VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

DOSTUPNÁ ŘEŠENÍ PRO CLUSTROVÁNÍ SERVERŮ

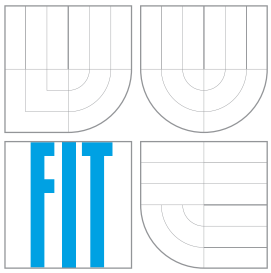
DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. VÁCLAV BÍLEK

BRNO 2008

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

DOSTUPNÁ ŘEŠENÍ PRO CLUSTROVÁNÍ SERVERŮ

AVAILABLE SOLUTIONS FOR SERVER CLUSTERING

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. VÁCLAV BÍLEK

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. TOMÁŠ KAŠPÁREK

BRNO 2008

Abstrakt

Cílem této diplomové práce je především důkladně zmapovat a zanalyzovat Open Source prostředky pro rozkládání zátěže a řešení vysoké dostupnosti, se zaměřením na oblasti úloh ve kterých jsou Open Source řešení typicky nasazována. Těmito oblastmi jsou především řešení síťové infrastruktury (routery, loadbalancery), obecně síťové a internetové služby a paralelní filesystémy. Další částí této práce je analýza návrhu, realizace a plánů dalšího rozvoje jednoho prudce se rozvíjejícího internetového projektu. Důsledkem takto dynamického rozvoje je nutnost řešení škálovatelnosti prakticky na všech vrstvách. Poslední část této práce pak představuje výkonnostní rozbor jednotlivých typů loadbalancingu v projektu Linux Virtual Server.

Klíčová slova

vysoká dostupnost, rozkládání zátěže, redundance, clusterování, škálovatelnost, IP failover, sdílené filesystémy, clustrové filesystémy, Linux Virtual Server, reverzní proxy, SQL replikace, SQL clusterování

Abstract

The goal of this master thesis is to analyse Open Source resources for loadbalancing and high availability, with aim on areas of its typical usage. These areas are particularly solutions of network infrastructure (routers, loadbalancers), generally network and internet services and parallel filesystems. Next part of this thesis is analysis of design, implementation and plans of subsequent advancement of an fast growing Internet project. The effect of this growth is necessity of solving scalability on all levels. The last part is performance analysis of individual loadbalancing methods in the Linux Virtual Server project.

Keywords

high availability, loadbalancing, redundancy, clustering, scalability, IP failover, shared filesystems, cluster filesystems, Linux Virtual Server, reverse proxy, SQL replication, SQL clustering

Citace

Václav Bílek: Dostupná řešení pro clustrování serverů, diplomová práce, Brno, FIT VUT v Brně, 2008

Dostupná řešení pro clustrování serverů

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Tomáše Kašpárka

.....
Václav Bílek
22. ledna 2008

Obsah

1 Úvod **3**

2 Vymezení pojmů **4**
2.1 Dostupnost . . . 4
2.2 Spolehlivost . . . 5
2.2.1 MTBF . . . 5
2.2.2 Failure rate . . . 6
2.3 Fault-tolerant system . . . 7
2.4 High Aviability . . . 8
2.5 Load Balancing . . . 8

3 High Aviability **9**
3.1 IP failover . . . 9
3.1.1 VRRP . . . 9
3.1.2 Dostupné implementace . . . 14
3.2 Stateful Firewall Failover . . . 16
3.2.1 pfsync . . . 17
3.2.2 Netfilter-HA . . . 17
3.2.3 Conntrack-tools . . . 18
3.3 Nadstavby nad IP failover . . . 19
3.3.1 Keepalived . . . 19
3.3.2 The High Availability Linux Project . . . 20
3.4 Cluster wide storage / Storage replication . . . 24
3.4.1 Sdílené médium / nízkoúrovňová replikace . . . 24
3.4.2 Sdílené Filesystémy . . . 26
3.4.3 Paralelní (distribuované) Filesystémy . . . 27

4 Load Balancing **31**
4.1 Linux Virtual Server . . . 31
4.1.1 LVS NAT . . . 33
4.1.2 LVS IP Tunneling . . . 33
4.1.3 LVS Direct Routing . . . 33
4.2 Load Balancing pomocí DNS . . . 34

5 HA a LB na aplikační vrstvě **36**
5.1 HTTP reverzní proxy . . . 36
5.1.1 Apache modproxy . . . 37
5.1.2 Squid . . . 37

5.1.3 Pound . . . 37
5.1.4 Varnish . . . 38
5.2 SQL replikace . . . 38
5.2.1 MySQL replikace . . . 38
5.2.2 PostgreSQL Slony-I . . . 39
5.3 SQL clustrování . . . 40
5.3.1 MySQL Cluster . . . 40
5.3.2 PGCluster . . . 42

6 Analýza návrhu a stávajícího stavu projektu živých sportovních výsledků 44
6.1 Výchozí podmínky, požadavky na návrh řešení . . . 44
6.2 Návrh řešení a jeho implementace . . . 45
6.3 Současný stav . . . 48

7 Možnosti dalšího rozvoje projektu 50
7.1 Linux Virtual Server . . . 50
7.2 MySQL replikace . . . 51
7.3 Akcelerace HTTP . . . 52
7.4 Nasazení distribuovaného filesystému pro sdílení dat . . . 53

8 Testování klíčových technologií pro rozvoj projektu 55
8.1 Linux Virtual Server . . . 55
8.2 Distribuované filesystémy - GlusterFS . . . 62

9 Závěr 65

Kapitola 1

Úvod

Snem každého administrátora je spravovat takový systém, u kterého se nemusí obávat výpadku či poruchy jednoho z jeho členů, který by měl za následek ať už částečnou nefunkčnost, nebo nefunkčnost systému jako celku. Dalším typickým „snem“ je neomezená škálovatelnost systému, kdy při nedostatku zdrojů datových, výpočetních či přenosových není problém rozšířit systém o další hardware a tím zdroje systému navýšit. Zároveň je většinou nutné zachovat transparentnost celého systému vůči uživatelům.

Jak jsem již naznačil takovéto požadavky na systém jsou spíše nedosažitelným ideálem než reálným cílem, nehledě na častou protichůdnost řešení těchto dílčích požadavků. Je však v lidské přirozenosti problémy překonávat, proto dnes pojmy jako *Load balancing, High availability*, nebo *Fault-tolerant system* jsou naplňovány v mnoha podobách a implementacích softwarových i hardwarových prostředků.

V dnešní době již není třeba dokazovat, že si Open Source Software vydobyl místo nejen tam, kde jsou hlavním kritériem nízké pořizovací náklady, ale i v rozsáhlejších systémech, kde již rozhodují o nasazené technologii především výše zmíněné požadavky na škálovatelnost, odolnost vůči výpadkům a transparentnost jejich použití vůči uživateli. Cílem této diplomové práce je především důkladně zmapovat a zanalyzovat Open Source prostředky pro rozkládání zátěže a řešení vysoké dostupnosti, se zaměřením na oblasti úloh ve kterých jsou Open Source řešení typicky nasazována. Těmito oblastmi jsou především řešení síťové infrastruktury (routery, loadbalancery), obecně síťové a internetové služby a paralelní filesystémy. Další částí této práce je analýza návrhu, realizace a plánů dalšího rozvoje jednoho prudce se rozvíjejícího internetového projektu. Důsledkem takto dynamického rozvoje je nutnost řešení škálovatelnosti prakticky na všech vrstvách. Poslední část této práce pak představuje výkonnostní rozbor jednotlivých typů loadbalancingu v projektu Linux Virtual Server a analýza možností nasazení distribuovaných filesystémů.

Kapitola 2

Vymezení pojmů

2.1 Dostupnost

Dostupnost (Availability) má více významů a způsobů jejího vyjádření v závislosti na oblasti použití. Pro naši potřebu se spokojíme s poloformálním vyjádřením[1]:

Dostupnost je definována jako procentuální vyjádření provozuschopnosti a komunikační operability systému ve chvíli, kdy je na něj vznesen požadavek.

Pro výpočet dostupnosti se v informatice a telekomunikacích často používají tyto obecné vztahy[2]:

$$A = \frac{Uptime}{Uptime + Downtime} = \frac{MTBF}{MTBF + MTTR} \tag{2.1}$$

Na první část vztahu 2.1 je třeba nahlížet správným způsobem, například uvažujme systém, který po určitou dobu[1] pracuje bez výpadku, jeho klienti jsou však připojeni přes síť která v tuto dobu zaznamená výpadek. Z pohledu administrátora daného systému se systém jeví jako 100% dostupný, z pohledu jeho klientů to již pravda není. Je tedy vždy třeba zvážit v jakém měřítku systém posuzujeme, tomuto výpočet dostupnosti přizpůsobit a zahrnout všechny relevantní zdroje výpadků.

Výrobci a provozovatelé systémů s oblibou používají pro popis dostupnosti svých řešení takzvaný devítkový systém. Častým, avšak většinou těžko dosažitelným cílem, je pětidevítková dostupnost, ta představuje dostupnost 99,999% po dobu jednoho roku, což odpovídá 5:15 [mm:ss] nedostupnosti za rok. V tabulce 2.1, jsou uvedeny užívané typy dostupnosti a jim odpovídající přípustné časy nedostupnosti.

[1] vzhledem ke které posuzujeme dostupnost tohoto systému

Dostupnost	Max. přípustná nedostupnost	
[%]	[hh:mm:ss/rok]	[hh:mm:ss/měsíc]
99	87:36:00	07:26:24
99,9	08:45:36	00:44:38
99,99	00:52:34	00:04:28
99,999	00:05:15	00:00:27

Tabulka 2.1: Devítkový systém, jeho časové vyčíslení.

2.2 Spolehlivost

Formální definice spolehlivosti (Reliability[2]), $R(t)$ je pravděpodobnost, že systém bude schopen nepřetržitě po dobu t, plnit funkci ke které byl navržen. Například systém se spolehlivostí 0,9999 pro jeden rok, má pravděpodobnost 99,99%, že bude fungovat po dobu jednoho roku bez poruch.

Spolehlivost je pouze jeden z mnoha faktorů ovlivňujících dostupnost, například 99,99% spolehlivost neznamená 99,99% dostupnost. Spolehlivost představuje schopnost systému fungovat bez přerušení, zatím co dostupnost představuje schopnost systému poskytovat svým klientům určitou službu. Tedy spolehlivost poskytuje metriku četnosti poruch určité komponenty, či systému jako celku, dostupnost vyjadřuje efekt nefunkčnosti systému způsobený poruchou.

2.2.1 MTBF

Jednou z běžně užívaných metrik spolehlivosti je *střední čas mezi poruchami (**MTBF**[2])*, což je průměrný časový interval[3] mezi dvěma po sobě jdoucími poruchami. Spolehlivost se zvyšuje, pokud se zároveň zvyšuje i MTBF.

Hardwarová MTBF – Představuje střední čas mezi poruchami hardwarové komponenty. Aby tato porucha nutně nepředstavovala i výpadek systému, používá se zpravidla hardwarová redundance a hardware s detekcí/korekcí chyb.[4]

Systémová MTBF – Představuje střední čas mezi poruchami systému. Poruchou systému rozumíme výpadek služby poskytované uživatelům, který lze odstranit pouze opravou systému. Redundance hardwarových komponent zvyšuje systémovou MTBF přesto, že MTBF jednotlivých komponent zůstává stejná. Protože porucha redundantní komponenty systému nutně neznamená poruchu systému, celková MTBF systému tak vzrůstá. Nicméně zvýšený počet komponent v systému znamená zvýšenou pravděpodobnost poruchy některé z těchto komponent. Při hardwarovém návrhu systému bychom se měli opírat o statistickou analýzu abychom byli schopni zjistit skutečný přínos spolehlivosti při použití redundance.

[2] Mean Time Between Failures
[3] většinou v hodinách
[4] například disková RAID pole; ECC paměti

MTBI [5] – Představuje dočasný systémový výpadek, u kterého není třeba oprava pro obnovení funkčnosti. S tímto typem výpadku se zpravidla setkáváme u systémů jejichž součástí je ASR(Automatic System Recovery), zpravidla jde o automatické zotavení systému po detekci vadného hardware.

MTTR [6] – Představuje střední čas do opravy. Je to jedna z nejčastěji užívaných metrik udržovatelnosti. Predikce udržovatelnosti pomocí MTTR, analyzuje potřebný čas k opravě a obnově funkčnosti systému po jeho poruše.

Zde bych se opět vrátil k vyjádření dostupnosti ve vztahu 2.1, konkrétně k jeho druhé části. Vystupují zde dva členy, MTBF a MTTR. Pokud chceme nějakým způsobem zvýšit dostupnost systému máme k dispozici právě 2 možnosti, zvýšit MTBF nebo(a zároveň) snížit MTTR. Jaké tedy máme možnosti pro první případ - zvyšování MTBF? Možností je více, zmiňme tedy ty hlavní:

- Volba kvalitních komponent s vysokou MTBF.
- Snižování počtu komponent systému, tedy i jeho komplexnosti.
- Redundance komponent.
- ...

V druhém případě, snižování MTTR se raději rovnou zamyslíme nad speciálním případem kdy $MTTR = 0$. Jde o situaci kdy jsme schopni jakoukoliv poruchu opravit za chodu systému bez vlivu na jeho funkčnost. Toto předpokládá dokonalou, nekonečně rychlou a spolehlivou detekci chyb v systému, zároveň jsou předpokladem mechanismy automatického (autonomního) zotavení z chyby, opět v nekonečně krátkém čase. Jak je jistě cítit z použitých výrazů o nekonečnu a dokonalosti, jde pouze o teoretický ideální stav, jemuž se snažíme přiblížit. Jakým způsobem a prostředky se o to snažíme bude předmětem značné části této diplomové práce, neboť se jedná o základní myšlenku *Vysoké Dostupnosti* a *Clusterování* jako prostředku k jejímu dosažení.

2.2.2 Failure rate

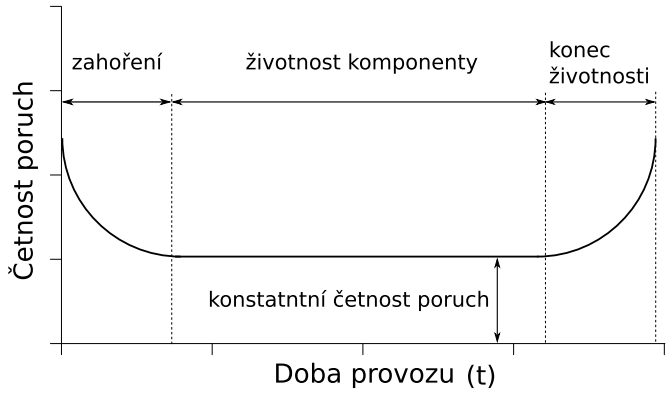
Četnost poruch je jednou z dalších metrik používaných k posuzování spolehlivosti. Je definována jako převrácená hodnota MTBF.

$$FailureRate = \frac{1}{MTBF} \tag{2.2}$$

Pro oba typy MTBF[7] existuje odpovídající *Failure Rate*. Pro většinu elektronických součástek platí, že jejich Failure Rate resp. MTBF po dobu jejich životnosti není konstantní, schéma vývoje této časové závislosti je pro většinu elektronických komponent shodný. Na obrázku 2.1 je znázorněn životní cyklus komponenty a odpovídající četnost poruch pro dané stáří komponenty. Na počátku životního cyklu komponenty je období, kdy je poruchovost zvýšená, příčinou jsou zpravidla skryté vady z výroby neodhalené výstupní kontrolou. Proto je běžnou praxí komponenty do důležitých systémů nechat odpovídající dobu

[5] Mean Time Between Interruptions
[6] Mean Time To Repair
[7] Hardwarová MTBF; Systémová MTBF

„zahořet“. Dále následuje období kdy je četnost poruch téměř konstantní, což odpovídá efektivní životnosti komponenty. Následuje období konce životnosti nebo opotřebení, kdy se opět četnost poruch zvyšuje.



Obrázek 2.1: Životní cyklus komponenty

2.3 Fault-tolerant system

Techniky odolnosti proti poruchám se obecně snaží zamezit tomu, aby chyby na nízké úrovni způsobily výpadek systému.*Systém odolný proti poruchám* je tedy takový systém, který při poruše kterékoliv své komponenty dokáže pokračovat ve své činnosti, ať už plnohodnotně, nebo v omezeném provozu. Základní požadavky na Fault-tolerant system lze specifikovat takto:

- Žádná komponenta v systému nesmí svou poruchou způsobit výpadek systému.
- Oprava jakékoliv komponenty musí být proveditelná za chodu.
- Systém musí být schopen detekovat vadnou komponentu a izolovat jí od zbytku systému.
- Systém musí být schopen zamezit propagaci chyb z vadné komponenty do systému.
- Musí byt dostupné mechanismy zotavení po chybě.

Přínosy systémů odolných proti poruchám jsou evidentní, existuje však mnoho důvodů, proč jejich nasazení může být problematické, případně zcela nevhodné, či nemožné. Zmiňme tedy ty hlavní:

- Zřejmě první věcí která nás napadne je *cena*. Cena je mnohem vyšší hned z několika důvodů, vyšší počet komponent (redundance), vyšší cena komponent (fault-tolerant komponenty) , kromě komponent které jsou jakýmsi HW backendem služby kterou systém poskytuje, je zde ještě mnoho speciálních komponent, které se starají právě o fault-tolerant chování systému (diagnostické, replikační, maskovací,... obvody). Kromě násobně vyšších pořizovacích nákladů se zde však objevují zvýšené náklady spojené s jeho provozem, větší energetická náročnost a s tím spojené větší nároky na chlazení. Fyzicky jde o větší zařízení, jsou zde tedy vyšší prostorové nároky. Díky mnohem vyšší složitosti systému jsou také vyšší nároky na jeho obsluhu.

- Stožitost testování - testovat komplexnější systém je vždy složitější, kromě toho existují situace, kdy nelze zaručit že redundantní komponenta nahrazující porouchanou bude fungovat po dostatečně dlouhou dobu potřebnou pro opravu, případně, že vůbec bude fungovat.
- Celkově složitost systému je vyšší, což sebou nese různá negativa:
  - vyšší náchylnost k chybám v návrhu
  - vyšší počet komponent vede k častějším poruchám
  - větší pravděpodobnost chyby obsluhy
  - složitost konfigurace a správy
  - ...
- Fault-tolerant systém svádí k laxnosti při řešení poruch.

2.4 High Aviability

Vysoká dostupnost je spíše marketingový než technický termín. Systémy označené jako *Vysoce Dostupné* zpravidla splňují tří a více "devítkovou" dostupnost (viz. tabulka 2.1). Oproti Fault-tolerant systémům, kde metrikou jejich kvality je kontinuita poskytované služby, HA systémy posuzujeme z hlediska dostupnosti, tedy nedostupnost u HA systémů nemusí znamenat vadu systému pokud jde o tak krátké, výpadky že celkový „Downtime"nepřesáhne hranici určenou třídou dostupnosti, do které daný HA systém patří.

2.5 Load Balancing

Load Balancing je technika rozkládání zátěže mezi více serverů za účelem dosažení vyššího výkonu, nižších latencí či snížení času nutného k výpočtu. „Loadbalancer"je zařízení, které toto rozkládání zátěže provádí, může se jednat jak o čistě hardwarové zařízení, tak o softwarovou implementaci běžící na běžném operačním systému. Loadbalancer zpravidla obsahuje několik algoritmů, podle kterých rozhoduje o přidělování příchozích dotazů. Nejběžnější jsou tyto:

- Round-robin – přiřazuje dotazy tak, jak přicházejí, postupně a dokola
- Nejrychlejší odpověď – dotaz je přiřazen serveru s momentálně nejnižšími latencemi
- Nejméně konexí – dotaz je přiřazen serveru s nejmenším počtem navázaných spojení
- a jejich váhové varianty, kdy algoritmus zohledňuje explicitně zadanou výkonnost backend serverů

Load balancing je možné provádět na mnoha vrstvách systému, například uvažme situaci kdy nadnárodní společnost pomocí DNS rozkládá požadavky na své webservery v datových centrech na jednotlivých kontinentech. Dále v každém datacentru je předřazena reverzní webcache, která kromě cachování ještě dělí požadavky na statická a dynamická data, požadavky na statická data posílá na k tomu určený webserver, požadavky na dynamická data posílá přes loadbalancer na webserverovou farmu, kde běží aplikační část webu společnosti. Aplikace na webserverové farmě při operacích nad databází dělí své dotazy na čtecí a zápisové, zápisové posílá na master SQL server, čtecí operace opět přes loadbalancer na několik slave replik ...

Kapitola 3

High Aviability

3.1 IP failover

Jednou z klíčových technik HA systémů je přebírání IP konfigurace, zpravidla se jedná o konfiguraci virtuálního rozhraní, která cestuje po uzlech clustreru v závislosti na momentálním stavu a dostupnosti jednotlivých serverů. Na protokolu VRRP[1] ukáži základní principy a možnosti a nastíním možné problémy. Dále uvedu používané otevřené implementace, ať už přímo tohoto protokolu, jeho odvozenin, či ideově odlišných řešení.

3.1.1 VRRP

V sítích kde je podstatná vysoká dostupnost, případně v sítích kde nemáme vliv na její kompletní architekturu (internet, komunitní sítě), je pravidlem redundance linek, routerů a switchů. Obecně se uplatňuje princip, že z bodu A do bodu B existuje několik navzájem nezávislých cest, které v případě výpadku jedné z nich nahradí její funkci. Toto je možné především díky protokolům dynamického routování, jako je RIP[2], OSPF[3] a BGP[4]. U koncových bodů sítě však tento přístup není vhodný z několika důvodů. Připojovat každou koncovou stanici duplicitním spojením je značně neekonomické, nehledě na nezanedbatelný nárůst kabeláže. Dalším důvodem může být neúměrně zvýšená složitost routovacího prostředí sítě a výpočetní/síťová režie, kterou dynamické routovací protokoly přinášejí.

VRRP je internetový protokol který je popsán v RFC3768, je založen na konceptu převzatém z proprietárního protokolu HSRP[5] firmy CISCO. Cílem tohoto protokolu je poskytnout nástroj pro zvýšení dostupnosti výchozí brány (default gateway) lokální sítě. VRRP zavádí koncept „virtuálního routeru“, což je vrstva abstrakce nad fyzickými routery na kterých VRRP běží. RFC zavádí několik pojmů, které stručně vysvětlím[4]:

VRRP router Router na kterém běží jedna či více instancí VRRP protokolu

Virtual router Abstraktní objekt představující výchozí bránu lokální sítě, sestává z VRID a VRRIP

Virtual router ID VRID numerický identifikátor (1-255) Virtuálního routeru, na daném segmentu sítě musí být unikátní

[1] Virtual Router Redundancy Protokol http://tools.ietf.org/html/rfc3768
[2] Routing Information Protocol http://tools.ietf.org/html/rfc2453
[3] Open Shortest Path First http://tools.ietf.org/html/rfc4750
[4] Border Gateway Protocol http://www.faqs.org/rfcs/rfc1771.html
[5] Hot Standby Router Protocol http://www.faqs.org/rfcs/rfc2281.html

Virtual router IP VRIP IP adresa Virtuálního routeru

Virtual MAC address VMAC Virtuální router nevyužívá MAC adresu rozhraní routeru, ale virtuální MAC adresu, která se odvozuje od VRID

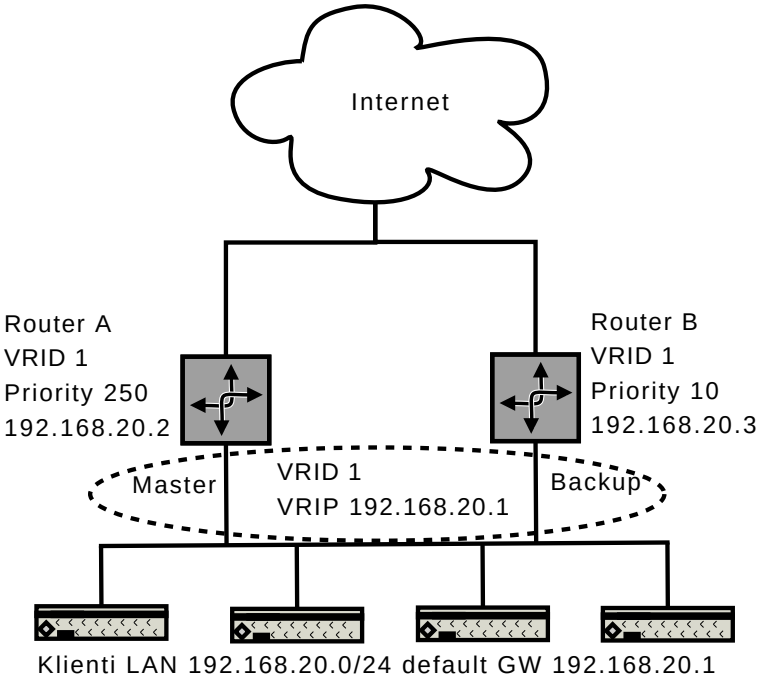
Master Instance VRRP která vykonává routování pro daný Virtuální router, v jednom okamžiku pro jedno VRID může existovat vždy jen jediný Master

Backup Instance VRRP která není ve stavu Master ale je připravena jeho funkci převzít, pro dané VRID jich může být libovolný počet

Priority Numerická hodnota (1-254) ohodnocující každou instanci VRRP, podle ní se určuje při výpadku Master instance její náhrada z řad Backu instancí. Čím vyšší číslo, tím vyšší priorita

Owner Pokud se shoduje na některém z routerů VRIP a adresa fyzického rozhraní, je instance VRRP na tomto stroji označena jako Owner s prioritou 255, pokud tento stroj běží je vždy ve stavu Master

Nyní na příkladu vysvětlím funkci protokolu. Na obrázku 3.1 je znázorněna nejjednodušší možná konfigurace dvou routerů využívajících VRRP a poskytujících službu výchozí brány pro klienty lokální sítě.



Obrázek 3.1: Jednoduchý příklad VRRP

Pokud je instance VRRP ve stavu Master, odesílá multicastové pakety na registrovanou adresu multicastové skupiny VRRP (224.0.0.18), čímž oznamuje ostatním instancím VRRP se shodným VRID, že je pro tuto skupinu Mastrem a byla mu přidělena určitá priorita.

Toto má 2 důvody:

1. Pokud je spuštěna instance s vyšší prioritou, může tato instance vyvolat volbu nového Mastera a převzít jeho funkci.
2. Instance ve stavu Backup naslouchá těmto zprávám a pokud po určitou dobu tuto zprávu neobdrží vyvolá volbu nového Mastera, protože je pravděpodobné, že stávající Master přestal plnit svou funkci.

Na obrázku 3.1 je router A ve stavu Master což je implikací jedné z následujících podmínek.

- Router A má vyšší prioritu než router B
- Router A je Owner (vlastní VRIP), což mu přiřazuje prioritu 255
- Pokud mají oba routery stejnou prioritu, router A má vyšší IP adresu.

V našem případě jsme nastavili prioritu routeru A na 250 a routeru B na 10. Pokud bychom tyto priority nenastavili, měly by oba routery prioritu 100 a router B by se stal Mastrem protože má vyšší IP adresu než router A.

Pád routeru Pokud Router A „spadne“, router B se to dozví tak, že mu přestanou chodit multicast pakety od routeru A. Normou je dán interval po který mají backup routery čekat na zprávu od Master routeru. Tento inerval každý backup router počítá následovně:

$$Master\_Down\_Interval = (3 * Advertisement\_Interval) \tag{3.1}$$

kde *Advertisement_Interval* je interval odesílání multicast paketů Mastrem a Skew_time je hodnota odvozená z priority routeru, zavádí se z důvodu omezení race condition přechodů mezi stavy. Stanovuje se takto:

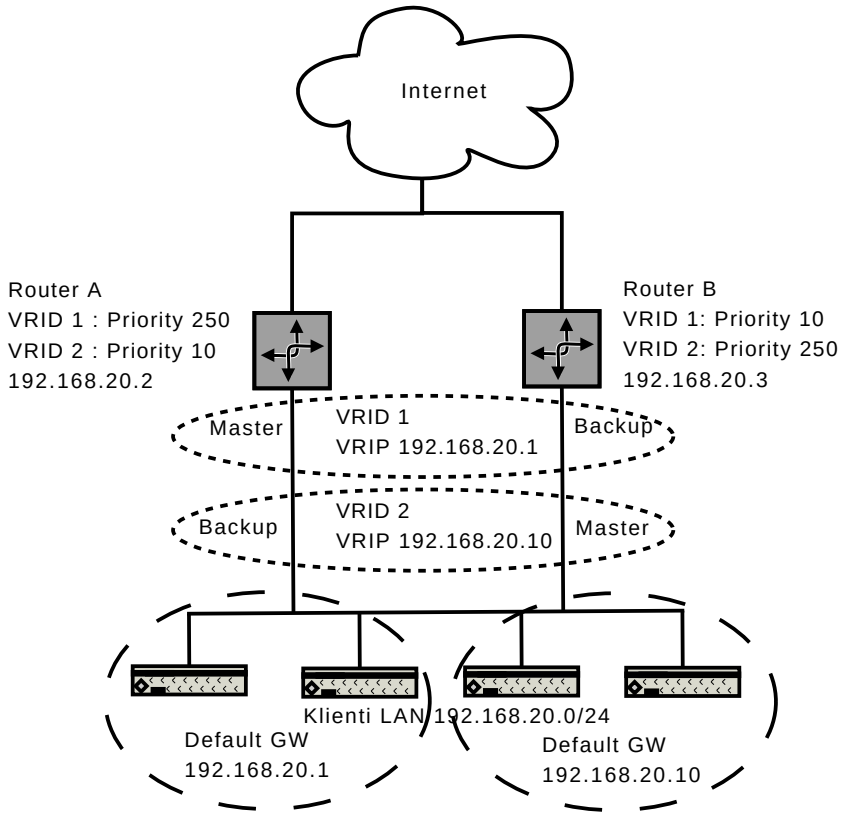
$$Skew\_time = \frac{256 - Priority}{256} \tag{3.2}$$

Restart routeru Po obnovení funkce routeru A mohou nastat 2 případy v závislosti na konfiguraci:

1. Pokud je router A zkonfigurován tak, aby po startu přešel do stavu Master, ihned inicializuje novou volbu rozesláním zprávy jako Master. Router B poté, co tuto zprávu obdrží, přechází do stavu Backup. V RFC je toto možné pouze v jediném případě, a to tehdy, je-li router A Owner (vlastní VRIP). Ve většině open source implementací se však toto nedodržuje a je možné explicitně stanovit zda router po startu přejde do stavu Master, či nikoliv, nezávisle na tom jestli je router vlastníkem VRIP.
2. Pokud je router A zkonfigurován tak, aby po startu přešel do stavu Backup, přejde do stavu Backup a čeká na první zprávu od momentálního Mastera (router B). Ve chvíli, kdy tuto zprávu obdrží, tak vzhledem k tomu, že má vyšší prioritu než router B, vyvolá novou volbu. Po této volbě přechází router A do stavu Master a router B do stavu Backup.

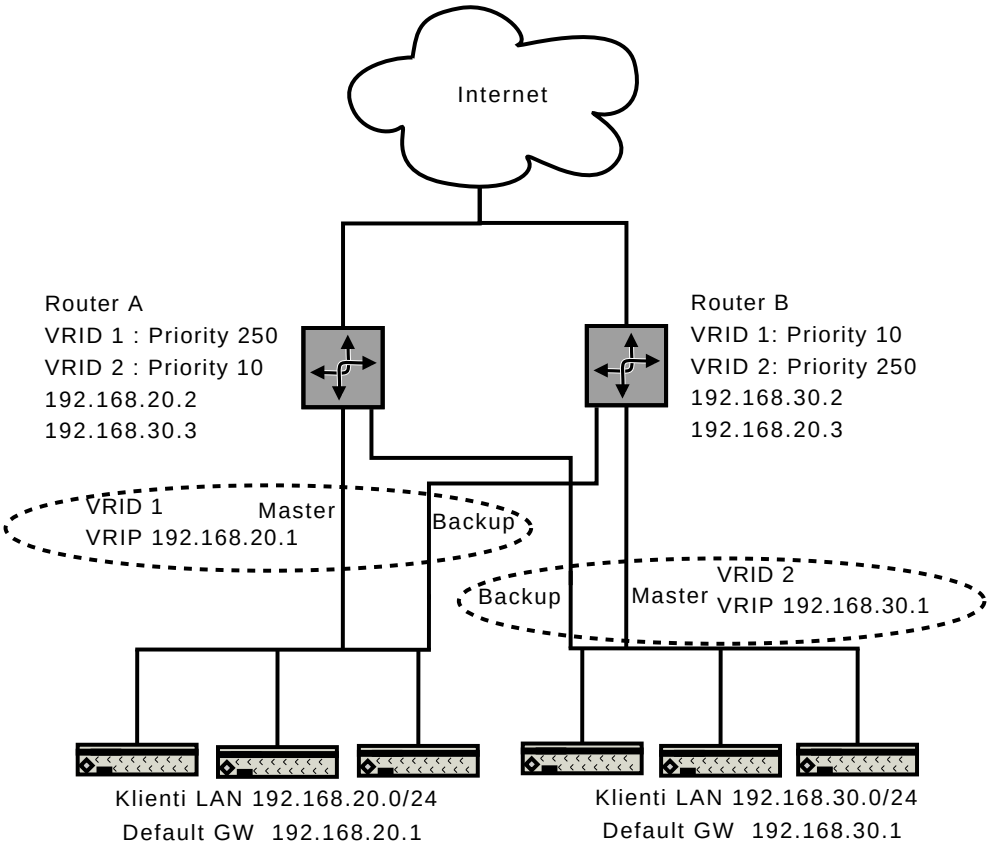
V obou případech oznamuje router B routeru A že přešel do stavu Backup tím že rozešle zprávu s prioritou nula (0). Poté co router A tuto zprávu obdrží, ví, že již může bezpečně přejít do stavu Master.

VRRP a Load Sharing Jedna z omezujících vlastností VRRP je, že v jednom okamžiku může provoz jednoho VRID routovat pouze jeden stroj. Tam kde není potřeba využít potenciál druhého stroje, který v tomto případě pouze nečinně čeká na výpadek stroje prvního, nemusí toto omezení znamenat problém. Jsou však situace, kdy takovéto plýtvání zdroji problémem je. Jeho řešením může být přidání dalšího VRID pro stejnou lokální síť viz. Obr.3.2 a rozdělení klientů sítě na dvě skupiny, kde každá skupina má vlastní výchozí bránu, odpovídající jednotlivým VRIP.



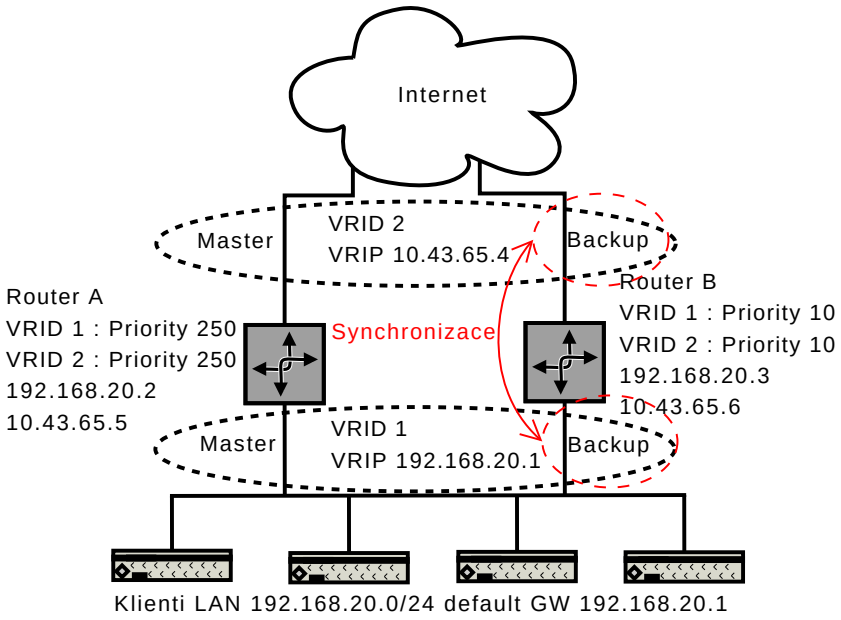
Obrázek 3.2: VRRP Load Sharing

Toto rozdělení stanic do dvou skupin je sice lehce proveditelné, například tak, že lichá stanice bude mít výchozí bránu nastavenou na VRIP1, a každá sudá stanice na VRIP2, obnáší však nežádoucí diferenciaci konfigurací. Tato rozdílnost konfigurací nám může vadit zpravidla pokud používáme klonování stanic případně nástroje pro centrální zprávu konfigurací. Je mnohem efektivnější, pokud tento model použijeme ve chvíli, kdy routery obsluhují několik lokálních sítí. Rozdělení na skupiny používající rozdílné výchozí brány je poté jednoduše řešeno příslušností jednotlivých stanic ke konkrétním podsítím viz (Obr. 3.3).



Obrázek 3.3: VRRP Load Sharing subnet grouping

VRRP na vnějším rozhraní routeru Ve všech předchozích případech jsme uvažovali IP failover pouze z pohledu vnitřní sítě. Dostupnost sítě z vnějšího světa byla zamýšlena pomocí dynamických routovacích protokolů, případně nebyla zamýšlena do těchto detailů. Dynamické routovací protokoly, jako řešení při nedostupnosti jednoho z routerů, je jen jednou možností, druhou možností je nasadit VRRP i na vnějším rozhraní routeru viz. (Obr. 3.4). Toto řešení přináší omezení v tom, že vnější rozhraní obou routerů musí být zapojeny do stejné podsítě. Tento model nasazení VRRP přímo nezapadá do konceptu zamýšleného v RFC, kde se problém nedostupnosti jednoho z routerů řeší pouze ze strany lokální sítě, kdežto při této konfiguraci se blížíme spíše nasazení obecného IP failover řešení, kde se nemusí jednat právě o routery ale obecně o jakoukoliv službu využívající IP protokol. Pokud VRRP nasadíme na obě rozhraní routerů vyvstává problém se synchronizací těchto dvou VRID tak aby nedocházelo ke křížové nedostupnosti vnější sítě viz (Obr. 3.4), kdy nefunkčnost jednoho rozhraní routeru musí způsobit přechod všech rozhraní na daném stroji do stavu Backup. Toto však RFC nikterak neřeší, a proto jednotlivé implementace se s tímto problémem vypořádávají různým způsobem, případně ho neřeší vůbec. Nutno zmínit, že tento problém se samozřejmě vyskytuje i tehdy, když na vnějších rozhraních běží některý z dynamických routovacích protokolů, i v tomto případě je z pochopitelných důvodů nutné nějakým způsobem řešit synchronizaci stavů vnějších a vnitřních rozhraní.



Obrázek 3.4: VRRP na vnějším rozhraní, seskupování rozhraní

3.1.2 Dostupné implementace

Zde bych se rád pozastavil nad dostupnými implementacemi, ať už přímo protokolu VRRP, jeho odvozenin, i nad implementacemi IP failover, které s VRRP nemají nic společného. Především bych zmínil hlavní rozdíly, výhody, nevýhody a stav vývoje.

Vrrpd Pod tímto názvem je možné nalézt více projektů. Většina z nich staví na implementaci Jeroma Etienne[6], její vývoj se však zastavil v roce 2000. Zde se zmíním o projektu, který v rozšíření této implementace došel nejdále, jedná se o projekt hostovaný na sourceforge.net[7], jeho vývoj se po dvou letech také zastavil. Tato implementace byla vyvíjena pouze pro linux, a funguje pouze na ethernetu, oproti RFC je přidáno řešení problému křížové nedostupnosti viz. (Obr. 3.4), toto není řešeno přímo synchronizací stavů vnějšího a vnitřního rozhraní, ale úpravou priority rozhraní při detekci problému na rozhraních zahrnutých do monitoringu danou instancí Vrrpd. Protože na linuxu není možné přiřadit jednomu síťovému rozhraní více MAC adres, není možné při dodržení RFC na jednom rozhraní provozovat více instancí VRRP. Ve Vrrpd lze toto obejít vypnutím podpory pro virtuální MAC adresu, tím je poté možné využít konfigurace pro rozložení zátěže viz (Obr. 3.2).

Tento projekt, ač se již dále nevyvíjí, poskytuje poměrně dobře zvládnutou funkčnost VRRP protokolu. Dalším jeho pozitivem může být fakt, že jde o software běžně dostupný přes balíčkovací systémy většiny distribucí. Otázkou ale je, jak dlouho se zde udrží vzhledem k tomu, že existuje aktivně vyvíjená alternativní implementace VRRP (Keepalived).

[6] http://www.off.net/~jme/vrrpd/index.html
[7] http://sourceforge.net/projects/vrrpd/

Carp Common Address Redundancy Protocol[8], jde o protokol, který byl vytvořen vývojáři OpenBSD jako reakce na licenční nesrovnalosti okolo protokolu VRRP. Funkčně je CARP zcela ekvivalentní protokolu VRRP, implementačně je však postaven na vlastní idei. Oproti implementacím VRRP již obsahuje podporu IPv6, zároveň je oproti VRRP navrhnut s důrazem na vysokou bezpečnost, dále je možné CARP využít pro loadbalancing. CARP řeší problém synchronizace vnitřních a vnějších rozhraní pouze v případě, že na obou rozhraních běží CARP preempt módu. Pak pokud jedno z rozhraní vypadne, dochází na druhém rozhraní k úpravě priority, a tím ke konzistentnímu přechodu stavů Master → Backup na obou rozhraních. Pokud však na některém z rozhraní CARP neběží, a zároveň je toto rozhraní podstatné pro správnou funkčnost routeru, je třeba použít nějaký externí mechanizmus, jako například IFSTATED[9], který se o tuto synchronizaci postará.

CARP existuje jako port pro všechny BSD klony. Pokud řešíme problém redundance routeru na těchto platformách, je CARP jasnou volbou.

Ucarp Userspace CARP [10], Port CARP protokolu do userspace démona, hlavní výhodou je podpora mnoha platforem jako Linux, OpenBSD, NetBSD, MacOSX. Tato multiplatformnost je docílena především oddělením té části kódu, která se stará o přenastavení IP konfigurace při přechodech mezi stavy, do shell skriptů, které démon volá ve chvíli, kdy k těmto přechodům dochází. Tyto skripty pak řeší nekompatibilitu konfigurace IP pro jednotlivé platormy, zároveň je zde prostor pro řešení synchronizace vnějších a vnitřních rozhraní, které samotný démon neřeší.

Ucarp je stále aktivně vyvíjen a je součástí většiny linuxových distribucí. Jeho hlavní výhodou je možnost použití na mnoha Unixových platformách.

Keepalived Keepalived[11] je projekt poskytující robustní failover backend pro LVS[12] load balancing. Tento projekt bude dále popsán v sekci „Nadstavby nad IP failover“, zde bych pouze zmínil jednu jeho část, kterou je právě implementace protokolu VRRP kterou obsahuje.

VRRP implementace v Keepalived původně vychází z projektu Jeroma Etienne.[13] Alexandre Cassen přidáním několika vlastností vytvořil stejnojmenný software[14], tento však sám označil za experimentální, užitečný pouze pro ověření funkčnosti návrhu. Následně se tento kód stal základem implementace VRRP v Keepalived, a dále je aktivně vyvíjen v rámci tohoto projektu.

Aby byla schopná implementace VRRP v Keepalived provozovat více instancí na jednom rozhraní, neimplementuje VMAC, protože linuxový kernel neumožňuje aby jedno síťové rozhraní mělo více MAC adres. Pro synchronizaci vnitřních a vnějších rozhraní (Obr.3.4) Keepalived poskytuje funkci tzv. „sync grouping“. Vzhledem k tomu, že Keepalived není pouze implementací VRRP, ale komplexním failover řešením, je možné zapojit do rozhodovacího procesu o přesunech jednotlivých VRID vlastní kontrolní skripty, které v závislosti na svém návratovém kódu ovlivňují prioritu virtuálního routeru. Lze si například představit, že každý z routerů bude připojen do internetu přes jiného providera, my budeme skrip-

[8] http://www.openbsd.org/cgi-bin/man.cgi?query=carp&sektion=4
[9] http://www.openbsd.org/cgi-bin/man.cgi?query=ifstated&sektion=8
[10] http://www.ucarp.org/
[11] http://www.keepalived.org/
[12] Linux Virtual Server http://www.linux-vs.org/
[13] http://www.off.net/~jme/vrrpd/index.html
[14] http://www.linuxvirtualserver.org/~acassen/

tem monitorovat například dostupnost stroje v peeringu těchto providerů. Pokud dojde k výpadku v síti providera přes kterého je připojen Master, skript toto zaznamená, Keepalived sníží prioritu tohoto stroje a dojde k přesunu služby na druhý router. Pokud by námi testovaný stroj v peeringu nebyl dostupný ani přes jednoho z providerů, dojde ke snížení priority u obou routerů, a k přesunu služby nedojde.

Tímto příkladem jsem chtěl naznačit limity protokolů jako je VRRP nebo CARP, kde se rozhodování o přesunu služby zakládá pouze na vzájemné dostupnosti strojů zapojených v rámci těchto protokolů.

Heartbeat Heartbeat je základní součástí projektu Linux-HA.[15] Jednou z jeho funkcí je i detekce „mrtvých uzlů“ a migrace služeb (IP adresa je chápána jako služba), což společně představuje IP failover řešení. Linux-HA je jedním z nejpoužívanějších projektů v souvislosti s HA na Linuxu. Dále bude popsán v sekci „Nadstavby nad IP failover“.

Failover služeb, tedy i IP failover využívá komunikaci mezi jednotlivými uzly clusteru, kdy na základě této komunikace je rozhodováno o stavu jednotlivých uzlů a o migraci služeb v rámci clusteru. Základní 3 druhy zpráv, kterými instance Heartbeatu komunikují mezi sebou jsou:

Status message Někdy nazývaný jako „heartbeat“, je paket vysílaný broadcastem, multicastem nebo unicastem, jeho funkcí je pravidelně potvrzovat ostatním strojům v clusteru svou funkčnost.

Cluster transition message Tato zpráva je vyslána vždy, když má dojít k přesunu služby mezi stroji. Žadatel (ten který službu přejímá) nejdříve posílá žádost o uvolnění služby, ve chvíli, kdy je služba uvolněna je žadateli zaslána odpověď, a ten přesun služby dokončuje.

Retransmission request Heartbeat každý Status massage opatřuje sekvenčním číslem, pokud některý z uzlů clusteru zaznamená nesrovnalost v těchto sekvenčních číslech přicházejících paketů, zažádá tímto typem zprávy o opětovné zaslání.

3.2 Stateful Firewall Failover

Jednou z oblastí, kde Open Source operační systémy jako Linux a xxxBSD mají nepopiratelný úspěch, je použití v síťové infrastruktuře jako routery a firewally. U routerů a stateless firewallů je situace řešení vysoké dostupnosti poměrně jednoduchá, vzhledem k tomu že u těchto typů zařízení nedochází k řízení provozu na základě stavů spojení které skrz ně prochází, stačí nějakým způsobem zabezpečit synchronizaci momentálního nastavení routeru/firewallu, případně filtrovacích pravidel. Pro failover routeru, či firewallu, pak stačí použít některý z výše uváděných prostředků jako VRRP, CARP nebo Heartbeat.

U stateful firewallů je situace poněkud složitější. Stavový firewall na jedné straně přináší značný přínos v možnosti filtrace a řízení provozu na základě stavů spojení, která přes něj procházejí. Je tak například jednoduše možné povolit příchozí provoz do lokální sítě pouze pro spojení, která byla iniciována zevnitř (z lokální sítě do sítě vnější). Na straně druhé se zde objevuje několik problémů. Connection tracking znamená zvýšenou výpočetní zátěž pro daný firewall, kromě hlaviček paketů může být nutné analyzovat datovou část

[15] The High Availability Linux Project http://www.linux-ha.org/Heartbeat

až po aplikační vrstvu, zároveň zde vyvstává nutnost udržování tabulky jednotlivých spojení v operační paměti, což jednak zvyšuje paměťové nároky, v druhé řadě takováto data znemožňují jednoduchý failover tak jak je to možné u stateless firewallů. Takovýto failover je sice možný, ale přicházíme tím právě o data stavů spojení, důsledkem je zpravidla ztráta v té chvíli aktivních spojení.

Řešením je synchronizace těchto stavových dat mezi aktivním a stand-by firewallem. Jako první mně známý Open Source operační systém, ve kterém se funkce takovéto synchronizace objevila, je OpenBSD, kde ve verzi 3.3 spatřil světlo světa *pfsync.*[16] Přibližně o rok později se objevil ve FreeBSD 5.3, kam byl z OpenBSD portován. Na Linuxu je stav o poznání horší. Prvním pokusem byl projekt *Netfilter-HA*[17], ten však nebyl dokončen a v dnešní době se již nevyvíjí. Další projekt který se danou problematiku snaží řešit je *Conntrack-tools,*[18]jde o poměrně mladý projekt, jeho vývoj za poslední rok však vypadá velmi slibně a naplňuje mne pocitem, že i pro linuxovou platformu bude k dispozici velmi silný nástroj pro connection tracking synchronizaci.

3.2.1 pfsync

Pfsync je protokol používaný Packet Filterem (pf[19]) pro řízení a synchronizaci stavových tabulek spojení mezi firewally. Standardně jsou změny stavů v těchto tabulkách rozesílány pomocí multicastu přes synchronizační rozhraní. Konkrétně je využit IP protokol 240 a muticastová skupina 224.0.0.240. Je možné tímto způsobem spojovat stavové tabulky i několika strojů. Ve chvíli kdy je nastaveno synchronizační rozhraní, začíná daný stroj vysílat změny ve vlastní tabulce stavů, a zároveň přijímá multicasty od jiných strojů a začleňuje je do vlastní tabulky. Vzhledem k tomu, že samotný protokol neobsahuje žádnou podporu autentizace nebo šifrování, je doporučováno použití vyhrazené linky pro tuto synchronizaci. Což lze řešit například křiženým UTP kabelem, další možností je použití IPSec.

3.2.2 Netfilter-HA

Klíčovou částí projektu je jaderný modul *ct_sync*, který implementuje synchronizaci stavů spojení v rámci netfilteru. Byl vyvíjen na jádře 2.4.26, ke kterému kromě samotného modulu ct_sync je třeba přidat několik patchů které upravují řadu věcí v rámci netífilteru, a další části do něj přidávají. Samotná replikace se skládá ze dvou kroků, nejprve je třeba označit spojení která si přejeme synchronizovat, data o takto označených spojeních poté ct_sync odesílá přes rozhraní, které mu nastavíme při zavedení modulu. Nastavení může vypadat například takto:

```
# insmod ct_sync  cmarkbit=30 syncdev=eth1 state=slave id=1
# iptables -t mangle -A PREROUTING -m state --state NEW \
> -j CONNMARK --set-mark 0x40000000/0x40000000
```

Přepínání mezi stavy master/slave se po zavedení modulu ct_sync provádí pomocí rozhraní v /proc. Příklad přepnutí do stavu Master:

```
# echo 1 > /proc/sys/net/ipv4/netfilter/ct_sync/state
```

[16]http://www.openbsd.org/cgi-bin/man.cgi?query=pfsync
[17]http://svn.netfilter.org/cgi-bin/viewcvs.cgi/trunk/netfilter-ha/
[18]http://people.netfilter.org/pablo/conntrack-tools/index.html
[19]http://www.openbsd.org/faq/pf/

Jak již bylo řečeno nebyl tento projekt nikdy dotažen do stavu, kdy by jej bylo možné použít v produkčním prostředí a dnes se již dále nevyvíjí.

3.2.3 Conntrack-tools

Narozdíl od Netfilter-HA jsou *Conntrack-tools* user space rozhraním pro tabulku connection trackingu udržovanou v jádře OS. Conntrack-tools se skládají ze dvou částí:

conntrackd Démon zabezpečující komunikaci mezi jednotlivými stroji a synchronizaci tabulek spojení.

conntrack Je CLI (comand line interface) umožůjící přidávání, mazání a modifikaci jednotlivých záznamů v tabulce spojení. Zároveň umožňuje vypsání této tabulky, a to jako plaintext nebo XML.

Pro komunikaci s jádrem OS používají *netlink*,[20] konkrétně netlink_netfilter, conntrackd zároveň využívá služeb *Connection tracking events* pro odchytávání událostí v tabulce spojení. Funkce démona conntrackd je shodná pro všechny stroje, pro aktivní i pro ty ve stavu stand-by. Démon implementuje dvě cache tabulky, interní a externí. Interní cache je plněna daty o spojeních, které démon získává odchytáváním událostí zasílaných jádrem OS přes connection tracking events. Jedná se tak vlastně o lokální kopii conntrack tabulky jádra s tím rozdílem, že cache tabulka démona neobsahuje data o spojeních, která v konfiguraci označíme za nepotřebná. Tato interní cache je dále distribuována pomocí multicastu ostatním strojům. Externí cache tabulka slouží jako úložiště dat přijatých v rámci multicast komunikace, jde tedy o vzdálenou kopii conntrack tabulky aktivního stroje. Pro multicastovou synchronizaci jsou k dispozici dva způsoby:

Persistent Každá změna v interní cache se ihned propaguje ostatním strojům, v pravidelných intervalech se tabulka přenáší celá. Pravidelným přenosem celé tabulky se řeší resynchronizace nově začleněného stroje, zároveň je zde vyšší garance toho, že si tabulky na jednotlivých strojích navzájem odpovídají.

NACK V tomto módu stroj při začlenění požádá o resynchronizaci, dále se přenáší pouze změny. Může se tak stát, že po určité době si tabulky vzájemně zcela neodpovídají, výhodou však je mnohem nižší servisní provoz.

Ve chvíli, kdy dojde k výpadku aktivního stroje, a důsledkem toho k failoveru na stand-by stroj, je třeba provést zavedení dat z externí cache tabulky démona do conntrack tabulky jádra OS. O toto se zpravidla stará pomocí skriptu software, který řídí a provádí failover (Keepalived, Heartbeat...).

Vzhledem k tomu, že mezi funkcí démona na aktivním a stand-by stroji není žádný rozdíl a zároveň se jejich funkce nikterak neovlivňují, je možné takto docílit konfigurace Activ/Activ (viz. Obr. 3.3).

Ač je tento projekt stále ve stádiu vývoje, jeho funkcionalita již obsahuje všechny podstatné vlastnosti. Doufám tedy, že je jen otázkou času, kdy se toto řešení stane ekvivalentem pfsync z OpenBSD.

[20] http://www.tin.org/bin/man.cgi?section=7&topic=netlink

3.3 Nadstavby nad IP failover

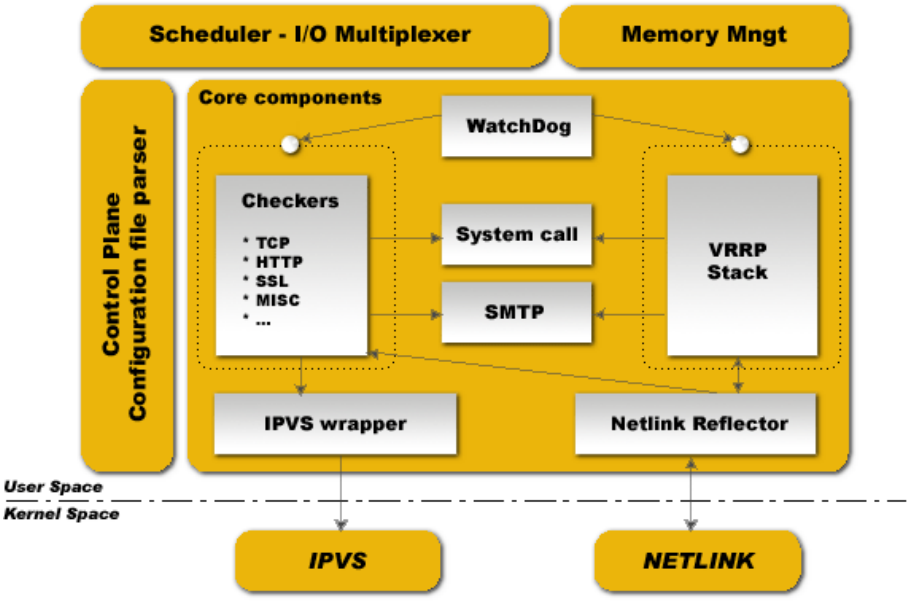
Nadstavbou je zde myšlen nějaký systém nebo sada nástrojů, umožňující správu, monitoring a migraci služeb v rámci clusteru. Zmíním zde řešení primárně zaměřená na vysokou dostupnost při rozkládání zátěže pomocí LVS[21] (Keepalived), kromě těchto dvou projektů se zde zaměřím na projekt Linux-HA[22].

3.3.1 Keepalived

Hlavním cílem toho projektu je poskytnutí robustního zázemí pro loadbalancery založené na Linux Virtual Server. Jeho funkci lze rozdělit do dvou skupin:

- Kontrola a failover loadbalanceru (VRRP).
- Kontrola funkce backend serverů a jejich dynamické začleňování/odstraňování ze skupiny aktivních serverů na základě jejich stavu.

Keepalived je kompletně napsán v C, při jeho vývoji byl kladen důraz na modulárnost jednotlivých funkčních částí. Démon při spuštění rozděluje svou činnost do tří procesů. Minimalistický hlavní proces slouží pro spuštění dvou synovských procesů, jejich následný monitoring a případný restart. Jeden z těchto procesů je instancí VRRP a stará se o failover loadbalanceru. Druhý proces implementuje kontrolu stavu služeb backend serverů a s tím spojené akce změn konfigurace LVS. Na obrázku 3.5 je znázorněn design implementace démona Keepalived, dále uvedu popis jeho hlavních částí.



Obrázek 3.5: Keepalived software design; Zdroj http://www.keepalived.org

Scheduler - I/O Multiplexer Keepalived nepoužívá POSIXová vlákna, místo toho implementuje vlastní abstrakční vrstvu vláken optimalizovanou pro síťový provoz.

[21] Linux Virtual Server http://www.linuxvirtualserver.org/
[22] The High Availability Linux Project http://www.linux-ha.org

Core components Představuje skupinu knihoven vytvořených za účelem maximální možné modulárnosti a zároveň minimalizace duplicit v kódu. Tyto knihovny poskytují například funkce: parsování html, časovač, formátování řetězců, síťové operace, management procesů, nízkoúrovňové operace nad TCP,...

WatchDog Hlavní proces monitoruje funkci svých synovských procesů pomocí zasílání zpráv přes Unix domain socket. Pokud se mu od monitorovaného procesu nedostane odpověď, použije sysV signál kterým prověří, zda je daný proces živý, a zrestartuje ho.

Checkers Jedna z hlavních činností démona. Odpovídá za kontroly funkčnosti služeb backend serverů. Výsledek testu může mít za následek odebrání backend serveru, resp. jeho zařazení zpět do skupiny aktivních serverů.

VRRP Stack Druhou hlavní činností je kontrola funkčnosti loadbalanceru a jeho případný failover, pokud dojde k výpadku. Tato část kódu běží jako samostatný proces monitorovaný svým rodičem a může být spuštěn samostatně bez aktivace součástí pro podporu LVS.

SMTP Pokud dojde k výpadkům, ať už na úrovni VRRP, nebo služeb backend serverů, je možné pomocí vestavěného SMTP klienta odeslat zprávu správcům systému.

IPVS wrapper Pokud některá z kontrol služeb backend serverů skončila s výsledkem, který vyžaduje úpravu pravidel IPVS (konfigurační pravidla jaderného kódu LVS), jsou tyto změny prováděny právě pomocí IPVS wraperu, který pomocí knihovny libipvs do jádra požadované změny propaguje.

Keepalived představuje velmi zdařilý projekt, s jistou nadsázkou je možné prohlásit, že realizace LVS loadbalanceru ve failover konfiguraci se při použití keepalived redukuje na editaci jednoho konfiguračního souboru. Stejné funkčnosti by bylo možné dosáhnout například pomocí projektu Linux-HA, jednalo by se však o řádově složitější operaci.

3.3.2 The High Availability Linux Project

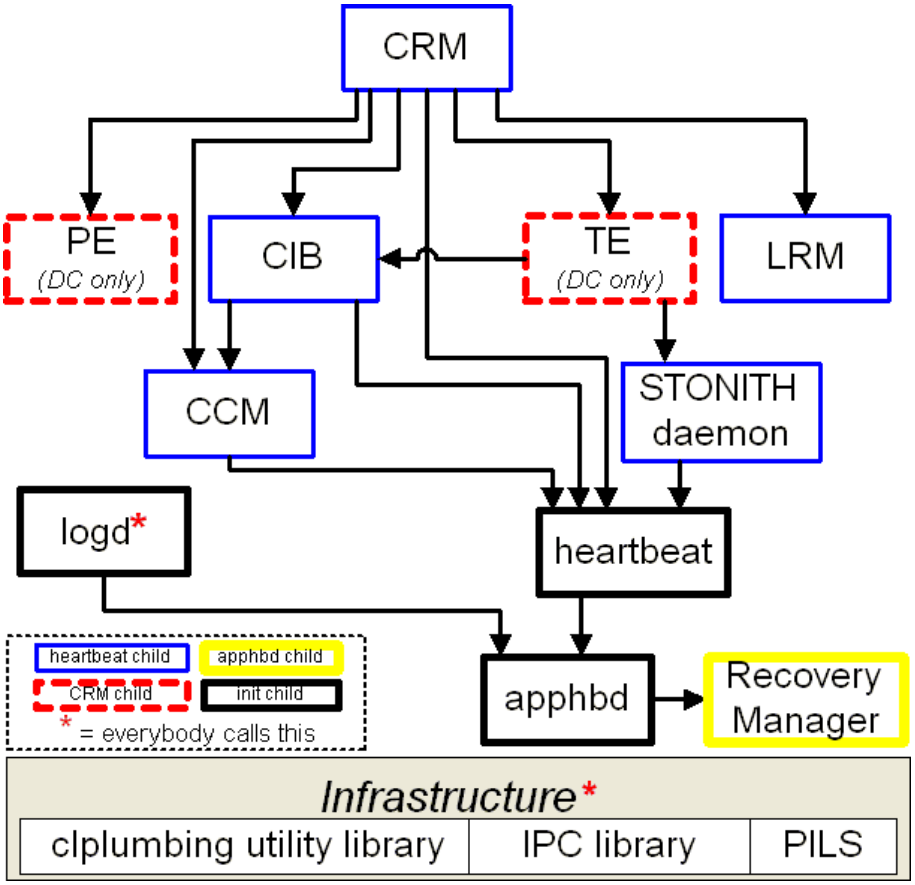
Linux-HA je jedním z nejpoužívanějších Open Source nástrojů pro řešení vysoké dostupnosti. Jeho vývoj vede Alan Robertson[23], jako zaměstnanec IBM Linux Technology Center. Primární vývojovou platformou je Linux, jde však o vysoce přenositelný software, a proto je ho možné provozovat i na FreeBSD, OpenBSD a Solarisu. Mezi hlavní vlastnosti projektu paří:

- Není omezen počet uzlů v clusteru, projekt je vhodný pro clustery o mnoha uzlech i pro nejjednodušší clustery o dvou uzlech.
- Monitorování služeb; Při výpadku může být služba automaticky restartována, nebo odmigrována na jiný uzel.
- Fencing; Technika znemožnění přístupu vadného uzlu ke zdrojům vyžadujícím exkluzivní vlastnictví.
- Propracovaný management zdrojů, jejich závislostí a omezení.

[23] http://www.linux-ha.org/AlanRobertson

- Časová pravidla umožňují různé politiky v závislosti na čase.
- Pro většinu běžně používaných služeb (Apache, DB2, Oracle, PostgreSQL, ...) jsou již předpřipravené management skripty.
- Grafické rozhraní pro monitoring a správu clusteru.

Nejznámější komponentou projektu je *Heartbeat*, běžně se zaměňuje název projektu s názvem této komponenty, pokud se tedy někde hovoří o Heartbeatu, je třeba z kontextu vytušit zda je tím myšlen celý projekt, nebo pouze jedna jeho komponenta. V kontextu tohoto projektu jsou služby a vlastně cokoliv, co je spravováno v rámci clusteru (IP adresy, filesystémy, komunikační linky, služby), označovány jako zdroje, které je možné dle definovaných pravidel a omezení přesouvat po uzlech clusteru. Na obrázku 3.6 je blokové schéma komponent projektu, které se pokusím stručně popsat a nastínit jejich funkci a interakci.



Obrázek 3.6: Architektura Linux-HA; Zdroj: http://www.linux-ha.org

heartbeat komunikační modul, zajišťuje služby intra-cluster komunikace, přenos konfiguračních dotazů, informace a testování dostupnosti uzlů clusteru, službu skupinové náležitosti

CRM Cluster Resource Manager je mozkem celého systému, spravuje zdroje, rozhoduje o jejich umístění v rámci clusteru, plánuje jakým způsobem přejít z momentálního stavu do stavu požadovaného. Všechny ostatní komponenty s CRM spolupracují.

PE CRM Policy Engine vytváří graf přechodů z aktuálního stavu do vyžadovaného stavu určeného konfigurací, zohledňuje aktuální stav, umístění jednotlivých zdrojů a stav uzlů clusteru.

TE CRM Transition Engine provádí graf přechodů vytvořený PE, komunikací s LRM jednotlivých uzlů se snaží provést všechny akce z tabulky předepsaných operací nutných pro přechod do nového stavu.

CIB Cluster Information Base je automaticky replikované úložiště informací dostupných CRM jako jsou stavy uzlů a zdrojů, lokace funkčních zdrojů, konfigurace, závislosti a omezení.

CCM Consensus Cluster Membership poskytuje službu vyhodnocující náležitost uzlů do skupiny vzájemě propojených uzlů (kde každý uzel slyší zprávy od všech ostatních uzlů ve skupině).

LRM Local Resource Manager přebírá pokyny od CRM a provádí spouštění, zastavování a monitoring zdrojů na uzlu kde je spuštěn.

Stonith Daemon je démon implementující službu restartu uzlů clusteru.

logd non-blocking logging daemon může logovat přes syslog nebo do souboru.

apphbd Application Heartbeat Daemon, implementuje časový watchdog pro aplikace schopné poskytovat pravidelné informace o svém stavu.

Recovery Manager navazuje na apphbd, kde ve chvíli, kdy aplikace přestane komunikovat nebo se nečekaně ukončí, provádí akce k nápravě tohoto stavu (zabití, restart aplikace).

Vývojový cyklus clusteru se může vyvíjet například následovně:

1. Veškerá komunikace probíhá přes modul heartbeatu, ten kromě toho zajišťuje sledování výpadků v komunikaci s ostatními uzly.
2. Změny v dostupnosti ostatních uzlů postupuje heartbeat dále do CCM, ten na základě takovéhoto podnětu inicializuje proceduru stanovení náležitosti uzlů do clusteru.
3. Ve chvíli kdy je tato procedura uspěšně dokončena předá CCM informaci o aktuálním členství uzlů v clusteru CRM a CIB.
4. Ve chvíli kdy se CIB aktualizuje, jsou na toto upozorněny všechny komponenty, které ji využívají.
5. Poté, co CRM zjistí změnu v CIB, inicializuje PE.
6. PE na základě nových dat v CIB se rozhodne, zda je třeba provést nějaké změny, pokud ano, vytvoří graf přechodu do nového stavu a předá ho CRM.
7. CRM tento graf předá TE.
8. TE přes CRM kontaktuje jednotlivé LRM s akcemi z grafu přechodu.
9. Vždy, když je jednotlivá akce ukončena, nebo vyprší její časový limit, je to oznámeno TE.

10. TE pokračuje v plnění akcí z grafu přechodů až do jeho dokončení.

11. Po úspěšném dokončení všech akcí TE toto hlásí CRM.

Na první pohled by se mohlo zdát, že řízení HA systému je celkem jednoduchá záležitost, kde stačí při výpadku službu zrestartovat, případně ji někam přemigrovat. Na druhý pohled se objevují problémy, jejichž řešení určuje kvalitu daného HA systému. Jedním z těchto problémů je tzv. *Split-Brain* a možné metody jeho řešení jsou *Fencing* a *Quorum*.

Split-Brain je problém vyplývající z nemožnosti rozpoznání vadného komunikačního kanálu od výpadku stroje se kterým přes tento kanál komunikujeme. Vzniká tak situace, kdy nejsme schopni s jistotou tvrdit, zda je protější strana naživu nebo ne. Takováto situace je u clusteru velmi nepříjemná, protože migrace služeb se zpravidla zakládá na informaci o tom, že uzel na kterém se služba nacházela je nedostupný. Pokud pomineme neškodnou variantu kdy je protější strana skutečně mimo provoz, vzniká stav, ve kterém se dva stroje snaží provozovat totožnou službu nezávisle na sobě (Split-Brain). Pokud jde o službu, která vyžaduje exkluzivní přístup k některému zdroji clusteru, je vysoce pravděpodobné, že se začnou dít špatné věci. Jako příklad je možné si představit cluster o dvou uzlech, které mezi sebou sdílejí přes Fibre Channel diskové pole. Pokud přestane fungovat komunikační kanál mezi těmito dvěma uzly, inicializuje do té doby neaktivní uzel převzetí služby. Ve chvíli, kdy tento uzel připojí, v té době používaný svazek, a začne na něj zapisovat, dojde na tomto svazku k destrukci dat. Předejít něčemu takovému je možné pokud si dokážeme s jistotou odpovědět na otázku: „Je druhá strana skutečně mrtvá?“ resp. „Nebude již druhá strana zapisovat na sdílený svazek?“ Metoda kterou lze na tyto otázky odpovědět je *Fencing*.

Fencing představuje myšlenku vybudování ohrádky (fencing) okolo vadného uzlu tak, aby nemohl dále přistupovat ke zdrojům, ke kterým potřebujeme exkluzivní přístup. Řešením našeho problému s hledáním odpovědi tedy není hledání skutečné a správné odpovědi, ale přizpůsobení reality našim potřebám.

Existují vzásadě dva typy fencingu:

Resource fencing pokud víme jaké zdroje vadný uzel využíval, které z nich potřebujeme od tohoto uzlu „odstřihnout“ a máme prostředky, jak toto zajistit, jedná se o fencing zdroje. Příkladem může být, když jsme schopni nastavit zakázání přístupu na portu řadiče sdíleného pole.

Node fencing násilnější metodou je fencing celého uzlu, kdy znemožníme přístup k jakémukoliv zdroji. Běžnou metodou je hardwarový reset stroje, případně jeho vypnutí, ač se jedná o řešení ne zcela elegantní, rozhodně je efektivní. Tato technika se nazývá STONITH[24].

Fencing může být spolehlivou technikou pouze pokud se nespoléháme na spolupráci s vadným uzlem, ale jsme schopni požadovaných opatření docílit nezávisle na něm.

Nyní si již dokážeme odpovědět na otázku jestli je druhá strana naživu, vyvstává však další problém. Obě strany si myslí, že druhá strana je mimo provoz a je třeba ji blokovat (fencing), tímto způsobem bychom se nejspíše dostali do nekonečné smyčky restartů, proto je třeba nějakým způsobem bez komunikace mezi takto oddělenými uzly clusteru vybrat jeden který bude dále pokračovat.

[24] Shoot The Other Node In The Head

Quorum je mechanizmus rozhodnutí o přisouzení privilegií pro další provoz služeb na jednom z uzlů clusteru. Nejběžnější metodou je prostá majorita hlasů. Tento způsob poměrně dobře funguje u clusterů s vyšším počtem uzlů, v našem případě, kdy jsou uzly jen dva, je tato metoda nepoužitelná. V našem případě je možné použít quorum disk, což představuje externí SCSI zařízení, které umožňuje přístup pouze na jednom portu v jednom okamžiku. Další alternativou je Quorum server. Tyto prostředky však nesmí sloužit jako primární rozhodovací mechanizmus, ale pouze jako rozhodčí, pokud mají obě strany stejný poměr hlasů, jinak se tyto prostředky stávají novým Single Point Of Failure.

Projekt Linux-HA lze bez váhání označit za vyspělý nástroj pro vysoce dostupné systémy. Zde nastíněné problémy řeší bezezbytku, další jeho předností je vlastní testovací systém, kterým lze prověřit možné problémy při návrhu a realizaci systémů. V neposlední řadě se jedná o aktivně vyvíjený projekt se silným zázemím a kvalitním vedením.

3.4 Cluster wide storage / Storage replication

Téměř každá aplikace dnes pracuje s daty, pokud chceme u aplikací docílit vysoké dostupnosti, musíme nejprve zajistit vysokou dostupnost dat nad kterými tyto aplikace pracují. Řešení je široká škála v závislosti na požadavcích aplikace, technickém řešení clusteru na kterém aplikace běží nebo požadavcích na konzistenci dat. Pokud nebudeme uvažovat případy, kdy jsme schopni konzistenci v rozumné míře zaručit čistě administrativně případně pomocí běžných nástrojů (rsync, scp, atp.), rozdělují se používané techniky na dvě skupiny. První z nich je cílená především do oblasti vysoké dostupnosti, její princip spočívá ve sdílení datového úložiště na úrovni blokového zařízení. Druhou technikou jsou paralelní filesystémy, které samy o sobě představují homogenní vysoce dostupné systémy a kromě sdílení dat mezi uzly clusteru se snaží řešit především výkonostní a datovou škálovatelnost.

3.4.1 Sdílené médium / nízkoúrovňová replikace

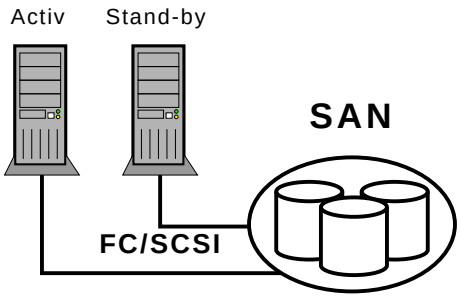
Pro sdílení blokového zařízení (externí Raid pole, SAN, DAS) jsou dnes používány především technologie a protokoly jako Fibre Channel, SCSI, iSCSI. Pro operační systém se takovéto zařízení jeví zcela transparentně jako lokální SCSI disk. Podpora dvou zároveň připojených systémů je řešena v režii daného zařízení, nikterak ale neřeší kolize IO operací těchto dvou systémů připojených k jednomu LUN[25]. Jsou tedy dvě možnosti jakým způsobem je možné provozovat dva systémy připojené k jednomu LUN tak, aby si navzájem nerozbíjely filesystém na sdíleném zařízení. První možností je cluster v konfiguraci Activ/Stand-by (viz. Obr.3.7), kde filesystém na daném zařízení je vždy připojen pouze na jednom z uzlů.

Pokud zařízení podporuje SCSI reservation je možné tuto vlastnost využít pro *Fencing* Stand-by uzlu. Oproti metodám jako je *STONITH* jde o mnohem elegantnější řešení situací jako je Split-Brain.

Druhou možností je připojení filesystému na obou uzlech zároveň, předpokladem ovšem je použití takového filesystému, který něco takového umožňuje. Více o těchto filesystémech v sekci *Sdílené Filesystémy*.

Není nutné zdůrazňovat, že tyto technologie jsou poměrně drahé a pokud mají splňovat požadavky na vysokou dostupnost (online vzdálená replikace, multipath atp.), zpravidla se cena násobí. Je samozřejmé, že tyto technologie mají v řadě situací nezastupitelné místo,

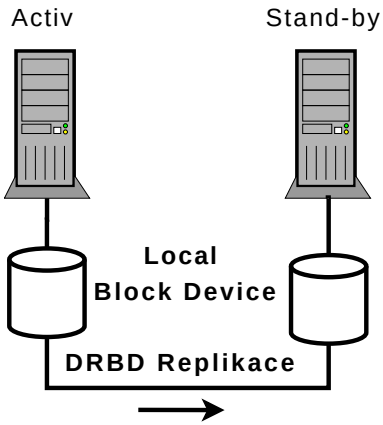
[25] Logical Unit Number



Obrázek 3.7: Sdílené blokové zařízení přes Fibre Channel/SCSI/iSCSI

jsou však situace, kdy jde o „kanón na vrabce„a v těchto případech je dobré seznámit se s alternativami. Jednou z nich je DRBD[26].

DRBD je zjednodušeně síťový RAID1. Implementací je virtuální blokové zařízení, které všechny zápisy kromě lokálního provedení propaguje přez IP síť na druhý stroj (viz Obr.3.8).



Obrázek 3.8: Replikace DRBD

DRBD zařízení může být na každém uzlu ve stavu Primární nebo Sekundární[27], k zápisu musí docházet na primárním zařízení, to provede zápis na podřízeném fyzickém blokovém zařízeni (/dev/sdX atp.), žádost o zápis odešle na druhý uzel, kde se zápis také provede.

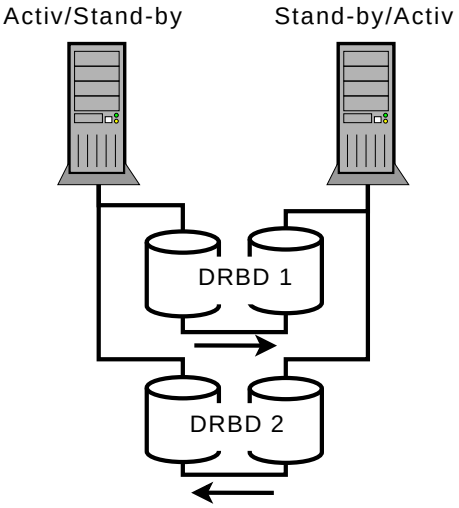
Pokud dojde k výpadku primárního uzlu, může dojít k failoveru na sekundární uzel. Ten má aktuální data, přepne se do primárního režimu a aplikace může pokračovat tam, kde skončila. Poté, co se původní primární uzel zotaví, musí dojít k synchronizaci dat, synchronizována jsou pouze data, která se změnila, tato synchronizace probíhá na pozadí.

Asi nejčastěji provozovanou konfigurací je Activ/Stand-by (Obr.3.8), kdy všechny aktivní služby běží pouze na jednom uzlu a druhý uzel nečinně čeká na výpadek prvního uzlu.

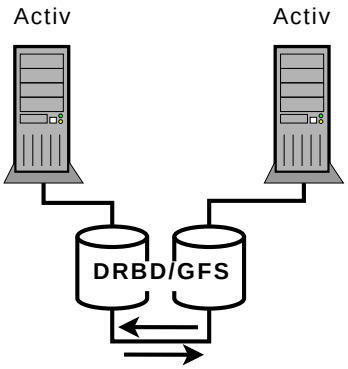
Další možností je dvojitá Activ/Stand-by viz. konfigurace Obr.3.9, kdy je třeba zkonfigurovat minimálně dvě DRBD zařízení, přičemž každý uzel bude mít minimálně jedno

[26] http://www.drbd.org
[27] od verze 8.0 je podporována konfigurace Primarní-Primární

zařízení v primárním stavu. Lze tak na každém uzlu provozovat nějakou skupinu služeb, která je datově nezávislá na skupině služeb běžících na druhém uzlu. Například na jednom serveru poběží web a mail server a na druhém DNS a LDAP server, každá z těchto skupin bude mít vlastní DRBD zařízení. Tato konfigurace skrývá riziko v možnosti přetížení při failoveru služeb. Aby při failoveru k přetížení nedocházelo, musí se zatížení serverů stabilně pohybovat pod 50% nebo musí existovat náhradní řešení představující například vypnutí méně důležitých služeb.



Obrázek 3.9: DRBD duální Activ/Stand-by



Obrázek 3.10: DRBD Activ/Activ

Další možností jak je možné DRBD použít, je Activ/Activ (Obr.3.10). Taková konfigurace vyžaduje nasazení sdíleného filesystému, zároveň zde také platí riziko přetížení při failoveru služeb. V dnešní době neexistuje mnoho aplikací které by dokázaly fungovat ve více instancích nad stejným datovým prostorem. Otevírá se zde ale prostor pro speciální použití, které by jinak fungovat nemohlo. Například je tímto způsobem možné provozovat live migraci XENovských domén, nebo vytvořit multipath iSCSI target.

Projekt DRBD představuje určitě zajímavou alternativu sdílených datových zařízení, těžko jim může konkurovat například v rychlosti nebo latencích IO operací, ale tam kde tyto parametry nejsou na prvním místě dokáže nabídnout kvalitní řešení za nejlepší možnou cenu (zadarmo). Momentálně projekt obsahuje modul do linuxového jádra a userspace nástroje pro správu. Jako jeden z hlavních cílů projektu je protlačení do vanila jádra, na čemž vývojáři intenzivně pracují.

Podobné funkcionality jako má DRBD lze docílit i kombinací NBD[28] a MD[29], jde však o složitější operaci, kdy je třeba ve vlastní režii řešit například resynchronizaci. Kladem tohoto řešení oproti DRBD je například to, že vše potřebné se již nachází ve vanila jádře.

3.4.2 Sdílené Filesystémy

Pokud potřebujeme sdílet mezi uzly clusteru blokové zařízení, je třeba aby na něm byl filesystém který se vyrovná se současným přístupem jednotlivých uzlů. V dnešní době jsou v

[28] Network Block Device
[29] MultiDevice (RAID,LVM)

Linuxu používány především GFS[30], který získal Redhat od společnosti Sistina Software a uvolnil ho pod GPL licencí, a OCFS[31] který vyvíjí Oracle také pod GPL licencí. Oba filesystémy jsou již delší dobu ve vanila jádře. Základním rozdílem oproti běžným filesystémům je distribuovaný Lock manager, který umožňuje jednotlivým uzlům koordinovat IO operace nad sdíleným blokovým zařízením.

3.4.3 Paralelní (distribuované) Filesystémy

Distribuovaný filesystém lze definovat jako síťový filesystém který se rozkládá přes několik uzlů/úložišť, kde každý z nich obsahuje pouze část z celého filesystému. Hlavním důvodem pro použití distribuovaných filesystémů je jejich škálovatelnost, kterou překonávají omezení klasických storage řešení. Zaměřují se na škálovatelnost kapacitní, výkonnostní, škálovatelnost možného počtu připojených klientů atp.. Dále se zde zaměřím na architekturu a vlastnosti dvou, dle mého názoru nejzajímavějších Open Source projektů. Prvním bude *Lustre*[32], druhým *GlusterFS*[33]. Podoba jejich názvů, a problém, který řeší, je víceméně to jediné, co tyto projekty spojuje. Každý z nich poskytuje odlišný pohled na věc, řeší vše po svém a nabízejí odlišné vlastnosti.

Lustre Tento projekt byl donedávna vyvíjen pod křídly společnosti Cluster File Systems Inc.. V říjnu letošního roku dokončil Sun Microsystems akvizici této společnosti a spolu s tím i této technologie. Jedná se o vyspělé, časem a komerčním nasazením prověřené řešení, například 14 z top-30 superpočítačů používá tento filesystém. Lustre podporuje až desetitisíce klientů, petabyty prostoru a propustnost ve stovkách GB/s.

Projekt jako takový se skládá ze tří částí:

- Patchované linuxové jádro, změny oproti standardnímu jádru vyžaduje především serverová část lustre, zároveň změny přináší výkonnostní výhody.
- Jaderný modul který implementuje lustre klienta a server.
- Userspace nástroje pro ovládání filesystému.

Lustre filesystém se skládá ze čtyř funkčních jednotek (viz. Obr.3.11).

MGS Management Server, nejedná se přímo o součást filesystému ale o jednotku poskytující konfigurační informace všem ostatním částem Lustre. MGS umožňuje provádět live úpravy konfigurace.

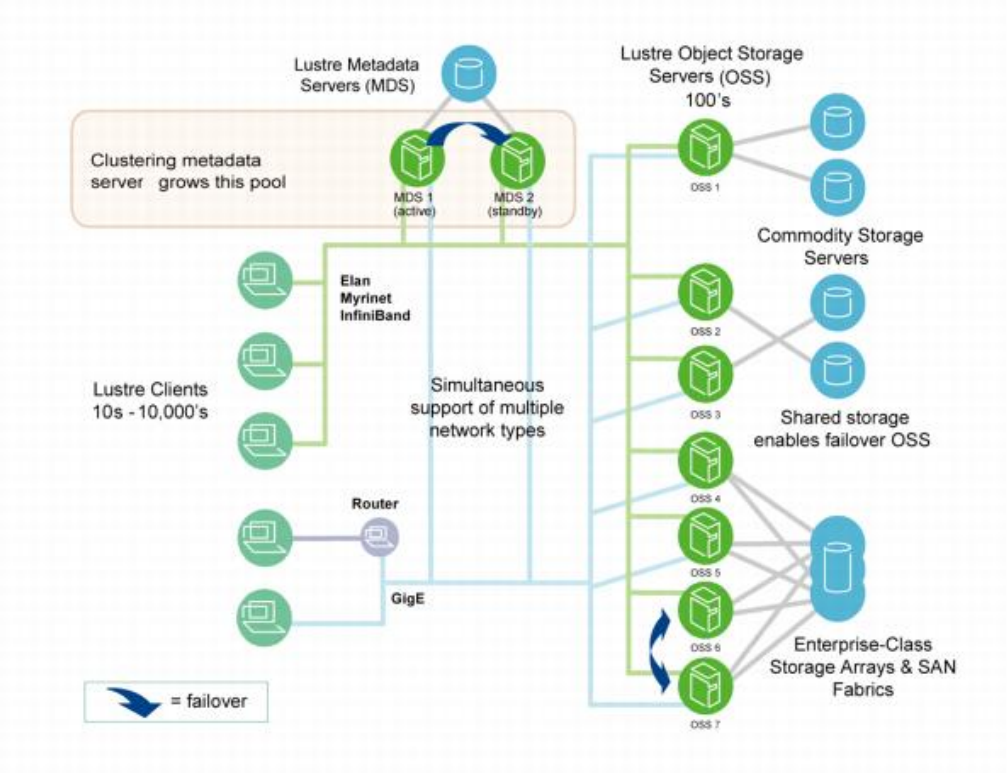
MDT Metadata Target je datové úložiště pro metadata jednoho filesystému. Metadata Server (MDS) je server, na kterém je jeden nebo více MDT. MDT obsahuje souborovou hierarchii (namespace) a atributy souborů, jako práva a odkazy na datové objekty uložené v OST.

OST Object Storage Target představuje úložiště datových objektů ze kterých se skládají jednotlivé soubory. Object Storage Server (OSS) poskytuje data z jednoho nebo více OST.

[30] http://sources.redhat.com/cluster/gfs/
[31] http://oss.oracle.com/projects/ocfs2
[32] http://www.lustre.org/
[33] http://www.gluster.org/glusterfs.php

Lustre klienti představují rozhraní mezi linuxovým VFS a Lustre servery. Všichni klienti vidí připojený filesystém jako celistvý, koherentní a synchronizovaný prostor umožňující souběžné čtení a zápis nad jedním souborem.

Klienti a servery spolu komunikují pomocí síťového API (LNET), které odstiňuje rozdíly mezi různými transportními technologiemi (podporované jsou např. Ethernet, InfiniBand, Myrinet...). Jednotlivé soubory jsou pomocí stripingu rozdělovány do jednotlivých OST, tím je možné škálovat propustnost a IO zátěž zvýšením počtu OST/OSS, zároveň tak není omezena maximální velikost souboru velikostí samotného OST.



Obrázek 3.11: Architektura LUSTRE; zdroj www.lustre.org

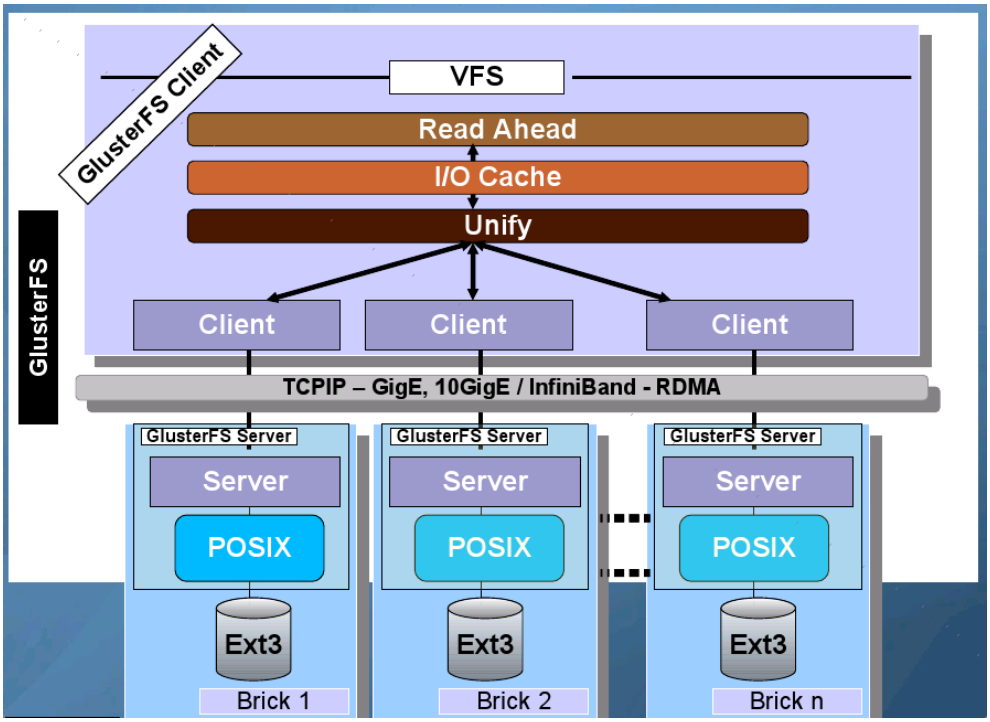
Vysoká dostupnost a failover je u Lustre podmíněna použitím sdíleného datového úložiště (SAN atp.), Lustre sám osobě žádný failover mechanizmus neimplementuje, spoléhá se na dostupná řešení typu Heartbeat. Typicky se používá konfigurace MDS Aktiv/Stand-by, OSS Aktiv/Aktiv, kde každý z OSS poskytuje jinou skupinu OST, a při výpadku jednoho z OSS, druhý OSS jeho OST přebírá.

Existují dva způsoby chování navázaných spojení (transakcí) při výpadku, *Failout* a *Failover*. V prvním případě při výpadku serveru transakce na klientu timeoutuje a aplikace vidí chybu EIO. V druhém případě (standardní nastavení) klient po zotavení nebo failoveru serveru danou transakci dokončí. Data jsou zapisována synchronně a po zotavení klient pouze zopakuje nekomitnuté transakce. Díky tomu je možné provádět například upgrade a údržbu bez ovlivnění operací klientů.

Lustre je cílen do segmentu výpočetních clusterů a podobných enterprise řešení, což je znát na jeho koncepci. Jednou z nepříjemností je nutnost držet se verzí jádra, pro které

existuje patch. Šance, že by se Lustre někdy dostal do hlavní větve linuxového jádra, je celkem mizivá, vzhledem ke změnám, které jsou pro jeho funkci nutné.

GlusterFS Začátek vývoje tohoto projektu se datuje do roku 2006, přesto se objevují zprávy (hlavně během posledního půl roku) o jeho úspěšných produkčních nasazeních. GlusterFS je kompletně implementován v uživatelském prostoru. Pro běh serverové části je třeba POSIXový operační systém, klientská část vyžaduje podporu FUSE[34] v jádře (Linux, FreeBSD, OpenSolaris). Kromě provozu přes TCP/IP podporuje GlusterFS propojení pomocí InfiniBand. Celý koncept GlusterFS je postaven na myšlence tzv. translátorů a jejich řetězení. Kromě translátorů klienta a serveru jsou prakticky všechny vlastnosti a rozšíření implementovány pomocí translátorů. Translátory jsou pro klienta a server shodné a je možné je aktivovat na obou stranách.



Obrázek 3.12: Schema modulárního uspořádání GlusterFS; zdroj: www.gluster.org

Translátory se dělí do několika skupin podle svého zaměření:

Výkonnostní translátory – do této skupiny patří translátory které se nějakým způsobem snaží zvýšit výkon GlusterFS, jako Read Ahead Translator, Write Behind Translator, Threaded I/O Translator, IO-Cache Translator.

Clusterovací translátory – implementují vlastnosti spojené s požadavky na paralelní filesystémy

- AFR Automatic File Replication Translator implementuje replikaci obsahu svazku na téměř libovolný počet uzlů, lze přirovnat k síťovému RAID-1.

[34] Filesystem in Userspace http://fuse.sourceforge.net/

- Stripe, tento translátor umožňuje škálovat výkonnost a ruší omezení maximální velikosti souboru vyplývající z kapacity filesystému jednotlivého uzlu.
- Unify je translátor spojující několik svazků/translátorů do jednoho velikého filesystému, lze v něm definovat jaký plánovač chceme pro takto vzniklý svazek použít, k dispozici jsou: ALU (Adaptive Least Usage), NUFA (Non-Uniform Filesystem Access) , Random, Round-Robin, Switch.

Ladící translátory

Translátory úložiště – GlusterFS závisí při operacích nad blokovým zařízením na lokálních filesystémech jako EXT3 nebo XFS, k těm se připojuje pomocí Posix translátoru.

Protokolové translátory – do této skupiny patří translátor klienta a serveru.

Speciální funkce – do této skupiny patří například translátory: filter (filtrování dle jména nebo atributů souborů), posix-locks, trash (implementuje funkci odpadkového koše).

Díky této modulárnosti je možné docílit konfigurací maximálně přizpůsobených našim potřebám. Lze dosáhnout chování obdobné JBOD, RAID-1, RAID-0, RAID-10 a 01. Plánování IO je možné na dvou úrovních, na úrovni souborů (Unify translátor), a na úrovni bloků (Stripe translátor). Jejich kombinací lze získat svazek, na kterém jsou některé soubory stripovány a ostatní jsou po uzlech rozkládány celé. Dalším zajímavým příkladem adaptivity GlusterFS je použití AFR. Zřejmě nejjednodušším a nejpřímočařejším použitím AFR je na straně klienta, kde přes něj spojíme svazky dvou uzlů. Klient tak bude veškeré zápisové operace duplikovat na tyto dva uzly, nepříjemné ale je, že se tím zvyšujeme zátěž klienta a zdvojnásobuje se množství dat přenášených mezi klientem a servery. Pokud AFR použijeme na straně serveru je možné tuto synchronizaci přenést na vyhrazenou síť propojující pouze servery, zvýší se tak propustnost sítě mezi servery a klienty, zároveň se odlehčí klientům. Negativem této konfigurace je zvýšení latence, díky prodloužení řetězu translátorů.

Tento projekt přichází s originálním řešením, jeho vývoj je velmi rychlý a nové funkce a vylepšení přibývají téměř každý týden. Odvážím se tvrdit, že tento projekt přináší řešení pro mnoho případů, pro které donedávna paralelní filesystémy nebyly vhodné. Jeden z důvodů proč GlusterFS vznikl, byla přílišná složitost zprovoznění některého v té době dostupného distribuovaného filesystému, v tomto ohledu GlusterFS rozhodně vítězí nad ostatními Open Source projekty.

Kapitola 4

Load Balancing

Dobře fungující rozkládání zátěže je jednou z podmínek pro uspokojivý provoz téměř jakéhokoliv clusteru. Rozkládání zátěže je možné provádět na mnoha vrstvách a rozličným způsobem, zde bych se chtěl zaměřit na rozkládání zátěže u internetových služeb, nebo obecněji u služeb fungujících nad TCP/IP protokoly.

4.1 Linux Virtual Server

Začátek vývoje tohoto projektu sahá do dob vývoje Linuxového jádra řady 2.0 a 2.2 (1998/1999), standardní součástí vanila jádra se stává v roce 2004. Linux Virtual Server (LVS) implementuje v rámci linuxového jádra přepínač na 4 vrstvě (ISO/OSI), ten umožňuje rozkládat TCP a UDP spojení mezi několik Real (Backend) serverů.

Terminologie:

Director – stroj na kterém běží LVS, který přímá spojení od klientů a předává je Real serverům

Real Server – jeden z aplikačních serverů farmy, běží na něm služba kterou se snažíme rozkládat

Virtual IP (VIP) – virtuální IP adresa pod kterou je daná služba prezentována klientům, vlastní ji Director

Real server IP (RIP) – IP adresa Realserveru

Rozkládání zátěže v LVS funguje na principu multiplexingu TCP spojení/UDP datagramů mezi několik Real serverů. Ve chvíli kdy přijde do Directoru paket inicializující nové TCP spojení, nebo UDP stream, je mu dle plánovacího algoritmu přiřazen jeden z Real serverů,na který je paket dále přeposlán. Další pakety náležící tomuto spojení/streamu jsou následně odesílány vždy na jemu přiřazený stroj, čímž je zaručena integrita spojení.

Virtuální služba kterou Director spravuje může být určena dvěma způsoby. Trojicí IP adresa, port a protokol nebo označením paketů pomocí iptables. Trojice údajů v prvním případě odpovídá cíli, na kterém klient očekává službu, ke které se chce připojit. Druhý případ se využívá pro seskupování více virtuálních služeb do jedné virtuální služby, zaprvé z důvodu zjednodušení konfigurace při vysokém počtu virtuálních služeb, zadruhé z důvodu persistence různých služeb (např. aby klient využívající http i https byl obsloužen vždy stejným serverem pro oba potokoly).

Algoritmy, které rozhodují o rozdělení spojení mezi Real servery, jsou implementovány jako jaderné moduly aby pro implementaci dalších algoritmů nebylo třeba zasahovat do jádra LVS.
K dispozici je nyní těchto několik algoritmů:

Round-Robin – rozděluje spojení rovnoměrně mezi Real servery.

Weighted Round-Robin – rozděluje spojení proporcionálně dle nastavené váhy (výkonnosti) Real serverů

Least-Connection – přiděluje více spojení těm serverům které mají méně aktivních spojení

Weighted Least-Connection – stejný jako Least-Connection, navíc je brána v potaz výkonnost jednotlivých serverů

Locality-Based Least-Connection – používá se pro loadbalancing cílových adres (používá se u cache clusterů), pro jednu cílovou IP adresu je přiřazen vždy stejný Real server, pokud je přetížený nebo nedostupný, je spojení přiřazeno serveru s nejmenším počtem aktivních spojení a další požadavky pro danou IP adresu jsou posílány tomuto serveru

Locality-Based Least-Connection with Replication – chová se podobně jako předchozí algoritmus s tím rozdílem, že pro každou cílovou adresu je vedena skupina Real serverů. Nové spojení je přiřazeno tomu serveru ze skupiny náležící dané cílové adrese, který má nejméně aktivních spojení. Pokud jsou všechny servery v dané skupině přetížené, je do této skupiny přiřazen nejméně zatížený server z clusteru. Pokud není do skupiny po určitou dobu přidán nový server, je z ní vyloučen nejzatíženější server.

Destination Hashing – spojení jsou přiřazována na základě statické hash tabulky kde je klíčem cílová adresa

Source Hashing – spojení jsou přiřazována na základě statické hash tabulky kde je klíčem zdrojová adresa

Shortest Expected Delay – spojení jsou přiřazena serveru s nejnižším očekávaným zpožděním, očekávané zpoždění se stanovuje z váhy (výkonnosti) serveru a počtu aktivních daného serveru

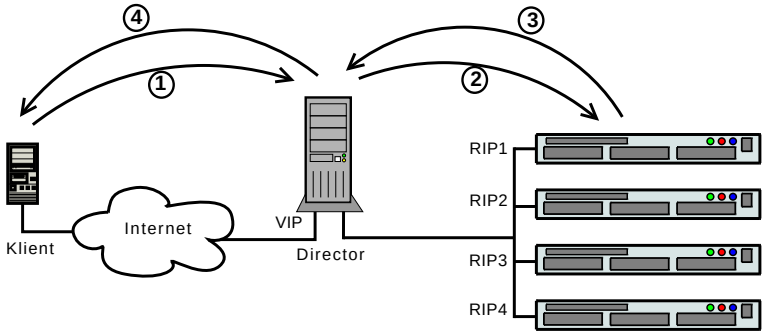
Never Queue – tento algoritmus pracuje ve dvou režimech. Pokud je některý server volný, je mu přiřazeno spojení i v případě, že existuje server, u kterého je nižší očekávané zpoždění. Pokud není žádný server volný, je spojení přiřazeno dle algoritmu Shortest Expected Delay.

Kromě loadbalancingu je v LVS myšleno na vysokou dostupnost samotného loadbalanceru, v rámci něj je implementován démon pro synchronizaci stavových dat procházejících spojení. Démon pro synchronizaci s backup loadbalacery využívá UDP multicast.

LVS má tři různé způsoby jakými může přeposílat pakety jednotlivých spojení: Network Address Translation (NAT), IP-IP tuneling, Direct Routing.

4.1.1 LVS NAT

Ve chvíli kdy se klient pokusí navázat spojení se službou kterou zaštiťuje LVS, přijde na Director paket s cílovou adresou odpovídajíci VIP. Director zjistí zda IP adresa, port a protokol odpovídají virtuální službě kterou poskytuje. Pokud ano, rozhodne dle zavedeného plánovacího algoritmu, kterému Real serveru bude toto spojení přiřazeno a zavede jej do své tabulky spojení. Poté je v paketu přepsána cílová IP adresa a port na hodnoty odpovídající zvolenému Real serveru a paket je mu přeposlán. Pokud příchozí paket patří tomuto spojení, je dle tabulky nalezen odpovídající Real server, paket přepsán a přeposlán. Pakety které vrací Real server jsou na Directoru opět přepsány tak, aby zdrojová adresa odpovídala VIP. Ve chvíli, kdy je spojení ukončeno, nebo timeoutuje, je záznam z tabulky spojení odstraněn.



Obrázek 4.1: LVS NAT

4.1.2 LVS IP Tunneling

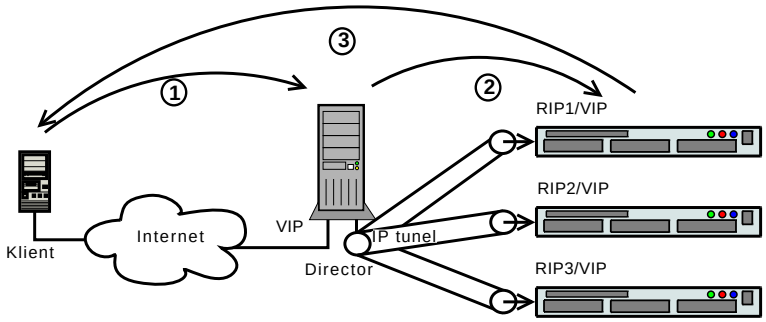
Nevýhodou použití NAT v LVS je, že pakety od Real serverů zpět ke klientovi musí projít Director, tím se může stát sám loadbalancer úzkým hrdlem. Další dvě techniky, IP Tuneling a Direct Routing tímto problémem netrpí, odpovědi klientům u nich mohou odcházet jinou cestou, než přes Director, skrz který přišly.

Způsob zpracování nového spojení na Directoru je shodný jako u NAT, až do chvíle, kdy má Director paket přeposlat. Zde paket nikterak nepřepisuje, ale zapouzdří ho do IP datagramu a odesílá cílovému Rael Serveru. Ten takto zapouzdřený paket rozbalí a zpracuje. Podmínkou je schopnost operačního systému využívat IP tuneling a možnost nastavení VIP na non-arp nebo skrytém rozhraní. Odpovědi klientům Realserver odesílá přímo, bez účasti loadbalanceru.

Výhodou IP tunelingu oproti Direct routing je možnost umístění Real serverů mimo lokální ethernetový segmet, mohou se dokonce nacházet i v geograficky rozdílných místech. Na druhou stranu tam, kde nemáme potřebu servery rozmisťovat odděleně, podává Direct Routing vyšší výkony.

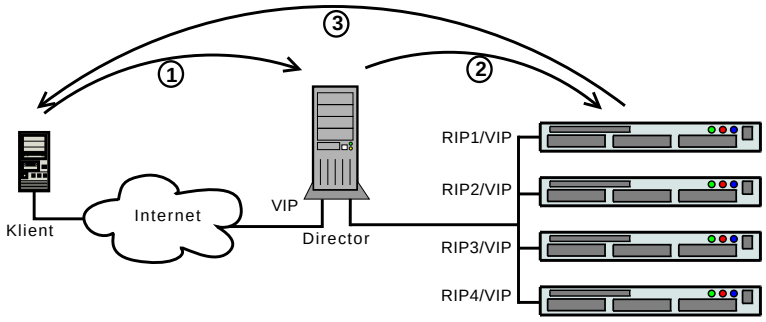
4.1.3 LVS Direct Routing

Podobně jako u IP Tunelingu, je zpracování příchozího spojení shodné s procedudou u NAT metody až do chvíle, kdy má Director paket přeposlat dále. Nedochází ani k přepisování IP adresy ani k zapouzdření, ale k přenastavení MAC adresy rámce na MAC adresu cílového Real serveru. Proto je nutné, aby Director a Real servery byly ve stejném segmentu sítě,



Obrázek 4.2: LVS IP tunneling

zároveň je nutné aby Real servery měly možnost nastavení VIP na non-arp, nebo skrytém rozhraní, protože není možné na jednom segmentu sítě provozovat několik zařízení se stejnou IP adresou. Tím že Real servery nebudou navenek ohlašovat vlastnictví VIP adresy se toto omezení obchází. Pakety s odpověďmi klientům Realservery odesílají přímo.



Obrázek 4.3: LVS Direct Routing

4.2 Load Balancing pomocí DNS

DNS loadbalancing spočívá v přídělení několika IP adres jednomu doménovému jménu, většina DNS serverů s takovouto konfigurací vrací každému klientovi vždy postupně jednu z těchto IP adres. Tím se docílí rozložení dotazů od jednotlivých klientů na různé servery. Loadbalancing pomocí DNS se používá v zásadě ve dvou případech.

- V případě, že potřebujeme rozložit zátěž na několik strojů a potřebujeme to udělat rychle, bez dalších zásahů do infrastruktury a nasazování jakékoliv další technologie. Většinou se ale jedná pouze o předstupeň plnohodnotného loadbalancingu pomocí některé jiné techniky, protože tento způsob neposkytuje téměř žádnou kontrolu nad rozdělováním práce jednotlivým serverům. Zároveň je nemožné dynamické začleňování/odstraňování serverů do/z clusteru v závislosti na jejich přetížení nebo stavu.

- Druhý případ kdy se DNS loadbalancing nasazuje je naopak v případech, kdy narážíme na technické limity použité techniky, nebo potřebujeme zátěž rozložit efektivněji

(například geograficky). Například, pokud máme cluster poskytující nějakou službu pro klienty z celého světa a provoz tohoto clusteru překročí kapacitu maximální možné konektivity (např. 1Gbit), je možné postavit druhý cluster připojený další 1Gbit linkou a zátěž mezi nimi rozdělit právě pomocí DNS. Druhou možností je rozdělit cluster na několik menších a ty následně umístit každý na jeden kontinent, zátěž mezi nimi poté řídit pomocí DNS, kde se DNS server bude rozhodovat o odpovědi kterou odešle na základě geografické lokace zdroje dotazu.

První případ je řešitelný standartními prostředky (DNS server, /etc/hosts ...). Druhý případ již není jednoduše řešitelný pomocí Open Source prostředků, protože chybí implementace DNS serveru, který by obsahoval podobnou rozhodovací logiku. Jediný náznak možného řešení je v DNS serveru Bind, kde pomocí ACL je možné nadefinovat rozdílné zónové záznamy pro různé klientské sítě či IP adresy. Pokud si však uvědomíme, jaký typ společnosti je schopen takovouto techniku využít, je spíše pravděpodobné doprogramování této vlastnosti do stávajícího Open Source DNS serveru vlastními silami, nebo pronajmutí této služby od některé společnosti, která se na to specializuje.

Kapitola 5

HA a LB na aplikační vrstvě

Loadbalancing na úrovni síťových spojení je ideově poměrně jednoduchý a lze s ním dosáhnout propustnosti na hranicích možností síťového HW. Tento způsob však není možné použít ve chvíli, kdy zátěž (dotazy) potřebujeme rozkládat na základě informací dostupných až ve vyšších vrstvách, typicky v 7 vrstvě (HTTP, FTP, SQL, ...). V takových případech je třeba implementovat řešení přímo pro protokol dané aplikace. Tím se sice vzdáváme obecné použitelnosti, ale naopak získáváme přístup k architektuře dané aplikace a je tak možné kromě samotného loadbalancingu a vysoké dostupnosti implementovat funkce s „přidanou hodnotou“, typicky se jedná o funkce, které mají za cíl odlehčit, zrychlit, nebo zabezpečit práci backend serverů. Mezi nejběžnější takto implementované funkce patří komprese, šifrování, cachování a filtrace.

V této kapitole se budu věnovat dvěma tématům o kterých si myslím, že v dané problematice převyšují všechny ostatní. První z nich je použití reverzní proxy jako webový loadbalancer/akcelerátor. Druhým je replikace a clustrování relačních databází, konkrétně MySQL a PostgreSQL.

5.1 HTTP reverzní proxy

Důvodů pro nasazení reverzní proxy před farmu web serverů může být několik. Mezi hlavní patrně patří možnost ovlivňovat výběr cílového serveru na základě URL nebo jiné informace dostupné v rámci HTTP protokolu, možnost modifikovat dotazy, cachovat obsah poskytovaný backend servery, SSL/TLS offloading atp.. Nejčastěji se reverzní proxy nasazuje v případech, kdy aplikační server má vyšší paměťové nároky, které vzrůstají spolu s počtem připojených klientů a server není schopen efektivně obsloužit větší počet paralelních spojení. Reverzní proxy je zpravidla navrhována s důrazem na minimalistické paměťové nároky neměnící se příliš s počtem připojených klientů, dále je kladen důraz na minimalizaci režie při obsluze velkého počtu paralelních spojení. Díky těmto vlastnostem je možné odstínit backend servery od zátěže vyplývající z přímé komunikace s klienty, kde proxy server od backend serveru odpověď dostává maximální možnou rychlostí, kterou umožňuje backend server a komunikační linka mezi ním a proxy. Nedochází tak k prostojům u procesů serveru způsobených pomalou komunikací s klienty. Tato režie se přenáší na proxy, která je pro tento druh zátěže optimalizovaná. Pokud reverzní proxy implementuje cachování, SSL offloading, nebo podobnou funkčnost, mluvíme pak o webovém akcelerátoru. Ten dokáže přebrat část práce backend serverů, a tu vykonávat zpravidla efektivněji než samotný backend server. Hlavním výkonnostním parametrem reverzní proxy je maximální počet pa-

ralelně držených spojení, při kterém ještě nedochází k výraznému nárůstu zpoždění při odbavení dotazů. Dalším parametrem je maximální propustnost pro určité velikosti dotazů, u cachujících reverzních proxy je významná také propustnost cachovaných objektů. U webových projektů s vysokou zátěží přináší reverzní proxy zpravidla možnost zapnutí persistentních spojení (keepalive) mezi proxy a klienty, což pro ně představuje výhodu v možnosti načítat více objektů v rámci jednoho TCP spojení. Odpadá tím režie spojená s navazováním a ukončováním TCP spojení, zpravidla se na straně klienta projeví zvýšenou rychlostí načítání obsahu, zároveň zde ale vyvstává požadavek, aby proxy dokázala efektivně držet vysoký počet paralelně otevřených spojení bez výrazného zpomalení odbavování dotazů.

5.1.1 Apache modproxy

Apache je zřejmě nejpoužívanější Open Source webserver, jeho součástí jsou moduly (mod_proxy, mod_cache), které umožňují použít Apache jako reverzní proxy. Výhodou Apache je jeho konfigurovatelnost díky které je možné přizpůsobit ho téměř pro jakýkoliv úkol. Rozsáhlost a komplexnost Apache ale může být chápána negativně. Pro software fungující jako reverzní proxy je nespornou výhodou, pokud je jeho kód kompaktní a lehce auditovatelný, což Apache rozhodně nesplňuje. Zároveň složitost jeho konfigurace oproti jednoúčelovému proxy serveru je vyšší.

Propustností se řešení pomocí Apache může srovnávat s ostatními zde uvedenými projekty, zásadní problém však nastává ve chvíli, kdy potřebujeme paralelně držet mnoho otevřených spojení (1000 a více). Projevuje se zde architektura Apache, kdy každé spojení má vyhrazen jeden proces/thread, teoretickým řešením by mohlo být použití event MPM[1], ten je ale dostupný až od verze Apache 2.2 a je stále v experimentálním stavu. Schopnost sledovat stavy backend serverů a rozdělovat dotazy na jejich základě je možné až od verze 2.2.

5.1.2 Squid

Squid je zřejmě nejpoužívanější cachovací proxy server, jeho kořeny sahají do doby vzniku www. Funkce reverzní proxy byla do projektu přidána až v průběhu doby. Tyto dvě skutečnosti jsou často Squidu vytýkány, jeho architektura je omezena dobou vzniku tohoto projektu, není například možné využít výhod multiprocesoru. Další nepříjemností je, že pro reverzní proxy není implementován mechanizmus dohledu nad stavem backend serverů a jejich výpadky musí být řešeny jiným způsobem.

Pokud odhlédneme od architekturní zastaralosti, Squid je stabilní a rozšířené řešení se solidním výkonem, širokou konfigurovatelností a velmi dobrou dokumentací.

5.1.3 Pound

Pound je oproti předchozím jen a pouze reverzní proxy, neimplementuje vlastní webserver ani cachování, zaměřuje se na minimalizaci vlastní režie, bezpečnost a jednoduchost konfigurace. Mezi jeho funkce patří: Loadbalancing s řešením dostupnosti backend serverů, možnost zachování perzistence spojení na základě adresy klienta, jeho autentizace, parametru URL, podle cookie, nebo http hlavičky.Dále je možné využít SSL offloading.

[1] http://httpd.apache.org/docs/2.2/mod/event.html

Pound poskytuje již několik let stabilní řešení, jeho vývoj je stále aktivní. Nesnaží se však příliš rozšiřovat funkčnost, spíše udržovat stabilitu stávajícího kódu s občasným vylepšením funkčnosti.

5.1.4 Varnish

Varnish[2] je webový akcelerátor, jeho vývoj začal v roce 2006 a ke konci téhož roku byla uvolněna první stabilní verze. Základním principem tohoto projektu je moderní design s cílem maximálního výkonu a propustnosti. Hlavní funkční částí je cachování které oproti Squidu neimplementuje vlastní správu paměti pro přesuny objektů mezi RAM a diskem, v tomto ohledu Varnish spoléhá výhradně na operační systém a jeho management virtuální paměťi[3]. Mezi zajimavé funkce patří například možnost externího zneplatnění objektů nebo prefetching objektů u kterých se blíží doba expirace.

Přesto že jde o velmi mladý projekt a jeho vývoj je poměrně dynamický, architekturně i kvalitou kódu je na velmi vysoké úrovni. Webové stránky projektu uvádí, že výkonnostní rozdíl oproti Squidu se může pohybovat mezi 10ti až 20ti násobkem. Projekt celkem rychle nabírá uživatelskou komunitu, která doufejme povede k dalšímu zkvalitnění, již tak velmi pěkného projektu.

5.2 SQL replikace

5.2.1 MySQL replikace

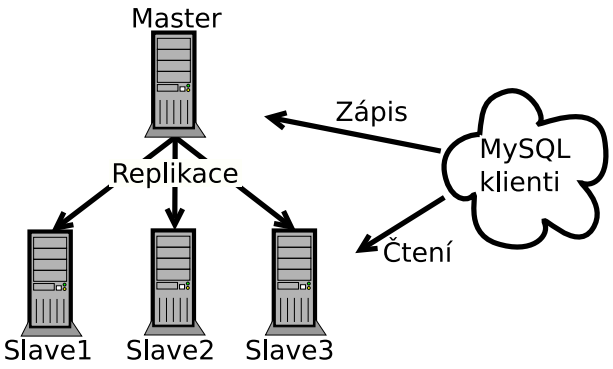
Replikace v MySQL je dostupná již od verze 3.23, jde o široce používanou techniku, asi nejčastějším důvodem jejího nasazení je snaha o škálování výkonnosti. Při replikaci se servery dělí na Master a Slave servery, kde na Master server jsou směrovány všechny zápisové operace, ty Master provádí a zároveň vytváří jejich binární log. Slave servery se připojují k Master serveru a podle binárního logu zápisových operací provádí zápisy na vlastní databázi. Slave servery mohou poskytovat jen čtecí operace a zároveň musí být schopny provádět stejný počet zápisových operací jako Master. Proto MySQL replikace přináší zvýšenou výkonnost pouze u takových aplikací, kde převládají čtecí operace. Z poměru čtecích a zápisových operací lze stanovit efektivní počet Slave serverů, nad který se již celková výkonnost nezvyšuje.

Důvodů pro nasazení replikace ale existuje více, než jen zvýšení výkonu, především je možné využít tímto způsobem získané redundance dat. Replikace se proto používá například jako řešení „Disaster Recovery“, kde se replika databáze může nacházet na geograficky jiném místě. Replikace nevyžaduje aby Slave server byl neustále připojen ke svému Masteru, pouze je nutné synchronizovat tak často, aby na Masteru ještě existoval binární log se všemi daty od poslední synchronizace. Díky tomu je možné replikaci používat i přes ne zcela spolehlivá média jako je internet nebo lze replikaci obecně využít jako efektivní způsob zálohování.

Rozkládání dotazů mezi jednotlivé servery musí být řešeno mimo MySQL, stejně tak rozlišování mezi čtecími a zápisovými operacemi je nutné řešit externě. Klasickým řešením dělení dotazů na čtecí a zápisové je v rámci aplikace, kde pro oba typy dotazů máme zvláštní spojení do databáze. Rozkládání čtecích dotazů je možné řešit mnoha způsoby, od řešení v rámci aplikace přes DNS round-robin, po síťové loadbalancery, jako je Linux Virtual Server nebo jeho komerční hardwarové obdoby. Pokud nechceme nebo z nějakého

[2] http://varnish.projects.linpro.no/

[3] http://varnish.projects.linpro.no/wiki/ArchitectNotes



Obrázek 5.1: Typické vyžití MySQL replikace

důvodu nemůžeme modifikovat naší aplikaci tak, aby byla schopná sama rozlišovat čtecí a zápisové operace, můžeme použít řešení jako je MySQL Proxy[4] nebo SQL Relay.[5] Tyto projekty poskytují službu loadbalancingu a connection poolingu, zároveň se snaží určitým způsobem řešit dělení čtecích a zápisových operací. Nasazení těchto proxy však musí být dobře promyšleno, mohou se stát úzkým hrdlem mezi aplikací a databází, zároveň mohou být „Sinlge Point Of Failure“.

Failover, stejně jako loadbalancing MySQL nikterak neřeší a musí být řešen externě. Je možné při výpadku například uvažovat o povýšení jednoho ze Slave serverů na Master, přitom je potřeba zajistit splnění několika požadavků, především je třeba zjistit, který ze Slave serverů je v replikaci nejdále, ten povýšit na Master. Dále je třeba zajistit napojení zbylých Slave serverů na nového Mastera, zajistit propagaci změny Mastera klientům. Je nutné, aby starý Master po svém zotavení nepřebral funkci Mastera zpět, nejdříve je nutné synchronizovat data s aktuálním Master serverem, což zpravidla znamená ruční zásah. Mnohem jednodušší situace je, pokud použijeme konfiguraci se dvěma master servery, které se mezi sebou replikují, tím se zjednodušuje failover ze strany klientů, který je možné řešit síťovým loadbalancerem. Dále odpadá hledání nástupce vypadnuvšího Mastera, zároveň mizí problém s resynchronizací po zotavení, o tu se postará replikace vůči druhému Master serveru. Zřejmě nejprůhlednější řešení failoveru Master serveru je při použití sdíleného média (SAN, DRBD,...), pro data databáze a konfiguraci Activ/Stand-by, pomocí nějakého HA řešení jako je Heartbeat.

Replikace mezi Master databází a Slave servery je asynchronní, nikterak není zaručeno zda budou Slave servery v určitém časovém bodě vzájemně obsahovat shodná data ani není zaručen stav replikace vůči Master databázi. Slave servery, zvláště při vysokém zatížení, mohou „zaostat“ ve zpracování binárního logsouboru, z čehož plyne možná nekonzistence mezi jednotlivými servery. Je důležité, aby si této skutečnosti byl programátor vědom, a spolusouvisející operace prováděl vždy v rámci jednoho spojení.

5.2.2 PostgreSQL Slony-I

Slony-I[6] je Master - Slave asynchronní replikační systém podporující hierarchické napojení slave serverů a online změnu Master serveru. Celý systém se skládá ze tří částí,

[4] http://forge.mysql.com/wiki/MySQL_Proxy
[5] http://sqlrelay.sourceforge.net/
[6] http://main.slony.info/

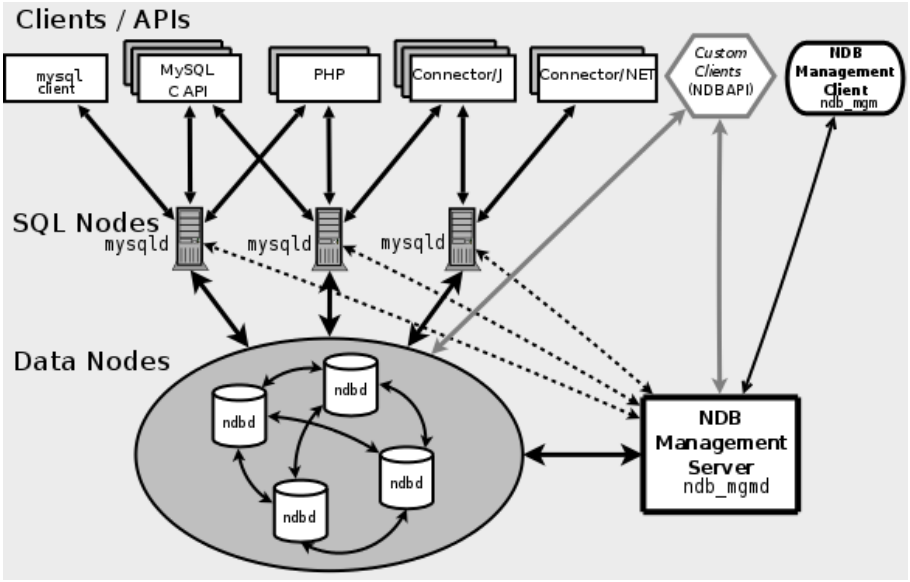
démona zajišťujícího komunikaci mezi stroji a samotnou replikaci, uživatelského rozhraní pro ovládání replikace a část nacházející se v databázi (v její datové části) podpůrné tabulky, triggery atp.. Replikace se nastavuje na úrovni tabulek a sekvencí, kde několik těchto objektů tvoří tzv. *Set*, pro který se definuje replikační schéma. Jednotlivé stroje v clusteru mohou mít funkce Origin, Provider, Subscriber, přičemž Origin je Master server nebo také Master Provider - zdroj všech replikačních změn, Subscriber odpovídá Slave serveru. Protože je možné hierarchické napojení, může být Subscriber zároveň i Provider. Díky tomu, že je replikace definována pro skupiny tabulek a sekvencí, je možné, aby jednotlivé stroje v clusteru vystupovaly v různých rolích pro jednotlivé sestavy (Sets). Replikace je založena na triggerech, které při změně v tabulkách nebo sekvencích vygenerují událost, ta je odchycena replikačním démonem a dle replikačního schématu propagována na Subscribery.

Slony neposkytuje žádnou infrastrukturu pro loadbalancing, sledování stavu jednotlivých serverů nebo automatický failover. Tyto funkce stejné jako u replikace MySQL musí být řešeny externě. Failover sám osobě je ve Slony implementován. Spočívá v předefinování replikačního schématu. Pro loadbalancing je možné použít některé z proxy řešení, jako je Pgpool,[7] nebo SQL Relay[8].

Slony je zřejmě jediný skutečně používaný projekt pro replikaci PostgreSQL, jeho vývoj je stabilní a drží krok s vydáváním nových verzí PostgreSQL. Pokud bych měl porovnat snadnost implementace Slony-I a MySQL replikace, rozhodně vítězí MySQL.

5.3 SQL clustrování

5.3.1 MySQL Cluster



Obrázek 5.2: Architektura MySQL Clusteru; zdroj: dev.mysql.com

MySQL Cluster[9] představuje technologii poskytující komplexní clusterové řešení navržené s důrazem na vysokou dostupnost a co nejlepší výkonovou škálovatelnost. Návrh počítá s

[7] http://pgpool.projects.postgresql.org/
[8] http://sqlrelay.sourceforge.net/
[9] http://www.mysql.com/products/database/cluster/

využitím komoditních technologií a architekturou „shared nothing“. Srdcem clusterového řešení je Storage Engine NDB[10], který poskytuje redundantní a škálovatelné úložiště dat databáze.
Cluster se dělí na tři části(viz. Obr.5.2):

Management node Představuje řídící proces (pojem *node* zde představuje spíše proces s určitou funkcí než počítač na kterém běží) poskytující ostatním nodům clusteru konfiguraci, starající se o spuštění, zastavení clusteru, provádění záloh a podobné činnosti.

Data node Odpovídá instanci NDB, stará se o ukládání dat.

SQL node Představuje proces přistupující a operující nad daty clusteru. Prakticky se jedná o instanci mysql serveru využívající Storage Engine NDB.

Oproti Mysql Replikaci, která provádí asynchronní replikaci, Mysql Cluster s využitím dvoufázového commitu implementuje replikaci synchronní.Tím je zaručeno, že po commitnutí transakce jsou na všech strojích shodná data. Data v NDB tabulkách jsou automaticky rozdělena přes všechny Data nody, čehož je docíleno pomocí hašovacího algoritmu nad primárními klíči tabulek. Pro koncového uživatele je tato funkce zcela transparentní. Počet Data nodů tak přímo ovlivňuje výkon clusteru. Zároveň jsou data udržována redundantně, stupeň této redundance je konfigurovatelný, typický stupeň redundance je 2. Veškerá data na Data nodech jsou držena v paměti, požadavky na její velikost lze stanovit přibližně takto:

*(Velikost databáze * Stupeň redundance) / Počet Data nodů*

Ukládání dat na disk Data nodů je prováděno asynchronně pomocí checkpointů a transakčního logu, to je možné právě díky synchronní replikaci.

MySQL Cluster bezesporu poskytuje kvalitní HA systém, u kterého redundanci jednotlivých komponent máme zcela ve svých rukou a můžeme tak výsledné řešení přizpůsobit našim potřebám, ať už výkonnostním, nebo požadavkům na vysokou dostupnost. Bohužel jsou zde i některá omezení, která toto řešení má, většina z nich vyplývá z omezení NDB, mezi ně například patří:

- Veškerá data jsou držena v paměti, od verze MySQL 5.1 bude možné data kromě indexů držet na disku místo v paměti.
- Není možné použít fulltext vyhledávání.
- Není implementována referenční integrita.
- Oddělení transakcí je možné pouze na úrovni READ COMMITTED (například InnoDB poskytuje tyto úrovně oddělení: READ UNCOMMITTED; READ COMMITTED; REPEATABLE READ; SERIALIZABLE).
- Database autodiscovery je implementována až od verze 5.1 (pokud máme v clusteru více SQL nodů, a na jednom z nich vytvoříme databázi, ostatní k ní nebudou mít přístup dokud na nich nespustíme CREATE DATABASE jmeno_databaze).
- Zamykání tabulek funguje pouze na SQL nodu, kde byl zámek vyvolán.

[10] **N**etwork **D**ata**b**ase http://dev.mysql.com/doc/refman/5.0/en/storage-engines.html

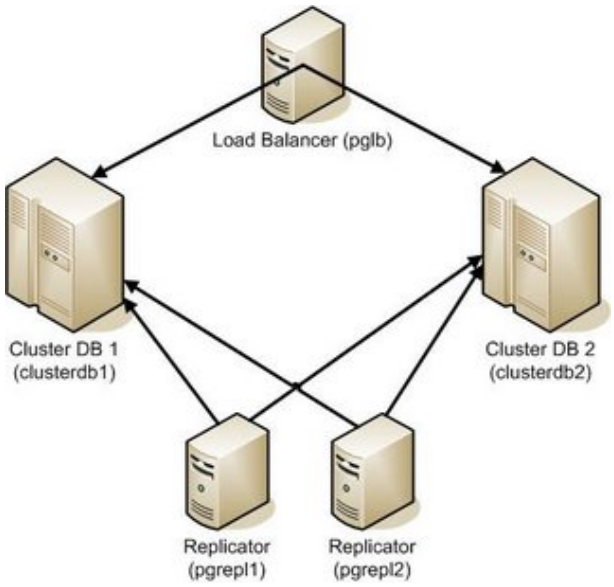
- ...

MySQL Cluster může být skvělé řešení pro aplikace, které jsou od počátku vytvářeny s vědomím omezení která zde jsou. Pro aplikace, které byly vyvíjeny s běžným Mysql serverem a následně řeší problém škálovatelnosti a vysoké dostupnosti, může být přechod na Mysql Cluster až příliš bolestivý a zásahy do stávající aplikace příliš rozsáhlé, v těchto případech je zpravidla vhodnější použít MySQL replikaci.

5.3.2 PGCluster

PGCluster[11] je synchronní multi-master replikační systém pro PostgreSQL. Projekt se skládá z upravené verze PostgreSQL a vlastních serverů pro loadbalancing a replikaci, konkrétně cluster na bázi PGCluster obsahuje tři stavební bloky:

- Loadbalancer, který se stará o rozkládání dotazů mezi databázové servery a o vyčlenění nefunkčních DB serverů z clusteru.
- Databázové servery provádějí loadbalancerem přidělené dotazy.
- Replikační servery zprostředkovávají replikaci zápisových operací mezi DB servery.



Obrázek 5.3: Architektura PGClusteru; zdroj : odyssi.blogspot.com

Díky synchronní replikaci nedochází ke zpoždění mezi zápisy na jednotlivé DB servery, transakce je potvrzena vždy až po zapsání dat na všechny DB servery v clusteru. Loadbalancing je u PGClusteru oproti MySQL clusteru součástí řešení a není třeba použít externích řešení jako je LVS nebo PGPool. Failover vzhledem k tomu že jde o multi-master systém spočívá pouze ve vyčlenění vadného serveru z clusteru, což je součástí řešení jak na straně loadbalanceru, tak na straně replikačního serveru.

Vývoj projektu celkem spolehlivě reaguje na nové verze PostgreSQL, je zde ale problém vyplývající z malé rozšířenosti tohoto projektu. Pokud se budeme snažit o konfiguraci,

[11] http://pgfoundry.org/projects/pgcluster/

která je shodná nebo podobná konfiguraci provozovanou některým z vývojářů, máme vysokou pravděpodobnost, že výsledek bude fungovat, pokud se ovšem pokusíme o konfiguraci nikým neprověřenou máme, naopak vysokou pravděpodobnost, že strávíme mnoho času odlaďováním chyb a výsledek rozhodně není zaručen. Další problém vyplývá z architektury multi-master se synchronní replikací a je jím výkon. Pokud cluster zpracovává jen čtecí dotazy násobí se výkon počtem DB serverů, stačí ovšem malé procento zápisových operací a výkon padá na úroveň jednoho DB serveru. S vyšším počtem serverů a zvyšujícím se poměrem zápisových operací se propad ještě prohlubuje.

Kapitola 6

Analýza návrhu a stávajícího stavu projektu živých sportovních výsledků

V této kapitole se budu zabývat historií projektu od jeho počátku až po současnost. Především se zaměřím na popis výchozí situace pro návrh řešení, samotný návrh včetně uvažovaných, ale zavržených variant řešení. Kromě samotné implementace zmíním problémy vyplývající ze vzrůstajícího provozu a jejich řešení. Nakonec zhodnotím současný stav s ohledem na dosaženou výkonnost, splnění předpokladů a požadavků vznesených před, a v rámci zpracování návrhu.Dále budu hodnotit vzhledem k dalšímu rozvoji tohoto projektu.

6.1 Výchozí podmínky, požadavky na návrh řešení

Jde o webový projekt specializující se na poskytování výsledků sportovních utkání v reálném čase. Vysoké nároky tohoto projektu na servery a infrastrukturu, na které běží, vyplývají ze snahy poskytovat tuto službu (stavy utkání), s co možná nejmenším zpožděním oproti skutečnému vývoji jednotlivých utkání. Z pohledu uživatelů je tato minimalizace zpoždění realizována automatickou aktualizací rámu (iframe) s tabulkami výsledků sledovaných zápasů ve webovém prohlížeči. Typický uživatel si po příchodu na stránky projektu vybere utkání, která ho zajímají, sestaví si z nich vlastní výsledkovou tabulku, a poté již jen sleduje změny této stránky, zpravidla do konce všech jím sledovaných utkání. Pokud budeme uvažovat návštěvu o délce trvání fotbalového zápasu (90 minut) a automatickou aktualizaci po 20 vteřinách, představuje návštěva jednoho uživatele přibližně 300 dotazů. Takové číslo je sice poměrně vysoké, ale nikterak závratně, hlavně s ohledem na rozložení těchto dotazů do delšího časového úseku (90 min.). Faktorem, který způsobuje výkonnostní náročnost, je časová kumulace přístupů uživatelů. Tato kumulace se projevuje hned v několika časových rovinách:

- Vzrůstající a snižující se návštěvnost odpovídá sezónám v jednotlivých sportech, nejvyšší návštěvnost je v průběhu jara a podzimu, kdy probíhají soutěže ve všech hlavních sportech.
- Stejným způsobem návštěvnost kopíruje počet utkání v jednotlivých dnech v týdnu, kde nejexponovanějšími dny jsou víkendové.

- Posledním faktorem kumulace je rozložení zápasů ve dni. Výkonnostní špičky jsou zde dosahovány v odpoledních až večerních hodinách.

Kromě této časové kumulace přístupů je zde ještě fakt neustále se zvětšující komunity uživatelů, který představuje další požadavek na výkon a škálovatelnost tohoto projektu.

Tento projekt vznikl přibližně před dvěma roky (zima 2006), pro svůj běh využívá klasickou kombinaci Linux, Apache, MySQL, PHP. Od počátku byl tento projekt velmi ambiciózní a předpokládal se prudký růst návštěvnosti, proto byl umístěn na vyhrazený server v konfiguraci P4 3GHz, 4GB RAM. Nárůst návštěvnosti byl však tak výrazný, že byla evidentní nutnost nasazení takového řešení, které dokáže takovýto růst absorbovat. Po přibližně třech až čtyřech měsících narazil server, na kterém projekt fungoval, na své výkonostní hranice (cca 200-250 dotazů za vteřinu, 10Mbit/s). Bylo to v době před začátkem letních prázdnin, kdy končí většina soutěží a následuje klidnější letní období, kdy návštěvnost dále neroste. Letní prázdniny se tak staly pomyslnou lhůtou pro návrh a implementaci řešení, které při takovémto růstu návštěvnosti obstojí. Požadavky na nové řešení se dají shrnout do několika bodů:

- Výsledný systém musí být škálovatelný takovým způsobem, aby nebyla nutná změna architektury nového řešení po dobu přibližně jednoho a půl, až dvou let. Škálovatelnost minimálního řešení by měla být alespoň desetinásobná.
- Růst výkonu celku by měl být lineárně závislý na počtu výpočetních jednotek (serverů), nejlépe by měl být roven součtu výkonnosti těchto jednotek.
- Při zvyšování počtu serverů nesmí zároveň růst složitost administrace systému jako celku.
- Pro vývojáře projektu musí nové řešení představovat zcela transparentní náhradu za jedno serverové řešení.

Poslední požadavek vychází především z nedostatku času na zásahy do aplikace a z prakticky nulových zkušeností vývojářů s vývojem a provozem aplikací v distribuovaném prostředí. Dalším aspektem, který bylo vhodné zohlednit v návrhu, byly technologie v té době používané ve firmě, která měla výsledný systém provozovat.

6.2 Návrh řešení a jeho implementace

Návrh nového řešení je možné rozdělit do tří oblastí. První z nich je návrh základní výpočetní jednotky (webserveru), další oblastí je řešení loadbalancingu a vysoké dostupnosti. Poslední oblastí jsou podpůrné technologie nutné jednak pro samotnou funkci webserverové farmy, a zadruhé technologie pro správu a monitoring.

Základní výpočetní jednotka umožňující škálování výkonu celého systému je v tomto případě webserver. Jeho návrhu bylo nutné věnovat velikou pozornost, především jeho maximální unifikaci a odstranění nebo ošetření všech částí systému, které by znemožňovaly hromadnou administraci. Z technologického hlediska byla volba celkem jasná a odvíjela se od technologií již v té době používaných. Konkrétně OS: Debian GNU/Linux, dále vlastní Apache, Mod PHP a eAccelerator. Hardwarově jsou tyto stroje navrženy především s ohledem na maximální poměr cena/výkon, a to v oblasti nižší cenové hladiny. Záměrně nebylo počítáno s HW, který by tyto stroje prodražoval jako ECC RAM, druhý disk pro zapojení do RAID, hotswap atp.. Důvodem bylo odstínění výpadků jednotlivých strojů na úrovni

loadbalanceru, nebylo tedy třeba takovýmto způsobem zvyšovat spolehlivost jednotlivých strojů a peníze bylo efektivnější investovat například do rychlejších CPU.

Aplikace pro svůj běh využívá MySQL databáze. Vzhledem k tomu, že obecně škálovatelnost databázových systémů je řádově nižší a obtížnější, byl návrh MySQL backendu prakticky opačný oproti webserverům. Počáteční řešení bylo navrženo jako výkonný jednoserverový systém. Hardware pro tento server byl volen s důrazem na maximální spolehlivost a výkonnost, cena tohoto serveru přibližně odpovídala pětinásobku ceny jednoho webserveru. Předpokládal jsem, že tento stroj dokáže zvládnout zátěž od pěti až osmi webserverů. Od počátku bylo jasné, že i tato část systému časem narazí na výkonnostní strop a bude nutné řešit její škálovatelnost. Ve fázi, ve které se projekt nacházel, však nemělo význam o takovém řešení uvažovat.

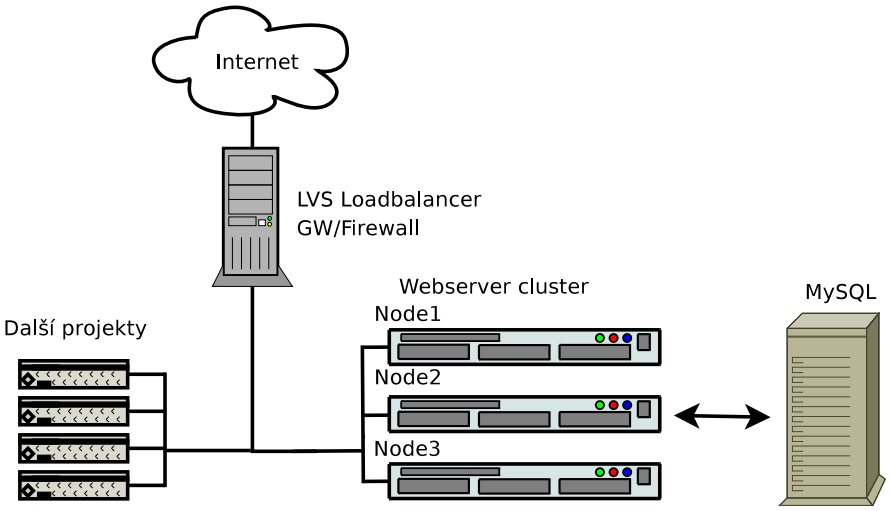
Zásadním rozhodnutím byla volba technologie pro rozkládání zátěže mezi webservery. Řešení na bázi DNS bylo zavrženo hned na začátku, hned z několika důvodů. DNS round robin neposkytuje prakticky žádnou kontrolu nad rozložením zátěže mezi jednotlivé servery. Statisticky bychom sice měli dosahovat čistě poměrného rozložení klientů na jednotlivé servery, v praxi však mohou nastat situace, kdy se tak neděje a nemáme žádný nástroj, kterým bychom si požadované chování vynutili. Příčinou takovýchto negativních projevů mohou být například špatně nakonfigurované cachovací DNS/HTTP-proxy servery velikých providerů. Kromě nemožnosti zcela zaručit poměrné rozložení zátěže na jednotlivé servery je zde prakticky nulová možnost ovlivnit rozložení zátěže na základě výkonnosti, nebo momentálního zatížení jednotlivých serverů. Jako další nepominutelné negativum tohoto řešení je minimální možnost řešení výpadků jednotlivých serverů.

Další možností jak řešit rozkládání zátěže, bylo pomocí reverzní HTTP-proxy. Pro naše účely se nejlépe hodil Pound[1]. Uspokojivě řeší rozkládání zátěže mezi servery, zároveň řeší i dostupnost jednotlivých backend serverů. Kromě těchto nutných vlastností by toto řešení přinášelo poměrně mocný nástroj pro filtraci dotazů, řešení persistence na úrovni HTTP protokolu a SSL offloading. Při zběžném měření propustnosti se ukázala poměrně slabá škálovatelnost tohoto řešení. Loadbalancer, který hardwarovou konfigurací odpovídal konfiguraci backend webserverů, zvládal zpracovávat provoz přibližně pro 3 až 4 backend servery. Takové řešení by pouze odsunulo nutnost hledat jiné řešení, kde by místo problému loadbalancingu webserverů nastal problém s loadbalancingem proxyserverů. Nezanedbatelná je také režie, kterou toto řešení přináší, ta se dá ospravedlnit přidanými funkcemi nad HTTP protokolem. V našem případě však neexistoval explicitní požadavek některé z těchto funkcí, proto lze tuto režii označit jako zápornou vlastnost.

Konečná volba padla na Linux Virtual Server. Jako Layer-4 switch implementovaný v rámci Linuxového jádra nabízí díky svým minimálním výpočetním a paměťovým nárokům maximální možnou propustnost, kterou je možné pomocí softwarového řešení dosáhnout (na tomto operačním systému). To, že tento loadbalancer pracuje na 4. vrstvě, představuje zároveň klady i zápory. Mezi klady patří především vysoká propustnost, nízké výpočetní a paměťové nároky, možnost použití pro jakoukoliv službu v rámci TCP a UDP protokolů. Hlavní nevýhodou je především tato univerzálnost, ze které vyplývá absence jakýchkoliv funkcí na úrovni jednotlivých L7 protokolů. Vzhledem k minimálním hardwarovým nárokům tohoto řešení byl LVS loadbalancer nasazen na bráně/firewallu, která kromě webserverové farmy propojuje s internetem ještě několik nezávislých projektů. Protože LVS se nachází na výchozí bráně webserverů, bylo nutné použít u LVS metodu NAT viz. Obr.4.1. Pro správu

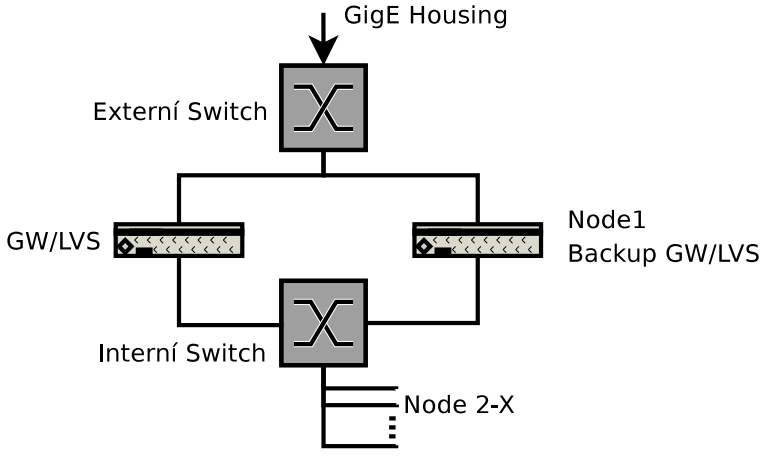
[1] http://www.apsis.ch/pound/

konfigurace LVS a řešení dostupnosti jednotlivých backend serverů byl použit Ldirector. Tento daemon je součástí projektu Linux-HA, přidává do něj podporu právě pro Linux Virtual Server spolu s mechanismy kontrol backend serverů a jejich dynamickou správu v rámci clusteru.



Obrázek 6.1: Architektura realizace clusteru.

Pro prvotní nasazení nebylo, z důvodu zvýšených finančních nákladů a prodloužení doby implementace, uvažováno o failover řešení samotného loadbalanceru. Zároveň ale bylo nutné zajistit efektivní náhradu tohoto stroje pokud, by došlo k jeho poruše. Rozhodl jsem se k tomuto účelu využít jeden z uzlů farmy. Oba stroje jsou do sítě zapojeny totožným způsobem (viz. Obr.6.2), čímž je eliminována nutnost fyzické manipulace se zapojením při náhradě hlavní GW/LVS. Synchronizace náhradního systému s hlavním



Obrázek 6.2: Zapojení záložního loadbalanceru

je prováďěna jednou denně pomocí dump/restore na vyhrazený diskový oddíl. Nahrazení funkce hlavní GW/LVS náhradním uzlem tedy spočívá v restartování záložního uzlu s root filesystémem nasměrovaným právě na tento vyhrazený diskový oddíl. Dodnes nebylo nutné tohoto opatření využít nouzově, bylo však úspěšně využito při stěhování farmy do nového

racku k bezvýpadkovému přepnutí.

Data, nad kterými farma operuje nejsou příliš často měněna a webservery samotné z lokálního filesystému pouze čtou. Díky tomu nebylo nutné implementovat pro synchronizaci a distribuci dat nějaké složité řešení. Tato distribuce je tak řešena pomocí automatické synchronizace FTP prostoru přístupného vývojářům projektu s filesystémy jednotlivých uzlů farmy. Samotná synchronizace je prováděna pomocí rsync.

Podobným způsobem jako data jsou synchronizovány i konfigurace a software, který nepochází ze zdrojů distribuce (vlastní skripty, Apache, PHP atd.). Pro účely hromadné správy uzlů je používán wraper nad ssh a několik skriptů, které tento wraper využívají. Typický administrátorský úkon se skládá z provedení konkrétní akce na jednom z uzlů a následném spuštění skriptu který změny zpropaguje na všechny ostatní uzly. Jedním z problémů, který znemožňoval hromadnou administraci uzlů, byla konfigurace virtualhostů Apache. Vzhledem k tomu, že LVS je použito v módu NAT, je nutné Apache navázat na rozhraní, které má na každém z uzlů jinou IP adresu (192.168.1.X). Konfigurace virtualhostů, ve kterých se tato IP adresa vyskytuje, je tímto pro každý uzel unikátní. Řešením bylo využití m4 makroprocesoru. Je tak možné nahradit deklaraci IP adresy proměnnou, kterou na každém uzlu lokálně m4 nahradí konkrétní IP adresou.

Další částí systému, kterou bylo třeba přizpůsobit clustrovému prostředí, je logování, a to jednak samotného systému, a zadruhé logování Apache. Pro obě tyto části bylo zpočátku úspěšně použito klasické řešení se vzdáleným logováním na centrální logovací server pomocí syslogu. V případě logování dotazů Apache však toto řešení nedokázalo překročit hranici 2000 dotazů za vteřinu, a vše nad tuto hranici bylo tiše zahozeno. První směr, kterým jsem se vydal při řešení tohoto problému, bylo hledání jiné, výkonnější implementace syslogu. Výsledkem byl však překvapivý závěr, že 2000 zalogovaných zpráv za vteřinu je poměrně slušný výsledek, a ačkoliv je možné najít o něco výkonější implementaci, její výkonnostní strop bude maximálně o desítky procent vyšší. V tomto případě ale potřebujeme řešení s propustností o stovky procent vyšší. Ke slovu tak přišla tvorba vlastní jednoúčelové klient-server logovací aplikace. V současné době ve špičkách touto logovací aplikací protéká přibližně 15 Mbit/s což odpovídá 6000 zalogovaných zpráv za vteřinu. Tento provoz na logovacím serveru generuje oproti syslogu prakticky neznatelné zatížení.

6.3 Současný stav

V současné době se farma skládá z 11 webserverů a jednoho MySQL serveru.

- Ve špičkách farma vyřídí:
  - 6 000 dotazů za vteřinu
  - provoz dosahuje 220 Mbit/s
  - loadbalancer zpracuje 80 000 paketů za vteřinu
  - MySQL vyřídí 3 000 dotazů za vteřinu což odpovídá datovému provozu okolo 50 Mbit/s
- za jeden den Apache zaloguje přez 30 GB dat
- za jeden den farma vygeneruje provoz o objemu až 1 TB
- za jeden měsíc farma vygeneruje provoz o objemu přiližně 15 TB

Přibližně každého půl roku se provoz zdvojnásobuje, a zatím nic nenasvědčuje tomu, že by se měl tento trend měnit. Do dnešního dne škálování tohoto projektu představovalo v zásadě jen zvýšení počtu webserverů. Začínají se ale objevovat signály, které naznačují blížící se hranice stávajícího řešení. Zároveň s růstem projektu se mění úhel pohledu a kriteria hodnocení jednotlivých částí systému.

Prvním signálem, který naznačil výkonostní limity LVS loadbalanceru, bylo překročení 200 Mbit/s, kdy se začalo projevovat přetížení systému. Při zkoumání příčin přetížení jsem zjistil nevhodné automatické nastavení obsluhy přerušení síťových karet, kdy přerušení od obou obsluhoval pouze jeden procesor. Po nastavení afinity přerušení tak, aby každá síťová karta měla vyhrazen vlastní procesor, se výkonostní strop zvedl. Můj osobní odhad je, že při tomto nastavení farma zvládne 9-11 tisíc dotazů za vteřinu.

Díky skvělé práci vývojářů projektu a jejich nekonečným optimalizacím bylo možné dodnes projekt provozovat pouze na jednom MySQL serveru. Ten je však již na hranici svých možností a v současné době se již pracuje na implementaci Mysql replikace. Tato metoda byla zcela jasnou volbou, více než 95% dotazů je čtecích, MySQL replikace je prověřené a pro škálování čtecích operací velmi výkonné řešení.

S nárůstem počtu uzlů začíná být distribuce dat pomocí rsync poměrně neohrabaná a pomalá.

S růstem projektu začíná nabírat na významu řešení vysoké dostupnosti na všech úrovních systému.

Kapitola 7

Možnosti dalšího rozvoje projektu

V této kapitole se budu zabývat možnostmi dalšího rozvoje projektu, především z pohledu odstranění stávajících výkonnostních omezení, zlepšení funkčních vlastností a zajištění dlouhodobé škálovatelnosti. Uvedu zde myšlenky, které jsou již ve stádiu implementace, zároveň nastíním možnosti, které jsou prozatím jen ve formě návrhu řešení. Celkové schéma všech navrhovaných změn a vylepšení je v příloze 2.

7.1 Linux Virtual Server

Klíčovou technologií která v současné době zajišťuje škálovatelnost je LVS loadbalancer který rozděluje požadavky mezi jednotlivé webservery. Současné řešení má stále výkonnostní rezervy, avšak při zachování stávajícího růstu projektu je možné očekávat dosažení těchto limitů během maximálně jednoho roku. Je tedy namístě uvažovat o úpravách a změnách stávajícího řešení.

Omezení vyplývají především z rozhodnutí nasadit LVS loadbalancer na gateway, další omezující faktor je, že tato gateway je společná ještě pro další projekty. V době návrhu původního řešení byla tato varianta výhodná především z pohledu efektivního využití infrastruktury a celkového zjednodušení. V současné době již oddělení infrastruktury webserverové farmy od ostatních projektů nepředstavuje zbytečné plýtvání prostředky a z hlediska budoucí škálovatelnosti je nevyhnutelné.

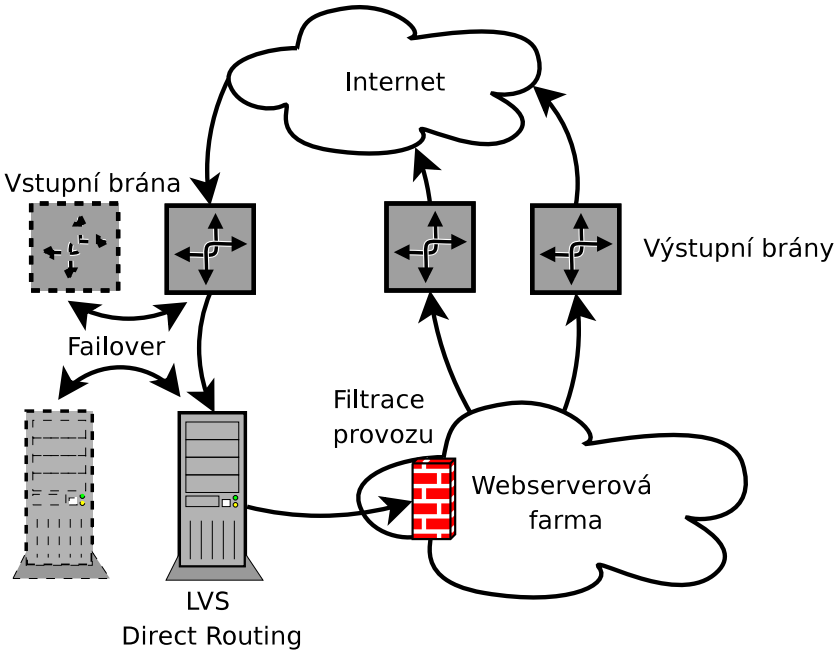
Zdroje úzkých míst s propustnosti LVS jsou především tuto dva:

- LVS je vzhledem k umístění na bráně použit v módu NAT, čímž veškerý provoz, jak příchozí tak odchozí, musí projít přes loadbalancer. Teoreticky tak vzniká hranice 1 Gbit/s kterou tímto řešením nelze překročit.
- Hranice 1 Gbit/s, je pouze hypotetická a v současném řešení nedosažitelná. Tato hranice je dále snižována tím, že gateway provádí filtraci provozu a connection tracking pro ostatní projekty. Obrázek o konkrétních dopadech kombinace těchto dvou technologií si lze udělat z dat naměřených a prezentovaných v další kapitole *Testování propustnosti Linux Virtual Server*. Hlavním důvodem je duplicita connection trackingu LVS a Netfilteru, přičemž tato funkce je v LVS implementována mnohem efektivněji.

Nové řešení by tato úzká místa mělo řešit. Evidentně je nutné oddělit infrastrukturu farmy od ostatních projektů. Zároveň je třeba oddělit funkce brány, LVS loadbalanceru a filtrace provozu. Nabízí se tedy použití LVS v módu Direct Routing (viz. Obr.4.3), kde loadbalancer

zpracovává pouze příchozí povoz. Odchozí provoz z webserverů je odesílán přímo bez účasti loadbalanceru pomocí brány, která může být pro každý z webserverů jiná. Příchozí provoz je tak stále limitován teoretickou hranicí 1 Gbit, příchozí provoz se skládá především z HTTP dotazů, které jsou datově poměrně malé, a tato hranice tedy nepředstavuje problém. Odchozí provoz je možné škálovat zvyšováním počtu bran, přes které webservery vracejí své odpovědi. Filtraci provozu je možné přesunout přímo až na webservery a eliminovat tak úzké místo na bráně vstupního provozu.

Zároveň je vhodné uvažovat o failover řešeních u vstupní brány a loadbalanceru, dostupnost výstupních bran je možné řešit přímo na webserverech.



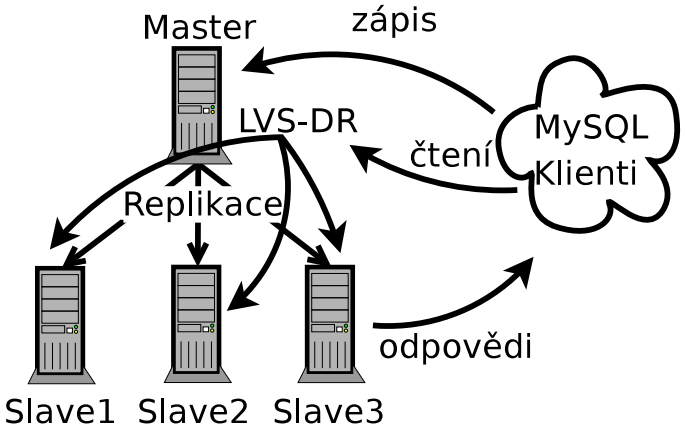
Obrázek 7.1: Návrh řešení LVS

7.2 MySQL replikace

MySQL replikace přináší poměrně jednoduché a výkonné řešení škálování výkonu čtecích operací , je zde ale několik bodů, u kterých je dobré se pozastavit.

Pro rozdělení čtecích a zápisových operací je vzásadě možné použít dvě metody. První z nich je využití některého z proxy řešení, jako je SQL Relay, nebo MySQL Proxy, ta však mají jistá omezení. Především se nasazením takovéto technologie zvyšuje složitost celkového řešení, nezanedbatelné jsou také hardwarové nároky na jejich provoz. Zároveň se toto řešení může samo o sobě stát úzkým hrdlem, nehledě na nutnost řešení redundance. Dle mého názoru, pokud máme tu možnost, tak je vždy lepším řešením zahrnout logiku rozdělování čtecích a zápisových operací přímo do aplikace, která databázi využívá. Co si ovšem v aplikaci můžeme ušetřit, je rozkládání zátěže mezi jednotlivé repliky a detekce jejich dostupnosti. V tomto ohledu se lze s výhodou spolehnout na LVS, a některou z jeho nadstaveb, jako je Ldirector. Nasazení LVS zde nutně nemusí znamenat dodatečné hardwarové nároky, je možné si například představit, že Master replika bude zároveň poskytovat službu LVS pro své Slave repliky. Výkonnostní dopad na Master repliku by při omezeném

počtu Slave replik byl, dle mého názoru, zanedbatelný.



Obrázek 7.2: MySQL replikace, LVS rozkládání čtecích operací.

7.3 Akcelerace HTTP

Při prvotním návrhu bylo uvažováno o použití reverzní HTTP proxy pro rozkládání zátěže, v té době ovšem nebylo k dispozici řešení, které by funkčně, případně výkonnostně dostačovalo požadavkům zadání. V současnosti však začíná být tato technologie opět zajímavá, nikoliv z důvodu rozkládání zátěže, ale pro její další vlastnosti.

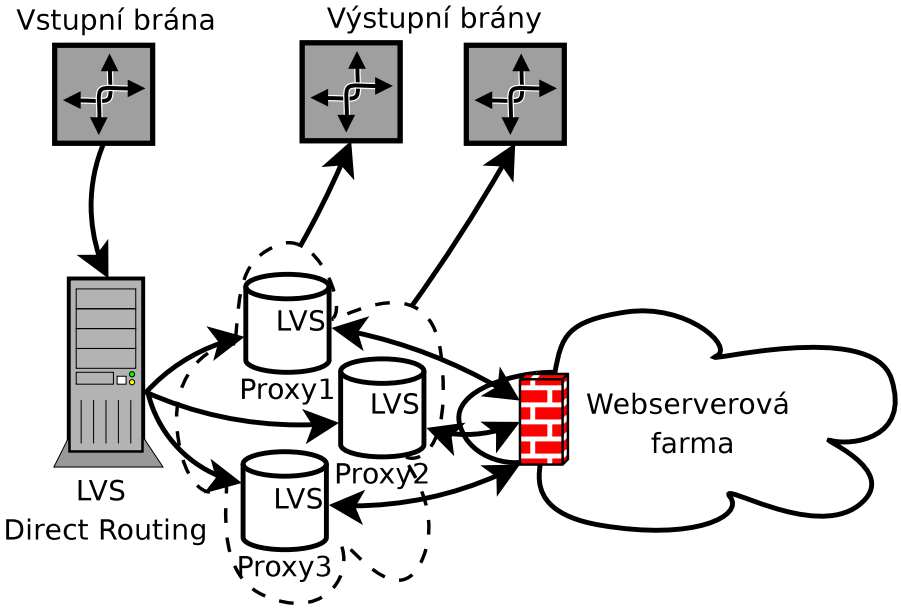
Především ve výkonnostních špičkách se projevuje vliv pomalých klientů, kdy má Apache blokované procesy do doby než klient odebere veškerá data. Negativně se toto může projevit tak, že Apache má spuštěno maximální počet procesů, a přesto není procesor vytížen na 100%. Neefektivně vysoký počet spuštěných, a zároveň aktivních procesů, představuje mnoho přepínání mezi kontexty, čímž je dále snižována celková výkonnost systému. Obecně je toto jeden z hlavních důvodů pro nasazování reverzních HTTP proxy.

Jednou z metod pro docílení vysoké propustnosti webové aplikace představuje cachování. Zpravidla se uplatňuje na všech vrstvách aplikace a souvisejících technologiích. Reverzní proxy zde představuje řešení pro cachování HTTP dotazů. V našem případě se od HTTP cachování dají očekávat značné přínosy i při minimálním TTL cachovaných objektů (TTL výsledkových tabulek se může pohybovat maximálně v jednotkách vteřin).

Pro cachovací reverzní proxy máme na výběr prakticky jen dvě možnosti Squid a Varnish. Především výkonnostní předpoklady mě vedou k preferenci projektu Varnish. Ani jeden z těchto projektů nedisponuje v současné době uspokojivým řešením rozkládání zátěže mezi backend servery. Podobně, jako u MySQL replikace, je možné k tomuto účelu s výhodou využít LVS. Podobně, jako u MySQL, je možné i zde zkombinovat reverzní proxy a LVS (viz. Obr.7.3). Zátěž LVS zde již nebude nijak extrémní, jednak z celkového počtu dotazů bude daný stroj zpracovávat jen poměrnou část odpovídající rozdělení mezi proxy servery. Tento počet dotazů je dále snížen o dotazy zodpovězené proxy serverem z cache.

Kromě předpokládaných výkonnostních přínosů jsou zde také negativa. Patrně hlavním negativem je značné zvýšení složitosti řešení. A to jak z hlediska infrastruktury, tak z hlediska vývoje aplikace. Dalším problémem je logování. Je třeba zajistit kompatibilitu logu proxyserverů se stávajícím systémem zpracování statistik, ty jsou pro komerční projekt

hlavním nástrojem pro prodej reklamy, která ho živí. Jednou z funkcí kterou nám Varnish znemožní, je persistence, v našem případě sice není potřeba, je ovšem nutné tento fakt mít na paměti.



Obrázek 7.3: Akcelerace HTTP; kombinace proxy a LVS

7.4 Nasazení distribuovaného filesystému pro sdílení dat

Distribuce a synchronizace dat poskytovaných webservery je v současné době řešena pomocí *rsync*. Kromě poměrně jednoduchého a přímočarého nasazení přináší tento způsob výhody ve zvýšené redundanci dat a nesnižujícím se výkonu s rostoucím počtem webserverů. Zároveň ale s přibývajícím počtem webserverů nabývají na významu nevýhody tohoto řešení. Především je tu nutnost explicitního řízení synchronizace. Synchronizace samotná způsobuje prodlevy mezi konzistentními stavy všech kopií. Se zvyšujícím se počtem webserverů se tyto prodlevy prodlužují, při synchronizaci většího množství dat se může takováto prodleva pohybovat i v řádech minut. Zároveň je zde problém s rozšiřováním kapacity stávajících serverů.

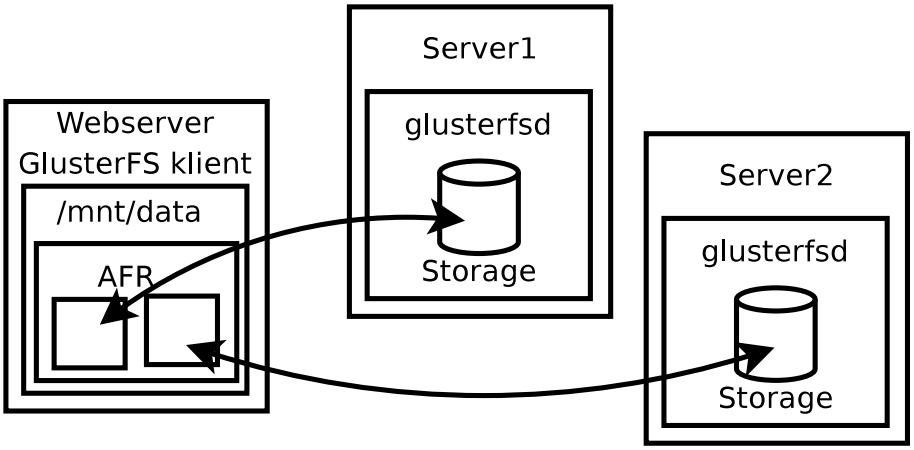
Pro data, která je nutné mezi webservery sdílet, jako jsou například PHP-sessions, log soubory aplikace atp., je dnes využíváno NFS. To však vnáší do systému „Single Point Of Failure“zároveň výkonnostní škálovatelnost NFS je přinejmenším problematická.

Donedávna byla odpovědí na tyto problémy buď řešení extrémně drahá nebo řešení, která se nedokáží s těmito problémy vypořádat bezezbytku. S projektem GlusterFS[1] se zde objevil distribuovaný filesystém s vlastnostmi, které dokáží na tyto problémy odpovědět. Zároveň jeho použití je natolik jednoduché a nekomplikované, že příprava a realizace nasazení nepředstavuje časově náročnou operaci jak je tomu zpravidla u jiných distribuovaných filesystémů.

V našem případě GlusterFS přináší výhody především svou snadnou škálovatelností a vestavěnou datovou i failover redundancí. Vzhledem k nízkému počtu zápisových operací,

[1] http://www.gluster.org/docs/index.php/GlusterFS

které je nutné tímto zabezpečit je zřejmě nejvhodnější způsob implementace pomocí AFR translátoru na klientské straně GlusterFS. Čtecí operace je možné rozložit mezi jednotlivé servery čímž je zaručena výkonnostní škálovatelnost. Zápisové operace klient provádí x-krát dle počtu požadovaných replik, nízký počet těchto operací tak nepředstavuje zvláštní výkonnostní požadavky. Pokud by se v budoucnu situace s počtem zápisových operací změnila, je možné i toto řešit rozšířením o další sadu replik, nad kterou použitím Unify translátoru rozložíme ukládaná data, a tím i zátěž. Výpadky jednotlivých serverů a synchronizaci dat po obnově funkce zde transparentně zajistí AFR translátor.



Obrázek 7.4: Využití GlusterFS na webserverové farmě

Kapitola 8

Testování klíčových technologií pro rozvoj projektu

V této kapitole se budu věnovat experimentům prováděným za účelem zjištění výkonnostních parametrů a stavu současné implementace vybraných technologií. Vybrány byly ty, u kterých se plánuje nasazení, nebo u nich je třeba provést změny pro zajištění udržitelnosti rozvoje projektu. Data z provedených testů představují chování těchto technologií na konkrétním hardware v určitém konkrétním zapojení za laboratorních podmínek. Získané výsledky tedy nelze brát jako absolutní a určující pro jakékoliv jiné prostředí,ale poslouží spíše k osvětlení chování a závislostí daných technologií.

8.1 Linux Virtual Server

Linux Virtual server je pro tento projekt zcela klíčovou technologií. Proto je nutné mít alespoň přibližnou představu o výkonnostních limitech jeho jednotlivých typů na dnes používaném hardware a o chování systému pod extrémní zátěží. Zároveň je třeba získat přehled o vlivu dalších komponent OS typicky kombinovaných s touto technologií. Na webových stránkách projektu LVS byly uveřejněny testy podobného zaměření, ale díky jejich stáří(většina provedena okolo roku 2000/2001) je jejich vypovídací hodnota pro dnešní případy prakticky nulová.

Při přípravě měření bylo třeba se rozhodnout jakým způsobem testovat. Zda použít přístup „syntetického testu“ pro cílené otestování konkrétní vlastnosti systému, nebo zda si vybrat některou z typicky používaných síťových služeb a jejím provozem se snažit simulovat různé druhy zátěže. Zvolil jsem první z těchto variant a to z několika důvodů. Především při testování pomocí konkrétní služby je přinejmenším složité otestovat určitou vlastnost LVS, lze však dobře zmapovat chování této konkrétní služby ve spojení s LVS. Takové měření však ztrácí smysl pro posouzení chování jakékoliv jiné služby a právě to by mělo být cílem těchto měření - získat přehled o chování a vlastnostech LVS v takovém rozsahu abychom byli schopni alespoň přibližně odhadnout výkonnost při konkrétním nasazení. Dalším důvodem, proč jsem se rozhodl netestovat pomocí konkrétní služby, je extrémní náročnost na výpočetní výkon a infrastrukturu potřebnou k takovému testování.

Primárním testovaným parametrem LVS je propustnost paketů v závislosti na dalších veličinách. Jako hlavní testovací nástroj jsem použil *Testlvs*.[1] Jde o aplikaci určenou přímo pro testování propustnosit LVS. Její funkce spočívá v generování SYN paketů s cílovou

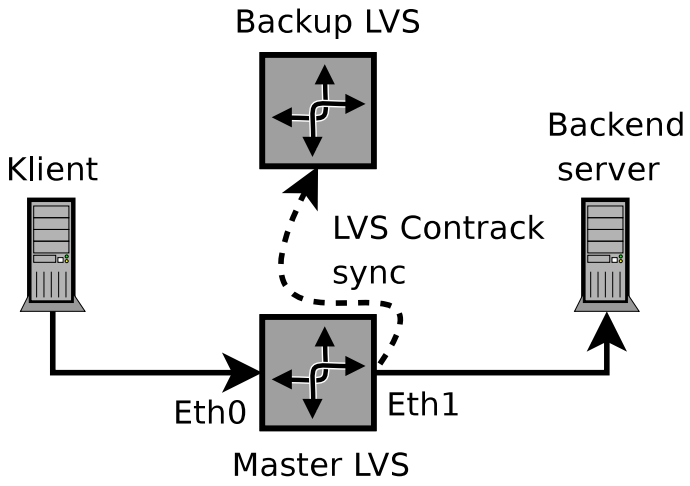
[1] http://www.ssi.bg/~ja/#testlvs

adresou a portem nastavenou na virtuální službu poskytovanou LVS. Pro testování lze nastavit typ protokolu (TCP,UDP), velikost paketu, počet zdrojových adres které aplikace v paketech podvrhne atd.. Testováno tedy bylo:

- Závislost propustnosti paketů na jejich velikosti.
- Vliv počtu klientů (zdrojových IP adres).
- Vliv synchronizace stavů spojení s Backup LVS.
- Propustnost paketů jednotlivých typů LVS (Direct Routing; IP Tunneling; NAT).
- Vliv Netfilteru a Connection trackingu na propustnost paketů.

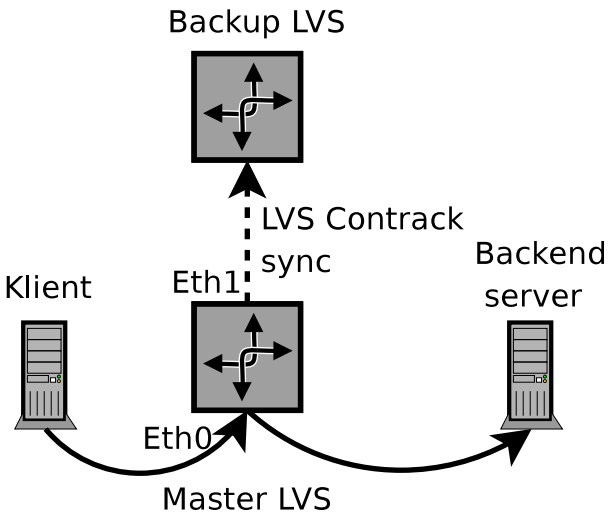
Testování probíhalo na 4 strojích se shodným hardware: CPU C2D 2,6GHz; 2GB RAM; on-board GigE síťová karta Intel (ICH9)(podrobněji viz. Příloha 1) vše propojeno 8 portovým GigE switchem. Operační systém Debian GNU/Linux 4.0; jádro 2.6.23 modulární, vlastní kompilace; ovladače síťové karty e1000[2] ve verzi 2.6.12, modul použit bez jakýchkoliv parametrů.

Původním záměrem bylo použít 2 síťové karty ve stroji, který realizoval funkci LVS, aby se zapojení co možná nejvíce blížilo situaci kdy je LVS nasazen na bráně (viz. Obr.8.1). Po zapojení jsem provedl úvodní testy a zjistil, že druhá síťová karta (PCI 32bit,33MHz) není schopna přenášet více než přibližně 60MB/s. Pokud jsem generoval provoz na on-board síťové kartě dosáhl jsem bez problémů na plnou rychlost GigE linky. Ve chvíli, kdy jsem generoval provoz na obou síťových kartách, jejich propustnost spadla přibližně na polovinu. Po přenastavení maximální četnosti přerušení přídavné síťové karty se sice propustnost lehce zvedla ale pro další testování jsem se rozhodl tuto kartu nepoužít. Výsledky by byly jejím výkonem zkresleny. Zapojení jsem proto přeuspořádal tak, aby LVS využívalo pouze síťovou kartu na základní desce (viz. Obr.8.2).



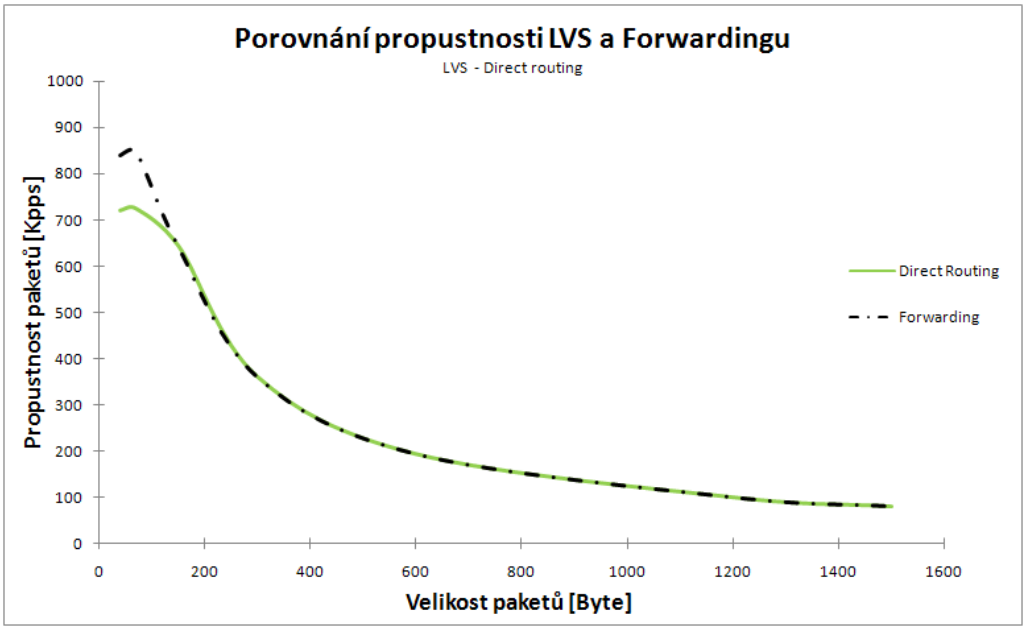
Obrázek 8.1: Plánované zapojení

[2]http://sourceforge.net/projects/e1000/



Obrázek 8.2: Skutečné zapojení

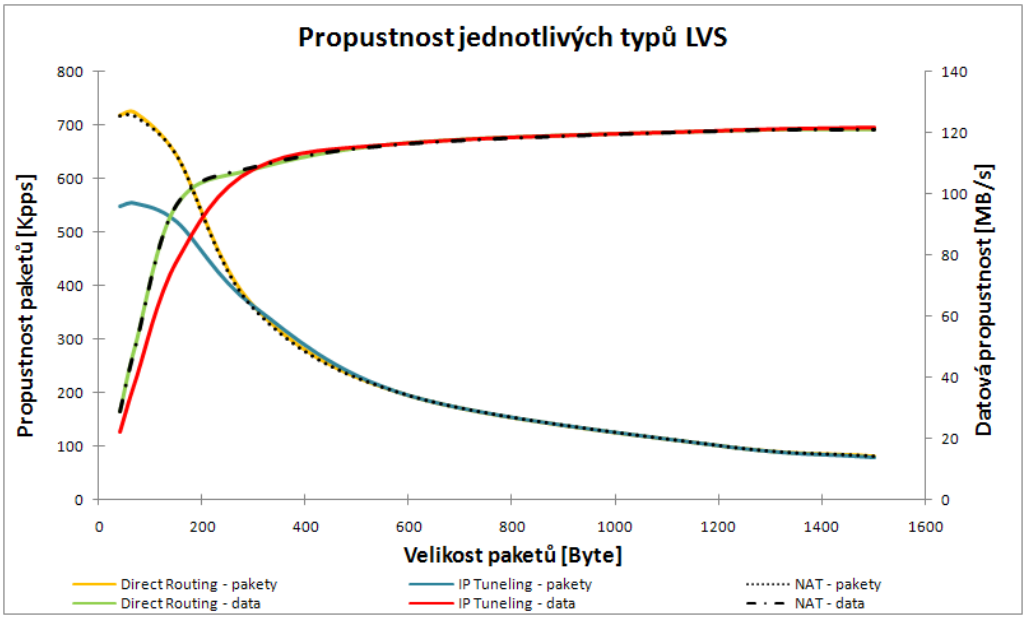
Jako první jsem měřil čistou propustnost IP Forwardingu. Především jde o referenční data, vůči kterým lze následně vyhodnocovat měření provedená na LVS. Při tomto měření byly odstraněny všechny moduly související s LVS i Netfilter, to proto abychom získali data pokud možno neovlivněná technologiemi, které následně budeme testovat. V předchozí větě jsem se nevyjádřil přesně, moduly nebyly „odstraněny“, ale po rebootu nebyly vůbec zavedeny. Jak bude předvedeno u poslední skupiny testů, i toto má svůj význam a výsledky jsou tímto ovlivněny.



Obrázek 8.3: Porovnání propustnosti Forwadingu a LVS - DR

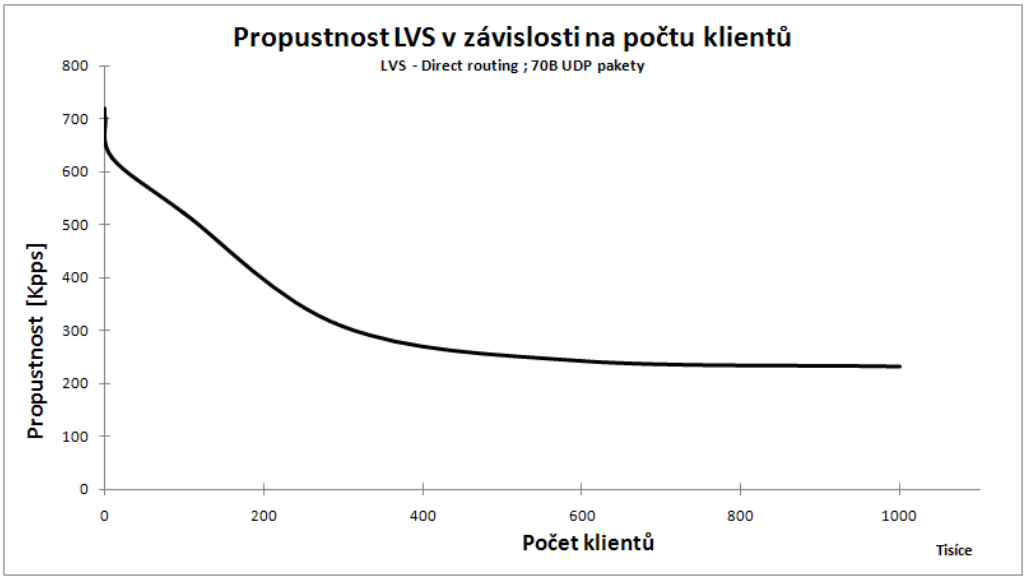
Na prvním grafu(Obr.8.3) je znázorněno srovnání propustnosti IP Forwardingu s LVS v módu Direct Routing pro různé velikosti paketů. Rozdíl v propustnosti pro malé pakety

(okolo 70B) se pohybuje mezi 10 a 20 procenty. Tento rozdíl představuje režii kterou LVS-DR přidává ke zpracování forwardovaných paketů. Z grafu je patrné, že již při velikosti paketů okolo 150B přestáváme být omezeni výpočetním výkonem, ale začíná se projevovat blížící se hranice datové propustnosti síťové linky. Proto se při testování dalších vlastností a závislostí pohybujeme právě v oblasti paketů o velikosti 70B, tím by mělo být zaručeno že testujeme chování LVS a ne přímo hardwaru na kterém běží.



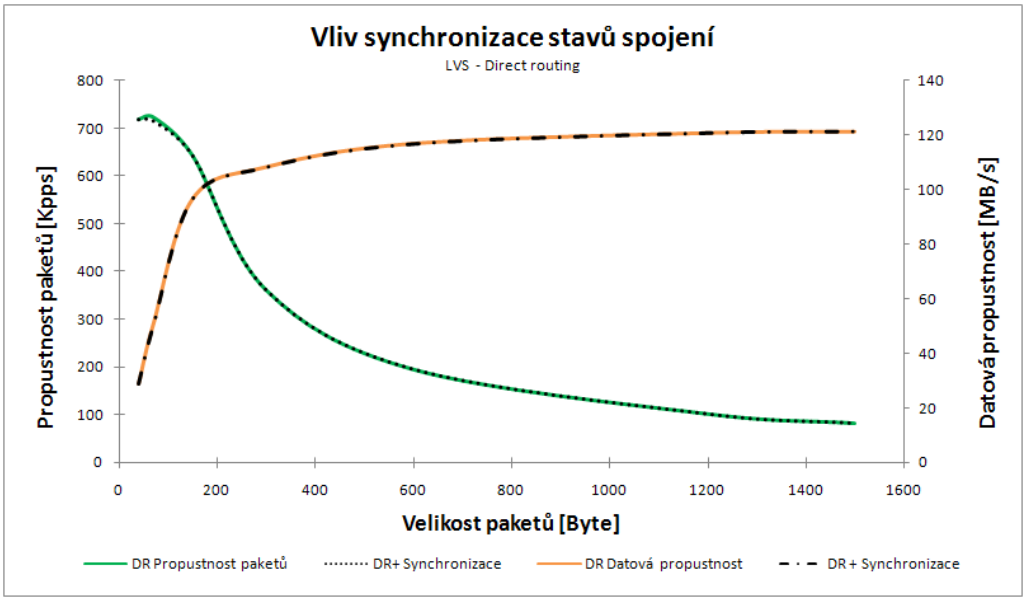
Obrázek 8.4: Porovnání propustnosti jednotlivých typů LVS

Na dalším grafu(Obr.8.4) se dostáváme k jednomu z hlavních záměrů tohoto testování a tím je proměření výkonnosti jednotlivých typů LVS. Graf obsahuje na jednu stranu data, která se dala předpokládat a na stranu druhou obsahuje data poměrně překvapivá. Očekávanými výsledky mám na mysli výkonnostní propad IP Tunelingu, kde je jasný zdroj další režie - enkapsulace IP datagramů. Překvapivé pro mne bylo zjištění, že Direct Routing dosahuje totožných výsledků jako NAT. Graf by tímto navozoval pocit, že tyto dva módy LVS jsou zcela srovnatelné a je možné od nich v reálném nasazení očekávat srovnatelné výsledky. Je zde ovšem několik *ale*. Prvním z nich je to, že u NAT musí být loadbalancer zároveň vstupní i výstupní bránou backend serverů, tím propustnost padá na polovinu. Typicky je příchozí provoz násobně menší než provoz odchozí, může tak lehce dojít k situaci, kdy odchozí linka loadbalanceru je již plně saturována, přitom ale příchozí provoz pro LVS neznamená výkonnostní maximum. Další *ale* vyplývá ze spojení loadbalancer - brána. Tato kombinace zpravidla navíc zahrnuje filtrování provozu, což jak bude prezentováno u poslední skupiny testů, představuje další snížení propustnosti LVS.



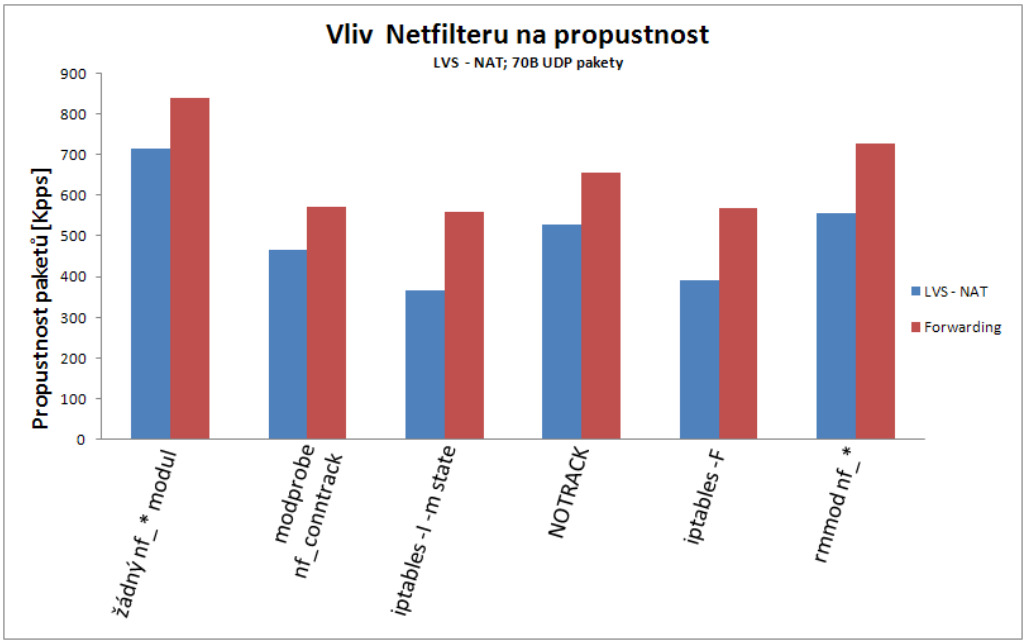
Obrázek 8.5: Závislost propustnosti LVS na počtu klientů

Dalším testem bylo určení vlivu počtu klientů (unikátních IP) na propustnost LVS. Zde se dal opět předpokládat pokles propustnosti s rostoucím počtem klientů. Osobně mě však překvapilo jak strmý tento propad, především v první polovině křivky,je. Předpokládám že tento propad souvisí s velikostí hash tabulky spojení a velikostí cache procesoru, respektive s klesající hitrate této cache při zvětšující se tabulce spojení. Zároveň předpokládám, že v reálném nasazení propad nebude tak dramatický. V reálném provozu nemá každý paket zcela náhodnou zdrojovou adresu, tak jak tomu bylo v tomto testu. Pakety z jedné adresy zpravidla přicházejí v určitých kvantech, čímž se hitrate cache procesoru opět zvyšuje.



Obrázek 8.6: Vliv synchronizace stavů spojení na propustnost LVS

Tento graf představuje vliv synchronizace stavů spojení na propustnost LVS. Opět je zde lehce překvapivý závěr a to, že vliv tato funkce nemá žádný. Rozdíly v naměřených datech jsou pouze v rámci chyby měření. Zapojení v této konfiguraci jsem využil alespoň k ověření funkčnosti této vlastnosti při failoveru serveru. Zde se ovšem žádné překvapení nekonalo a vše fungovalo jak mělo. Po převzetí IP konfigurace klienti pokračovali ve spojeních navázaných před failoverem.



Obrázek 8.7: Vliv Netfilteru na propustnost VLS

Poslední skupina testů si klade za cíl ozřejmit vliv Netfilteru na propustnost LVS, nebo routeru bez LVS. Z Netfilteru jsem se především zaměřil na connection tracking, který je jeho součástí a do jisté míry duplikuje činost LVS. Měření probíhalo následujícím způsobem.

Postup byl shodný pro měření LVS i samotného IP Forwardingu. Po rebootu stroje byly postupně prováděny následující kroky, před každým z nich byla změřena propustnost.

1. Po rebootu byla změřena propustnost systému bez zavedených modulů Netfilteru, případně i LVS.
2. Dalším krokem bylo zavedení modulu nf_conntrack.
3. Vytvoření pravidla v iptables využívajícího connection tracking. Konkrétně: `iptables -I FORWARD -m state --state NEW -j ACCEPT`
4. Zavedení pravidla, které vyloučí pakety patřící testu propustnosti z následného zpracování Netfilterem: `iptables -t raw -I PREROUTING -s 10.0.0.0/8 -j NOTRACK`
5. Vymazání všech vytvořených pravidel.
6. Odstranění všech modulů náležících netfilteru z jádra.

Propustnost se mění celkem dle předpokladů, propad po zavedení modulů, další propad po vytvoření pravidel, zvýšení propustnosti po vytoření „zkratky“pro pakety. Zajímavé ovšem je, že některé z těchto kroků mají nevratný charakter. Například propustnost v bodě 1. by měla odpovídat propustnosti v bodě 6., stejně tak i v bodech 2. a 5. by měla být propustnost shodná.

Z naměřených dat lze odvodit následující hypotézy:
Netfilter a connection tracking se o něco méně negativně projevuje na routeru bez LVS.
Pokud potřebujeme extrémně výkoný loadbalancer je nutné filtraci provozu provádět mimo něj.

Pokud na loadbalanceru filtraci provádět musíme, je vhodné zvážit nutnost použití connection trackingu. Zároveň je vhodné vyloučit provoz určený pro LVS ze zpracování Netfilterem.

8.2 Distribuované filesystémy - GlusterFS

Přínosy, které by představovalo nasazení distribuovaného filesystému, konkrétně GlusterFS, byly vysvětleny v minulé kapitole. GlusterFS odbourává tři základní problémy které se běžně vyskytují u ostatních distribuovaných filesystémů:

- Pro provoz serveru ani klientů není třeba upravovat linuxové jádro (podpora FUSE je v něm již poměrně dlouho).
- Aplikace využívající tento filesystém není třeba překompilovat se zvláštními knihovnami, či jakkoliv jinak upravovat.
- Tento filesystém je POSIXový.

Každá z těchto výhod představuje v nejednom případě pomyslnou hranici, která rozlišuje řešení kde převládají klady od řešení, které se nevyplatí nasazovat.

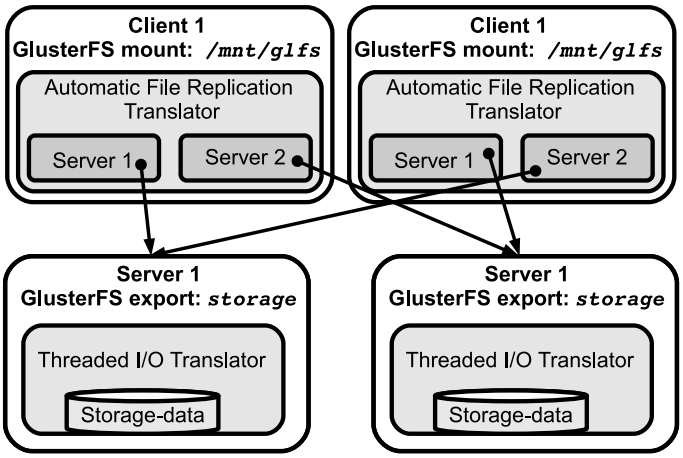
Na rozdíl od předchozího experimentu, zde není cílem prověřovat výkonnost či škálovatelnost tohoto filesystému. Tím se již zabývali jiní a jejich závěry lze nalézt na stránkách projektu.[3] Jako cíl tohoto experimentu jsem si stanovil posouzení reálné nasaditelnosti tohoto projektu do produkčního prostředí. Hodnotit budu od instalace přes konfiguraci a běh až po chování při výpadcích a jiných nestandardních událostech.

Testovací hardware je shodný s hardwarem použitým pro testování LVS, 2 GlusterFS servery a 2 jejich klienti. Použitý software byl následující: linuxové jádro 2.6.23; FUSE 2.7; GlusterFS 1.3.7.

Nejsnadnější částí zprovozňování GlusterFS je bezesporu instalace, která představuje klasický postup `./configure; make; make install;`, pro kompilaci není v systému třeba nic zvláštního (FUSE, flex, bison).Dalším krokem je konfigurace. Zde GlusterFS dává veliký prostor pro vlastní invenci, variabilita konfigurací je skutečně nezměrná. Já jsem se přidržel dvou konfigurací které jsou na webu projektu zveřejněny jako vzorové. Protože jeden z hlavních cílů experimentu bylo otestování chování při simulovaných výpadcích, byly obě konfigurace řešením replikace dat. Jedna na straně klienta (viz. Obr.8.8), druhá na straně serveru s vyhrazenou replikační sítí (viz. Obr.8.9). Zde bych se zastavil nad problémy s konfigurací. Při zprovozňování těchto dvou konfigurací jsem narazil na řadu problémů vyplývajících z poměrně nízkého věku projektu a jeho rychlého rozvoje. Například na několika místech v dokumentaci jsem narazil na nekonzistentní definice parametrů translátorů. Pravděpodobně nešlo o chyby, ale pouze o změny v syntaxi případně ve funkčnosti mezi jednotlivými verzemi GlusterFS. Bohužel dohledat správné definice pro konkrétní verzi není zcela jednoduché. Uvedu jeden konkrétní příklad. Pro využití AFR na straně klienta je na webu projektu popsán postup i s konkrétní konfigurací. V definici translátoru AFR však chybí parametr `option replicate *:2`. Tento parametr již nebude potřeba od verzí vyšších než 1.3.7, žádná taková verze však zatím nebyla vydána. Problém je v tom, že svazek s takovouto definicí se poměrně nečekaně korektně sestaví, je možné jej používat a data se replikují. Ve chvíli, kdy jeden ze serverů odpojíme, není již možné zařadit jej zpět a provést synchronizaci. Logování sice nemlčí ale na chybu konfigurace nic neukazuje a pro

[3] http://www.gluster.org/docs/index.php/GlusterFS#GlusterFS_Benchmarks

vyřešení problému je třeba značné dávky intuice. Po doplnění chybějícího parametru již vše fungovalo dle předpokladů.



Obrázek 8.8: GlusterFS AFR na straně klienta

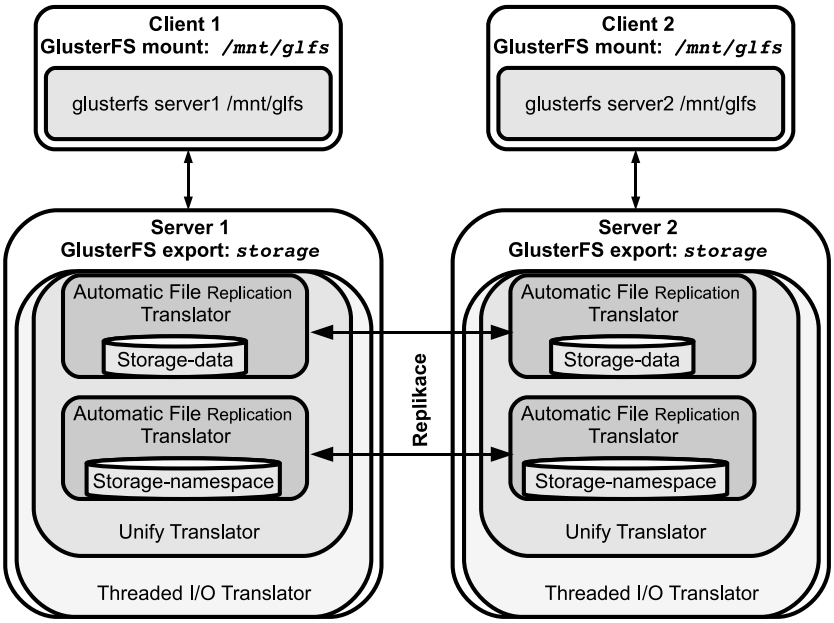
První testovaná konfigurace je zřejmě nejjednodušším možným použitím AFR. O distribuci a synchronizaci dat mezi replikami se stará sám klient. Všechny zápisové operace jsou paralelně zasílány na všechny zúčastněné servery. Při testování výpadků se systém choval dle očekávání, detekoval nedostupnost daného serveru a pokud zrovna prováděl čtecí operace z tohoto serveru, tak po vypršení spojení operaci dokončil ze serveru druhého. Doba tohoto timeoutu je přibližně 2 minuty (standardní doba vypršení TCP spojení), tuto hodnotu je možné přenastavit v client translátoru pomocí parametru `option transport-timeout X`. Pokud klient prováděl v době výpadku zápisové operace, tak je dokončil jen na dostupném serveru. Po opětovném zprovoznění serveru jsou data synchronizována, ne však automaticky ale při prvním přečtení. Synchronizaci serverů si lze vynutit například takto:

```
$find /mnt/glusterfs -type f -exec head -c 1 {}  \; >/dev/null
```

Automatická synchronizace je plánována od verze 1.4.

Druhá konfigurace vychází z tutoriálu[4], který se snaží řešit násobný provoz mezi klientem a servery při použití AFR na straně klienta. Základní myšlenkou je přenesení AFR na servery a provoz synchronizace po vyhrazené síti. Vyvstává zde ovšem problém s dostupností přípojného bodu (mount point), ve chvíli kdy spadne spojení se serverem ke kterému je klient připojen přestává připojený svazek fungovat až do doby, kdy se spojení obnoví. Tento problém je možné řešit pomocí DNS round robin, kdy klient po rozpadu spojení periodicky resolvuje doménové jméno serveru a pokouší se připojit až do chvíle, kdy se mu to podaří. Ve verzi 1.4 je plánován High Availability translátor, který bude toto řešit. Při testování chování při výpadcích jsem narazil na problém, který se mi nepodařilo vyřešit. Při zabití jednoho z procesů serveru se systém choval korektně, klient připojený na druhý server mohl dále pokračovat ve využívání svazku, po opětovném spuštění procesu serveru se obnovila jeho funkce a synchronizovala se změněná data. Pokud jsem ale jeden ze serverů odpojil od sítě, nebo vypnul, přestal fungovat i druhý server s chybovým hlášením o rozpadu řetězu translátorů. Dalším nemilým překvapením u této konfigurace byl

[4] http://www.gluster.org/docs/index.php/GlusterFS_High_Availability_Storage_with_GlusterFS



Obrázek 8.9: GlusterFS AFR na straně serveru

slabý výkon. Je možné, že použití Infiniband pro replikační síť by výkon zlepšilo, při použití gigabitového ethernetu však byly některé operace nad filesystémem prakticky nepoužitelné.

Po nejasnostech s konfigurací ve chvíli, kdy jsem již zaručeně provozoval správně nakonfigurovaný filesystém, jsem již nenarazil na žádný problém a systém se choval zcela stabilně. Výkon první z testovaných konfigurací byl poměrně dobrý, výkonnostní propad oproti výkonu lokálního filesystému byl srovnatelný s propadem například u NFS serveru. Překvapilo mě značné zatížení procesoru jak na serverech, tak na klientech. Celkově si myslím, že tento filesystém je možné provozovat v produkčním prostředí. Je však nutné věnovat dostatek času konfiguraci, testování a lazení výkonu před nasazením.

Kapitola 9

Závěr

Prvním z cílů této diplomové práce bylo zmapování a zhodnocení Open Source řešení z oblasti vysoké dostupnosti, rozkádání zátěže a škálovatelnosti. V rámci této části mé práce jsem se snažil o realistický pohled na konkrétní problémy z daných oblastí a o posouzení možností jejich řešení pomocí Open Source prostředků. Touto částí jsem si vytvořil teoretický a přehledový základ vědomostí nutných pro další části práce.

V další části jsem se zabýval hodnocením návrhu, vývoje a současného stavu webového projektu, pro který jsem zajišťoval návrh a realizaci technologií, které tento projekt využívá pro své fungování. Na tuto část mé práce jsem navázal návrhem dalšího rozvoje této infrastruktury, především se zaměřením na dlouhodobě udržitelnou škálovatelnost projektu a zajištění jeho bezvýpadkového provozu.

V poslední části diplomové práce jsem provedl rozbor podstatných vlastností technologií klíčových pro rozvoj tohoto projektu.

V rámci celé diplomové práce jsem se snažil o obecný náhled na konkrétní řešené problémy a to takovým způsobem, aby byl výsledek obecně uplatnitelný a využitelný.

Hlavní přínos této diplomové práce vidím v rozboru konkrétního případu s jasně stanovenými výkonnostními parametry a ve stanovení východisek jednotlivých problémů které je nutné v rámci rozvoje takovéhoto projektu řešit.

Literatura

[1] WWW stránky. Wikipedia Availability.
http://en.wikipedia.org/wiki/Availability

[2] Enrique Vargas. High Availability Fundamentals.
http://www.sun.com/blueprints/1100/HAFund.pdf

[3] WWW stránky. Wikipedia Load Balancing.
http://en.wikipedia.org/wiki/Load_balancing_%28computing%29

[4] Ed. R. Hinden. RFC 3768: Virtual Router Redundancy Protocol (VRRP).
http://www.ietf.org/rfc/rfc3768.txt

[5] Vic Cross. Virtual Router Redundancy Protocol on VM Guest LANs.
http://www.redbooks.ibm.com/redpapers/pdfs/redp3657.pdf

[6] WWW stránky. Jerome Etiene Vrrpd.
http://www.off.net/j̃me/vrrpd/index.html

[7] WWW stránky. Projekt VRRP.
http://sourceforge.net/projects/vrrpd/

[8] WWW stránky. OpenBSD Programmer's Manual CARP.
http://www.openbsd.org/cgi-bin/man.cgi?query=carp&sektion=4

[9] WWW stránky. OpenBSD Programmer's Manual IFSTATED.
http://www.openbsd.org/cgi-bin/man.cgi?query=ifstated&sektion=8

[10] WWW stránky. Projekt UCARP.
http://www.ucarp.org/

[11] WWW stránky. Keepalived Documentation.
http://www.keepalived.org/software_design.html

[12] WWW stránky. Alexander Cassen Vrpd.
http://www.linuxvirtualserver.org/ãcassen/

[13] WWW stránky. Projekt Linux-HA.
http://www.linux-ha.org/

[14] WWW stránky. OpenBSD Programmer's Manual PFSYNC
http://www.openbsd.org/cgi-bin/man.cgi?query=pfsync

[15] WWW stránky. Projekt Netfilter-HA.
http://svn.netfilter.org/cgi-bin/viewcvs.cgi/trunk/netfilter-ha/

[16] WWW stránky. Projekt Conntrack Tools.
http://people.netfilter.org/pablo/conntrack-tools/index.html

[17] WWW stránky. The OpenBSD Packet Filter.
http://www.openbsd.org/faq/pf/

[18] WWW stránky. Linux Programmer's Manual NETLINK.
http://www.tin.org/bin/man.cgi?section=7&topic=netlink

[19] WWW stránky. The Linux Virtual Server.
http://www.linuxvirtualserver.org/

[20] WWW stránky. Projekt DRBD.
http://www.drbd.org

[21] WWW stránky. Global File System Project.
http://sources.redhat.com/cluster/gfs/

[22] WWW stránky. Oracle Cluster File System.
http://oss.oracle.com/projects/ocfs2

[23] WWW stránky. Lustre cluster filesystem.
http://www.lustre.org/

[24] The Lustre Operations Manual.
http://manual.lustre.org/images/4/48/LustreManual_1.6_man_v19.pdf

[25] WWW stránky. GlusterFS Wiky.
http://www.gluster.org/docs/index.php/GlusterFS

[26] WWW stránky. Filesystem in Userspace.
http://fuse.sourceforge.net/

[27] WWW stránky. Apache MPM event.
http://httpd.apache.org/docs/2.2/mod/event.html

[28] WWW stránky. Varnish Project.
http://varnish.projects.linpro.no/

[29] WWW stránky.Varnish Notes from the Architect.
http://varnish.projects.linpro.no/wiki/ArchitectNotes

[30] WWW stránky. MySQL Proxy Project.
http://forge.mysql.com/wiki/MySQL_Proxy

[31] WWW stránky.SQL Relay.
http://sqlrelay.sourceforge.net/

[32] WWW stránky. Slony-I enterprise-level replication system.
http://main.slony.info/

[33] WWW stránky. PgPool-II Project.
http://pgpool.projects.postgresql.org/

[34] WWW stránky. MySQL Reference Manual - Replication.
http://dev.mysql.com/doc/refman/5.0/en/replication.html

[35] WWW stránky.MySQL Reference Manual - MySQL Cluster.
http://dev.mysql.com/doc/refman/5.0/en/mysql-cluster.html

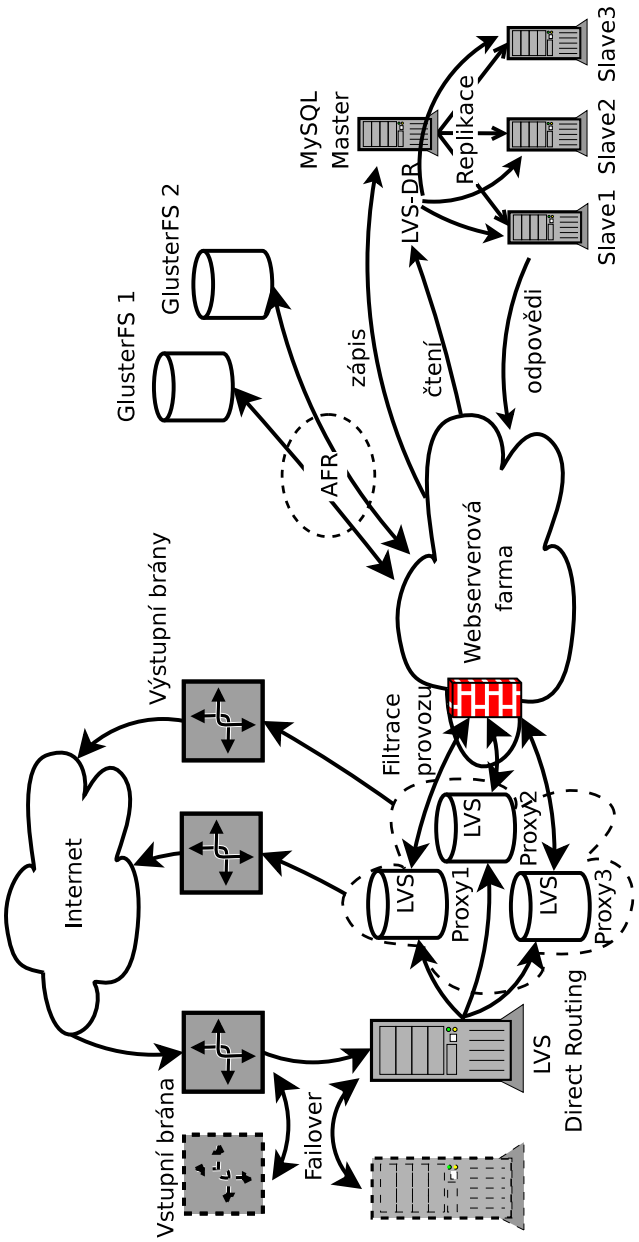
[36] WWW stránky. PGCluster Project.
http://pgfoundry.org/projects/pgcluster/

Příloha 1 Hardwarová konfigurace testovacích strojů

```
processor       : 1
vendor_id       : GenuineIntel
cpu family      : 6
model           : 15
model name      : Intel(R) Core(TM)2 Duo CPU     E6750  @ 2.66GHz
stepping        : 11
cpu MHz         : 2666.408
cache size      : 4096 KB
physical id     : 0
siblings        : 2
core id         : 1
cpu cores       : 2
fdiv_bug        : no
hlt_bug         : no
f00f_bug        : no
coma_bug        : no
fpu             : yes
fpu_exception   : yes
cpuid level     : 10
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush
dts acpi mmx fxsr sse sse2 ss ht tm pbe nx lm constant_tsc arch_perfmon pebs bts pni monitor ds_cpl
vmx smx est tm2 ssse3 cx16 xtpr lahf_lm
bogomips        : 5333.09
clflush size    : 64
```

```
00:00.0 Host bridge: Intel Corporation 82G33/G31/P35/P31 Express DRAM Controller (rev 02)
00:01.0 PCI bridge: Intel Corporation 82G33/G31/P35/P31 Express PCI Express Root Port (rev 02)
00:03.0 Communication controller: Intel Corporation 82G33/G31/P35/P31 Express MEI Controller (rev 02)
00:19.0 Ethernet controller: Intel Corporation 82801I (ICH9 Family) Gigabit Ethernet Controller (rev 02)
00:1a.0 USB Controller: Intel Corporation 82801I (ICH9 Family) USB UHCI Controller #4 (rev 02)
00:1a.1 USB Controller: Intel Corporation 82801I (ICH9 Family) USB UHCI Controller #5 (rev 02)
00:1a.2 USB Controller: Intel Corporation 82801I (ICH9 Family) USB UHCI Controller #6 (rev 02)
00:1a.7 USB Controller: Intel Corporation 82801I (ICH9 Family) USB2 EHCI Controller #2 (rev 02)
00:1b.0 Audio device: Intel Corporation 82801I (ICH9 Family) HD Audio Controller (rev 02)
00:1c.0 PCI bridge: Intel Corporation 82801I (ICH9 Family) PCI Express Port 1 (rev 02)
00:1c.1 PCI bridge: Intel Corporation 82801I (ICH9 Family) PCI Express Port 2 (rev 02)
00:1c.2 PCI bridge: Intel Corporation 82801I (ICH9 Family) PCI Express Port 3 (rev 02)
00:1c.3 PCI bridge: Intel Corporation 82801I (ICH9 Family) PCI Express Port 4 (rev 02)
00:1c.4 PCI bridge: Intel Corporation 82801I (ICH9 Family) PCI Express Port 5 (rev 02)
00:1d.0 USB Controller: Intel Corporation 82801I (ICH9 Family) USB UHCI Controller #1 (rev 02)
00:1d.1 USB Controller: Intel Corporation 82801I (ICH9 Family) USB UHCI Controller #2 (rev 02)
00:1d.2 USB Controller: Intel Corporation 82801I (ICH9 Family) USB UHCI Controller #3 (rev 02)
00:1d.7 USB Controller: Intel Corporation 82801I (ICH9 Family) USB2 EHCI Controller #1 (rev 02)
00:1e.0 PCI bridge: Intel Corporation 82801 PCI Bridge (rev 92)
00:1f.0 ISA bridge: Intel Corporation 82801IH (ICH9DH) LPC Interface Controller (rev 02)
00:1f.2 IDE interface: Intel Corporation 82801IR/IO/IH (ICH9R/DO/DH) 4 port SATA IDE Controller (rev 02)
00:1f.3 SMBus: Intel Corporation 82801I (ICH9 Family) SMBus Controller (rev 02)
00:1f.5 IDE interface: Intel Corporation 82801I (ICH9 Family) 2 port SATA IDE Controller (rev 02)
01:00.0 VGA compatible controller: nVidia Corporation G70 [GeForce 7300 GT] (rev a1)
03:00.0 IDE interface: Marvell Technology Group Ltd. 88SE6101 single-port PATA133 interface (rev b2)
07:01.0 Ethernet controller: Intel Corporation 82543GC Gigabit Ethernet Controller (Copper) (rev 02)
07:03.0 FireWire (IEEE 1394): Texas Instruments TSB43AB22/A IEEE-1394a-2000 Controller (PHY/Link)
```

Příloha 2 Celkové schéma navrhovaného zapojení



Obrázek 9.1: Celkové navrhované schéma zapojení