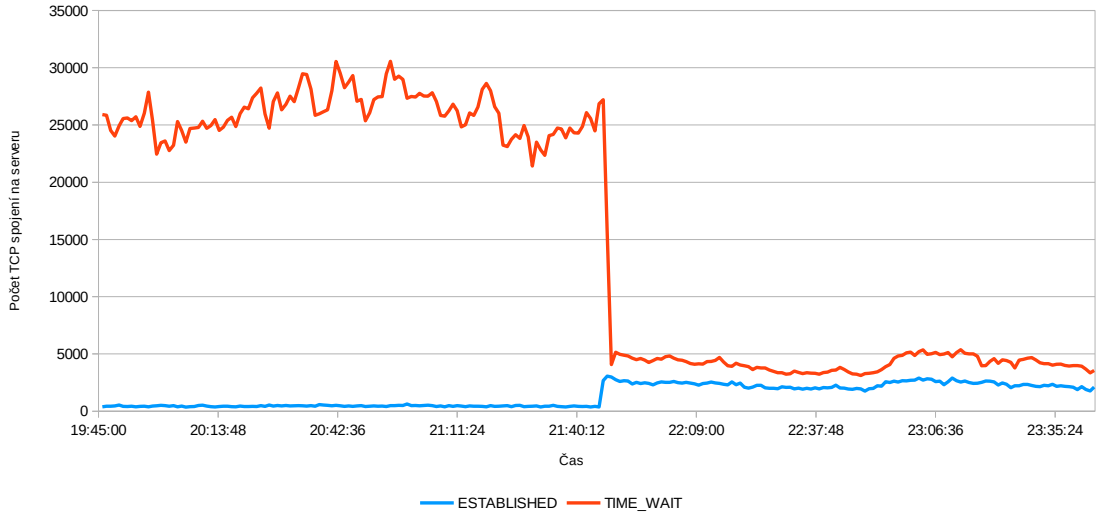
Na obrázku 4.4 je viditelný vliv HTTP cache na množství požadavků a využití sítě serveru. V tomto případě došlo k ušetření poloviny šířky pásma a webový server odbavuje polovinu požadavků.

U opravdu hojně navštěvovaných webů je potřeba myslet i na vedlejší efekt cachování na straně klienta. V momentě, kdy se na server nahraje nový obsah, který ještě klienti nemají stažený, dojde ke stažení. S rostoucím počtem změn a uživatelů roste i náročnost na síť a další prostředky serveru. Může proto dojít k situaci, kdy se aktualizací webu zapříčiní přetížení vlastních serverů. V roce 2008 se tato situace stala serverům seznam.cz.

4.3 Keep-alive spojení

Během načítání stránky se běžně stáhne několik desítek dalších souborů z daného serveru v podobě stylů, obrázků atd. Za účelem zrychlení načítání webu byla ve specifikaci HTTP/1.1 navržena možnost recyklace TCP spojení. Server klientovi v HTTP odpovědi řekne pomocí hlavičky `Connection: keep-alive`, že dané spojení má nechat otevřené a že jej může použít pro další HTTP požadavky. Tímto způsobem se ušetří režie na vytváření dalších spojení.

Samozřejmě má i tato technologie určité nevýhody. Používáme-li webový server, který využívá metodu zpracování prefork nebo vlákna, musíme při plánování počtu procesů a zdrojů počítat s rezervou pro keep-alive spojení. Většina webových serverů podporuje možnost omezit počet souběžných spojení ve stavu keep-alive. Stejně tak je podporován timeout pro ukončení spojení v případě, kdy nepříjde žádný požadavek, z důvodu, aby spojení nebylo navázáno napořád. Běžné hodnoty jsou v řádu jednotek vteřin. U webových serverů řízených událostmi je tento problém minimální, spojení konzumuje jen zanedbatelné množství paměti.



Obrázek 4.5. Vliv nastavení keep-alive na počet TCP spojení na serveru

Nemá-li webový server povoleno keep-alive spojení, vytváří klient pro každý požadavek nové spojení. To má za následek vyšší počet spojení, která server musí udržovat hlavně v TIME_WAIT stavu. Dobře je to viditelné na grafu 4.5, který porovnává počet spojení ve stavu ESTABLISHED a TIME_WAIT před a po zapnutí keep-alive.

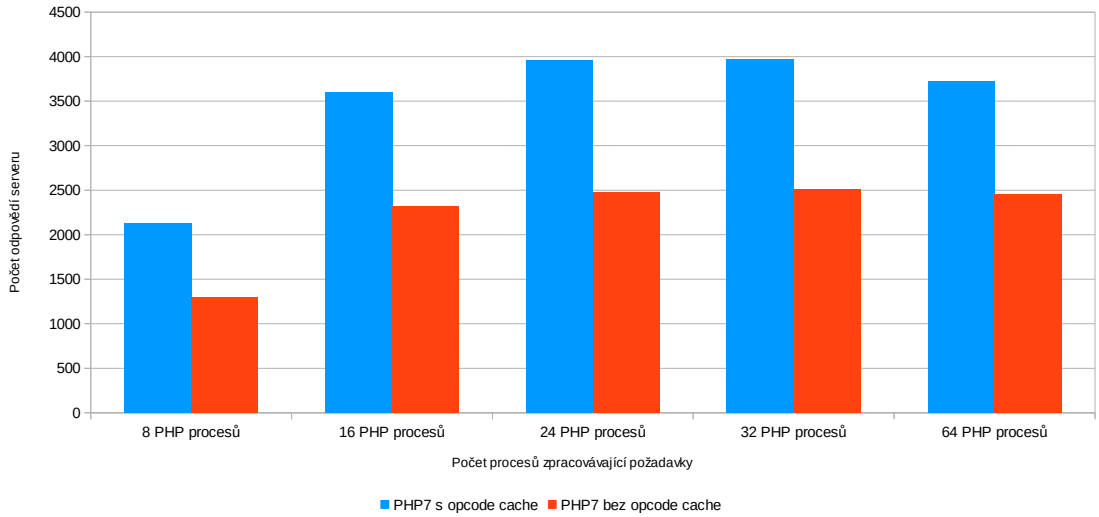
Kapitola 5
Optimalizace PHP

V této kapitole budou blíže rozebrány možnosti optimalizace PHP za účelem dosažení vyššího výkonu. Dále bude vyzkoušeno a porovnáno více verzí PHP. Nakonec zmíníme několik vlastností PHP, které mohou působit problémy.

5.1 Opcode cache

Pod pojmem optimalizace PHP si většina z nás představí opcode cache, která velmi často zapříčiňuje výrazný rozdíl ve výkonu. Opcode cache je systém, který kompiluje PHP kód do bajtkódu (bytecode) a ukládá si jej do paměti. To umožňuje používat již připravený bajtkód místo kompilace PHP při každém požadavku. Princip je velmi podobný činnosti HHVM, které toho však vykonává víc. Zevrubně jsou funkce opcode cache a její různé implementace popsány ve čtvrté kapitole zdroje [4]. Od PHP verze 5.5 [5] je již integrovaná implementace Zend Opcache a není tudíž potřeba instalovat žádná rozšíření.

Všechna měření probíhala na PHP verzi 7, přičemž na databázovém serveru byla zapnuta query cache.



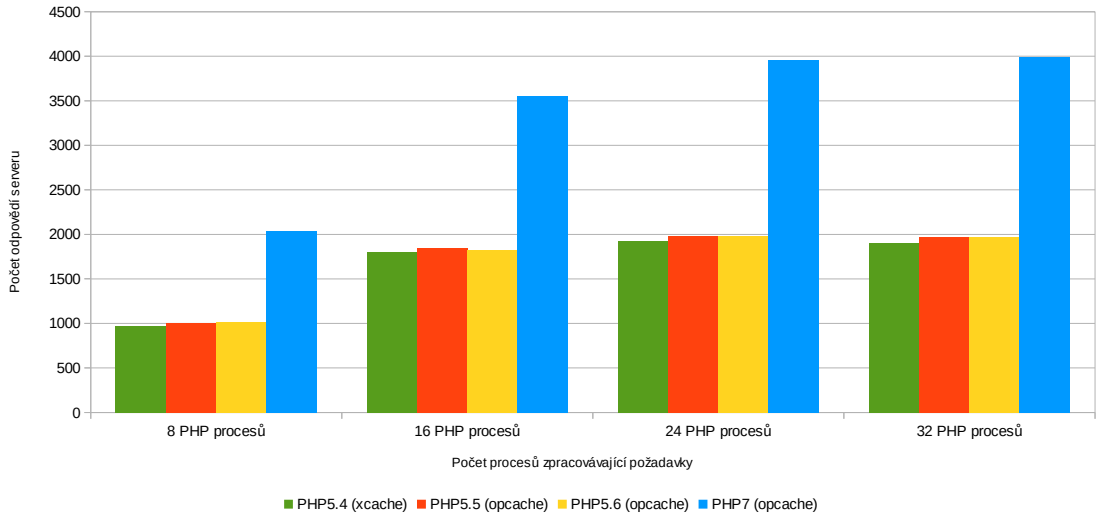
Obrázek 5.1. Vliv opcode cache na výkon PHP

Zapnutím opcode cache na serveru se zvýšil výkon z 2509 zodpovězených požadavků na 3973 v konfiguraci s 32 PHP procesy. Tato změna, která ve většině případů nemá žádný vliv na funkčnost prováděného kódu, přináší výrazné zlepšení výkonu.

Při měření bylo také vidět, že v momentě, kdy PHP nemusí čekat na vykonání SQL dotazů a dostane data ihned, je schopno vytížit všechna vlákna serveru na 100%. Další přidávání procesů zpracovávajících požadavky tedy nepřináší žádný růst počtu odpovědí od serveru.

5.2 Srovnání výkonu různých verzí PHP

Měření různých verzí PHP bylo provedeno v konfiguraci, při které byl webový server nginx v kombinaci s PHP-FPM v jednotlivých verzích. Na databázovém serveru musela být zapnuta query cache. Důvodem bylo PHP 7, jež dokázalo přetížit MySQL server, čímž došlo k negativnímu ovlivnění výsledků. Měřicí nástroj Siege byl spuštěn v 64 vláknech.



Obrázek 5.2. Porovnání výkonu několika verzí PHP

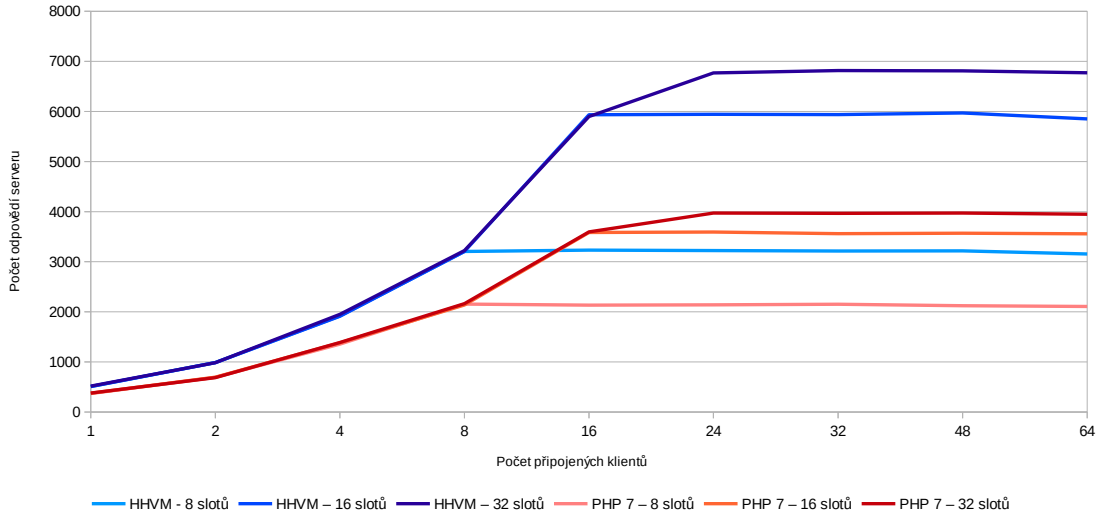
Z naměřených hodnot je vidět v obrázku 5.2, že s každou verzí PHP 5 dochází k nepatrnému zlepšení výkonu a server je schopen vrátit více odpovědí. V právě připravované verzi PHP 7 udělali vývojáři velký kus práce a dokázali zvýšit výkon PHP dokonce až na dvojnásobek.

5.3 Srovnání výkonu PHP 7 a HHVM

PHP verze 7 je PHP nové generace, ve kterém byla přepsána část kódu tak, aby dodávalo vyšší výkon a sníženo množství paměti. Vývojáři v PHP 7 výrazně přepsali interní struktury pro práci s proměnnými [6], čímž docílili zvyšeného výkonu oproti předchozí verzi PHP 5. Aktuálně však změny nejsou plně kompatibilní se starší verzí PHP. Na tom nicméně vývojáři pracuji.

S ohleden na vysoký výkon HHVM i PHP 7 bylo měření prováděno se zapnutou query cache na MySQL databázi a všechna spojení s databází byla komprimovaná. Naměřené hodnoty nám ukazují, že HHVM je výrazně rychlejší. PHP 7 je aktuálně skoro o polovinu pomalejší než řešení od Facebooku.

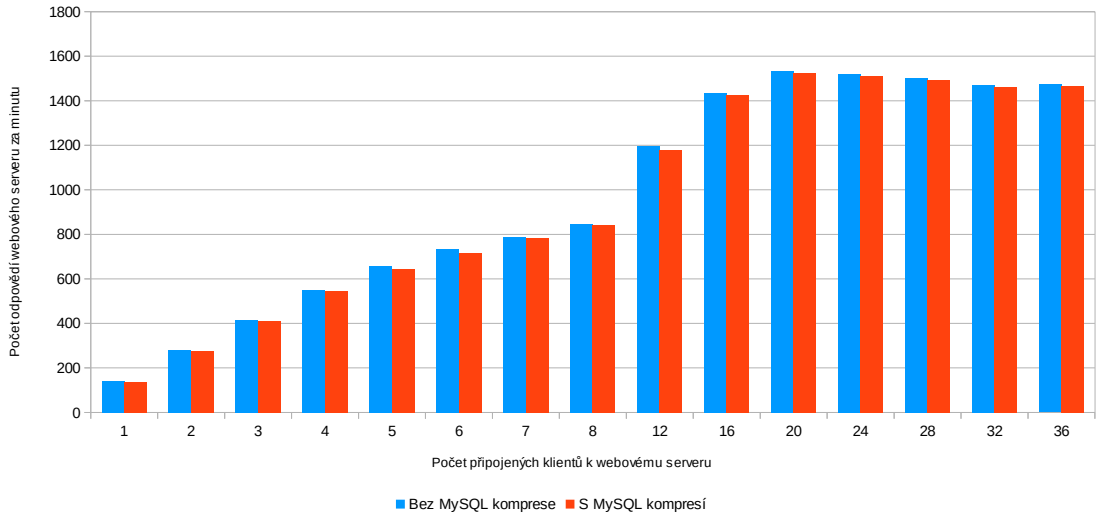
Porovnáme-li hodnoty měření s hodnotami, které jsem naměřil v kapitole 5.2 zjistíme, že HHVM je schopno obsloužit třikrát víc požadavků než aktuálně stabilní verze PHP 5.6.



Obrázek 5.3. Srovnání výkonu PHP 7 a HHVM

5.4 MySQL komprese

Stejně jako v HTTP je i v MySQL protokolu podpora komprese dat při přenosu. MySQL server umožňuje komprimovat zprávy delší než 50 bajtů. To je nekonfigurovatelná hodnota zapsaná ve zdrojovém kódu jako konstanta.



Obrázek 5.4. Vliv MySQL komprese na výkon webu

Graf na obrázku 5.4 zobrazuje měření se zapnutou a vypnoutou MySQL kompresí v konfiguraci Nginx a PHP-FPM s 32 procesy. Měření ukázalo, že MySQL komprese nemá na výkon webu žádný vliv nebo má dokonce nepatrně záporný.

5.5 Realpath cache

V PHP je kód webu často rozdělen do mnoha malých souborů. Při zpracování požadavků dochází k jejich načítání podle aktuální potřeby. Pokaždé když se v PHP načítá soubor, musí o něm PHP interpret z jádra systému získat informace. V případě, že

jsou v cestě souboru symbolické odkazy, musí jádro systému také všechny přeložit. Při získávání informací se volá systémové volání `stat`, které v případě chybějících dat v bufferu souborového systému může vést až ke čtení z disku. Protože čtení z disku nepatří mezi nejrychlejší operace a snažíme se mu pokud možno vyhnout, obsahuje PHP `realpath cache`, v níž si uchovává informace o používaných souborech.

Realpath cache se nastavuje pomocí přepínače `realpath_cache_size`, jenž udává velikost paměti použitou pro cache, a přepínače `realpath_cache_ttl`, udávajícího dobu, po kterou budou data v cache uchována. V PHP existuje funkce `realpath_cache_get`, která vrátí seznam všech souborů v cache, a funkce clearstatcache, jež vynutí smazání cache.

I přestože je většinou systémové volání `stat` odbaveno z bufferu souborového systému, u zatížených systémů, kde je navíc mnoho souborů, se občas stává, že `stat` začne být pomalý a dojde k přetížení systému. Toto chování jsem několikrát pozoroval při použití filesystému XFS s velkým množstvím souborů a také v situaci, kdy byly soubory připojeny přes síť pomocí NFS.

5.6 Výkon open basedir

Nastavení `open_basedir` je jedním z bezpečnostních mechanismů PHP interpreta. Tato vlastnost má za úkol omezit přístup z PHP k souborům, které jsou mimo vymezené adresáře. Pokusíme-li se tedy například funkcí `file_get_contents` přečíst soubor mimo povolené adresáře, PHP vrátí chybu a čtení se nezdaří.

Zjišťujeme-li příkazem `strace`, jak se volá funkce `include` při vypnutém nastavení `open_basedir`, vidíme na výpisu 5.5, že se nejdříve získají informace o souboru a následně se soubor otevře.

```
lstat("/var/www/incl.php", {st_mode=S_IFREG|0644, st_size=0, ...}) = 0
open("/var/www/incl.php", O_RDONLY) = 13
```

Obrázek 5.5. Výstup příkazu strace při volání PHP funkce include bez open_basedir

Navodíme-li stejnou situaci, ale s aktivním nastavením `open_basedir`, vidíme na výpisu 5.6, že PHP interpret musí vykonat mnohem více práce.

```
lstat("/var/www/incl.php", {st_mode=S_IFREG|0644, st_size=0, ...}) = 0
lstat("/var/www", {st_mode=S_IFDIR|0755, st_size=4096, ...}) = 0
lstat("/var", {st_mode=S_IFDIR|0755, st_size=4096, ...}) = 0
lstat("/var/www/incl.php", {st_mode=S_IFREG|0644, st_size=0, ...}) = 0
lstat("/var/www", {st_mode=S_IFDIR|0755, st_size=4096, ...}) = 0
lstat("/var", {st_mode=S_IFDIR|0755, st_size=4096, ...}) = 0
lstat("/var/www/incl.php", {st_mode=S_IFREG|0644, st_size=0, ...}) = 0
lstat("/var/www", {st_mode=S_IFDIR|0755, st_size=4096, ...}) = 0
lstat("/var", {st_mode=S_IFDIR|0755, st_size=4096, ...}) = 0
lstat("/var/www", {st_mode=S_IFDIR|0755, st_size=4096, ...}) = 0
lstat("/var", {st_mode=S_IFDIR|0755, st_size=4096, ...}) = 0
open("/var/www/incl.php", O_RDONLY) = 13
```

Obrázek 5.6. Výstup příkazu strace při volání PHP funkce include s open_basedir

Celý tento problém je podrobně rozebrán v nahlášené chybě pod číslem 52312 [7]. Realpath cache je vypnutá, když je nastavena hodnota `open_basedir`, PHP pak z bezpečnostních důvodů kontroluje všechny rodiče v adresářovém stromu až ke kořenu. Tato kontrola je volána vícekrát z několika míst v PHP, proto v našem případě pozorujeme tři opakování.

Toto chování se dá pozorovat i u nastavení `safe_mode`, které se od PHP verze 5.3 již nepoužívá.

5.7 Zámky na PHP sessions

Sessions jsou běžně používány pro ukládání informací o přihlášeném uživateli či o uživatelově právě rozpracované práci. Na velkém množství webů slouží sessions pouze k identifikaci uživatele a ve většině případů se z ní jen čte.

PHP má pro ukládání sessions unifikované rozhraní, čímž umožňuje implementaci mnoha způsobů uložení dat. Sessions mohou být uloženy na disk v podobě souborů, do MySQL databáze, Memcache či jakékoli jiné databáze, pro kterou je napsán ovladač (session handler). Samotný ovladač ani nemusí být napsán v jazyce C, může být napsán přímo v PHP samotném. K registraci vlastního sessions ovladače slouží funkce `session_set_save_handler`.

Mnoho dostupných ovladačů má však slabinu v podobě zámku, což může způsobit nemalé problémy. Nejčastěji se sessions ukládají do souborů na souborovém systému, přičemž jeden soubor reprezentuje jednu session. V tomto případě se PHP snaží udržovat session konzistentní, v jeden moment ji proto může používat pouze jeden proces. Jakmile se v PHP aplikaci zavolá funkce `session_start` dojde k zamknutí souboru se session a žádný jiný proces tuto session nemůže používat. Po dokončení požadavku je soubor se session opět odemknut. Bude-li webový server zpracovávat požadavek jednoho uživatele dlouhou dobu a stejný uživatel otevře další okno stejném webu, uvidí pouze bílou obrazovku, protože PHP proces bude čekat na zámek, jenž drží první proces. Tímto způsobem může uživatel otevírat stále nová okna, dokud nezaplní všechny procesy serveru, čímž způsobí jeho výpadek.

PHP podporuje možnost uzavřít session funkcí `session_write_close` a dále používat obsah session jen pro čtení. Uzavření session však musí být vyvoláno z PHP kódu. Jinou možností řešení problému se zámky je použití ovladače, který session nezamyká a který při každé změně detekuje a sloučí pouze změněné hodnoty. Tato metoda nicméně není univerzální a v některých situacích může způsobit problémy s konzistencí dat.

Ve webovém prostředí, kde jsou sessions nejčastěji užity k identifikaci uživatele, se však můžeme chytrým návrhem aplikace vyhnout tomu, abychom sessions vůbec použili. Jeden z návrhů je popsán ve zdroji [8] na straně 40. Principem je, že sessions PHP vůbec nepoužívá a místo toho se uloží do cookies identifikace uživatele (například jeho ID nebo uživatelské jméno) a přidá se k tomu podpis celého obsahu cookies. Je-li podpis vytvořen dobrou hashovací funkcí a přidáme-li do podepisovaného textu i skrytou část, nebude případný útočník schopen tento podpis podvrhnout. Přijde-li pak na server požadavek, kterému nesouhlasí podpis, může být takovýto požadavek odmítnut. Tato metoda bohužel zvyšuje náročnost na procesor, protože s každým požadavkem se musí ověřit validita podpisu. Úplně se díky ní však vyhneme použití sessions a nemusíme řešit problém, jaké uložiště pro ně použít a jak toto uložiště škálovat a sdílet mezi více servery.

5.8 Plánování zdrojů

Nejhorší situace, která pro administrátora může nastat, je, když aplikace spotřebuje všechny dostupné prostředky serveru a server přestane odpovídat. Většinou se na takovýto server nedá ani přihlásit a těžko se zjišťuje, co se stalo.

Díky tomu, že PHP zpracovává každý požadavek v separátním procesu, můžeme dosti přesně naplánovat zdroje serveru a podobným situacím se tak vyhnout. Každá zpracovávaná stránka potřebuje ke svému uspěšnému dokončení určité množství času procesoru a určité množství paměti.

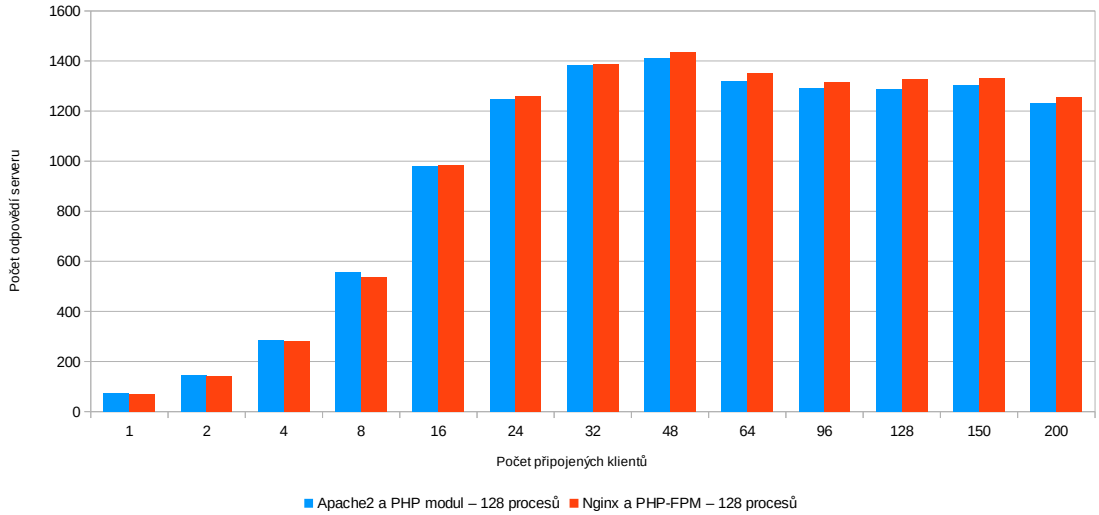
Množství paměti, kterou budeme potřebovat, můžeme vypočítat tak, že vynásobíme maximální počet PHP procesů hodnotou memory_limitu a přidáme k tomu nějakou paměť pro další aplikace běžící na serveru. Memory_limit je součást PHP určující maximální množství paměti, jež může být při zpracování požadavku alokována. Při překročení tohoto limitu je vyhozena fatální výjimka a celé zpracování je ukončeno.

Při plánování počtu jader nebo procesorů je vhodné udělat výkonnostní testy na provozované aplikaci a podle toho se rozhodnout. Obecně však platí, že PHP většinu času čeká na nějaké další zdroje dat, díky čemuž můžeme na serveru spustit více PHP procesů než má server dostupných jader. Je to však různé pro každé prostředí a aplikaci.

Zpracování každého požadavku v jednom procesu má i další výhodu. Pomocí nástrojů jako je strace se můžeme podívat, co daný proces dělá, nebo v systému vidíme bez hlubšího zkoumání, kolik spotřebovává paměti. Vždy máme jistotu, že bude zpracováván jen jeden požadavek v jeden moment.

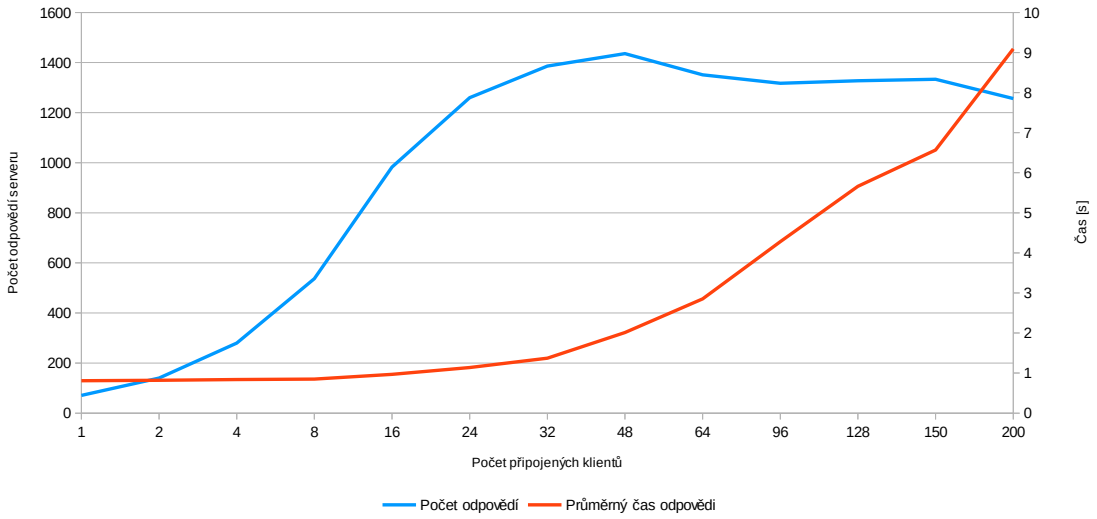
5.9 Výběr webového serveru

S plánováním zdrojů serveru úzce souvisí výběr webového serveru. Kombinace Nginx a PHP-FPM je často spojována s vysokým výkonem. Porovnal jsem tedy výkon této konfigurace s často používaným webovým serverem Apache2 s PHP jako modulem. Měření proběhlo pro několik kombinací počtu PHP procesů a počtu připojených klientů. Query cache na MySQL databázi byla vypnuta, stejně jako MySQL komprese.



Obrázek 5.7. Rozdíl výkonu Nginx a Apache2

Z obrázku 5.7 je vidět jen nepatrný rozdíl ve výkonu. Pro zpracování PHP požadavků tedy není důležité, který webový server bude použit. Při měření však nebyly zahrnuty požadavky na statický obsah. Hlavní výhoda konfigurace Nginx a PHP-FPM spočívá v tom, že zatímco PHP-FPM má počet procesů omezen na 128, Nginx může na požadavky na statický obsah dále odpovídat. To ale není možné u konfigurace s Apache2 a PHP modulem, protože všech 128 procesů Apache2 je blokováno vykonáváním PHP kódu.



Obrázek 5.8. Závilost počtu odpovědí serveru na době generování odpovědi

Dalším zajímavým jevem viditelným na obrázku 5.8 je závislost mezi počtem odpovědí serveru a průměrným časem, který server potřeboval k odpovědi. Na grafu lze pozorovat, že server dosáhl svého maximálnáho výkonu při 48 připojených klientech. V té době byla ale doba načítání již dvojnásobná v porovnání s dobou, kdy měl server minimální zátěž. Důležitou informací zde je, že procesor byl vytížen téměř na 100% již při 32 připojených klientech. Další zvyšování počtu připojených klientů vedlo k výraznému prodlužování čekací doby na odpovědi serverů. Počet odpovědí serverů však zůstal stále stejně vysoký, nebyl tedy zaznamenán žádný výrazný propad.

Kapitola 6
Optimalizace serveru MySQL

MySQL server je podle wikipedie [9] druhý nejrozšířenější relační databázový server. O jeho optimalizaci již bylo napsáno mnoho knih. Výborným zdrojem je [10], která podrobně rozebírá fungování MySQL. Z tohoto zdroje a také z [11] jsem vycházel při přípravě této kapitoly.

V rámci této kapitoly budou popsány dva nejčastější zdroje problémů s MySQL databází.

6.1 Storage engine

Databázový server MySQL disponuje unikátní vlastností, tj. že si lze vybrat u každé tabulky její typ (storage engine), a tím i ovlivnit způsob uložení dat na serveru. Mezi základní typy patří MyISAM, Innodb, Blackhole a Memory.

Memory tabulky existují v paměti serveru a při jeho restartu dojde ke ztrátě všech dat. Tento druh tabulek je vhodný pro data, která si můžeme dovolit kdykoli ztratit. Je ale třeba si uvědomit, že používání memory tabulek může způsobit problémy při MySQL replikaci. Používá-li se `row based` replikace a jeden ze serverů je restartován, může dojít k situaci, kdy mají oba servery nekonzistentní data a v případě manipulace s daty dojde k rozpadu replikace.

Blackhole tabulky na serveru nikdy nedrží žádná data. Jakékoli čtení vždy vrátí prázdný výsledek. Blackhole tabulky jsou používány při replikaci, kdy dotazy typu INSERT, UPDATE, DELETE jsou provedeny, ale dochází pouze k jejich zápisu do binlogu. Změny jsou následně replikovány na další servery. Použití blackhole tabulek je ale vzácné.

Nejčastějším typem tabulek je MyISAM. Dlouhou dobu byl tento druh tabulek v MySQL nastaven jako výchozí. V MyISAM je implementováno mnoho funkcí, jež v ostatních úložištích chybí. Příkladem mohou být prostorové indexy či fulltext indexy [1]). Hlavní nevýhodou MyISAM je způsob práce s daty. Při jakékoli změně dojde k zamknutí celé tabulky. Změny dat může v jeden moment provádět jen jeden SQL dotaz, ostatní musejí čekat, než je tento ukončen. To způsobuje, že i PHP procesy čekají na data z databáze, což vede ke zpomalení a pádu celého systému. Problém se zámky se týká i některých dotazů čtoucích data. V seznamu procesů je pak u ostatních procesů vidět `Waiting for table level lock`. MyISAM nepodporuje transakce a nesplňuje pravidla ACID. Dalším problémem jsou nutné opravy po pádu systému – MyISAM tabulky mohou zůstat v nekonzistentním stavu. Na vývoji MyISAM se v současné době již nepracuje.

InnoDB sice nemá všechny možnosti MyISAM, při práci s tabulkami však zamyká pouze ty řádky, jež zámek potřebují. Tím InnoDB umožňuje, aby v jeden moment upravovalo tabulku více procesů. InnoDB také splňuje všechna pravidla ACID. InnoDB je doporučený storage engine pro tabulky.

[1]) Od verze 5.6.4 jsou fulltext indexy podporovány i v InnoDB, avšak implementace není stejná.

6.2 Query cache

Cachování dotazů na úrovni databáze je velmi diskutovaným tématem. Na jedné straně může tato metoda výrazně zvýšit výkon databáze, na straně druhé však může negativně ovlivnit výkon celého systému a způsobit problémy. Je proto důležité vědět, jak tato vlastnost funguje, a podle toho ji v závislosti na prostředí použít.

Databázový server MySQL používá ke cachování dotazů pouze jednu globální cache, která je sdílena všemi relacemi. Do cache jsou uloženy všechny dotazy typu SELECT, přičemž je uložen vždy specifický SQL dotaz a k němu daný výsledek, který se při prvním zpracování také vrátí klientské aplikaci. Příjde-li na server identický požadavek, MySQL databáze na něj již odpoví daty z cache bez toho, aby jej musela znovu zpracovat. Dotaz však musí být naprosto totožný s původním dotazem, a to i včetně komentářů a mezer.

Na první pohled se zdá, že MySQL cache přináší spoustu výhod, jako například ušetření procesorového času a zrychlení aplikace. Má však i své stinné stránky. Obecně je známo, že v programech, kde se používá cachování, je složitou otázkou invalidace cache. Není tomu jinak ani u MySQL databáze. K invalidaci cache dochází pokaždé, když dojde ke změně dat v určité tabulce. MySQL nedrží žádné informace o tom, který záznam z tabulky je uchován v cache. Při změně dat v tabulce tedy invaliduje všechny výsledky v cache, jež s danou tabulkou souvisejí. V případě tabulek, do kterých se zapisuje často, dochází neustálým vyprazdňováním ke snížení efektivnosti cache. Při invalidaci cache jsou využívány zámky. Proto může aplikace na tyto zámky čekat, a to i desítky vteřin. V přehledu dotazů, které MySQL server v jeden moment zpracovává, pak vidíme `Waiting for query cache lock`.

Každé prostředí je jiné a někde query cache může pomoci, zatímco jinde může naopak velmi uškodit. Je proto potřeba si chování vyzkoušet na specifické aplikaci a pokud query cache nepřináší velmi výrazný zisk, je lepší ji vypnout.

Pro správu query cache v MySQL se používají dva základní konfigurační parametry. Query_cache_type ovládá chování samotného cachování, přičemž má tři možné hodnoty:

- 0 - Cache je úplně vypnutá a dojde k úplnému zpracování každého dotazu
- 1 - Dojde ke cachování všech dotazů, které nezačínají SELECT SQL_NO_CACHE.
- 2 - Databázový server cachuje jen dotazy, které to samy explicitně vyžadují, tzn. všechny, které začínají SELECT SQL_CACHE.

Druhým parametrem pro nastavení cache je velikost paměti, která bude pro cache vyhrazena. Velikost paměti se ovládá parametrem query_cache_size.

Kapitola 7
Optimalizace jádra

V poslední kapitole bude podrobněji rozebráno několik důležitých nastavení linuxového jádra. Nastavení sama o sobě nezvyšují výkon tak výrazně jako předchozí optimalizace, hrají ale důležitou roli v systémech s vysokým počtem spojení. Je proto důležité je sledovat a navýšit jejich hodnotu v případě potřeby. Výchozí nastavení ve většině linuxových distribucí není vhodné pro velmi zatížené systémy. Všechna zde uvedená nastavení se dají měnit několika způsoby, nejpoužívanějším z nich je soubor /etc/sysctl.conf.

V oficiální dokumentaci od vývojářů je toho popsáno málo, nicméně jako velmi detailní zdroj informací byl použit [12].

7.1 Počet otevřených souborů

Parametr `fs.file-max` omezuje celkový počet otevřených souborů v systému. V případě překročení limitu je do logu zapsána hláška VFS: `file-max limit 1231582 reached`. Aktuální počet otevřených souborů je možné získat ze souboru /proc/sys/fs/file-nr.

7.2 Lokální rozsah portů

Parametr `net.ipv4.ip_local_port_range` určuje rozsah portů, které server může používat pro odchozí spojení vytvořená na serveru. Příchozí spojení nejsou tímto ovlivněna, protože jsou vždy vedena na jeden konkrétní port. Většina aplikací ale vytváří další spojení, například na databázi, a nebude-li mu linuxové jádro moci přidělit číslo portu, skončí takové spojení chybou.

7.3 Počet opakovaného odeslání nepotvrzeného paketu

Parametry `net.ipv4.tcp_syn_retries` a `net.ipv4.tcp_synack_retries` ovlivňují, kolikrát bude znovu odeslán paket, na který od druhé strany nepřišlo potvrzení. Hlavně druhý parametr je důležitý na serverech, na něž je směrován veřejný provoz, obzvláště pak v momentech útoku. Server se při útoku typu SYN flood snaží několikrát odpovědět na příchozí SYN, na straně klienta ale nikdo neodpovídá. Výhodou TCP spojení je to, že když se klientské straně nepodaří navázat spojení, snaží se o to znovu. Proto můžeme na serveru snížit počet pokusů o znovuzaslání paketu s odpovědí na velmi nízkou hodnotu. Sníží se tím doba, než server uvolní alokované zdroje pro spojení, na která nikdo neodpovídá. První zmíněné nastavení pak ovlivňuje, kolikrát bude zaslán SYN, snaží-li se server vytvořit spojení.

7.4 Doba ukončení spojení

Parametr `net.ipv4.tcp_fin_timeout` je hodnota ve vteřinách a určuje, jak dlouho bude spojení drženo ve stavu TIME_WAIT před tím, než bude úplně ukončeno. Potřebujeme-li rychle uvolňovat porty a zdroje pro TCP spojení, je vhodné tuto hodnotu snížit.

7.5 Fronta příchozích spojení

V linuxovém jádře musí TCP spojení projít při vytváření několika frontami. Aby mohla přijímat spojení, musí každá serverová aplikace vytvořit socket, který bude naslouchat ve stavu LISTEN. V kódu této aplikace následuje nekonečná smyčka. Obsahuje funkci accept čekající na nové spojení a také kód, který spojení obslouží. V momentě, kdy aplikace obsluhuje spojení, nemůže příjmat další spojení. S tím jí pomůže jádro, které pro takováto spojení spravuje frontu. Velikost této fronty se určuje v momentě, kdy aplikace ve funkci listen žádá kernel o port, který chce používat.

V případě webových serverů se velikost fronty dá nastavit v konfiguraci a nemusí se upravovat zdrojové kódy. Tato hodnota má však i svůj limit nastavený v jádře. Tento limit se ovládá nastavením `net.core.somaxconn`. Výchozí hodnota je 128 a dá se zvýšit až na 65 535.

V předchozím případě jsme se podívali na již navázaná spojení, TCP spojení se však musí nejdříve navázat. Linuxové jádro obsahuje ještě druhou frontu, jež obsahuje spojení, kde SYN paket byl již přijat, ale ještě nepřišlo potvrzení na paket SYN+ACK. Nastavení velikosti této fronty se provádí pomocí `net.ipv4.tcp_max_syn_backlog`.

Jádro obsahuje ještě jednu frontu, a to frontu pro všechny příchozí pakety na síťovém rozhraní, jež postupně zpracovává. Hodnota `net.core.netdev_max_backlog` nastavuje maximální počet paketů, které mohou být vloženy do fronty v případě, že jsou pakety příjmány rychleji, než je jádro může zpracovat.

Kapitola 8
Vyhodnocení

Závěrem bych rád vyhodnotil a shrnul všechna měření a provedené optimalizace.

Porovnáním webového serveru Apache2 s PHP modulem a Nginx s PHP-FPM jsem zjistil, že výkon obou řešení při zpracování PHP požadavků je velmi podobný. Důvodem, proč je druhý webový server natolik ceněn, je fakt, že v této konfiguraci jsou odděleny PHP procesy od samotného webového serveru, který mezitím ještě může zpracovávat požadavky na statický obsah.

Ačkoli přináší vývoj PHP stále nové možnosti, zlepšení výkonu v rámci řady PHP 5 je zanedbatelné. Cena za hosting webu tudíž rozhodně není hlavním kritériem pro přechod na vyšší verzi, protože k výrazné změně nedojde. To ale neplatí pro právě připravovanou verzi PHP 7, jež přináší až dvojnásobný nárůst výkonu. Použijeme-li alternativní PHP interpret HHVM od Facebooku, jsme schopni z naší aplikace získat až třikrát vyšší výkon v porovnáním s použitím PHP 5. U HHVM však musíme počítat s možností nekompatibility některých funkcí. Nástup stabilní verze PHP 7 bude pro hostingové prostředí znamenat velký krok kupředu. Pevně věřím, že se v mnohých datacentrech sníží spotřeba energie, protože bude zapotřebí menšího množství serverů. Není-li přechod na PHP 7 nebo HHVM možný, lze výkon PHP výrazně zlepšit zapnutím opcode cache. Pokud bych já sám vyvíjel novou aplikaci v PHP, snažil bych se ji od začátku psát tak, aby byla kompatibilní s HHVM. Za zvážení by určitě stálo použití jazyka Hack, jenž z PHP vychází.

V původním návrhu měření jsem nepočítal s možností použití query cache na MySQL databázi. Výkony PHP 7 a HHVM však byly natolik nečekané, že query cache zapnuta být musela. Obecně je však efektivnost query cache v MySQL sporná. Přestože v našem prostředí výrazně pomohla, nemusí vést její zapnutí vždy k pozitivním výsledkům. V některých měřeních došlo také k úplnému zaplnění 1Gbps portu na databázovém serveru. Musel jsem tedy zapnout kompresi dat mezi webovým a databázovým serverem. Samotné zapnutí komprese nepřineslo žádné zlepšení výkonu, využití sítě však kleslo čtyřikrát.

Co se týče protokolu HTTP, učinil jsem dvě zásadní optimalizace. Nejdříve jsem zapnul HTTP kompresi, což vedlo ke snížení využití sítě na polovinu, bez dopadu na výkon procesoru, a to díky výrobcům procesorů, kteří kompresní funkce integrovali do procesorů. Dalšího snížení zatížení sítě jsem docílil zapnutím cachování statického obsahu v prohlížečích uživatelů. V momentě jeho zapnutí došlo ke skokovému snížení množství přenesených dat o další polovinu. Druhým pozitivním efektem bylo výrazné snížení počtu požadavků na server. Tyto změny jsem prováděl a jejich účinek testoval na reálné aplikaci s reálnou zátěží.

V rámci této práce jsem ověřil, že výkon webu může být několikrát zvýšen i bez zásahu do kódu aplikace, pouze díky správnému nastavení serveru.

Věřím, že práce splní i svůj druhý cíl a jejím přečtením získají čtenáři větší přehled o možných optimalizacích aplikací psaných v PHP, při nichž nebylo třeba zasáhnout do kódu. Uplatní-li čtenáři některé z mnou popsaných optimalizací v praxi, mohou významně snížit počet výpadků svých webových aplikací.

Rád bych práci v budoucnu dále rozšiřoval, což se týká například kapitol o optimalizaci MySQL serveru a linuxového jádra. Hodlám též pokračovat ve zkoumání možností nových protokolů vznikajících kolem HTTP, jež mají zrychlit přenos dat mezi serverem a uživatelem. Budu potěšen, pokud zde mí stávající i budoucí kolegové naleznou inspiraci pro vlastní práci. Přínosem též bude každá diskuse, již tato práce podnítí.

Literatura

[1] Libor Dostálek. *Velký průvodce protokoly TCP/IP a systémem DNS*. 2. aktual.vyd. vydání. Praha: Computer Press, 2000. ISBN 80-722-6323-4.

[2] David Gourley a Brian Totty. *HTTP*. 1st ed. vydání. Sebastopol, CA: O'Reilly, 2002. ISBN 15-659-2509-2.

[3] D. Robinson a K. Coar. The Common Gateway Interface (CGI) Version 1.1. *The Common Gateway Interface (CGI) Version 1.1*. 2004, (RFC3875),

[4] Armando Padilla a Tim Hawkins. *Pro PHP application performance*. New York: Distributed to the Book trade worldwide by Springer Science Business Media, c2010. ISBN 14-302-2898-9.

[5] *PHP 5.5.0 release announcement*.
`http://php.net/releases/5_5_0.php`.

[6] *Upgrading PHP extensions from PHP5 to NG*.
`https://wiki.php.net/phpng-upgrading`.

[7] *PHP :: Bug #52312 :: PHP safe_mode/open_basedir lstat performance problem*.
`https://bugs.php.net/bug.php?id=52312`.

[8] Stephen Corona. *Scale PHP Applications*. 2014.
`http://www.amazon.com/Scaling-PHP-Applications-Stephen-Corona-ebook/dp/B00BHN3SVW`.

[9] *MySQL*. 2001-2015.
`http://en.wikipedia.org/wiki/MySQL`.

[10] Baron Schwartz, Peter Zaitsev a Vadim Tkachenko. *High performance MySQL*. 3rd ed. vydání. Cambridge [Mass.: O'Reilly, c2012. ISBN 14-493-1428-7.

[11] *Mysql performance blog*. 2015.
`http://www.percona.com/blog/`.

[12] Christian Benvenuti. *Understanding Linux network internals*. Sebastopol: O■Reilly, c2006. ISBN 05-960-0255-6.

Příloha A
Seznam použitých zkratek

ACID	Zkratka pro vlastnosti: Atomicity, Consistency, Isolation, Durability
bytecode	Přenositelný kód, který je nezávislý na platformě. K jeho spuštění je potřeba běhové prostředí, které zajistí překlad do strojového kódu podle aktuální platformy.
CGI	Common Gateway Interface
Debian	Populární linuxová distribuce
gettext	Knihovna sloužící k překladům
HHVM	Alternativní PHP interpret vyvinutý společností Facebook.
HTTP	Hypertext Transfer Protocol
HTTP cookie	Jedna z hlaviček HTTP protokolu, která slouží k přenášení dat mezi požadavky
PHP	Víceúčelový skriptovací jazyk spouštěný na straně serveru. Je navržen k vývoji webů, ale používá se také jako obecný programovací jazyk.
PHP session	Mechanismus pro zachování dat mezi jednotlivými HTTP požadavky uživatele
Siege	Linuxový nástroj, který dokáže zasílat HTTP požadavky na server
TCP	Transmission Control Protocol
URL	Uniform resource locator