Univerzita Hradec Králové
Fakulta informatiky a managementu
Katedra informatiky a kvantitativních metod

DISTRIBUOVANÉ SYSTÉMY

Autor: David Šec
Studijní obor: Aplikovaná Informatika

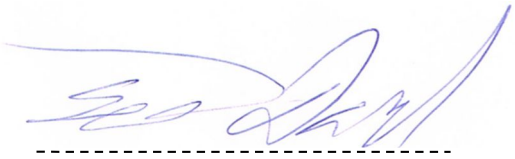
Vedoucí práce: *Horálek Josef, Mgr. Ph.D.*

Hradec Králové 1. září 2015

Prohlášení:

Prohlašuji, že tato práce je mým původním autorským dílem, které jsem vypracoval samostatně.

Všechny zdroje, prameny a literaturu, které jsem při vypracování používal nebo z nich čerpal, v práci řádně cituji s uvedením úplného odkazu na příslušný zdroj.



V Hradci Králové dne: 27.07.16 Šec David

Poděkování:

Tímto bych rád poděkoval vedoucímu práce *Mgr. Josefu Horálkovi Ph.D* za cenné rady a ochotu při konzultacích. Dále děkuji celé své rodině za podporu, bez které by tato práce nemohla vzniknout.

Anotace:

Cílem práce je podrobně představit principy distribuovaných systémů s důrazem na meziprocesovou komunikaci a synchronizaci. Dále představit principy distribuovaných databází. V praktické části implementuje vlastní řešení distribuovaného systému s několika fyzickými uzly s cílem sběru dat z okolního prostředí a na jejich základě vykonávat zadané akce.

Annotation:

Title: Distributed systems

Purpose of this diploma thesis is to introduce principles of distributed systems with emphasis on the multi-process communication and synchronisation. In the next parts introduce principles of distributed database systems. In the practical part implement own solution of distributed system with several physical nodes with main objective to collect data from environment and perform action on the basis of it.

Osnova práce

1 Úvod

Distribuované systémy, spolu s algoritmy pro distribuci dat jsou poměrně novou oblastí informatiky. I přesto však staví na principech známých v mnoha příbuzných oblastech informatiky již řadu let. Jedná se zejména o poznatky z oblasti matematických výpočtů a přístupu k informacím dostupných za pomoci určitého komunikačního média.

O distribuovaných systémech se začalo mluvit s masivním rozvojem vysokorychlostních počítačových sítí. Původním záměrem pro rozvoj počítačových sítí byla pouze výměna data a programů mezi uživateli. Po rozšíření osobních počítačů se stala právě počítačová síť přirozeným nástrojem pro sdílení periferních zařízení, paměťových kapacit, ale i uživatelských zátěží. Zajistila tak propojení tisíců zařízení a umožňovala rychlou odezvu v řádech jednotek milisekund. V současnosti budované počítačové sítě umožnují běžně přenést enormní objemy dat při rychlostech přesahujících jednu miliardu bitů za sekundu.

Máme-li zajištěno dostatečně kvalitní spojení mezi počítači, Nabízí se myšlenka spojení několika počítačů dohromady tak, aby jednotlivé komponenty spolupracovaly na dosažení určitého cíle. Toho můžeme dosáhnout například rozložením úlohy na několik menších samostatných úloh, a ty následně rozeslat na jednotlivé počítače – výpočetní uzly. K zajištění optimální spolupráce mezi sebou tyto uzly komunikují za pomoci mechanismu zasílání zpráv.

V diplomové práci budou postupně představeny základní vlastnosti distribuovaných systémů spolu s popisem vývoje výpočetního modelu a přehledem vybraných distribuovaných systémů s krátkým popisem každého z nich. Následně bude představena komunikace mezi procesy, které je hojně využíváno v praktické části. Součástí kapitloly je krátké představení síťových modelů, spolu s možnými úskalími, které síťová komunikace přináší. Kapitola synchronizace je věnována zejména vzájemné synchronizaci procesů a s tím spojenou synchronizací vnitřních hodin. Jsou představeny vybrané algoritmy pro jejich synchronizaci a vzájemné vyloučení procesů spolu se způsoby jejich použití. I když je synchronizace v praktické části zmíněna spíše okrajově, tvoří nedílnou součást distribuovaných systémů a tvoří tak plnohodnotnou kapitolu teoretické části. Předposlední kapitola představuje obecné principy distribuovaných databází spolu se způsoby uložení dat a následnou distribucí transakcí a distribuovaným zamykáním objektů. V praktické části je následně podrobně popsána implementace vlastního řešení distribuovaného systému pro automatické řízení inteligentní domácnosti.

2 Distribuovaný systém

Existuje řada definic distribuovaného systému. Jednou z nich by mohla být například od [1].

> *A distributed system is a collection of independent computers that appears to its users as a single coherent system.*
>
> *(Tanenbaum, 2007, p. 2)*

Ten popisuje distibuovaný system jako systém, který je složen z více autonomních strojů (počítačů), keré se uživateli jeví jako jeden systém, se kterým je schopen interagovat. Jednotlivé stroje spolu komunikují mechanismem zasílání zpráv. Za tímto účelem musí být propojeny sítí. Symslem distribuovaného systému je efektivně koordinovat práci všech strojů k dosažení určitého cíle, případně koordinovat používání sdílených zdrojů.

Takto propojené počítače tak mohou spolupracovat na libovolné vzálenosti. Mohou se nacházet na rozdílných kontinentech, nebo jen v různých místnostech jedné budovy.

Důležitými vlastnostmi distribuovaných systémů jsou :

Concurrency – V prostředích, kde dochází ke sdílení zdroů mezi uživateli je zcela běžné, že systém musí současně obsloužit několik užívtelů naráz. Pokud je to nutné, mohou uživatelé rovněž sdílet určitá data, nebo periferie. Systém může být dále rozšiřován připojováním nových uzlů podle aktuální potřeby a zátěže daného systému.

No global clock – V distribuovaných systémech neexistují žádné společné hodiny, ani přesné měření globálního času. Proto se zavádějí tzv. logické hodiny, které zajišťují konzistentní měření času v celém systému. Ty slouží k tomu, aby systém dokázal indetifikovat pořadí, v jakém se dané události staly. Nezajišťují však žádným způsobem měření reálného času [2].

Independent failures – Ve výpočetním systému s jedním procesorem dojde v případě slehání hardware ke kompletnímu kolapsu celého systému. V takovém případě nemůže uživatel se systémem pracovat. S rostoucím počtem komponent navíc stoupá i pravděpodobnost vzniku poruchy. Pokud dojde k selhání některé z komponent v distribuovaných systémech, zbylé komponenty převezmou jeho úlohu a umožní udržet celý systém v chodu. Případná chyba tak neohorzí probíhající výpočet. V mnoha distribuovaných systémech dochází v pravidelných intervalech ke kontrole ostatních uzlů tak, aby případná chyby byla okamžitě detekována, diagnostikována její závažnost a případně automaticky opravena.

2.1.1 Transparentnost

Základní charakteristikou *transparentních* distribuovaných systémů je minimalizovat nepříjemné vlivy sdílení na uživatele. Fakt, že jsou některé procesy a zdroje distribuovány napříč internetovou sítí by měl mít minimální dopad na uživatele samotného. Ten by měl mít v ideálním případě pocit, že pracuje pouze s lokálními zdroji na svém počítači. Takto navržené systémy mohou být ozančeny za transparentní. Transparentnst distribuovaných systémů může být brána z několika pohledů. Jejich přehled znázorňuje Tabulka 1.

TRANSPARENTNOST	POPIS
ACCESS *(PŘÍSTUPOVÁ)*	Proces má přistupovat ke sdíleným zdrojům stejně jako by byly umístěny lokálně
LOCATION *(LOKAČNÍ)*	Uživatel, nebo proces nemusí vědět, kde konkrétně se požadované zdroje nacházejí.
MIGRATION *(MIGRAČNÍ)*	Jednotlivé zdroje se mohou přemisťovat bez vědomí uživatele
RELOCATION *(REALOKAČNÍ)*	Jednotlivé procesy mohou být v případě potřeby přesunuty na libovolný procesor
REPLICATION *(REPLIKAČNÍ)*	Uživatel nemůže říci kolik kopií objekty se aktuálně v systému nachází
CONCURRENCY *(KONKURENTNÍ)*	Uživatel neví s kolika dalšími uživateli sdílí dané zdroje
FAILURE *(TRANSPARENCE SELHÁNÍ)*	Uživatel se nemusí dozvědět o případném selhání daného zdroje

Tabulka 1: Transparentnost v distribuovaných systémech (ISO, 1995); převzato z [2] (upraveno)

Přístupová transparentnost řeší rozdíly mezi vnitřní reprezentací dat a způsob, jakým k těmto datům přistupovat. Základním účelem je odstínit uživatele od často rozdílné vnitřní architektury daného stroje, ale také od vnitřní repreznetace dat na často rozdílných operačních systémech. Je tak vždy potřeba zohlednit určitá specifika daného operačního systému, například jmennou konvenci, přístupová práva, nebo další atributy každého souboru.

Lokační transparentnost označuje fakt, že uživatel nemusí jednoznačně určit, ve které části distribuovaného systému se daný zdroj nachází. Pro zajištění lokační transparentnosti tvoří důležitou roli vhodné pojmenování jednotlivých zdrojů v uzlu. Za tímto účelem se používá jednoznačný identifikátor objektu (někdy nazýván rovněž jako binární jméno). Jednoznačně přiřazená binární hodnota by však byla pro uživatele

jen těžko zapamatovatelná. Proto se často zavádějí tzv. symbolická jména objektů. Zavedení symbolických jmen má rovněž další výhody. Dle [3] lze chápat transformaci symbolických jmen na binární jako matematické zabrazení. K jednomu symbolickému jménu tak může být přiřazeno více shodných objektů. Takto je dle [1] možné efektivně přepínat mezi jednotlivými ekvivalentními objekty v případě výpadku některého z nich, nebo rovnoměrně distribuovat úlohy mezi dostupné zdroje s cílem rozložení zátěže napříč uzly. Symbolická jména musí být globálně jednoznačná. Dále mohou být upřednostňovány lokálně dostupné zdroje, které jsou z daného uzlu lépe přístupné, nejsou přetížené apod.

Lokační transparentnosti může být dosaženo například vhodným přiřazením symbolických jmen ke zdrojům tak, aby odrážely název objektu, případně i jeho polohu v distribuovaném systému [1].

Migrační transparentnost popisuje možnosti přesouvání vybraných objektů bez toho, aby se změnil způsob jakým lze k objektu přitupovat. Migrační transaprentnost je spojená s transparentností lokační. Kdy jsou objekty identifikovány logickým jménem a jejich fyzické umístění v systému nehraje zásadní roli pro přístup k němu.

S výše popsanými souvisí i *realokační transparentnost*. Přemístění objektu, nebo procesu do jiné části systému nesmí mít dopad na uživatele. Kromě přemísťování objektů mohou být některé objekty replikovány. Jedná se o vytváření kopií daného objektu, které jsou následně strategicky rozmístěny v jednotlivých částech systému. Replikace se nejčastěji vuyžívá v distribuovaných databázích s cílem snížit přenášené objemy dat a čas potřebný pro vykonání dotazu. *Replikační transparentnost* popisuje, že systém by měl sám určit, která data je vhodné replikovat a sám zajistit synchronizaci všech kopiií – replik.

Konkurentní transaprentnost řeší přístup současný přístup několika uživatelů k jednomu objektu. Autor [2] tento přístup nazývá jako „konkurentní přístup“. Systém musí zajistit, aby se data nacházela v konzistentním stavu (viz kapitola Transakce) a nedocházelo tak ke vzájemným přepisům dat.

Poslední *transparence selhání* řeší případy, kdy dojde k náhlému výpadku některého z uzlů systému. Systém by měl v případě výpadku uzlu sám vyhledat ekvivalentní zdroje a převezít jeho úlohu do doby, než dojde k opětovnému zotavení daného uzlu.

2.1.2 Přizpůsobivost

Autonomie – Jednotlivé počítače tvořící distribuovaný systém jsou na sobě zcela nezávislé a samostatně funkční. Mohou mít zcela odlišný hardware, operační systém a podobně.

Decentralizované řízení a rozhodování – Každý procesor vykonává rozhodnutí nezávisle na ostatních. V distribuovaných systémů jsou řídící algoritmy rozprostřeny po celé síti. Vzhledem k absenci sdílené paměti a neexistence globálního stavu je velmi obtížné zaručit, aby pohled jednotlivých komponent na systém jako celek byl stále jednotný. Toho lze docílit pouze přísnou synchronizací. To však vede k prodloužení doby požadované akce. Proto se dle [3] velmi často distribuované řízení realizuje tak, že v ustáleném stavu, kdy systém běží stabilně, je řízení prováděno centralizovaně. V případě výpadku centralizovaného řízení dojde pomocí decentralizovaných algoritmů k opětovné obnově centralizovaného řízení [2].

Migrace procesů a prostředků – V případě potřeby mohou být procesy i prostředky přemístěny na jiný počítač. Důvodem může být například automatické rozprostření zátěže.

Vyvažování výpočetní zátěže – Úlohy mohou být automaticky přemístěny na méně vytížený procesor, kde bude úloha dokončena bez toho, aby se musel celý výpočet opakovat [2].

2.1.3 Otevřenost a rozšiřitelnost

V distribuovaných systémech je každá služba specifikována vlastním rozhraním. To je často definováno pomocí Interface Definition Language (IDL), které definuje atributy a metody, které daná služba využívá, návratové hodnoty a případné výjimky, které může služba vyvolat. Výhodou oteřeného systému je, že systém lze nakonfigurovat z různých komponent třeba od různých vývojářů. Lze tak jednoduše zaměňit určitou komponentu za jinou implementující stejné rozhraní bez toho, aby to nějak ovlivnilo stávající komponenty systému. Otevřený distribuovaný systém by měl být rovněž snadno *rozšiřitelný*.

V rozšiřitelném systému by mělo být snadné přidat určité části, které běží na rozdílném operačním systému, nebo zaměnit celý souborový systém za jiný. V praxi je však úplná flexibilita systému jen velmi obtížně proveditelná. Obvykle je vytipována pouze určitá omezená množina funkcí, která musí být systémem zabezpečena ve všech uzlech [1].

2.1.4 Škálovatelnost

Škálovatelnost distribuovaného systému je schopnost reagovat na vzrůstající požadavky klietů. Měl by umožnit snadné přidávání nových prostředků, pokud to aktální sitauace vyžaduje. Podle [4] můžeme měřit Škálovatelnost systémů v několika různých dimenzích. Základní dimenzí je *numerical (zátěžová škálovatelnost)* a má řešit vzrůstající počet uživatelů a s tím spojený počet objektů a požadavků na distribuovaný systém.
Geografická dimenze řeší výkonnost a použitelnost systému bez ohledu na to, do kolika vzdálených lokalit je systém rozptýlen, nebo jaké jsou vzdálenosti mezi nimi.
Poslední *Administrativní* dimenze určuje způsob, jakým má být řešena správa jednotlivých části distribuovaného sytému.

The administrative dimension consists of the number of organizations that exert control over pieces of the system.

(Neuman, 1994, p. 1)

2.1.5 Výkonnost

Úloha, kterou uživatel zpracovává pomocí distribuovaného systému by neměla být výrazně pomalejší, než kdyby byla zpracovávána na klasickém (jednoprocesorovém) systému. V distribuovaném systému je však pro dosažení podobné rychlosti potřeba výkonnější hardware. To je způsobeno vyšší odezvou při přenosu zpráv po síťovém médiu, ale i vyšší režií složitějšího systému [2].

2.2 Vývoj distibuovaných systémů

Výpočetní model je dle [5] ucelená představa o tom, kde jsou aplikace uchovávány jako programy a kde skutečně běží, zda (a jak) jsou aplikace rozděleny na části a jakým způsobem tyto části vzájemně spolupracují, kde a jak se uchovávají a zpracovávají data, kde se nachází uživatel, případně kdy a jakým způsobem komunikuje se svými aplikacemi.

Některé výpočetní modely vůbec napočítají s existencí počítačové sítě, jiné její existenci předpokládají a další naopak přítomnost počítačové sítě pro svůj běh vyžadují. Správné objasnění výpočetních modelů je nezbytné pro pochopení vzniku distribuovaných systémů [2].

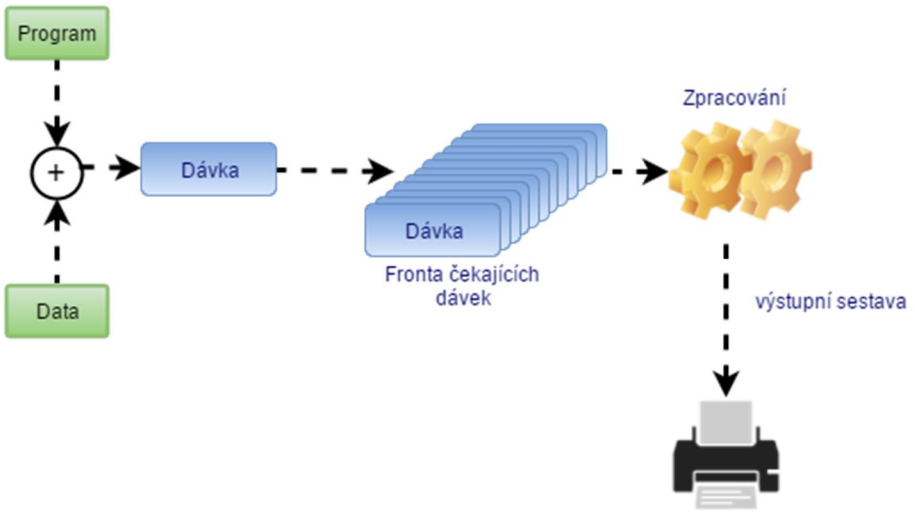
Výpočetní model se postupem času vyvíjel (a stále se i vyvíjí) od absobutní centralizace zpracování inforamcí po absolutní decentralizaci. Postupně se hledaly různé kompromisy jako reakce na určité nedostatky, které s sebou tato řešení nesla. Cílem následující kapitoly je představit nejdůležitější výpočetní modely a nastínit trendy, jakými by

se mohly ubírat do budoucna. Dále je třeba zdůraznit, že výpočetní model je vlastností konkrétní aplikace, nikoliv počítačové sítě. V reálných systémech je zcela obvyklé, že různé aplikace využívají různé výpočetní modely.

2.2.1 Dávkové zpracování

Dávkové zpracování (*batch processing*) je prvním a také historicky nejstarším výpočetním modelem. Pochází ještě z dob prvních počítačů, které byly velmi nákladné a bylo spíše vynuceno dobou a potřebou sdílet omezené hardwareové a softwareové prostředky s co největším počtem uživatelů. Vyznačovaly tím, že uživatel neměl bezprostřední kontakt s úlohou, kterou vytvořil. Mohl pouze dopředu přesně specifikovat, co má konkrétní program vykonat. Následně instrikce spolu s daty zabalil do balíčku (tzv. dávky). Ta měla nejčastěji podobu děrného štítku, nebo svitku. Ten byl následně fyzicky přenesen k počítači a zařazen ke zpracování do fronty dalších dávek čekajících na zpracování. V okamžiku, kdy na příslušnou dávku přišla řada, server ji zpracoval a výsledky opět zabalil, tak aby mohly být předány uživateli. Výstup probíhal zpravidla na tiskárnu. Nevýhodou takového přístupu byla dlouhá doba obrátky (doba od odeslání dávky po obdržení výstupní sestavy).

I když měl tento přístup řadu omezení, dokázal velmi efektivně využít dostupné zdroje. V určité formě je používán v řadě specifických oblastí dodnes. Princip dávkového zpracování zobrazuje Obrázek 1.

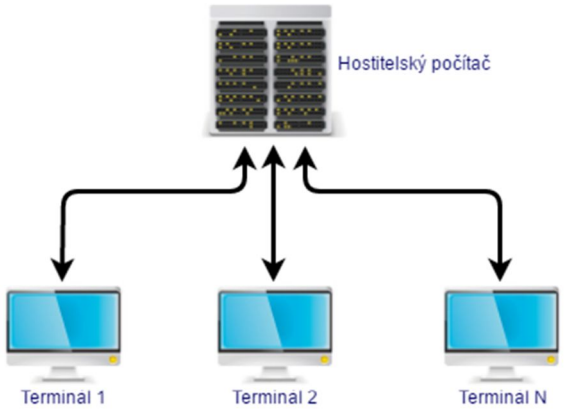


Obrázek 1: Podstata dávkového zpracování převzato z [2] (upraveno)

2.2.2 Host-Terminál

Model host-terminál vznikal jako reakce na neinteraktivní dávkové zpracování.

Hlavní výpočetní kapacitu poskytoval hostitelský počítač, který hostoval veškeré systémové zdroje. Těmi mohla být nejen výpočetní kapacita, ale i úložný prostor, periferie, ale i operační systém spolu s aplikacemi a jejich daty. Hostitelský počítač navíc umožňoval připojit N uživatelských terminálů, které tyto zdroje využívaly. Každý z terminálů sloužil pouze pro interakci uživatele s aplikací a vykreslování výstupů (viz Obrázek 2).



Obrázek 2: Model host-terminál

Oproti dávkovému zpracování umožňovala reagovat na probíhající výpočet při zachování současného přístupu více uživatelů. Toho bylo docíleno sdílením času, kdy jednotlivé úlohy byly v určitých časových intervalech střídány a vytvářely tak iluzi současně běžících programů.

2.2.3 Éra osobních počítačů

S příchodem osobních počítačů si lidé slibovali především vyšší komfort a nezávislost na ostatních, tj. bez nutnosti sdílet výpočetní zdroje s ostatními uživateli. I když se tyto požadavky podařilo splnit, záhy se objevily problémy nové. Dříve se daný problém řešil vždy na jednom místě, nyní však bylo nutné řešit stejný problém opakovaně na různých místech. Uživatelé jsou tak mnohem více odkázáni sami na sebe. Dále bylo nutné vzřešit složitější sdílení dat a programů.

S rostoucím využitím osobních počítačů vyvstával problém, které zdroje sdílet a které naopak využívat izolovaně pouze na osobním počítači. Místo dříve používaného modelu host/terminál mohl nyní každý uživatel disponovat vlastní výpočetní kapacitou a mohl tak provádět řadu úloh přímo na svém osobním počítači a to bez nutnosti připojení k síti. Postupem času se však ukázalo, že potřeba sdílet některé zdroje je stále aktuální a

je mnohem výhodnější, pokud se uživatel může o některé zdroje podělit s ostatními. Jednalo se zejména o drahé periferie, tiskárny a skenery, ale i důležitá firemní data a databáze.

Předpokladem pro úspěšné sdílení dat je, že uživatel nesmí zpozorovat výraznější rozdíly mezi přístupem k lokálním a sdíleným datům. Proto je nutné zabezpečit dostatečně rychlé přenosové technologie a mechanismy sdílení musí být implementovány transparentně. S těmito požadavky vznikaly první sítě LAN, které řešily potřebu sdílení souborů a periferií s několika uživateli [2].

2.2.4 Klient-Server

Model klient-server představuje určitý kompromis, mezi absoutní centralizací, kdy byly veškeré zdroje sdíleny, a absoutní decentralizací, kdy byly veškeré prostředky

Obrázek 3: Model Klient-Server

svěřeny do rukou uživatele. V modelu klient-server je hlavní myšlenkou ponechat data v místě kde se nachází, zatímco výstupy se generují na straně uživatele. Z tohoto důvodu musely být aplikace rozděleny na dvě části. První část zpracovávající data běžela na serveru. Druhá část na klientské stanici zajišťovala interakci s uživatelem a vykreslení uživatelského rozhraní. Klient i server spolu komunikují zasíláním dotazů a odpovědí. Komunikaci zahajuje vždy klient zasláním požadavku na server. Vhodnou formulací dotazů lze rovněž přispět ke značné redukci přenášených dat. Je možné provozovat komunikaci nejen v lokální síti, ale i v celosvětové síti internet. Princip komunikace zobrazuje Obrázek 3.

V současnosti se od dvouvrstvých aplikací opouští a je nahrazována architekturou třívrstvou, kde dochází na straně serveru k oddělení aplikační a datové vrstvy.

2.3 Popis nejznámějších distribuovaných systémů

2.3.1 World Wide Web

Celosvětová síť Internet je asi nejznámějším prostředkem komunikace mezi počítači. Tato síť spojuje miliardy počítačů po celém světě a umožňuje jim tak komunikovat s libovolným počítačem připojeným do internetu.

Autoři [6] a [7] popisují jeho strukturu jako distribuovaný systém složený z milionů různých serverů rozmístěných po celém světě a poskytujících libovolný obsah. Jedním z příkladů distribuovaného systému je webový server. Server předkládá uživateli obsah ve formě HTML souborů. Součástí obsahu může být rovněž odkaz na libovolné další soubory (nebo webové stránky) uložené na webu za pomoci jednoznačného identifikátoru URL.

Tím může při prohlížení obsahu vzniknou iluze jednoho velkého systému. Ve skutečnosti se však jedná o nepřeberné množství nezávislých webových serverů, které se často nacházejí v různých lokalitách, spravují je různí administrátoři a často mívají odlišný operační systém. Přesně v tom spočívá podstata distribuovaných systémů. Zpřístupnit uživateli systém (v tomto případě webový obsah) jako celek bez ohledu na počet uzlů, topoligii, nebo způsob komunikace. Ta probíhá výhradně prostřednictvím jednotného komunikačního rozhraní. Tím je v případě webového serveru protokol HTTP, resp. HTTPS. Koncový uživatel je však od této komunikace odstíněn a zprostředkovává ji za něho webový prohlížeč.

2.3.2 Apache Hadoop

Apache Hadoop je open source projekt společnosti Apache. Vznikla jako řešení inspirované koncepcí MapReduce od společnosti Google. Jedná se o komplexní distribuované výpočetní prostření vynikající svou spolehlivostí a snadnou škálovatelností postavené nad souborovým systémem HDFS (Hadoop Distributed File System) a programovacím paradigma MapReduce. Systém samotný může běžet na jednom počítači, ale mnohem častěji bývá nasazován do výpočetních clusterů čítajících stovky, nebo tisíce počítačů [8].

Hlavní podstata výpočtu spočívá v rozdělení rozsáhlých dat na menší bloky, které rozesílá ostatním počítačům. Zde pak dochází k paralelnímu zpracování dat a maximálnímu zefektivnění dané úlohy.

S rostoucím počtem uzlů je však nutné řešit i možné výpadky. Pro cluster s řádově tisíci uzly je pravděpodobnost chyby některého z uzlů poměrně vysoká. Klíčovou vlastností

systému je replikace dat, kdy je každý blok dat uložen vždy na několik uzlů tak, aby případný výpadek některých uzlů neohrozil průběh výpočtu. Díky replikaci je tak možné úlohu dokončit na jiném uzlu. Počet replik můžeme určit pro každý soubor zvlášť.

Tento systém dokáže velmi spolehlivě zpracovat rozsáhlé soubory dat a těší se velké oblibě zejména v oblastech distribuovaných výpočtů a Big Data, kde často napomáhá k hromadnému zpracování dat před jejich analýzou, nebo se určitou mírou podílejí na analýze dat samotné [9].

2.3.3 NFS

NFS (*Network file system*) je distribuovaný souborový systém vyvinutý firmou SUN Microsystems. Přes určitá omezení i fakt, že v současnosti existuje mnoho sofistikovanějších distribuovaných souborových systémů (např. AFS), jeho snadná implementace a široká nativní podpora napříč platformami jej činí stále hojně využívaným distribuovaným souborovým systémem.

NFS umožňuje transparentní přístup ke vzdáleným souborům prostřednictvím počítačové sítě. K tomu využívá stejnojmenný protokol. Ten je navržen tak, aby umožnoval přístup bez ohledu na typ počítače, operační systém, síťové architektuře, nebo protokol transportní vrstvy.

Přístup ke vzdálenému hostitelskému serveru je uskutečňován stejně jako přístup uživatele k lokáním souborům. To je docíleno použitím NFS na straně serveru v kombinaci s funkcí jádra na straně klienta. Platformní nezávislost zajišťuje použití univerzálního RPC mechanismu - vzdáleného volání procedur (viz kapitola Remote Procedure Call – RPC). Transparentnost pak dle [3] zajišťuje XDR (eXternal Data Representation).

3 Meziprocesová komunikace

V distribuovaném systému spolu spolupracuje vždy skupina procesů. Aby mohla být tato spolupráce efektivně koordinována, musejí spolu jednotlivé procesy komunikovat. Meziprocesová komunikace tvoří hlavní rozdíl mezi jednoprocesorovým a distribuovaným systémem.

V distribuovaných systémech nelze oproti jednoprocesorovým systémům předpokládat exitenci společné paměti, ke které by měly přístup všechny procesy a která by sloužila k výměně dat. Bylo nutné zavést jiné principy komunikace a synchronizace. Informace o stavu jednotlivých procesů jsou dle [2] zasílány v podobě zpráv.

Dle [3] může být propojení jednotlivých uzlů v distribuovaném systému realizováno pomocí komunikačních linek. Komunikační spoje mohou být jak dvoubodové, tak i vícebodové. Pokud chce proces A komunikovat s prcesem B, musí nejprve sestavit zprávu ve svém adresním prostoru. Následně jej odešle za pomoci systémového volání prostřednictvím sítě na počítač, kde běží proces B. Ten zprávu převezme a zpracuje.

Základní myšlenka se zdá velice jednoduchá, ale ve skutečnosti se musejí obě strany předem dohodnout na způsobu komunikace. Za tímto účelem jsou zaváděny standardizované internetové protokoly [1].

Pro komunikaci se většinou používají standardní protokoly známé z oblastí počítačových sítí, lze však implementovat i vlastní řešení a komunikační protokoly odpovídající požadavkům a danému způsobu komunikace.

3.1 Síťová architektura a topologie sítě

Řízení počítačové sítě vyžaduje velmi komplikovaný software rozdělený do hierarchie vrstev. Každá z těchto vrstev představuje distribuovaný systém, ve kterém pracují distribuované algoritmy zvané komunikační protokoly.

(*Klimeš, 2007, p. 7*)

Dle [10] je z hlediska sítě nezbytné rozlišovat síťovou architekturu a síťovou topologii. Zatímco síťovou architekturou se nazývají metody sloužící pro komunikaci po síťovém médiu, topologií sítě se rozumí fyzická síťová infrastruktura sloužící k propojení jednotlivých systémů.

V některých případech může konkrétní architektura směřovat k jediné možné topologii, jindy bude naopak určité topologii vyhovovat pouze jediná možná architektura. Taková vazba mezi architekturou a topologií sítě však není vždy natolik pevná a jednoznačně daná.

Nejčastějšími faktory pro výběr architektury mohou být geografické rozložení sítě, organizační struktura, počet uživatelů, požadavky na výkon, nebo kvalifikace personálu pro správu infrastruktury.

3.2 Referenční model ISO/OSI

Referenční model OSI vznikal jako jednotný standard vydaný organizací ISO pro komunikaci prostřednictvím sítě. Norma ISO/IEC 7498-1 v obecné podobě představuje způsob komunikace od úrovně fyzického připojení až po samotný přenos dat.

Celý proces síťové komunikace je poměrně komplikovaný, proto je potřeba ho rozdělit do několika vrstev. Každá z vrstev využívá služeb nižší vrstvy a naopak poskytuje své služby vrstvě vyšší.

Model ISO/OSI rozděluje síťovou komunikaci do sedmi vrstev.

Fyzická vrstva – pracuje s daty na úrovni bitového proudu. Řeší zejména jednotlivé napěťové úrovně pro hodnoty logické 0/1, navazování a ukončování komunikace, případně modulace signálu pro dané přenosové médium.

Linková vrstva – poskytuje správu spojení dvou sousedních systémů, řízení mechanismu pro určení optimální cesty, segmentace zpráv do datových rámců. Součástí linkové vrstvy je rovněž správa přenosové rychlosti.

Síťová vrstva – tato vrstva má na starosti směrování paketů v síti.

Transportní vrstva – řeší integritu přenášených dat, rozdělení zprávy na pakety a jejich opětovné spojení na straně příjemce. Potvrzování přijetí zprávy (ACK).

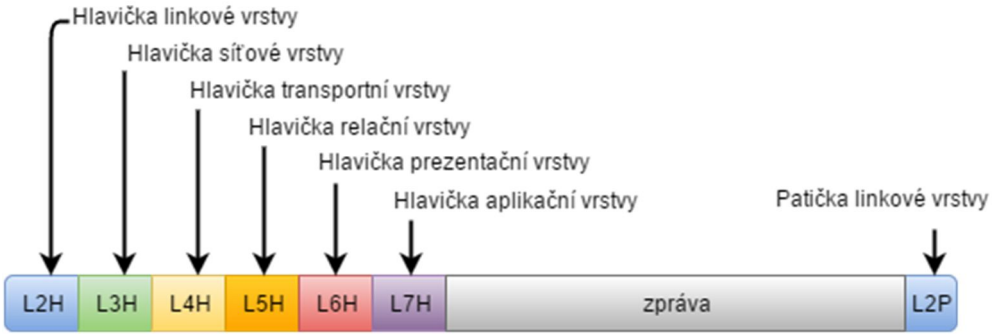
Relační vrstva – zajišťuje prostředky pro udržení relací, řeší kvalitu spojení a jeho navázáni v případě ztráty spojení, případně bezpečnostní mechanismy.

Prezentační vrstva – má na starosti kódování znaků, případně pak kompresi a šifrování zprávy. Samozřejmostí je pak opětovná reprodukce zprávy do původní podoby na straně příjemce a její předání aplikační vrstvě.

Aplikační vrstva – tvoří rozhraní mezi uživatelskou aplikací a síťovou infrastrukturou.

Bližší popis modelu OSI i význam jednotlivých vrstev detailně popisuje například [10].

V průběhu odesílání jsou data postupně obalována dalšími informacemi nezbytnými pro přenos v dané vrstvě. Tomuto procesu se říká zapouzdření (encapsulation). U příjemce jsou naopak tato data opět postupně ořezávána do podoby původní zprávy. Jedná se o takzvaný proces odpouzdření (decapsulation). Proces zapouzdření a odpouzdření paketu zobrazuje Obrázek 4.



Obrázek 4: Princip zapouzdření zprávy; převzato z [1] (upraveno)

3.3 Model TCP/IP

Kromě soustavy ISO/OSI se v praxi spíše setkáme s modelem TCP/IP. Obdobně jako předchozí model rozděluje komunikaci do vrstev Model TCP/IP rozlišuje pouze čtyři vrstvy.

Vrstva síťového rozhraní – přebírá zodpovědnost za první a druhou vrstvu modelu ISO/OSI. Obsahuje informace pro odeslání na sousední zařízení. TCP/IP přímo nedefinuje protokol síťového rozhraní, řeší pouze návaznost předchozí vrstvy na vrstvu síťového rozhraní. Ta již používá své vlastní komunikační protokoly v závislosti na použitém médiu.

Síťová vrstva (IP) – řeší směrování sítí za pomoci IP paketů ze zdrojového na cílové zařízení.

Transportní vrstva – navazuje, nebo ukončuje spojení, rozděluje data do paketů, řeší pořadí a správnost přijatých paketů. V případě potřeby si může vyžádat chybějící, nebo poškozené části.

Aplikační vrstva – zodpovědnost za kódování, převod implementace OS na standardní komunikační rozhraní.

Oproti ISO/OSI předpokládá model TCP/IP jednoduchou nespojovanou komunikaci s maximálním důrazem na rychlost. Přenosové mechanismy se tak starají pouze o co nejefektivnější přenos dat. Celý přenos funguje na principu maximální snahy (best effort), kde síť jako taková řeší pouze samotný přenos dat a zajišťování spolehlivosti přenechává na koncových uzlech. Případná spojovanost a spolehlivost komunikace je proto řešena až na úrovni transportní vrstvy a je pouze na konkrétní aplikaci, zda si vystačí s nespolehlivým přenosem, nebo zda využije spolehlivého transportního protokolu. Eventuelně si může spolehlivost komunikace zajistit vlastní režií [11].

V poslední řadě je nutné říci, že pro některé služby je TCP/IP principiálně nevhodné. Jedná se zejména o multicastovou komunikaci $1{:}N$, tedy takovou komunikaci, kdy jsou přenášena stejná data od jednoho zdroje k více příjemcům současně. Podle [11] takové přenosy umožnila až služba MBONE, která však nebyla příliš úspěšná, a později ji nahradila IPv6 a IP Multicast Initiative.

3.3.1 TCP protokol

Transportní protokol TCP (Transmission Control Protocol), je protokol pracující na transportní vrstvě orientovaný na přenos se zvýšenou režií. Tato režie má sice za následek snížení rychlosti, ale přináší určité výhody. Mezi ně můžeme zařadit spolehlivost doručování, doručení paketů ve správném pořadí, nebo kontrolu toku dat. Jedná se o protokol spojového charakteru. Spojení se navazuje pomocí třífázového handshakingu. Přenos dat probíhá mezi dvěma body. Zpráva je rozdělena na pakety podle MTU (Maximum Transmission Unit) a následně je odesílá příjemci. Ten musí přijetí každého paketu potvrdit. Pokud nedojde do určité doby k potvrzní paketu, odesilatel je automaticky odešle znovu. Velikost MTU se může lišit podle druhu přenosového média. Dle [12] často bývá volena tak, aby se přizpůsobila nejmenšímu MTU na cestě mezi příjemcem a odesilatelem. Po dokončení přenosu je nutné spojení ukončit.

Potvrzování přijatých paketů a jejich číslování může být velice výhodné z hlediska bezpečnosti přenosu, Nevýhodou je pak vyšší náročnost přenosu, protože každý paket musí obsahovat řadu redundantních údajů. O tom svědčí i velikost TCP hlavičky, která je 20 bajtů u IPv4, resp. 40 bajtů u IPv6.

3.3.2 UDP protokol

UDP protokol (User Datagram Protocol) je druhý z protokolů fungujících na transportní vrstvě. Oproti TCP pokytuje nespojovou komunikaci bez garance doručení. Jeho hlavní výhodou je naopak velmi nízká režie přenosu. Díky tomu je ideální pro řadu aplikací, která upřednostňují rychlost přenosu před spolehlivostí. UDP rovněž umožňuje broadcastovou, nebo multicastovou komunikaci.

3.4 Spolehlivost komunikace

Až doposud jsme uvažovali námi navrženou komunikaci za spolehlivou a předpokládali jsme, že pokud klient vyšle libovolnou zprávu na adresu serveru, server tuto zprávu obdrží přesně v takové podobě, v jaké byla vyslána. V praxi tomu tak samozřejmě není a je potřeba řešit mechanismy zajišťující spolehlivé doručení zpráv spolu s detekcí chyb vznikajících při přenosu dat po síti. Toho je docíleno zavedením potvrzovací zpětné vazby do přenosu.

Samotný proces zasílání zpráv můžeme rozdělit na několik částí. V každé části si klient i server vyměňují informace o průběhu zaslání dané zprávy. Každá přijatá část je příjemcem zkontrolována a potvrzena její bezchybnost v podobě ACK zprávy (acknowledgement). Existuje celá řada mechanismů, jakými lze zprávy potvrzovat. Od potvrzení každého datagramu zvlášť, až po kontinuální zasílání určitého počtu paketů a jejich zpětné potvrzování protistranou. Potvrzovací zprávy mohou mít podobu samostatných paketů, nebo mohou být u obousměrné komunikace přímo součástí datagramů. Takové technice se říká *piggybacking*. Potvrzovací mechanismus může rovněž protistranu informovat formou NAK (non-acknowledgement) o rámcích, které byly doručeny s chybou a měly by být zaslány znovu [13].

V případech, kdy je zpráva příliš velká na to, aby se její obsah vešel do jednoho paketu, bývá automaticky rozdělena. Jednotlivé pakety jsou očíslovány a samostatně odeslány příjemci. V takových případech může příjemce potvrzovat buď celou zprávu, nebo každý paket zvlášť. První způsob má výhodu rychlejší reakce na chybný paket, ale za cenu vyšší režie sítě, zatímco pro potvrzení celé zprávy stačí jedna zpráva. Můžeme tím sice ušetřit řadu potvrovacích zpráv, avšak v případě chyby libovolné z částí zprávy musí být celá zpráva odeslána znovu. To obvykle vyžaduje větší čas potřebný na zotavení.

Kód	*Typ paketu*	*Od*	*Komu*	*Popis*
REQ	Request	Klient	Server	Žádost klienta o postytnutí služby
REP	Reply	Server	Klient	Odpověď serveru
ACK	Acknowledgement	Libovolná strana		Předchozí paket dorazil neporušen
AYA	Are you alive?	Klient	Server	Zjištění dostupnosti serveru
IAA	I am alive.	Server	Klient	Potvrzení o funkčním serveru.
TA	Try again.	Server	Klient	Server nemá prostor
AU	Address unknown	Server	Klient	Žádný z procesů nepoužívá tuto adresu

Tabulka 2: Typy paketů používané v komunikaci Klient/Server; převzato z [14] (upraveno)

Tabulka 2 zobrazuje přehled používaných typů zpráv v modelu Klient/Server. První tři typy zpráv slouží k zajišťení komunikace mezi oběma stranami, zbylé pak k ověření dostupnosti serveru.

I když většina protokolů transportní vrstvy disponuje poměrně robustními mechanismy pro zajištění spolehlivosti přenosu zpráv, neřeší problémy vznikající na straně serveru. Typická situace nastává v případě, že klient odešle žádost, ale neobdrží žádnou odpověď. V takových případech bývá velice obtížné stanovit přesnou příčinu a místo, kde chyba vznikla. Autor [1] popisuje možné důvody:

Klient se nemůže připojit na server – Tato situace nastává v případech, kdy se klient nedokáže připojit na server. Důvody mohou být problémy se sítí, chybně definované DNS, nebo nedostupnost serveru z důvodu údržby, odstávky, nebo výpadku. V těchto případech bývá uživateli zobrazena chybová hláška o nedostupnosti serveru.

Ztratila se zpráva s žádostí – Klient odeslal žádost na server, ale do vypršení timeout neobdržel odpověď. V této situaci je většinou automaticky znovu zaslána žádost. Pro případy, že první žádost byla správně doručena, musí server disponovat mechanismy pro detekci opakovaných žádostí a jejich ignorování. Více v kapitole Request/reply protokol.

Ztratila se zpráva s odpovědí – Volaná procedura byla vykonána a žádost odeslána. V průběhu přenosu zpět ke klientovi se však ztratila. Stejně jako v předchozím případě je klient nucen žádost zopakovat. Pokud server nedokáže tuto opakovanou žádost detekovat, vykoná proceduru opakovaně. To však může být v určitých případech nežádoucí.

Server havaroval při vykonávání žádosti – Server přijal požadavek na vykonání procedury, ale ještě před odesláním výsledku havaroval. V takových případech bývá velice obtížné stanovit, zda byla procedura vykonána (nebo alespoň její část) a server havaroval až při sestavování odpovědi, nebo ke spuštění procedury nedošlo vůbec. V takových případech by měl server umožňit návrat do stavu před uskutečněním volání.

Klient po odeslání žádosti havaruje – Klient havaruje ještě před zpraováním procedury serverem. Odpověď serveru tak nemůže být doručena a zobrazena. Takové volání popisuje [15] jako osiřelé volání (*orphaned invocation*). Server by pak měl rozhodnout, jakým způsobem bude s takovými daty dál nakládat.

V případě havárie musí server rozhodnout, jakým způsobem se budou interpretovat opakované žádosti o danou službu. Zda je možné proceduru kdykoliv bez problémů zpracovat, nebo je nutná další režie systému, aby se některé kritické operace nemohly spouštět opakovaně. Autor [1] popisuje techniky pro zajištění bezpečnosti komunikace následovně.

Technika alespoň jednou (*at-least-once*) je technika zaručující minimálně jedno zpracování zprávy. Klient vyšle žádost a pokud po vypršení timeout neobdrží odpověď ze serveru, tuto žádost periodicky opakuje dokud odpověď nedorazí. Tato technika je velice jednoduchá na implementaci, avšak často vede k mnohonásobnému vykonávání dané procedury. Je tedy vhodná pouze pro *idempotentní* operace. Tedy takové operace, při kterých opakování procedury neovlivní výsledná data. Jedná se například o operace čtení ze souboru.

Druhou technikou je technika nejvýše jednou (*at-most-once*). Ta sice nezaručuje provedení procedury, ale zároveň neumožní více než jedno volání procedury. Pokud do uplynutí časového linitu nepřijde žádná odpověď, je jeasné, že se procedura neprovedla vůbec, nebo byla spuštěna právě jednou a v jejím průběhu došlo k havárii serveru.

Poslední technikou je právě jednou (*exactly-once*), což je technika zaručuíjcí právě jedno provedení dané úlohy. Bývá realizováno jednoznačným identifikátorem každé zprávy podle kterých může server rozpoznat případné kopie a zamezit tak opakovanému volání již dokončené procedury. Tato technika je velice vhodná pro všechny neidempotentní operace jako je zápis do souboru, nebo databáze. Příkladem takových operací bývají často uváděny různé bankovní operace jako pohyby na účtech. Server musí zaručit atomicitu prováděné úlohy. Úlohy musí být dokončena celá, nebo musí být v případě chyby navráceny veštěré zdroje do původního stavu.

Autor [15] dále popisuje techniku zvanou možná (*maybe)*. Tato technika však nijak nezajišťuje provedení dané úlohy a funguje na principech best-effort.

3.5 Mechanismus zasílání zpráv

3.5.1 Request/reply protokol

Jeden z nejjednodušších způsobů komunikace dvou procesů je Request/reply protokol. Jedná se o způsob komunikace, kdy klient vyšle zprávu s žádostí (*request*) o určitou službu a server vykoná požadovanou službu a vrátí odpověď (*reply*) obsahující požadovaná data a návratovou hodnotu, která specifikuje, zda byla žádost úspěšně provedena . V opačné případě z ní lze často vyčíst i důvod neprovedení žádosti.

Samotný protokol je velmi jednoduchý, bývá implementován v transportní vrstvě a obsahuje pouze základní množinu požadavků spolu s množinou možných odpovědí.

Podle [2] mohou komunikační služby poskytované jádrem operačního systému obsahovat pouze dvě systémová volání. Jedno pro odeslání zprávy *send()* a druhé pro přijetí zprávy *receive()*.

Veškeré zprávy jsou ve výchozím stavu zasílány jako bezestavové, tj. bez předchozího navázování spojení. To umoňuje velice efektivní a rychlou komunikaci, ovšem pouze do doby, než dojde ke ztrátě zprávy. Pokud do určité doby nepřijde ze serveru odpověď, není zcela jasné, zda došlo ke ztrátě požadavku, nebo samotné odpovědi, případně zda ještě nedošlo k dokončení dané úlohy. V takovém případě většinou uživatel svou žádost zopakuje. Pokud došlo ke ztrátě odpovědi, tak opakování žádosti způsobí několikanásobné provedení dané úlohy. To může mít u určitých (neidempotentních) operací fatální následky. Této problematice se více věnuje kapitola Spolehlivost komunikace.

Z důvodu předcházení výše popsaným situacím obsahuje request/reply protokol takzvaný *discarding duplicate request messages* mechanismus. Česky lze tento název přeložit jako zahazování duplicitních žádostí. Jedná se dle [7] o mechanismus, kdy server po obdržení první žádosti o provedení úlohy ignoruje duplicitní žádosti od stejného uživatele na vykonání stejné úlohy a to obvykle do doby, než je odeslána odpověď. Tento mechanismus však stále neřeší ztrátu odpovědi.

Za tímto účelem byla navržena RRA (*Request/reply-acknowledge reply*) komunikace. K dokončení komunikace jsou potřeba tři zprávy – žádost, odpověď a potvrzovací zpráva o přijetí odpovědi. Potvrzovací zpráva obsahuje *requestId,* které bylo zasláno součástí

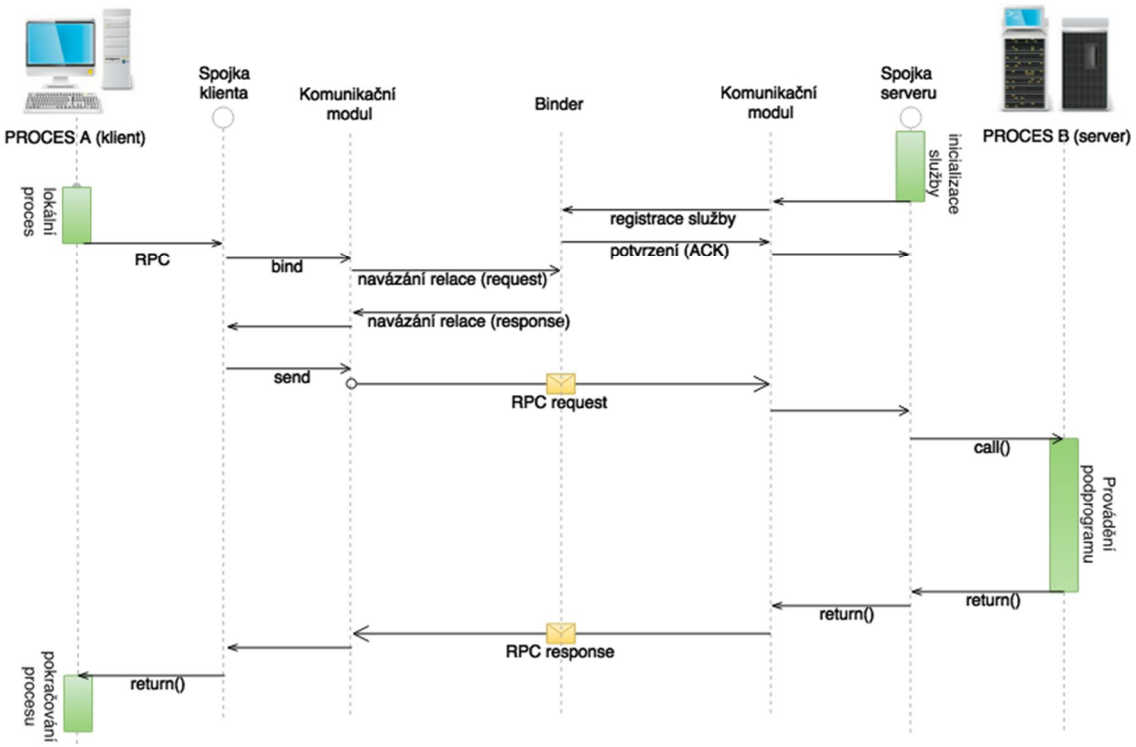
odpovědi. Přijetí requestId zároveň potvrzuje i přijetí všech zpráv s nižším requestId. Případná ztráta některé z potvrzovacích zpráv tak nehraje roli.

3.5.2 Remote Procedure Call – RPC

RPC (*Remote procedure call*) je jedním z prvních způsobů komunikace mezi dvěma procesy v distribuovaných systémech. Základní myšlenkou RPC je volání vzdálených procedur stejným způsobem, jak jej známe ze sekvenčních programovacích jazyků, kde zařízení volají podprogram za pomoci systémových volání. Ten úlohu vykoná a vrátí výsledek spolu s řízením hlavnímu programu. Celý proces komunkace je velice jednoduchý a může být efektivně realizován.

Pokud proces volá funkci, která se nachází na vzdáleném stroji za pomoci RPC, je volající proces pozastaven, veškeré parametry jsou vloženy do zprávy (*parameter marshaling*) a ta je následně doručena vzdálenému uzlu. Zde je zavolána odpovídající funkce, jejíž výsledek je opět uložen do zprávy a vrácen odesilateli. Ten po zpracování zprávy a předání parametrů obnoví řízení pozastaveného procesu a program pokračuje dál obdobným způsobem, jako by volal lokální funkci. V ideálním případě by vzdálené volání mohlo mít stejnou podobu, jako volání lokální funkce. V praxi lze však úplného odstínění všech chyb vznikajících při komunikaci dvou vzdálených systémů dosáhnout jen velmi obtížně. Dalším rozdílem je, že adresy v paměti volajícího procesu nejsou shodné s adresami volaného procesu. Předávání parametrů v podobě referencí na objekty není možné.

Systémy vzdáleného volání procedur jsou navrženy tak, aby byly nezávislé na konkrétním programovacím jazyce, nebo prostředí. Tomu musí odpovídat i použité datové typy ve zprávách. Protože různé programovací jazyky mohou mít různý popis datových typů, musí být i datové typy popisující rozhraní pro komunikaci RPC definovány tak, aby byly nazávislé na konkrétním programovacím jazyce. Jsou-li datové typy použité ve zprávách odlišné od těch lokálních, musíme mít k dispozici nástroj pro jejich konverzi. Většina RPC systémů používá automaticky generované spojky (*stubs*) jak na straně klienta tak i na straně serveru. Tyto spojky slouží jako vrstva zprostředkovávající komunikaci mezi procesem a zprávou. Má na starosti reperzentaci parametrů do zprávy a jejich následnou rekonstrukci [1]. Obrázek 5 zobrazuje RPC komunikaci mezi dvěma procesy.



Obrázek 5: Komunikace dvou procesů pomocí vzdáleného volání procedur (převzato z [51]; upraveno)

Autor [3] uvádí dva způsoby převodu mezi různými reprezentacemi.

Nejčastěji používaným způsobem je definovat standardní reprezentaci pro každý typ zprávy a požadovat po obou procesech konverzi lokálních datových typů dle tohoto standardu. Tento postup může být velmi efektivní v případě, že lokální repreznatace dat odpovídá vnitřní reprezentaci dat zprávy. V opačném případě je nutné zprávu konvertovat. Volba vhodného standardu je rovněž důležitá. Pokud by například oba procesy využívaly stejnou reprezenatci datových typů, který by se však lišil od standardu pro reprezentaci zprávy, bylo by nutné provádět konverzi na straně klienta i na staně serveru.

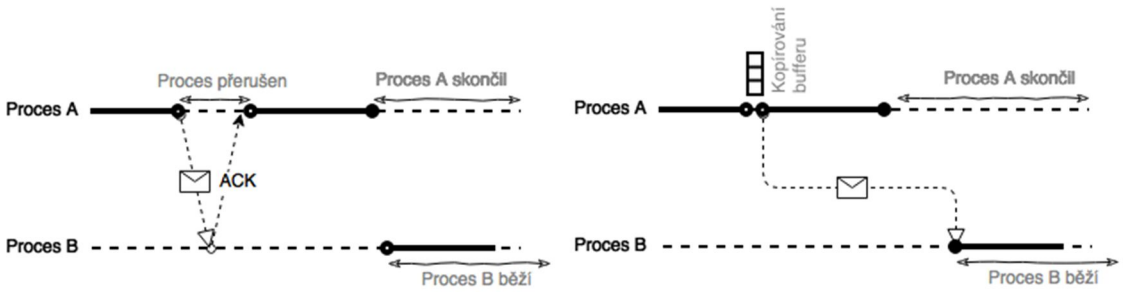
Prvním ze standardů používaných pro popis rozhraní v RPC komunikaci je Interface definition language. Postupem času však vznikalo mnoho dalších specifikačních jazyků, jako XDR od společnosti Sun Microsystems, AIDL pro RPC v systémech Android, případně protokoly založené na XML, jako SOAP a mnoho dalších, které umožňují kromě základních datových typů přenést i uživatelem definované strukturované typy.

Druhou možností je přenést data ve tvaru odpovídající lokální reprezantaci a po serveru požadovat jejich konverzi do podoby odpovídající jeho vnitřní reprezenatci. Ten by však musel umožňovat provádět libovolnou konverzi, která by se mohla v systému vyskytnout.

3.6 Synchronní /asynchronní komunikace

U synchronní komunikace je proces po dobu přenosu zprávy zablokován až do doby, než zpráva dorazí k příjemci. Veškeré další insrukce po volání procedury jsou vykovávány až v okamžiku doručení zprávy. Obdobně receive zablokuje proces B až do okamžiku obdržení zprávy.

Oproti tomu u asynchronní komunikace může proces po odeslání zprávy pokračovat v dalším výpočtu bez čekání na její doručení. Po odeslání zprávy jádro systému okopíruje zprávu do systémových bufferů, čímž se zamezí změnám zasílaných dat. Následně umožní další běh procesu. Výhodou je vyšší stupeň paralelismu. Základní rozdíly mezi synchronní a asynchronnní komunikací zobrazuje Obrázek 6.



Obrázek 6: Princip synchronní (vlevo) vs asynchronní (vpravo) komunikace

3.7 Další způsoby komunikace

3.7.1 Socket

Soket (Socket) je definován jako abstraktní rozhraní při komunikaci mezi dvojicí procesů vyvinutý společností Berkeley Software Distribution. Soket se nejčastějii používá nad transportními protokoly TCP/IP a může mít podobu klasického datagramu (datagram sockets) využívající ke komunikaci povětšinou UDP protokol, nebo datového proudu (stream sockets), které používají TCP protokol.

Komunikace procesů probíhá následujícím způsobem:

Nejprve je nezbytné na serveru specifikovat soket za pomoci série systémových volání socket(), následně funkce bind() sváže socket s lokální adresou a číslem portu na kterém bude proces komunikovat. Následujícím voláním listen() je komunikační kanál připraven naslouchat. Pokud je komunikace nespojová, je server připraven ihned přijímat pakety od klienta. U spojové komunikace je potřeba každý pokus o navázání relace nejpreve přijmout voláním accept(). Teprve poté je možné začít komunikovat. K tomu slouží volání send() a recv() Pro ukončení komunikace stačí, aby jedna ze stran ukončila spojení voláním close().

Klient se k serveru připojuje za pomoci kombinace IP adresy serveru a čísla portu. Následně se pokusí navázat spojení se serverem. Pokud dojde k potvrzení spojení, může začít proces komunikovat. Díky zdrojovému portu může server udržovat několik relací současně a obsloužit tak několik klientů v jednom okamžiku.

Číslo portu je 16 bitové číslo identifikující konkrétní aplikaci v rámci počítače. Na každém počítači tak můžeme specifikovat až 2^{16} různých portů, přičemž některé čísla portů bývají rezervována různým typům datové komunikace. Tyto čísla spravuje organizace IANA. Jedná se o autoritu, která se stará o přiřazení portů v Internetu. Dle [16] můžeme čísla portů rozdělit na tři rozsahy:

Dobře známé porty (0-1023) – porty používané obvyklými protokoly TCP/IP. Tyto porty spravuje IANA.

Registrované poty (1024-49151) – porty pro zasílání a příjem dat různých aplikací. Tato čísla portů si mohou výrobci u IANA registrovat

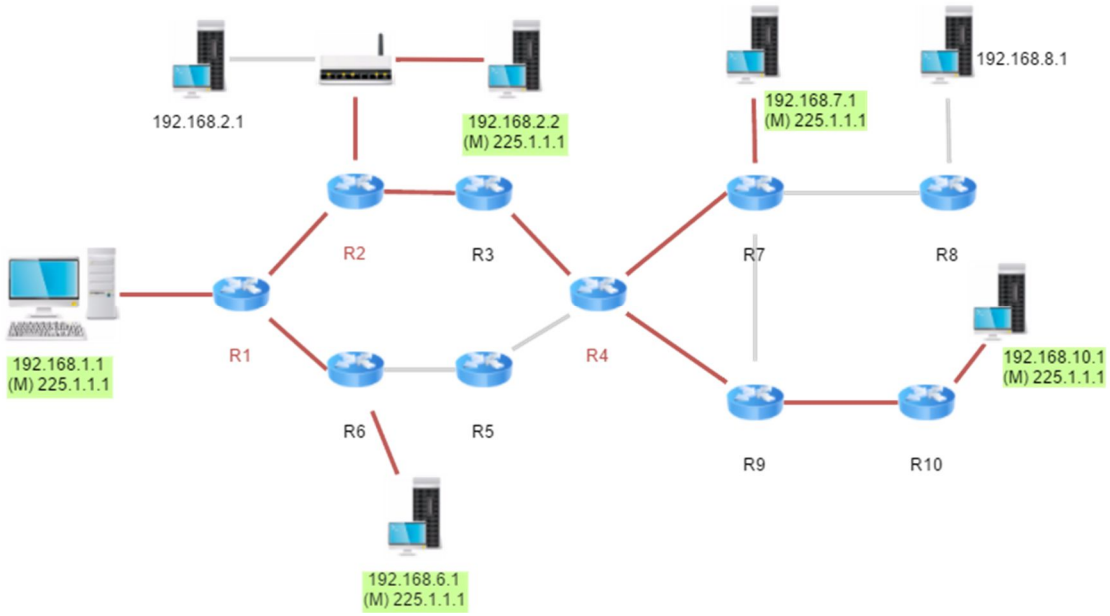
Dynamické a privátní porty (49152-65535) – čísla v této skupině nejsou nijak vyhrazena a jsou volně použitelná. Bývají dynamicky přidělována jako zdrojové porty.

3.7.2 Multicastová komunikace

Další důležitou úlohou distribuovaných systémů je zasílat zprávy několika příjemcům najednou. Zdrojový uzel přitom zasílá data pouze jednou. Následné kopírování zprávy mezi příjemce má na starosti přenosová infrastruktura. Tím dochází k výrazné redukci odesílaných dat. Takový způsob komunikace se nazývá multicastová komunikace (*multicast communication*). V minulosti bylo takové posílání správ možné pouze v rámci jedné podsítě pomocí broadcastů. Nyní je pomocí speciálních multicastových protokolů komunikace realizovatelná napříč internetem. Můžeme tak rozesílat zprávy všem procesům, nebo jen procesům v určité skupině, případně pouze vybraným skupinám procesů bez ohledu na jejich umistění v síti.

Jednou z možností, jak efektivně komunikovat s více příjemci je využít stávající protokoly IP, které podporují multicastovou komunikaci. V takovýh případech často hovoříme o *IP multicastu*.

Autor [17] popisuje způsob skupinové komunikace, kde je celá skupina příjemců identifikována jednou IP adresou. Tato IP adresa je z rozsahu D, tedy 224.0.0.0 až 239.255.255.255. Zdrojový proces odešle zprávu na danou IP adresu. Síťová infrastruktura se následně postatá o její distrubuci všem příjemcům dané skupiny. Členství ve skupině přitom může být velice dynamické. Každý ze členů skupiny se může kdykoliv odhlásit, nebo naopak přihlásit k jiné skupině. Každů z procesů může být zároveň členem několika skupin. Aby byla síť schopná identifikovat všechny příjemce, je nutné, aby se každý člen skupiny k příjmu dat zaregistroval u nejbližšího směrovače.



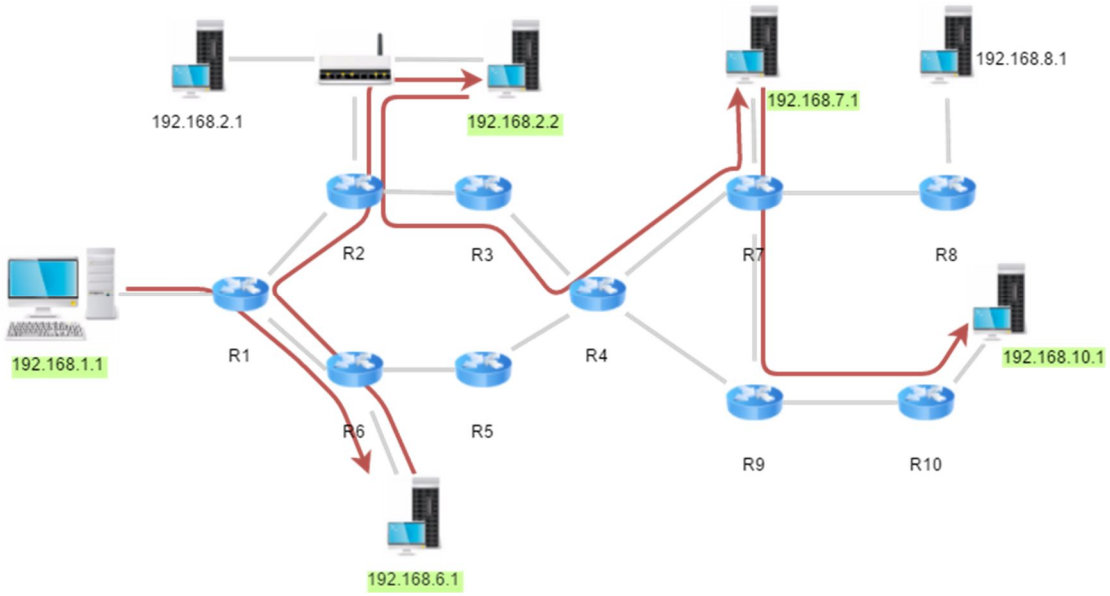
Obrázek 7: Princip IP multicast komunikace
převzato z [17] (upraveno)

K tomu musí zaslat zprávu pomocí IGMP (*Internet Group Management Protocol*) protokolu obsahující informace o členství v dané skupině. Router si tuto informaci zapíše do tabulky a informuje o členství routery v okolí. Routery pak musí umět spočítat optimální cestu k cíli a zajistit, aby každá zpráva dorazila ke každému příjemci právě jednou.

Princip IP multicastové komunikace zobrazuje Obrázek 7. Zeleně jsou znázorněny počítače ve skupině, červeně pak strom, kudy komunikace probíhá. Červeně označené směrovače udávají místa, kde je potřeba zprávu replikovat pro více příjemců.

Oproti IP multicastu existují ještě další způsoby jak komunikovat s více příjemci. Autoři [1] a [18] popisují multicastovou komunikaci implementovanou na úrovni aplikační vrstvy (*Application-Level Multicast*) jako vhodnou alternativu používanou v sítích, kde není možné využít IP multicast. Multicast na úrovni aplikační vrstvy funguje na principu vytváření virtuálních překryvných sítí (overlay networks) nad stávající síťovou infrastrukturou, ve kterých jsou zprávy replikovány a předávány jednotlivými členy dané skupiny. Oproti IP multicastu jsou tak za přenos multicastu zodpovědni sami příjemci. Předávání zprávy mezi členy je realizováno unicastově formou point to point komunikace. Nevýhodou tohoto řešení jsou vyšší nároky na šířku přenosového pásma z důvodu, že stejná zpráva je přenášena mezi členy několikrát po stejné lince.

Obrázek 8 zobrazuje princip multicastové komunikace. I když zůstala topologie sítě shodná, v porovnání s IP multicastem je jasně viditelné, že zpráva prochází některými uzly opakovaně.



Obrázek 8: Multicast na aplikační vrstvě

4 Synchronizace

V Centralizovaných systémech jsou veškeré problémy spojené se synchronizací procesů jako je přístup do kritické sekce, nebo vzájemné vyloučení procesů řešeny formou zámků, semaforů apod. Všechny tyto mechanismy však předpokládají určitou formu sdílené paměti ke které musí mít všechny interagující procesy přístup [2].

U distribuovaných systémů je synchronizace o mnohem složitější a bývá navržena různými způsoby. Synchronizace může být navržena na základě skutečného času, případně si může skupina procesů zvolit koordinátora, který bude řídit ostatní procesy.

4.1 Synchronizace logických hodin

U centralizovaných sytémů je čas naprosto jednoznačný. Pokud proces A požádá o systémový čas a krátce poté provede totéž i proces B, dostane proces B vyšší hodnotu, než proces A. U distribuovaných systémů toto není úplně jisté. V různých částech systému se může čas mírně lišit. To by způsobovalo problémy s určováním pořadí v jakém se jednotlivé události staly.

Autorři [2] a [1] popisují základní myšlenku L. Lamporta:

- Pokud se dva procesy nijak vzájemně neovlivňují, není nutné, jejich hodiny nijak synchronizovat, protože případné rozdíly by stejně nikdo nepoznal a nezpůsobí žádné problémy.
- Pokud však k takové interakci mezi procesy dochází, není nutné aby se shodly na přesném čase, ale na pořadí v jakém se jednotlivé události staly.

Uvedené definice mají znázornit fakt, že i když je synchronizace v distribuovaných systémech důležitá, nemusí být vždycky absolutní.

Pro synchronizaci logických hodin definoval Leslie Lamport relaci zvanou „předchází“ (*happens-before*). Tato relace značí, že všechy procesy se shodly na tom, že událost A se stala před událostí B. Relace předchází je dle [1] definována následovně:

- Pokud jsou události a a b události v jednom procesu a a se udála před b, potom je relace „předchází“ definována $a \rightarrow b$.
- Pokud je a událost zaslání zprávy jinému procesu a b událost přijetí zprávy, potom platí $a \rightarrow b$. Zpráva nemůže být přijata před jejím odesláním a.

Vazba předchází je *tranzitivní* tedy pokud platí, že $a \rightarrow b$ a zároveň $b \rightarrow c$, potom můžeme s jistotou říci, že $a \rightarrow c$.

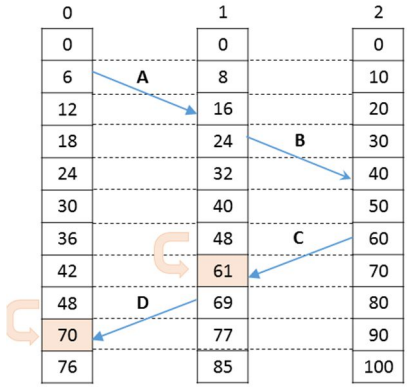
Pokud se události x a y odehrály ve dvou různých procesech, které spolu nijak nekomunikují, nelze říci, zda $x \rightarrow y$, nebo $y \rightarrow x$. Tyto dvě události jsou navzájem konkurenční a nedá se stanovit která událost předcházela druhou, ani čas, kdy se obě události staly.

Samotný algoritmus spočívá ve stanovení času C(a) na kterém se musí všechny procesy shodnout. Proces zasílá žádost spolu s časovou značkou Tm = Ci(a). Následně čeká,

než dorazí odpověď od všech ostatních procesů. Příjemce přijme zprávu v čase Cj(b). Výsledný čas Cj se stanoví jako maximum hodnot Cj(b) a Tm inkrementované o jedna podle vzorce

$$Cj = max(Cj(b), Tm + 1).$$

Následně odešle vlastní žádost s aktuální časovou značkou. Hodiny mohou být posunuty vždy pouze dopředu, nikoliv nazpět. Všechny přijaté zprávy by měly mít vyšší časovou značku. Proces synchronizace logických hodoin Lamportovým algoritmem znázorňuje Obrázek 9. Oranžová místa zorazují okamžik synchronizace hodin pomocí pozitivní zpětné vazby.



Obrázek 9: Synchronizace logických hodin - Lamportův algoritmus převzato z [1]

4.2 Synchronizace fyzických hodin

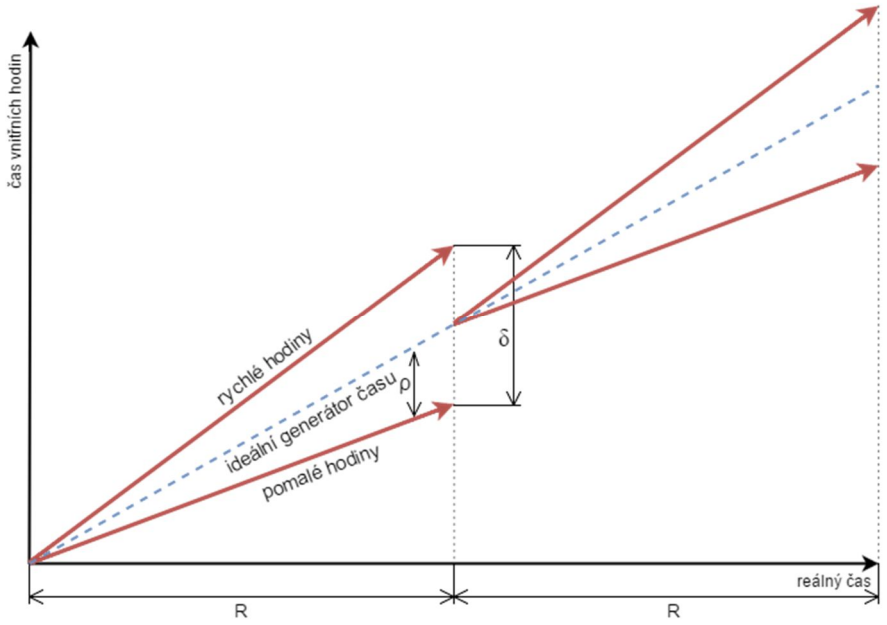
I když poskytuje Lamportův algoritmus dostatek prostředků k identifikaci pořadí, neumožňuje získat žádné informace o čase kdy se událost odehrála. To může být pro určité systémy kritické. Jedná se zejména o systémy reálného času, které již z principu své činnosti předpokládají přítomnost fyzických externích hodin. Z důvodu zajištění spolehlivosti bývá v jednom systému obvykle více než jeden generátor fyzického zdroje času. To však klade značné nároky na jejich vzájemnou synchronizaci. Ta musí v pravidelných intervalech probíhat tak, aby se hodnota generátoru nelišila od reálného času.

Má-li každý počítač vlastní vnitřní hodiny s časem *C* a zároveň zdroj skutečného času t, pak hodnota hodin na počítači *p* je *Cp(t)*. V ideálním případě, pokud by vnitřní hodiny měřily čas zcela přesně, platila by rovnost $Cp(t) = t$ pro každý okamžik *t*. Poměr $^{dC}/_{dt}$ by v takových případech byl vždy rovný jedné.

Takový ideální genreátor času je však nemožné sestrojit. Vnitřní hodiny počítačů obsahují oscilátory s výrobcem předepsanou garantovanou mírou přesnosti ρ, pro kterou v platí dle [19] vztah:

$$(1-\rho) \leq \frac{dC}{dt} \leq (1+\rho).$$

Dvoje hodiny stejného typu (stejné třídy přesnosti) se tedy mohou lišit maximálně o hodnotu $2\rho\Delta t$, kde Δt značí dobu od poslední synchronizace. Má-li být maximální rozdíl časů mezi dvěma libovolnými hodinami (*clock skew*) v systému menší, než δ, pak musí synchronizace fyzických hodin periodicky probíhat v časových úsecích R menších, než $\delta/2\rho$ [1].



Obrázek 10: Synchronizace fyzických hodin v pravidelných intervalech

Obrázek 10 znázorňuje princip synchronizace fyzických hodin v periodických intervalech R. Délka synchronizačního intervalu R závisí na maximální přípustné odchylce systému od reálného času. Jako zdroj reálného času se používá koordinovaný světový čas UTC (*Coordinated Universal Time*) s přihlédnutím k aktuálnímu časovému pásmu. Jako zdroje UTC mohou být použity časové servery NTP, nebo získáván ze systémů GPS, případně za pomoci radiového vysílání stanice DCF.

V následující kapitole budou představeny některé algoritmy pro synchronizaci fyzických hodin v distribuovaných systémech.

4.2.1 Cristianův algoritmus

Cristianův algoritmus je zástupcem algoritmů pro externín synchronizaci. Jedná se o synchronizaci, při které je pro určení přesného času použita komponenta nacházející se mimo samotný distribuovaný systém. Nejčastěji se jedná o externí časový NTP server.

Samotný algoritmus spočívá v periodickém dotazování na přesný čas v pravidelných intervalech $R < {}^{\delta}/_{2\rho}$. Časový server následně odešle odpověď s přesným časem. Pro přesnější odhad času je zohledňována i doba potřebná pro přenos odpovědi ze sereru zpět na klienta.

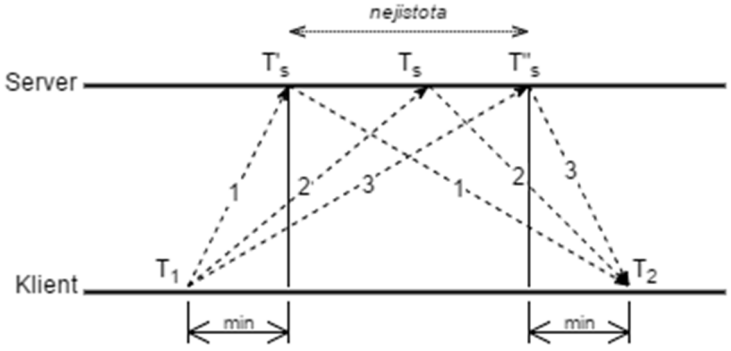
Pokud byla žádost vyslána v čase T_1 a odpověď obdržena v čase T_2, pak lze dobu mezi odesláním žádosti a přijetím odpovědi RTT (*round-trip time*) určit jako T_2-T_1. Pro urční korekce času nás ovšem zajímá pouze doba mezi sestavením zprávy serverem a jejím zpracování klientem.

Při přenosu zprávy se předpokládá, že cesta je symetrická, tedy žádost i odpověď je posílána po stejné trase a přenos žádosti i odpovědi trvají stejně dlouho. Doba pro sestavení a přenos odpovědi T_k můžeme dle [19] stanovit jako:

$$T_k = \frac{T_2 - T_1}{2}.$$

Výsledný čas T_r můžeme stanovit jako součet „aktuálního“ času zaslaného serverem T_s a vypočtené korekce T_k.

$$T_r = T_s + T_k$$



Obrázek 11: Cristianův algoritmus - chyba korekce

Z principu fungování počítačové sítě nelze v reálných podmínkách zpoždění přenosu považovat za symetrická a může docházet k nepředvídatelným prodlevám v obou směrech. To může vnášet do výpočtu korekce značnou míru nejistoty.

Na Obrázku 11 jsou jako *min* označeny minimální doby potřebné pro přenos žádosti a odpovědi. Tyto krajní případy při zachování RTT znázorňují přenosy 1 a 3. Reálná hodnota T_k se bude nacházet někde v intervalu $\langle T'_s, T''_s \rangle$. Ten můžeme určit jako

$$T_2 - T_1 - 2min.$$

I když můžeme spočítat velikost intervalu $\langle T'_s, T''_s \rangle$, ve kterém se hodnota T_k nachází, přesnou hodnotu nejsme schopni určit. V důsledku toho se odvíjí i absolutní chyba měření, která je dána právě velikostí intervalu $\langle T'_s, T''_s \rangle$. Výslednou chybu korekce lze určit jako:

$$T_r \mp \frac{(T_2 - T_1)}{2} - min.$$

Relativní chybu lze částečně eliminovat opakovaným dotazováním, přičemž se pro výpočet použije hodnota s nejnižším RTT. Vyšší přesnosti a odolnosti proti selhání lze dle [19] docílit také použitím několika časových serverů.

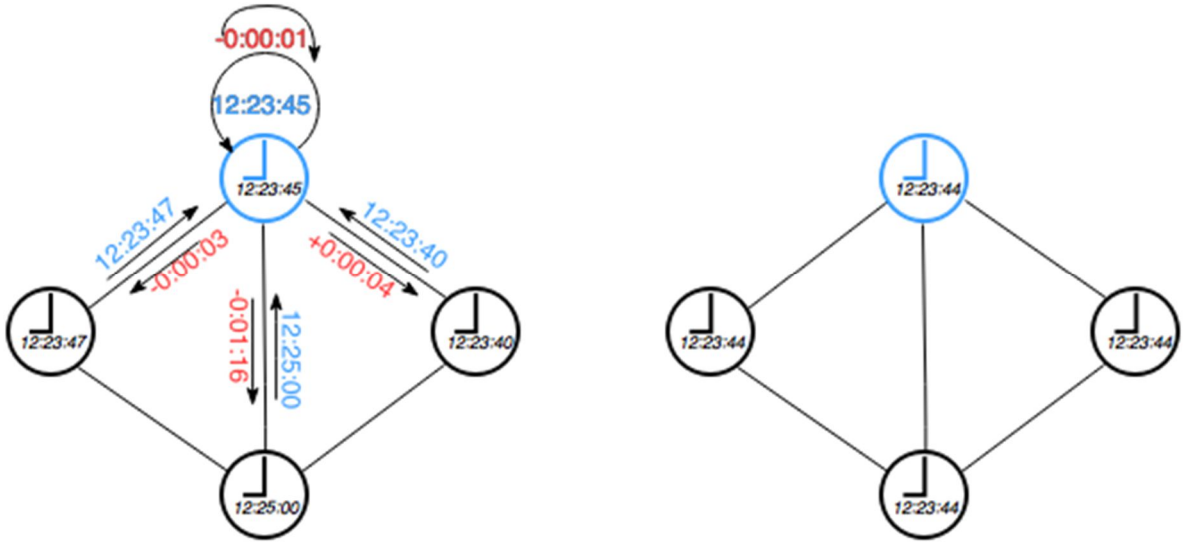
Podle [3] je rovněž nutné zajistit, aby nedocházelo k posouvání času směrem zpět. Proto má-li k takové situaci dojít, rozpočte se odchylka rovnoměrně do každého taktu až do příští synchronizace.

4.2.2 Berkeley algoritmus

Berkeley algoritmus je jedním z algoritmů pro interní synchronizaci fyzických hodin v distribuovaných systémech. Interní synchornizace nevyužívá žádné vnější zdroje času a je tak vhodná pro systémy bez přripojení k internetu, nebo pro sytémy, kde není důležité přesné určení reálného času, ale spíše udržení těsné synchronizace jednotlivých hodin. Jedním z příkladů mohou být senzorové sítě, kde bývají použity levnější oscilátory s nižší přesností a jednotlivé hodiny se tak mohou snadněji rozcházet.

Spočívá ve výběru procesu, který synchronizaci iniciuje. Tento proces následně požádá ostatní procesy o jejich systémový čas. Z nich spočítá průměrný čas v systému a všem procesům rozešle odpověď s údaji o kolik si mají posunout vlastní systémový čas, aby se shodoval s průměrným časem v systému. Vše se v pravidelných intervalech opakuje tak, aby rozdíl časů mezi libovolnými dvěma hodinami v systému nepřekročil maximální přípustnou odchylku δ. Pokud leží některá ze získaných hodnot mimo předepsané δ, je tato hodnota ingnorována a do výpočtu průměru není zahrnuta.

Důvod použití korekcí namísto určení přesného času, jaký má být nastaven je, že hodnotu korekce neovlivní zpoždění sítě jako tomu bylo v přechozím případě.



Obrázek 12: Berkeley algoritmus princip synchronizace
převzato z [19]; (upraveno)

Obrázek 12 znázorňuje princip synchronizace pomocí Berkeley algoritmu. Koordinátor je označen modře spolu s časy které od ostatních procesů obdrží. Červeně ozančené hodnoty jsou korekce rozeslané pro synchronizaci hodin v systému. Obrázek vpravo označuje stav systému po dokončení synchronizace.

4.3 Vzájemné vyloučení procesů

Přistupuje-li několik souběžně běžících procesů k jednomu sdílenému prostředku, je nutné koordinovat jejich přístup tak, aby nedocházelo k nekonzistenci dat. Část kódu ve které proces přistupuje ke sdílenému prostředku označujeme *kritickou sekcí*. Ta je vázána vždy ke konkrétnímu sdílenému prostředku. V kritické sekci sdružené s jistým prostředkem se v každém okamžiku může nacházet nejvýše jeden proces. Za kritickou sekci můžeme označit přístup ke sdílenému hardware, sdíleným datům, globálním proměnným, který může současně obsluhovat pouze jeden proces.

V distribuovaných systémech je vzájemné vyloučení řešeno výhradně prostřednictvím mechanismu zasílání zpráv. Podle [19] by vzájemné vyloučení procesů založené na zasílání zpráv mělo řešit následující úlohy:

Mějme n $(n > 1)$ procesů tvořících distribuovaný systém a ozančených čísly 0 až n-1. Topologie tvoří úplný graf, tudíž každý z procesů může přímo komunikovat s libovolným porcesem. Každý proces pravidelně požaduje přístup do kritické sekce, kde vykoná požadovanou operaci a následně kritickou sekci opustí.

Pak je třeba navrhnout protokol, který dokáže splnit následující podmínky:

- *Podmínka vzájemného vyloučení*: Pokud se proces P nachází v kritické sekci, pak žádný jiný poroces nesmí vstoupit do své kritické sekce sdružené se stejným prostředkem.
- *Prevence deadlock*: Jestliže se žádný z procesů nenachází v kritické sekci sdružené s jistým prostředkem a zároveň existuje jiný proces, který se chystá do této kritické sekce vstoupit, pak výběr procesu, který do kritické sekce vstoupí nesmí být odkládán.
- *Konečnost*: Každému z procesů musí být umožňeno vstoupit do kritické sekce v konečném čase. Každý proces, který do kritické sekce již vstoupil musí v konečném čase kritickou sekci také opustit.

Autor [1] popisuje následující postupy jak podmínky vzájemného vyloučení procesů splnit.

Distribuované algoritmy pro vzájemné vyloučení procesů můžeme rozdělit do dvou skupin.

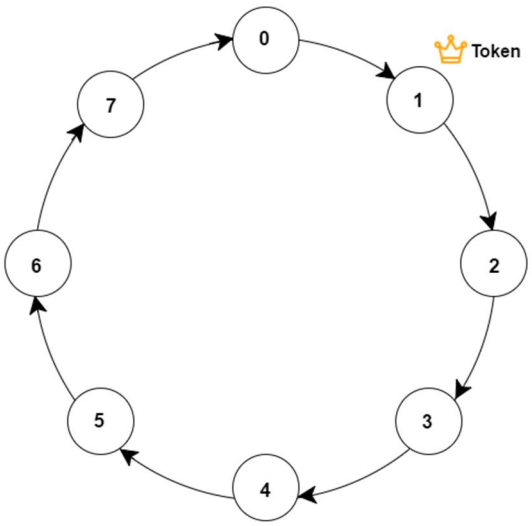
První je založena na principu zasílání speciálního typu zprávy - *tokenu* mezi procesy. V rámci jedné kritické sekce se může nacházet vždy pouze jeden token. Proces, který přijal token je oprávněn vstoupit do kritické sekce. Po návratu z kritické sekce odešle token dalšímu procesu. Pokud porces, který právě obdržel token nepožaduje vstup do kritické sekce, jednoduše jej předá dál. Následující řešení je velice jednoduché na implementaci zaručuje splnění všech tří výše uvedených podmínek. Avšak v případě ztráty tokenu, například z důvodu chyby procesu držící token, se musí poměrně komplikovaným způsobem obnovit řízení, vytvořit nový token a ověřit, zda je skutečně jediný, který se v rámci dané kritické sekce nachází.

Druhou skupinu tvoří algoritmy, které řídí vzájemné vyloučení na základě oprávnění. Pokud se proces chystá vstoupit do kritické sekce, musí nejprve požádat o přístup ostatní procesy. V tomto případě existuje mnoho způsobů, jakými lze vzájemné vyloučení zaručit. Vybrané algoritmy budou popsány v níže.

4.3.1 Token Ring algoritmus

Token Ring algoritmus poskytuje jednoduchý způsob jakým lze řídit přístup ke sdílenému prostředku. Spočívá v identifikaci množiny procesů a jejich seřazení. Na základě jejich pořadí je následně vytvořen pomyslný kruh. V něm je důležité, aby každý z procesů věděl, který proces následuje po něm. Po skončení procesu n-1 se token předává opět prvnímu procesu. Jakmile je vytvořen kruh je procesu s číslem 0 udělen token. Proces který obdrží token, může vstoupit do kritické sekce. Obrázek 13 zobrazuje 8 procesů seřazených do virtuálního kruhu. Proces 1 aktuálně obdržel token a má tak přístup ke sdílenému prostředku.

Jakmile vykoná kód kritické sekce, předá token následujícímu procesu. Ten pak může přistupovat ke sdílenému prostředku. Pokud některý proces aktuálně přístup nepotřebuje, přepošle token dál. V okamžiku, kdy žádný z procesů nepožaduje přístup ke sílenému prostředku, krouží token stále dokola. Vzhledem k pevně definovanému pořadí procesů nehrozí, že by některý z procesů dříve, či později neobdržel oprávnění k přístupu. Pokud proces požaduje přístu ke sdílenému prostředku, je mu vyhověno nejdéle po tom co n-1 procesů vykoná svůj kritický kód. Není však definováno za jak dlouho se tak stane.



Obrázek 13: Token Ring algoritmus
převzato z [1]; (upraveno)

To, že token dlouhou dobu nepřichází, ještě nemusí nutně znamenat, že se ztratil. Některý z procesů může vykovávat složitšjší kód kritické sekce a držet tak token delší dobu.

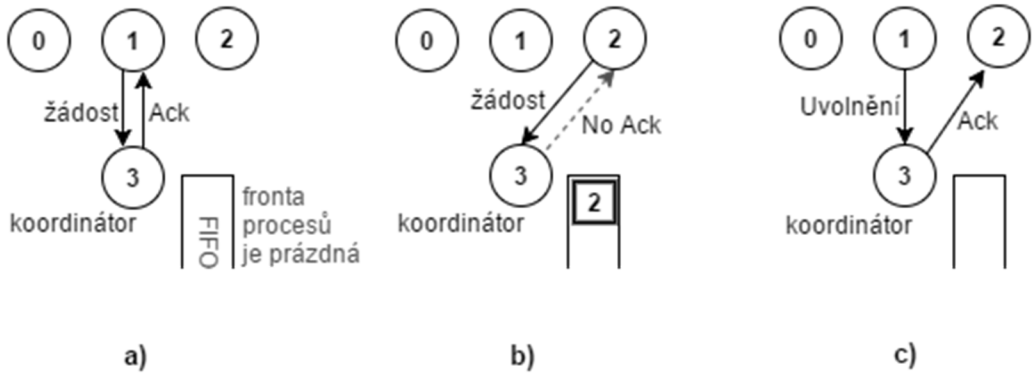
Určitým mechanismem pro detekci ztráty tokenu je potvrzování každého přijetí tokenu. V případě havárie některého z procesů nebude potvrzení zasláno. Proces je identifikován jako ukončený a může být z kruhu vyřazen [1].

4.3.2 Centralizovaný algoritmus

Dalším ze způsobů, jak řešit vzájemné vyloučení procesů je zvolit obdobný způsob jako u centralizovaných systémů. Autoři [1] a [19] popisují detailně princip algoritmu.

Veškerá oprávnění pro přístup jsou udělována jediným procesem – koordinátorem, kterého si v systému nejčastěji volí ostatní procesy. Pokud se některý z procesů požaduje přístup ke sdílenému prostředku, zašle koordinátorovi zprávu s žádostí o přístup k danému sdílenému prostředku. Ten v případě, že se již v kritické sekci nenachází jiný porces zašle potvrerní a proces může následně vstoupit do kritické sekce. Po vykonání kódu kritické sekce opět zašle koordinátorovi zprávu o uvolnění prostředku. Koordinátor zašle zpět potvrzení a proces může pokračovat ve vykonávání dalšího kódu.

Tímto mohou být splněny první dvě podmínky. Třetí podmínka konečnosti může být dle [1] splněna například implementováním FIFO fronty. Pokud proces zašle žádost v okamžiku, kdy ke sdílenému zdroji přistupuje jiný proces, koordinátor žádné potvrzení nezašle a zařadí proces do fronty. V okamžiku, kdy se sdílený prostředek uvolní zašle potvrzní s oprávněním prvnímu procesu ve frontě.



Obrázek 14: Centralizovaný algoritmus
převzato z [1]; (upraveno)

Obrázek 14 zobrazuje princip centralizovaného algoritmu pro vzájemné vyloučení procesů. Proces s číslem 1 požaduje přístup ke sdílenému prostředku. Koordinátor zasílá potvrzení s oprávněním (Obrázek 14.a) Následně se ke sdílenému prostředku pokusí připojit jiný proces. Koordinátor si jej uloží do fronty a prozatím oprávnění neuděluje (Obrázek 14.b). Potom co první porces ukončí kritický kód, zasílá koordinátorovi zprávu o navrácení zdrojů. Ten vybírá první proces ve frontě a následně mu zasílá oprávnění (Obrázek 14.c).

Jediný koordinátor v systému však tvoří slabý článek. V případě jeho selhání si sice mohou procesy zvolit nového koordinátora, ale je těžké rozlišit selhání od odmítnutí přístupu. V obou případech totiž proces požadující přístup ke sdílenému prostředku neobdrží žádnou odpověď.

4.3.3 Lamportův algoritmus

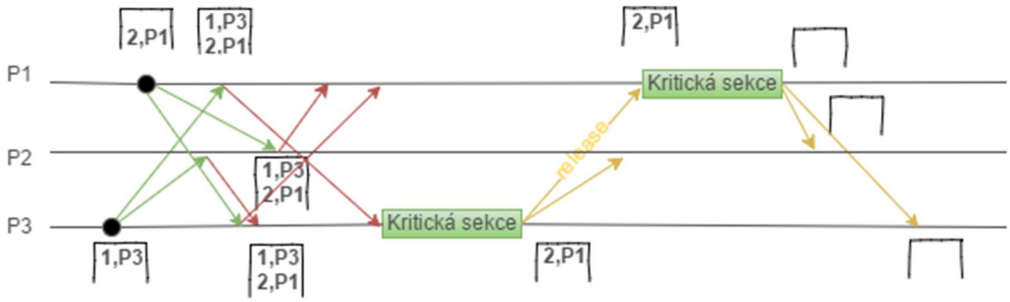
Lamportův algoritmus byl prvním z distribuovaným algoritmem používaným pro vzájemné vyloučení procesů. Později na základě jeho modifikace byly odvozeny další algoritmy pro vzájemné vyloučení procesů, jako Ricart–Agrawalův algoritmus, Maekawův algoritmus a řada dalších. Tři jmenované dále popisuje [19].

Algoritmus předpokládá úplnou topologii, prostředek komunikace mezi procesy tvoří FIFO fronty. Každý proces si udržuje svojí frontu žádostí o vstup.

Zákaladních pět pravidel pro správnou funkci algoritmu popisuje [19] následovně:

- Před vstupem do kritické sekce proces zašle všem ostatním procesům žádost obsahující časovou značku. Zároveň si žádost zařadí do vlastní fronty žádostí o vstup.
- Pokud proces od libovolného dalšího procesu obdrží žádost o vstup do kritické sekce, zkontroluje, zda se již ve frontě nenachází a pokud ne, přidá jej do své fronty. Pokud se proces nenachází v kritické sekci, zašle zpět potvrzení. V opačném případě zašle potvrzení až po dokončení kritické sekce.
- Proces vstupuje do kritické sekce, pokud se ve frontě nachází na první pozici. To znamená, že časová značka žádosti je ze všech nejnižší. A zároveň obdržel potvrzrní od všech procesů.
- Po dokončení kódu kritické sekce si proces odstraní žádost ze své fronty a zašle všem ostatním procesům zprávu o uvolnění zdroje.
- Pokud proces obdrží zprávu o uvolění zdroje, odstraní si proces ze své fronty.

Díky zasílání časových značek je zaručeno, že se každý proces dostane ke sdílenému prostředku. V okamžiku, kdy je žádost potvrzena všemi procesy (proces zaslal $n-1$ žádostí a přijal $n-1$ potvrzení) jsou všechny fronty identické. Vstup do kritické sekce tak může být posuzován na základě lokální fronty žádostí. Po vykonání kritické sekce musí proces odeslat všem ostatním procesům oznámení o uvolnění zdroje. Počet všech zpráv nutných pro vstup jednoho procesu ke sdílenému prostředku je $3(n-1)$.



Obrázek 15: Lapmportův algoritmus pro vzájemné vyloučení procesů

Obrázek 15 znázorňuje komunikaci mezi procesy před vstupem do kritickách sekcí. Znázorněné fronty zobrazují časovou značku a název procesu.

5 Distribuované databáze

Distribuované databáze můžeme definovat jako soubor databází, které patří do jednoho systému, ale fyzicky jsou rozprostřeny do několika oddělených lokalit. Veškerá distribuce dat probíhá v rámci počítačové sítě. V každém uzlu sítě se nachází vlastní systém řízení báze dat (SŘBD), který zajišťuje distribuci a konzistenci dat, správu transakcí a mnoho dalšího (bude detailně představeno níže).

Napříč literaturou můžeme najít mnoho různých decinic. Například [20] definuje distribuované databáze následovně:

We define a distributed database as a collection of multiple, logically interrelated databases distributed over a computer network. A distributed database management system (distributed DBMS) is then defined as the software system that permits the management of the distributed database and makes the distribution transparent to the users.

(Özsu, 2011, p. 3)

Oproti cenralizovaným databázím tak nabízí většinu výhod představených v distribuovaných systémech jako transparentnost, rozšiřitelnost, výkonost, škálovatelnost a v neposlední řadě odolnost proti výpadkům určitých uzlů. V případě vhodně zvolené replikace mohou být chybějící data získána z ostatních uzlů.

V případech, kdy jsou procesory fyzicky umístěny pohromadě, hovoříme o paralelních architekturách, respektive o *paralelních databázových systémech*. Takto navržené databáze mají zajišťovat vyšší výkon, nikoliv však centralizaci více databází. Jedná se tedy spíše o snahu vytvoření výkonnější centralizované databáze.

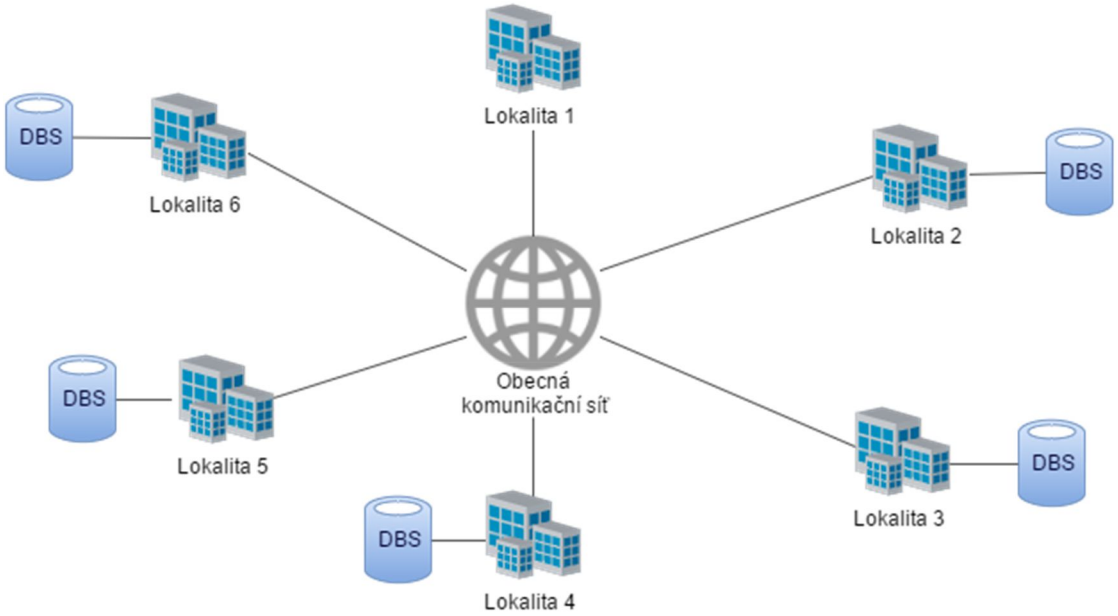
Distribuované databázové systémy vycházejí z potřeby nabídnout lokálně autonomní systémy s možností globálního či integrovaného pohledu na distribuovaná data. V distribuovaných databázích se dotazy a další požadavky na manipulaci s daty provádějí nezávisle v různých místech a částečné výsledky na základě komunikace mezi místy přispívají k vytvoření celkového výsledku.

(Klimeš, 2007, p. 104)

I když se jednotlivé distribuované databáze mohou ve svých implementacích výrazně lišit, [21] popisuje společné funkce které by měla každá distribuovaná databáze zajišťovat.

Transparentní distribuce dat – Uživatel by musí být schopen se systémem pracovat stejně jako by pracoval s lokálním databázovým systémem. Veškerá distribuce dat musí být uživateli skryta a neměla by mít na koncového uživatele negativní dopad.

Transparentnost transakcí – Každá transakce musí zajišťovat integritu dat napříč databázovými uzly. Každá transakce může být rozdělena do subtransakcí. Každá subtransakce ovlivňuje jeden databázový uzel.



Obrázek 16: Schéma distribuovaných systémů

I když každý distribuovaný databázový systém obsahuje několik systémů pro řízení báze dat, které mohou být specifické pro každou lokalitu, Každá lokalita obsahuje vlastní databázi s částmi (fragmenty) dat, které logicky patří do jednoho systému. Je tak schopna zpracovávat úlohy vyžadující přístup pouze k lokálním datům. Distribuovaný systém pro řízení báze dat spojuje tyto fragmenty do jedné logické databáze a vytváří tak komplexní pohled na data nezávisle na jejich umístění. Umožňuje tak zpracovávat dotazy napříč databázovými uzly.

Obrázek 16 zobrazuje prostředí distribuovaného databázového systému. SŘBD umístěné v různých lokalitách se mohou navzájem lišit. To může být způsobeno například snahou postupně integrovat několik stávajících systémů. V takových případech mluvíme o *heterogenních* databázových systémech.

5.1 Fragmentace

V distribuovaných databázových systémech bývají relace velmi často rozděleny do více pod-relací (fragmentů), které jsou následně distribuovány do různých lokalit. Cílem je logicky uspořádat data podle lokalit, ke kterým jsou vztažena. Aplikace využívající lokální data mohou využívat DDBS dané lokality. Výhody fragmentace popisuje například [21].

Fragmentace musí být provedena tak, aby nedocházelo ke ztrátám, nebo naopak zbytečnému vícenásobnému zápisu dat. [21] popisuje tři základní pravidla pro zajištění fragmentace:

Úplnost – Je-li relace R rozdělena na fragmenty R_1, R_2,… R_n, pak libovolný prvek z R se musí nacházet v alespoň jednom z fragmentů. Pravidlo zajišťuje, že v průběhu fragmentace nedojde ke ztrátě dat. Platí:

$$R_1 \cup R_2 \cup \ldots \cup R_n = I$$

Rekonstruovatelnost – Pokud je relace R rozdělana na fragmrnty R_1, R_2,… R_n, pak musí existovat relační operace pro jejich opětovnou rekonstrukci do relace R. Pravidlo zajišťuje opětovné spojení fragmentů do původní relace. V relačních databázích pro tuto operaci bývají definovány monožinové operace sjednocení, nebo spojení.

$$R = R_1 \cup R_2 \cup \ldots \cup R_n$$

Disjunktnost – Je-li relace R rozdělana na fragmenty R_1, R_2,… R_n, potom by se prvek dat z R_i neměl objevit v žádném jiném fragmentu. Toto pravidlo zajišťuje minimální redundanci. Výjimku tvoří případ vertikální frakgmentace, kde je nutné opakování primárního klíče z důvodu opětovné rekonstruovatelnosti.

$$R_1 \cap R_2 \cap \ldots \cap R_n = \emptyset$$

5.1.1 Typy fragmentace

Ke správnému rozdělení databáze na fragmenty je třeba provést důkladnou analýzu. Výslednou podobu rozdělení databáze může ovlivnit celá řada faktorů, jako logická organizace databáze, místo a frekvence spouštěných aplikací, výkon transakcí a aplikací, propustnost počítačové sítě případně frekvence a typy přístupu k datům. Mluvíme o tzv. kvantitativních a kvalitativních požadavcích na konstrukci databáze. Pro zjednodušení neuvažujeme v popisovaných příkladech žádný ze způsobů replikace dat. Předpokládáme tedy, že každý fragment dat je uložen právě v jedné lokalitě.

V následující kapitole budou představeny základní druhy fragmentace dat. Existují dva základní způsoby, jakými mohou být data fragmentována. Podle způsobu rozdělení dat

můžeme rozlišit vertikální a horizontální fragmentaci. Případně můžeme využít kombinace obou typů fragmentace, kdy je jedna část relace fragmentována vertikálně a druhá horizontálně. V takových případech mluvíme o fragmentaci smíšené.

Horizontální fragmentace

U *horizontální fragmentace* dochází k rozdělování relací na základě podmínky jednoho, nebo několika atributů v relaci. Výsledná n-tice tvořící fragment je dána selekcí $\sigma_P(R)$ na základě stanovených podmínek. Podmínky P_1, P_2, P_n musí zahrnovat všechny prvky relace R. Každy z fragmentů může být uložen v jiné lokalitě.

Mějme tabulku dodavatelů obsahující jména a města ve kterých sídlí (viz Tabulka 3.a). Pro jejich rozdělení do horizontálních fragmentů podle umistění poboček dané firmy můžeme aplikovat následující podmínky:

$$Dodavatel1 = \sigma_{mesto=Hradec\ Králové}\ Dodavatel$$
$$Dodavatel2 = \sigma_{mesto=Pardubice}\ Dodavatel$$
$$Dodavatel3 = \sigma_{mesto\neq Hradec\ Králové \wedge mesto\neq Pardubice}\ Dodavatel$$

Původní relaci R získáme opětovným sjednocením jednotlivých relací:

$$R = Dodavatel1 \cup Dodavatel2 \cup Dodavatel3.$$

Výsledkem budou tři fragmenty seskupené podle měst. Poslední pravidlo je voleno tak, aby byla splněna podmínka úplnosti. Tabulka 7.b zobrazuje výsledné fragmenty D_1 až D_3. Uvažujme vazbu 1:N mezi dodavatelem a tabulku zboží. Potom by dotaz na veškerý sortiment dostupný pro lokalitu Hradec Kálové mohl vypadat následovně:

$$zbozi_{HK} = \sigma\,(Dodavatel1 * zbozi)$$

Dodavatel

id	**nazev**	**mesto**
1	Novák	Hradec Králové
2	ABC print	Pardubice
3	Váňa	Hradec Králové
4	Slavíček a syn	Praha
5	FF service	Písek
6	AB comouters	Pardubice

a)

Dodavatel

	id	**nazev**	**Město**
D_1	1	Novák	Hradec Králové
	3	Váňa	Hradec Králové
D_2	2	ABC print	Pardubice
	6	AB comouters	Pardubice
D_3	5	FF service	Písek
	4	Slavíček a syn	Praha

b)

Tabulka 3: příklad horizontální fragmentace

Výhodou horizontální fragmentace je, že monoho aplikací využívá pouze lokální data z jednoho fragmentu. Fragmentace by měla být volena tak, aby docházelo k minimalizaci přenášených dat a vykonávání dotazů bylo maximálně efektivní.

Vertikální fragmentace

U *Vertikální fragmentace* dochází k rozdělování relací na základě projekce jednoho, nebo více atributů v relaci. Výsledné fragmenty jsou dány projekcí $\pi_{a1,a2,\dots an}(R)$, kde $a1, a2, \dots an$ tvoří atributy relace R spolu s primárním klíčem pro jejich opětovnou rekonstruovatelnost do podoby původní relace.

Mějme tabulku zaměstnanců, která obsahuje atributy s osobními údaji, pracovním zařazení a platu (viz Tabulka 4). Vertikální fragmentaci můžeme provést tak, aby první z fragmentů obsahoval osobní údaje zaměstanaců (viz Tabulka 4) a druhý informace o jejich pozici a aktuální výši platu (viz Tabulka 4). Pro rozdělení do fragmentů použijeme následující projekci:

$$Z1 = \pi_{osCislo,jmeno,prijmeni,datumNarozeni}(Zamestnanec)$$
$$Z2 = \pi_{osCislo,cPozice,plat,cPobocky}(Zamestnanec)$$

Původní relaci získáme přirozeným spojením $Z1 \bowtie Z2$.

Zamestnanec

osCislo	**jmeno**	**prijmeni**	**datumNarozeni**	**cPozice**	**plat**	**cisloPobocky**
604005	Petr	Pavel	31.03.1976	400	18100	65
839530	Lucie	Konečná	11.07.1964	198	19900	148
156039	Tomáš	Malý	31.08.1986	198	40000	37
655805	Libuše	Šťastná	08.08.1981	165	26100	70
644017	Jan	Novák	04.12.1962	347	19200	110
17752	Josef	Veselý	02.09.1977	490	18000	122

a)

Z1

Zamestnanec

osCislo	**jmeno**	**prijmeni**	**datumNarozeni**
604005	Petr	Pavel	31.03.1976
839530	Lucie	Konečná	11.07.1964
156039	Tomáš	Malý	31.08.1986
655805	Libuše	Šťastná	08.08.1981
644017	Jan	Novák	04.12.1962
17752	Josef	Veselý	02.09.1977

b)

Z2

Zamestnanec

osCislo	**cPozice**	**plat**	**cPobocky**
604005	400	18100	65
839530	198	19900	148
156039	198	40000	37
655805	165	26100	70
644017	347	19200	110
17752	490	18000	122

c)

Tabulka 4: Příklad vertikální fragmentace

V případě že fragmenty $Z1, Z2$ ... Zn obsahují všechny atributy původní relace R, mluvíme o tzv. *kompletní vertikální fragmentaci*.

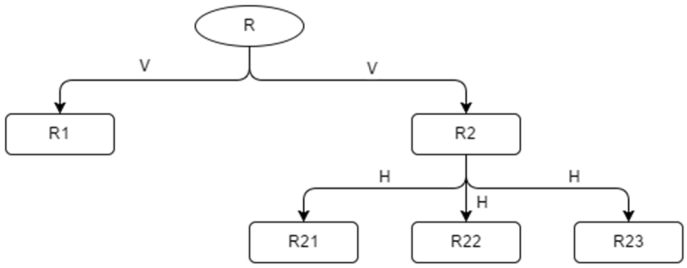
Platí zde:

$$Z1 \cup Z2 \cup \ldots \cup Zn = Attr(R)$$
$$Z1 \cap Z2 \ldots \cup Zn = PK(R),$$

kde $Attr(R)$ je množina všech atributů R a $PK(R)$ je primární klíč relace R.
Pro rekonstrukci původní relace R z fragmentů kompletní vertikální fragmentace je třeba aplikovat operaci přirozeného spojení NATURAL JOIN.

Smíšená fragmentace

Smíšená fragmentace je kombinace vertikální a horizontální fragmentace. Vzniká v případech kdy jsou některé fragmenty vertikální fragmentace dále rozděleny do horizontálních fragmentů, nebo naopak. Vzniká postupnou aplikací selekcí σ_{Ri} a projekcí π_{Ri} na původní relaci R. Pořadí aplikace relačních operátorů bývá často znázorňováno do stromové struktury. Jednu z možností znázorňuje Obrázek 17.



Obrázek 17: Stromová struktura smíšené fragmentace

Uvažujme tabulku zaměstnanců z předchozího příkladu (Tabulka 4: Příklad vertikální fragmentace) Pokud její data aplikujeme na stromovou strukturu znázorněnou na Obrázku 17, můžeme jednotlivé pod-relace definovat následovně:

$$R1 = \pi_{osCislo,jmeno,prijmeni,datumNarozeni}(Zamestnanec)$$
$$R2 = \pi_{osCislo,cPozice,plat,cPobocky}(Zamestnanec)$$
$$R21 = \sigma_{cPobocky \leq 50}\left(\pi_{osCislo,cPozice,plat,cPobocky}(Zamestnanec)\right)$$
$$R22 = \sigma_{cPobocky > 50 \wedge cPobocky \leq 100}\left(\pi_{osCislo,cPozice,plat,cPobocky}(Zamestnanec)\right)$$
$$R23 = \sigma_{cPobocky \leq 150}\left(\pi_{osCislo,cPozice,plat,cPobocky}(Zamestnanec)\right)$$

Původní relaci pak získáme postupným aplikováním operací spojení a sjednocení:

$$R = R1 \bowtie R2, \text{ kde}$$
$$R2 = R21 \cup R22 \cup R23$$

R1

Zamestnanec			
osCislo	**jmeno**	**prijmeni**	**datumNarozeni**
604005	Petr	Pavel	31.03.1976
839530	Lucie	Konečná	11.07.1964
156039	Tomáš	Malý	31.08.1986
655805	Libuše	Šťastná	08.08.1981
644017	Jan	Novák	04.12.1962
17752	Josef	Veselý	02.09.1977

R2

	Zamestnanec			
	osCislo	**cPozice**	**plat**	**cPobocky**
R21	156039	198	40000	37
R22	604005	400	18100	65
	655805	165	26100	70
R23	644017	347	19200	110
	17752	490	18000	122
	839530	198	19900	148

Tabulka 5: Smíšená fragmentace- rozložení jednotlivých fragmentů

Tabulka 5 zobrazuje rozložení jednotlivých fragmentů dle stromové struktury znázorněné na Obrázku 17.

5.2 Transakce

V následující kapitole budou popsány základní principy a vlastnosti distribuovaných transakcí. [21] definuje pojem transakce následovně:

A transaction consists of a set of operations that represent a single logical unit of work in a database system and moves the database from one consistent state to another.

(Chhanda, 2009, p. 104)

V databázích je základním prostředkem pro přístup k datům správná formulace dotazu v příslušném jazyce. V případě, že ke stejným datům přistupuje několik zdrojů naráz, musí systém zajistit, aby nedocházelo k nekonzistenci dat. Dále je potřeba rozlišovat, zda jsou data požadována pro čtení, nebo pro zápis. Zatímco při čtení hodnoty může přistupovat k jedné proměnné několik procesů najednou, v případě zápisu je nutné zajistit, aby nedocházelo k nekonzistenci dat. Použití transakcí zajistí, že uživatelé mohou přistupovat pouze ke konzistentním datům. Transakce zajišťují nejen konzistenci dat, ale také bezpečnost, tj že se kód volaný v transakci provede, celý nebo v případě chyby budou všechny proměnné navráceny do původního stavu před započetím transakce. [22] popisuje veškeré dění od začátku transakce po její dokončení jako nedělitelné. Až po dokončení transakce jsou dle výsledku transakce promítnuty výstupy do databáze. Případné selhání transakce neovlivní data vůbec. Pro zajištění nedělitelnosti musí dle [22] transakce splňovat následující vlastnosti (někdy známé jako ACID):

atomicita – zajišťuje, že transakce je považována za atomickou, tedy dojde k jejímu úplnému provedení, nebo se neprovede žádný z příkazů obsažených v transakci. Uživatel musí být informován o stavu transakce.

konzistence – před započetím transakce i po jejím dokončení musí být všechna data konzistentní. V průběhu provádění transakce se mohou (a často se tomu tak dle [20] děje) některá data nacházet v nekonzistentním stavu. Tato podmínka je nezbytná pro čtvrté pravidlo.

izolovanost - probíhající transakce musí probíhat skrytě před ostatními konkurenčními transakcemi. Operace probíhající uvnitř transakce probíhají mimo fyzickou databázi a jsou pro okolí neviditelné.

trvanlivost – V okamžiku potvrzení transakce jsou změny dat bezpečně uloženy a nemohou být ztraceny.

I když výše uvedená pravidla vycházejí z centralizovaných transakcí, platí i pro distribuované transakce, avšak v distribuovaném prostředí vyžaduje splnění všech podmínek komplikovanější algoritmy, vzhledem ke fragmentaci a případné replikaci dat.

V distribuovaných systémech můžeme distribuované transakce rozdělit podle dat se kterými pracují. Pokud může být transakce provedena pouze z lokálních zdrojů dané lokality, mluvíme o *lokální transakci*. Naproti tomu pokud jsou pro transakci vyžadována data napříč jednotlivými lokalitami, mluvíme o *transakci globální* [21]. Distribuovaný SŘBD musí zajistit atomicitu globálních transakcí, stejně jako transakcí v rámci dané lokality.

Autor [20] dále rozděluje transakce podle jejich struktury na Ploché transakce (*flat transactions*), Vnořené, nebo dle [9] také Hnízděné transakce (*nested transactions*), a Workflow modely (*workflows*),

5.2.1 Flat Transactions

Ploché transakce (jak je překládá např. [23]) mají jeden počátek transakce (*Transaction Begin*) a jeden konečný bod transakce (*Tranasaction End*). Pro jejich jednoduchost převládají ve většině vykonávaných transakcí právě ploché transakce. Svou povahou se však nehodí pro rozsáhlé operace s daty. S rostoucím počtem prováděných operací v transakci roste i pravděpodobnost selhání některé z nich. V takovém případě dochází k selhání celé transakce a navrácení všech vstupů do předchozího konzistentního stavu. Autor [23] dále uvádí, že ploché transakce nemohou spolupracovat s jinými transakcemi a nejsou tolerovány dílčí neúspěchy, a to ani v případech, kdy by neměly vliv na ostatní operace.

5.2.2 Nested Transactions

Vnořené transakce tvoří jistou alternativu částečně eliminující nevýhody flat transakcí. Tyto transakce vkládají vlastní počáteční a koncové body (begin a end) do těla vnější transakce. Takto ohraničené vnitřní transakce jsou často označovány jako sub-transakce, nebo dílčí transakce. Vkládáním dílčích transakcí do těla hlavní transakce vzniká hierarchická struktura, kterou můžeme vyjádřit jako strom transakcí, obsahující jednu kořenovou tranaskci a několik vnořených transakcí. Listy stromu jsou tvořeny vždy flat transakcemi. Rodičovskou transakcí nazývýme naopak transakci, do které je daná transakce vnořená. Takoto vnořenou tansakci nazýváme dceřinou transakcí (child-transaction). Všechny trasakce v hierarchii mají svou rodičovskou transakci. Výjimkou je pouze kořenová transakce, která rodiče nemá.

Každá dílčí transakce může provést commit nebo abort. Potvrzení dílčí transakce však nemá vliv na výsledný stav rodičovské transakce. Ta má po potvrzení k dispozici dílčí výsledky, ketré však nejsou rodičovským transakcím dostupné dokud tato transakce sama výsledky nepotvrdí. Výsledná data se promítnou do databáze až v okamžiku potvrzení kořenové transakce. Rollback vnořené transakce vyvolá i rollback všech potomků. Tato pravidlo je však v rozporu s vlastnostmi transakcí, konkrétně s posledním pravidlem o trvanlivosti dat. Proto splňují vnořené transakce pouze vlastnosti atomicita, konzistence a izolovanost.

Výhodou tohoto modelu je, že zdroj použitý v subtransakci může být ihned po jejím dokončení alokován jiným zdrojem, který tak nemusí čekat na dokončení celé transakce. Další výhodou je snažší zotavení transakcí v případě poruchy systému, kdy může být zotavena pouze určitá dílčí transakce, namísto zotavení celého stromu transakcí.

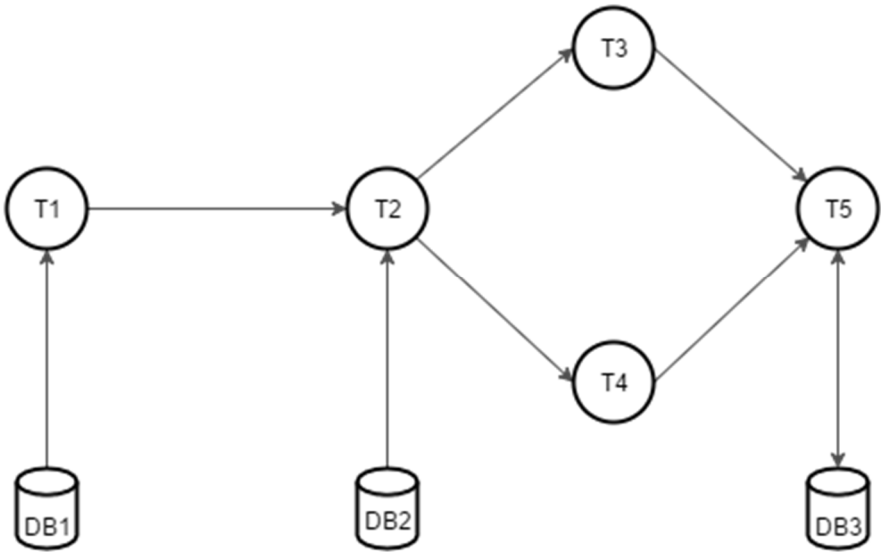
5.2.3 Workflows

Poslední kategorii transakcí tvoří workflow. Dle [21] je definován jako kolekce úloh nezbytných pro vykonání určitého business procesu. Podle [24] můžeme workflow dále rozdělit na:

Postupy orientované na lidi (*Human-oriented workflows*) – jedná se o postupy pro splupráci a předávání řízení mezi lidmi. V konečném důsledku však za splnění všech úkolů zodpovídají opět lidé.

Systémově orientované postupy (*System-oriented workflows*) – jedná se o postupy pro podporu specializovaných úloh, které lze provádět pomocí počítače, jako je kontrola souběžnosti a zotavení, automatické spuštění úlohy a oznámení.

Transakční postupy (*Transactional workflows*) - Jde o kombinaci obou výše zmíněných postupů, přičemž poskytují podporu pro koordinaci vykonávání více úloh, přičemž využívají transakčních vlastností pro jednotlivé úkoly nebo celé pracovní postupy. Mezi hlavní rysy transakčních postupů je důraz na transakčnost a tedy i splnění všech vlastností transakcí (viz kapitola Transakce5.2).



Obrázek 18: Příklad workflow;
převzato z [20]

Obrázek 18 zobrazuje příklad obecného postupu. Úlohy T1 a T2 označují činnost při které jsou jednotlivé vstupní údaje načítány z databází. Úlohy T3 a T4 jsou prováděny paralelně. Ke spuštění úlohy T5 dojde až v okamžiku dokončení obou předchozích úloh.

5.3 Dvoufázový potvrzovací protokol

Dvoufázový potvrzovací protokol (*Two-phase commit protocol*) je velice jednoduchý, ale efektivní způsob, jak zajistit atomicitu distribuovaných transakcí. Spočívá v ověřování, zda se všechny lokality zapojené do realizace tranaskce shodly na úspěšném potvrzení transakce ještě před tím, než jsou změny trvale uloženy. Existuje celá řada variant dvoufázového potvrzování transakcí, některé popisuje [20], [21], [9], případně [25]. Liší se především ve způsobech volby koordinátora, množstvím zaslaných potvrzovacích zpráv, nebo stanovováním výsledků transakce.

Předpokládejme řízení distribuované transakce koordinátorem, přičemž uzly, na nichž jsou uloženy příslušné fragmenty dat a podílející se na distribuované transakci nazývejme účastníky transakce. Protokol používá k ověření každé distribuované transakce dvě fáze.

První fáze známá jako hlasovací (*voting phase*), a druhá rozhodovací (*decision phase*). Obě fáze detailně popisuje [21].

V první fázi zašle koordinátor všem účastníkům transakce zprávu Prepare, ve které se koordinátor ptá všech účastníků, zda byla jejich lokální transakce úspěšná a souhlasí s jejím potvrzením. Každý z účastníků může kdykoliv požadovat zrušení transakce, pokud ji již předtím nepotvrdil. Pokud některý z účastníků požaduje zrušení transakce, nebo neodpoví ve stanoveném čase, Koordinátor rozešle všem účastníkům pokyn k jejímu zrušení. Naopak, pokud všichni zúčastnění odsouhlasí potvrzení svých lokálních transakcí, koordinátor zašle pokyn k potvrzení celé transakce. Pokud účastník transakce odpoví pozitivně, změní svůj stav na čekající a zablokuje všechny objekty související s transakcí (viz kapitola Zamykání). Následně čeká na pokyn koordinátora o provedení transakce (viz druhá fáze). V opačném případě zruší lokální transakci a uvolní příslušné objekty.

Ve fázi rozhodování se na základě obdržených odpovědí z předchozí fáze rozhoduje o tom, zda bude transakce zrušena, nebo potvrzena. Transakce může být potvrzrna pouze v případě, že byly potvrzrny všechny její lokální části. Pokud tedy koordinátor obdržel od všech účastníků souhlas s provedením transakce, rozešle commit všem uzlům. Účastníci následně provedou odpovídající operaci.

Zavedením časového limitu se zamezí situacím, kdy by došlo ke ztátě zprávy od některého z účastníků, nebo naopak od koordinátora k některému z účastníků. Veškeré odpovědi od účastníků, kteří do stanoveného časového limitu nepotvrdily své transakce, jsou automaticky považovány za negativní.

5.4 Zamykání

5.4.1 Problém konkurentního přístupu

Jestliže má být přistupováno ke sdílenému objektu, je potřeba zabezpečit korektní chování transakčního sytému. Nejčastějším problémem, který v databázích může nastat je, že dvě transakce přistupující ke stejným datům si budou tato data navzájem přepisovat. Názorný příklad zobrazuje Tabulka 6:

#	*Transakce I.*
1	local ← B
2	local ← local + $250
3	B ← local

#	*Transakce II.*
4	local ← B
5	local ← local + $250
6	B ← local

Tabulka 6: Problém konkurentního provádění transakcí; převzato z [19]

Tabulka znázorňuje dvě konkurentní trasakce I a II. Předpokládejme, že objekt B má před započetím transakcí hodnotu $B = \$1000$. Pokud budou transakce prováděny postupně, v pořadí (1 2 3 4 5 6), nebo případně v opačném pořadí (4 5 6 1 2 3) bude po dokončení obou transakcí hodnot proměnné $B = \$1500$.

Z důvodu vyšší propustnosti systému však bývá většina transakcí prováděna paraleleně. Pokud dojde k vykonání transakcí v pořadích (1 4 2 5 3 6), případně (1 2 4 5 6 3), bude výsledek po vykonní obou transakcí pouze $B = \$1250$.

Tento problém, kdy obě transakce přičítají hodnotu ke stejnému zdroji (účtu), přičemž obě transakce zakládají své výpočty na základě shodné hodnoty, bývá dle [26] označován jako problém ztracené aktualizace (*lost update problem*). V takových případech dochází k porušení konzistence databáze.

Za tímto účelem jsou zaváděny mechanismy pro serializovatelné provádění transakcí. Výsledek paralelního provádění transakcí musí být stejný, jako kdyby byly transakce prováděny sériově, tedy jedna po druhé. Zároveň nesmí paralelní zpracování vést k nižší průchodnosti systému než v případě seriového zpracování [19]. Jedním ze způsobů, jak takového stavu dosáhnout je zamykání objektů za pomoci zámků.

5.4.2 Zámky

Zámky jsou popisovány jako nejstarší, ale zároveň nejrozšířenější způsob ochrany proti konkurentnímu přístupu v podobě zamykání objektů [2]. Ten spočívá na velmi jednoduchém principu, kdy proces přistupující k libovolnému objektu daný objekt uzamkne, čímž ostatním znemožní další přístup k němu. Zámky bývají spravovány distribuovaně vrámci každé lokality. Každý server tak udržuje záznamy o vlastních lokálních zámcích. Autoři [2] a [3] rovněž uvádí existenci databázových systémů s centralizovanou správou zámků.

Takto definované zámky však výrazně sníží celkovou propustnost systému, proto bývají definovány zámky zvlášť pro čtení a zvlášť pro zápis. Autor [23] uvádí dva druhy zámků.

Exkluzivní zámky (*exclusive locks*) zajišťují, že k danému souboru může přistupovat pouze daný proces. Ostatním procesům není až do opětovného odemčení čtení ani zápis umožněn. Oproti tomu sdílený zámek (*shared locks*) zajišťuje, že daný objekt může číst několik procesů naráz, avšak žádnému není umožněna modifikace objektu.

Autor [2] dále uvádí pojem granularita zámku. Jedná se o množství objektů, které zámek ovlivní. Hrubá granulace je případ, kdy dochází k zamykání celého souboru, nebo databáze, naopak při jemné granulaci může být zámek vztažen přímo ke konkrétnímu záznamu v databázi . Vyšší úroveň granulace značně zvýší stupeň

paralelismu, avšak přesnější zámky vyžadují daleko vyšší režii systému. Zároveň bývají dle [2] nachylnější k deadlockům.

Autoři [7] a [9] uvádějí dva základní způsoby přístupu pro řešení zamykání objektů v transakcích.

5.4.3 Optimistický přístup

Při optimistickém přístupu se předpokládá, že ke konfliktním situacím dochází pouze zřídka. Spočívá v kontrole objektu, zda od předchozího čtení nedošlo k jeho modifikaci jiným procesem. Tento proces ověřování se nazývá dle [9] preCommit. Pokud proces před zápisem zjistí, že se objekt liší, je transakce zrušena.

5.4.4 Pesimistický přístup

Oproti tomu při pesimistickém přístupu, je přistupováno ke stejné situaci zcela opačně. Přístup předpokládá, že ke konkurentním operacím dochází často a je lepší jim předcházet. Jakmile první transakce záznam zamkne, každá další transakce skončí při pokusu získat stejný záznam chybou. Autor [9] dále popisuje nevýhody pesimistického přístupu k databázím v podobě možného vzniku deadlocků. Tedy situací, kdy dvě (nebo více) transakcí čeká na uvolnění zdroje, na který má vlastní zámek jiná z transakcí a žádná z nich tak nemůže pokračovat. Takové situace je nejprve nutné detekovat poměrně složitými algoritmy pro detekci smyček vybudováním grafu čekajících transakcí. Smyčka v grafu značí přítomnost deadlocku. Následně je nutné vhodně ukončit vybrané transakce aby se smyčka uvolnila.

Způsobů jakými lze rozhodnout o ukončení dané transakce může být hned několik [19]. Transakce mohou být ukončovány na základě časového razítka transakce, přičemž ukončena bude vždy nejmladší z transakcí. Cílem je ovlivnění co nejmenšího počtu dalších transakcí.

Další variantou může být ukončení transakce s nejvyšším počtem zámků, nebo transakce která se účastní více cyklů.

Dále mohou být při výběru zohledněny transakce, které již byly v minulosti přerušeny. Takové transakce doporučuje autor [23] z výběru vynechat.

5.4.5 Dvoufázové zamykání

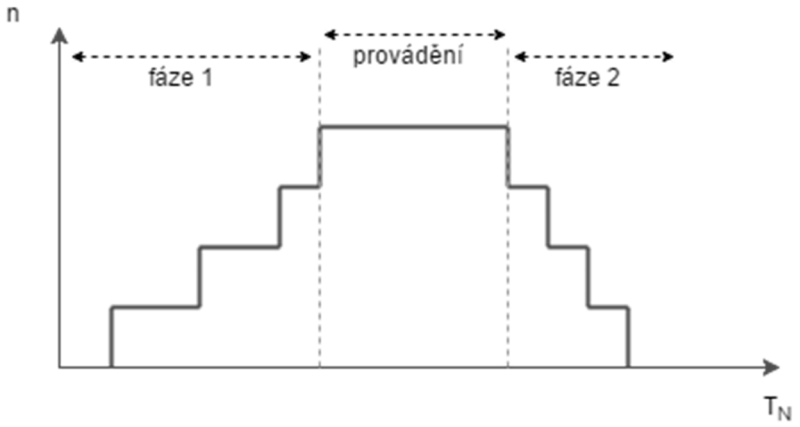
Zamykání a odemykání objektů až v okamžiku potřeby může způsobovat nekonzistenci a tvorbu deadlocků. Proto je ve většině transakčních systému implementována za pomoci dvoufázového zamykání.

V první fázi dochází k postupnému zamykání všech objektů požadovaných v transakci. Tato fáze je označována fako fáze nabytí (*acquisition phase)*.

Následně jsou provedeny požadované operace a až poté jsou ve druhé fázi zámky uvolněny. Druhá fáze uvolňování zámků je označována jako smršťující fáze (*shrink phase*, nebo také *release phase*). Obě fáze postupného zamykání a odemykání objektů znázorňuje Obrázek 19.

Druhá fáze se kvůli konzistenci dat často provádí až při ukončení transakce, ať již úspěšném nebo neúspěšném.

(Klimeš, 2007, p. 94)



Obrázek 19: Dvoufázové zamykání

Dvoufázové zamykání zabraňuje transakci získat nový zámek poté, co již některý ze zámků uvolnila. Dvoufázové zamykání však samo o sobě nevylučuje vznik deadlocků, nicméně [19] popisuje postačující podmínku pro eliminaci vzniku deadlocků v podobě zamykání objektů vždy v jednotném uspořádání – kanonickém pořadí. Dvoufázově uzamykaná transakce vždy splňuje podmínku sekvenčnosti.

6 Praktická část

V praktické části diplomové práce bude popsán vývoj systému inteligentního řízení domácnosti. Tento systém je realizován několika vzájemně propojenými zařízeními - uzly. Každý uzel je schopen zcela autonomně plnit zadanou činnost. Zároveň však může reagovat na povely uživatele. Uživatel tak může získat hodnotu libovolného senzoru, nebo iniciovat akci (např. měnit stavy GPIO pinů) na libovolném uzlu v síti. Sám uživatel komunikuje pouze prostřednictví webového rozhraní na jednom z uzlů. To následně zprostředkovává komunikaci s dalšími uzly systému.

6.1 Úvod do problematiky

Představa propojit většinu existujících systémů tak, aby poskytovaly uživateli komplexní přehled o všech situacích a umožnit mu tak provádět nejlepší možná rozhodnutí, ať už v interakci s uživatelem, nebo bez jeho přímého zásahu je snahou mnoha velkých firem napříč platformami.

V řídící technice je patrný výrazný trend přechodu od výkonných centrálních systémů k menším, které jsou rozmístěny podle potřeby a propojeny komunikační sběrnicí. Odpadá tak nutnost vést všechny signály do jednoho řídícího bodu. Řízení je tak distribuováno přímo do jednotlivých bodů a ty následně vykonávají vybrané akce. Takto navrženým systémům se říká distribuované řídicí systémy. Jejich využití najdeme v mnoha průmyslových aplikacích IIoT (Industrial Internet of Thinks), ale také v budovách, automobilech a mnoha dalších.

Master uzel má seznam všech senzorů v síti včetně IP adresy daného uzlu. Změny stavu GPIO jsou prováděny za pomoci realtime komunikace prostřednictvím služby Sockets. Reakce na požadavek tak probíhají prakticky ihned.

Každý uzel dokáže pracovat zcela autonomně. V případě výpadku internetového připojení je tak každý z uzlů schopen samostatně vykonávat zadanou činnost tj. dlouhodobý sběr dat ze senzorů a reakce na aktuální stav dle zadaného plánu.

6.2 Trendy v oblasti inteligentních domů

V dnešních domech se používá na 150 různých elektronických a dálkově ovládaných zařízení. Od regulace stínicí techniky, osvětlení, vytápění přes řízení alarmu, fotovoltaických panelů, audia, sauny až k ovládání přístupového systému, nebo monitorování objektu.

Každé zařízení se ovládá z pravidla vlastním ovládacím prvkem – vypínačem, dálkovým ovladačem, pohybovým senzorem, časovačem nebo dotykovou obrazovkou. Žádné zařízení tedy neví, co dělá jiné nebo co vlastně uživatel chce. To není moc inteligentní, a už vůbec ne pohodlné.

(*Loxone, 2016*)

Systémy inteligentních domů se v současnosti těší obrovské popularitě nejen ve světě, ale postupně si své místo nacházejí i u nás. Silným argumentem pro pořízení inteligentní stavby je výrazná úspora nákladů na její následný provoz. Dalšími argumenty pak mohou být reakce na potřeby uživatele, maximalizace uživatelského komfortu, ale i maximální bezpečí obyvatel.

Srdcem každého inteligentního domu je centrální systém, který díky potřebné infrastruktuře a jednotlivým aktivním i pasivním prvkům pomáhá automatizovat provoz domu. [27]

Srdcem každého inteligentního domu je Centrální řídicí systém. Ten představuje vysoce specializovaný kus hardware (často označovaný jako centrální řídící prvek, nebo server) s vlastním firmware a sadou předprogramovaných pravidel. Tato pravidla si pochopitelně může každý uživatel přizpůsobit vlastním potřebám. Takový server se většinou instaluje přímo do rozvodné skříně elektrické energie v blízkosti hlavního domovního jističe, případně do samostatně umístěného rozvaděče v její bezprostřední blízkosti. Do tohoto místa musí být rovněž svedeny veškeré vodiče od jednotlivých senzorů a aktivních prvků rozmístěných v domácnosti. Díky tomu může jediný senzor nebo vypínač ovládat nejenom světlo, ale i alarm, audio, větrání, vytápění, klimatizaci a spoustu dalších zařízení. Server bývá rovněž velmi často připojen k internetu, Celou domácnost tak můžeme spravovat v mobilní, nebo desktopové aplikaci a to často, i v případě, že se nenacházíme uvnitř ovládané budovy. Je tak možné na dálku vypnout spotřebiče v domě, manipulovat s osvětlením, nebo regulovat teplotu v domácnosti.

Takto navržená řešení však v sobě skrývají i celou řadu omezení. Tím prvním a asi nejzásadnějším je množství instalované kabeláže. V průměrně velkém domě může být celková vzdálenost všech datových kabelů i více než 700 metrů. Ty se tak stávají nedílnou součástí stavby a s jejím rozmístěním se musí počítat již ve fázi projektování inteligentního domu.

Pro dodatečnou integraci inteligentních systémů jsou tedy stávající technologie naprosto nevyhovující. Přinejmenším se nevyhneme velmi nákladné rekonstrukci celého objektu spojenou s instalací veškeré kabeláže. S tím jsou pochopitelně spojeny i vysoké náklady

za stavební práce. Proto je celé řešení předurčeno spíše novostavbám realizovaným tzv. „na zelené louce“.

Řada výrobců počítá s variantou integrace inteligentních prvků do stávajících budov a nabízejí i bezdrátové varianty pracující na principech technologie MESH.

Použitím bezdrátových technologií sice odpadá složitá a nákladná instalace vodičů, ale součástí každého senzoru musí být i část pro komunikaci na bezdrátové úrovni. Tím se krom větší velikosti zařízení zvyšuje také spotřeba elektrické energie, ale i cena každého senzoru. V neposlední řadě je uživatel odkázán pouze na speciální zařízení poskytované výrobcem řídící jednotky. Je tedy prakticky nemožné připojit jednotku k senzorům vyráběným konkurencí, nebo propojit nový systém se stávajícími systémy (např. pro zabezpečení objektu) od jiných výrobců.

6.2.1 Technologie Mesh

Jedná se o speciální topologii, kde jsou některé uzly propojeny s více než jedním uzlem v síti [28]. Umožňuje tak komunikaci i v případech, kdy dojte k výpadku některých prvků. Většina zařízení musí být schopna nejenom odesílat a přijímat vlastní data, ale i předávat dál data z jiných uzlů. Data mohou být distribuována mezi jednotlivé uzly buď za pomoci směrování, kdy se hledá optimální cesta k cíli, nebo tzv. záplavovým (flooding) algoritmem. V takovém případě jsou data vždy odeslána na všechny dostupné uzly s výjimkou těch, ze kterých byla přijata. Nespornou výhodou technologie Mesh je i fakt, že síť není potřeba nijak konfigurovat. Jednotlivé uzly po spuštění sami navážou spojení s ostatními zařízeními. Pro koncového uživatele tak odpadá nutnost každé zařízení složitě konfigurovat [29].

6.3 Alternativní systémy pro řízení inteligentních domácností

Kromě velkých společností nabízejících hotová řešení se na internetu objevilo mnoho zajímavých článků popisujících způsob, jakým lze řídit několik zařízení v domácnosti, nebo alespoň sbírat data o teplotě a vlhkosti v domácnosti. V následující kapitole jsou stručně popsány některé z nich.

Jedním z takových je projekt *Raspberry Pi as a Sensor Web node for home automation* [30]. Autoři představují centralizovanou podobu řídicího systému, kdy bylo využito jednoho zařízení Raspberry Pi jako řídící jednotky, ke které jsou za pomoci GPIO pinů připojeny jednotlivé periferie. Ani v tomto případě se však nevyhneme nákladné instalaci kabeláže. Dlouhé vzdálenosti mezi řídící jednotkou a senzory s sebou rovněž nesou určitá omezení. Právě z důvodu velkých vzdáleností mezi řídící jednotkou a senzorem je mnohdy vyloučeno použití digitálních senzorů. Přitom V kombinaci s Raspberry Pi

je čtení z digitálních senzorů velmi jednoduchou záležitostí. Ve většině případů je tedy autor nucen používat analogové teplotní senzory. Ty pracují na principu změny el. odporu se změnou teploty. S rostoucí vzdáleností napájecího vodiče však klesá napětí na výstupu v důsledku vnitřního odporu vodiče. S těmito ztrátami je nutné počítat a naměřenou hodnotu kompenzovat právě podle délky použitého vodiče.

Druhým, avšak zcela odlišným řešením je *Wireless Sensor Network System Design Using Raspberry Pi and Arduino for Environmental Monitoring Applications* [31]. Autor se sice zabývá pouze měřením teploty a vlhkosti spolu s ukládáním hodnot za účelem statistik, ovšem oproti předchozímu řešení využívá decentralizovanou strukturu řídicího systému. Ta spočívá v tom, že kromě centrální řídící jednotky (opět Raspberry Pi) mohou být připojeny další řídící jednotky v podobě zařízení Arduino UNO R3. Tyto jednotky mají stejně jako hlavní jednotka vlastní OS s možností naprogramovat si vlastní logiku. K základní jednotce je tak možné rozšířit množství senzorů o dalších až 6 analogových, nebo 14 digitálních a to s každou nově připojenou jednotkou. Každá jednotka dokáže sama periodicky zpracovávat naměřené hodnoty a následně je odesílat do hlavní jednotky, kde jsou ukládány do databáze. Zde mohou být efektivně tříděny a velmi rychle vyhledány na základě zvoleného SQL dotazu. O předání a vizualizaci hodnot se stará webový server Apache. Pro grafickou vizualizaci hodnot je použita java-skriptová knihovna Google charts [32].

O vzájemnou komunikaci a předávání hodnot mezi jednotkami se starají speciální moduly XBee Pro S2B. Jedná se o malá zařízení, pracující na principu Mesh sítě s využitím síťového protokolu ZigBee. Dosah jednoho zařízení může být až 1600m. Pro potřeby pokrytí domácnosti je tedy toto řešení naprosto dostačující. Zásadní nevýhodou je však cena tohoto modulu, která několikanásobně převyšuje cenu samotné jednotky. Dalším nedostatkem ZigBee techlologie je fakt, že veškerá komunikace probíhá zcela nezabezpečeně.

Toto řešení rovněž neuvažuje možnost vyžádat si aktuální teplotu pro danou jednotku, nebo ovládání výstupů v reálném čase jako tomu bylo u předchozího řešení.

I když všechny výše uvedené postupy popisují funkční řešení problému, vzhledem k určitým nedostatkům jsem se rozhodl pro řešení vlastní, které spojuje výhody předchozích projektů, ale zároveň redukuje, nebo úplně eliminuje omezení, která s sebou tato řešení nesla. Zároveň by mělo umožnovat aktivně informovat uživatele, nebo předpovídat určité situace tak, aby na ně mohl uživatel včas reagovat.

Na Univerzitě Hradec Králové již v minulosti vnikla celá řada úspěšných projektů v oblastech automaticatce a IoT navržených a postavených na základě malého jednodeskového počítače.

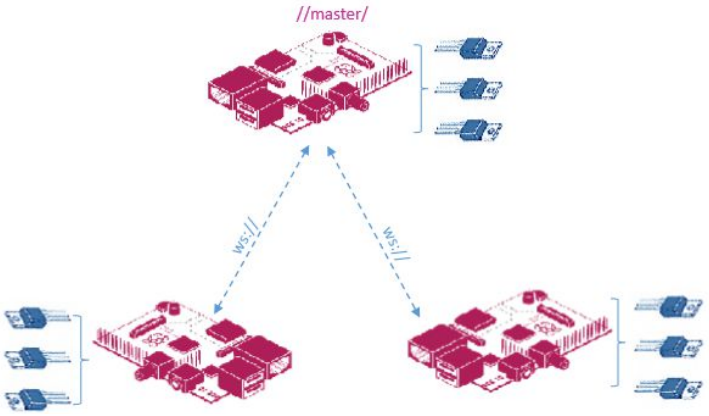
Namátkou to mohou být projekty:

Cloud Based Solution for Mobile Healthcare Application (viz [33])
A Smart Arduino Alarm Clock Using Hypnagogia Detection During Night [34],
Lower Layers of a Cloud Driven Smart Home System [35] , případně
Intelligent Heating Regulation [36].

6.4 Popis systému

Cílem práce je tedy navrhnout takový systém, který bude snadné integrovat i do stávajících staveb a to s cílem minimálního zásahu do stávajících domovních rozvodů. Systém musí být schopen snímat hodnoty v různých částech domu, zároveň musí umožňit ovládat vybrané elektrické spotřebiče včetně domovního osvětlení. Systém dále musí pracovat tak, aby se daly určité akce automatizovat bez zásahu uživatele. Systém bude rovněž schopný si určité hodnoty zajistit z veřejně dostupných internetových služeb. Jedná se zejména o údaje, které jsou v domácích podmínkách jen obtížně měřitelné, nebo se jedná o předpovědi pro nadcházející období. Tyto údaje mohou být následně využity pro interní potřeby systému a přispět k vyhodnocování určité akce. Uživateli musí být umožněno měnit vybrané parametry (např. požadovaná teplota v místnosti), nebo spínat určité okruhy manuálně prostřednictvím počítače, tabletu, případně mobilní aplikace.

Nabízí se tak myšlenka rozdělit (decentralizovat) řídicí systém do několika menších celků - uzlů, které mohou být strategicky rozmístěny v jednotlivých částech objektu. K těmto uzlům budou následně připojeny požadované senzory a v případě potřeby i relé moduly sloužící ke spínání elektrických okruhů o napětí až ~240V Vzájemná koordinace všech jednotek v domácnosti bude zaručena propojením jednotek prostřednictvím počítačové sítě ethernet, jak znázorňuje Obrázek 20.

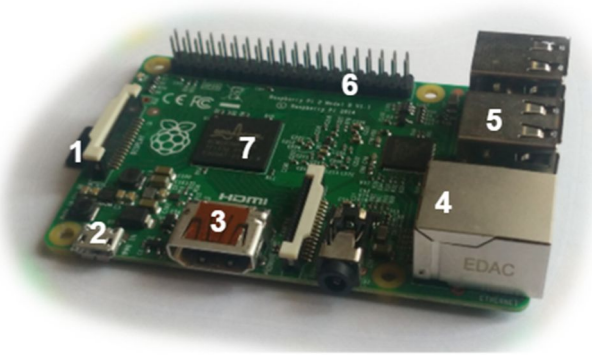


Obrázek 20: Schéma topologie systému

Díky standardizaci internetových protokolů je rovněž možné navzájem kombinovat různé způsoby připojení. K propojení uzlů uvnitř objektu je například možné využít stávajících ethernetových rozvodů, zatímco k připojení vzdálených objektů využít bezdrátových technologií WiFi, případně v kombinaci s veřejnou IP adresou, nebo VPN i CDMA/GSM mobilní technologie. Je tak možné monitorovat a spravovat pohyblivé objekty, ale i několik demograficky vzdálených budov (např. dům a chalupu, nebo jednotlivá odloučená pracoviště) z jednoho místa a zařízení.

6.5 HW technologie systému

Každou jednotku bude tvořit jedno zařízením Raspberry Pi 2 model B. Jedná se o miniaturní jednodeskový počítač, který byl vyvinut britskou firmou Raspberry Pi Foundantion. Raspberry Pi2 model B je již pátou generací. Na desce o velikosti 85 * 56 mm je osazen čtyřjádrový procesor Broadcom BCM2836 o frkvenci 4 * 0,9 GHz a operační pamětí 1GB RAM. Na desce se rovněž nachází čtveřice USB 2.0 portů, grafické jádro výstup HDMI a 3,5mm Jack pro audio vstup/výstup. Pro připojení k síti je připraveno rozhraní RJ45. Rozmístění jednotlivých rozhraní na desce znázorňuje Obrázek 21.



Obrázek 21: Raspberry Pi 2 model B
1) Micro SD karta; 2) Napájení microUSB 5V; 3) HDMI; 4) Ethernet RJ-45;
5) 4x USB2; 6) GPIO sběrnice; 7) CPU a GPU BCM2836

Zařízení je svými rozměry dostatečně malé na to, aby se vešlo například do standardní krabičky pro elektrickou instalaci, zároveň však poskytuje dostatečný výkon k chodu distribuované databáze, webového serveru a API pro poskytnutí dat dalším aplikacím. Zároveň umožňuje připojení řady periferií prostřednictvím GPIO sběrnice. Dále obsahuje 4 USB porty vhodné pro připojení dalších periferií a integrovanou síťovou kartu pro připojení k síti internet. V projektu budou použity celkem tři tyto jednotky. Jedna z jednotek bude označena jako *master* – hlavní uzel a ostatní jako *slave* – vedlejší uzly. Na master uzlu poběží webový server, který poslouží pro zobrazení naměřených

hodnot, poskytne přehled o jednotlivých elektrických okruzích. Dále bude zpřístupňovat veškeré hodnoty pro další aplikace ve formátu JSON. O vizualizaci dat se postará HTML5 v kombinaci s Java skriptem.

Každý uzel bude periodicky v zadaných intervalech snímat fyzicky připojené senzory. Veškeré hodnoty budou uchovány v databázi, odkud mohou být v případě potřeby získány. Tyto hodnoty se budou následně distribuovat mezi ostatními uzly tak, aby bylo možné efektivně získat přehled o všech senzorech v domácnosti.

Ke snímání budou použity převážně digitální senzory, které mohou být připojeny k libovolnému uzlu prostřednictvím vestavěné GPIO sběrnice. Těch může být takřka neomezené množství. V místnostech budeme bezprostředně měřit pouze teplotu a vlhkost. K tomu nám poslouží senzor teploty DS18b20 a senzor teploty a vlhkosti DHT11. První ze jmenovaných senzorů je teplotní senzor měřící teplotu. Tento velice malý senzor se schopen měřit teplotu v rozsahu od -55°C do +125°C. Kompletní přehled informací je možné nalézt na [37]. Výrobce rovněž nabízí variantu ve voděodolném pouzdru. Můžeme tak bez obav měřit i venkovní teplotu. Napájení je zajištěno přímo z jednotky Raspberry Pi a to napětím 3,3V. Senzor může rovněž pracovat v tzv. „parazitním režimu“, kdy napájení obstarává pouze datový vodič. Každý další senzor se připojuje paralelně ke stávajícímu datovému vodiči. Každý senzor je reprezentován unikátním sériovým číslem, pod kterým jsou v systému viditelná ve složce */sys/bus/w1/devices/*, kde se zobrazí adresář s názvem unikátního čísla senzoru. V něm je obsažen soubor se samotnými daty. Součástí tohoto souboru je rovněž informace o kontrolním součtu. Lze tak velmi jednoduše identifikovat chybné hodnoty a měření buď zopakovat, nebo jinak zareagovat na vzniklou situaci.

Druhým typem je digitální senzor DHT11. Ten v sobě skrývá digitální teplotní senzor a senzor vlhkosti integrované do jednoho pouzdra. Kompletní informace jsou dostupné na webu výrobce [38]. Rozsah měřených teplot je 0-50°C. U senzoru vlhkosti pak 20-90%. Tím je předurčen především do interiéru. Tento senzor se vyznačuje sice menší přesností, ale vzhledem k tomu, že se jedná v podstatě o dva senzory v jednom pouzdře, jedná se o velmi efektivní způsob jak měřit hodnoty v interiérech budov.

Jako akční člen byl zvolen relay board HL-85. Ten poskytuje osm relátek, umožňujících nezávislé spínaní až osmi okruhů o napětí až 250V a proudu do 10A. Lze tak ovládat i několik spotřebičů naráz a to do výkonu až 2500W. Deska se k uzlům připojuje prostřednictvím GPOI sběrnice a jednotlivé okruhy se spínají přepnutím vybraných pinů do stavu logické LOW. Napájení desky zajišťuje GPIO sběrnice prostřednictvím napájecího pinu +5V.

Jako zástupce dat, která získávaných z rozhraní třetích stran byly zvoleny předpověď počasí a rychlost větru. Obě hodnoty budou čteny ze služby openweathermap.org ve formátu JSON. Ta poskytuje bezplatně Rest API pro celou řadu lokalit a parametrů s frekvencí až 50 000 dotazů za den. Kompletní přehled parametrů je dostupný na webových stránkách společnosti [39].

Připojení všech uzlů do lokální sítě zajistí vestavěný síťový adaptér v kombinaci s běžným domácím switchem. V našem případě se jedná o malý pětiportový switch Tenda S5.

Jak již bylo v úvodu zmíněno, zařízení musí být schopna reagovat na povely uživatele a to s minimálním zpožděním. Za tímto účelem byla zvolena knihovna Java Sockets.

Jedná se o knihovnu sloužící pro mezi procesovou komunikaci v rámci jednoho stroje, ale i mezi vzdálenými stroji, v rámci rodiny protokolů PF_INET a PF_INET6. Používají se běžně pro kominikaci v internetu a obsahují IP adresy a čísla portů TCP a UDP [40].

Celý proces od vzniku požadavku až po vykonání akce bude probíhat následujícím způsobem. Uživatel ve své aplikaci vyvolá akci stiskem příslušného tlačítka. Tím vyšle HTTP požadavek na master server, který bude mít podobu:

http://master/relay/?name=osetleni&state=true

Zde je master serverem podle identifikátorů identifikováno cílové zařízení a následně mu je zaslán požadavek o provedení určité akce ve formě Socketu. Cílové zařízení požadavek zpracuje a následně provede zadanou akci. V případě potřeby pak vrátí zprávu o úspěšném vykonání dané akce. Ta je obratem zobrazena uživateli.

6.6 Implementace

Před prvním spuštěním je nutné předpřipravit všechny paměťové karty, které budou sloužit pro běh samotného operačního systému. Na výběr máme hned z několika linuxových distribucí jako Raspbian, Ubuntu, a mnoho dalších. Případně je možné nainstalovat Windows 10 IoT Core , který nabízí firma Microsoft jako bezplatnou alternativu k malým linuxovým IoT distribucím. Kompletní seznam podporovaných OS najdeme na stránkách výrobce [41].

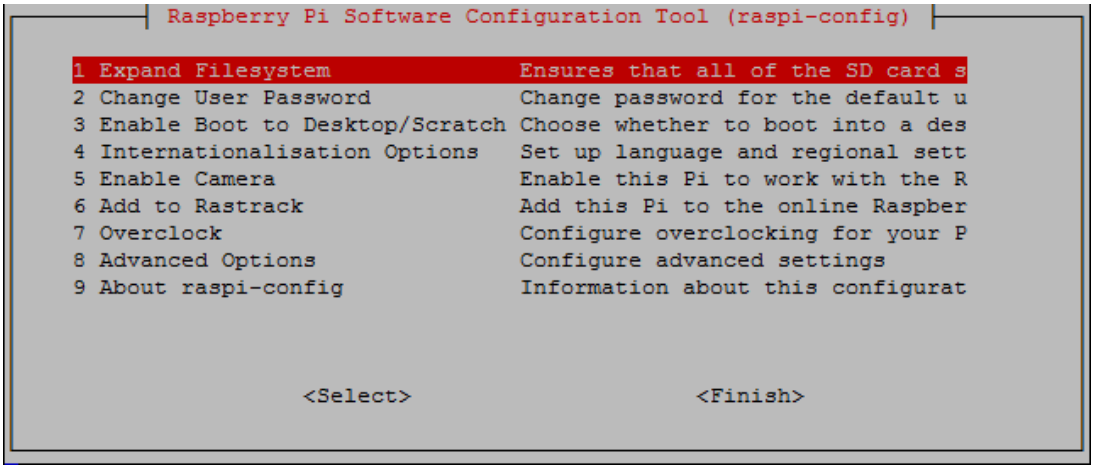
Samotná příprava karty spočívá v překopírování obrazu systému na paměťovou kartu zařízení. V našem případě byl použit obraz NOOBS, který nabízí výběr ze všech nabízených OS, Obraz vybraného OS se následně stáhne a průvodce instalací Vás provede několika kroky. Po dokončení instalace ještě provedeme několik nezbytných nastavení,

jako změna výchozího uživatelského jména a hesla, případně povolení vzdálené správy za pomoci SSH.

K tomu nám poslouží příkaz v terminálovém rozhraní:

```
raspi-config
```

Po odeslání příkazu se zobrazí jednoduchá nabídka, kde nastavíme vše potřebné (viz Obrázek 22)



Obrázek 22: Konfigurační utilita Raspberry Pi; zdroj autor

Následujícím krokem bude instalace databázového serveru. Pro naše účely byla použita database MySQL server ve verzi 5.5.44.

Instalaci provedeme příkazem:

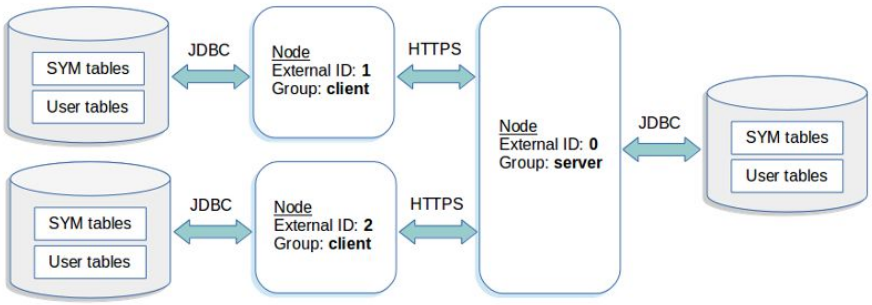
```
sudo apt-get install mysql-server
```

Po instalaci je ještě nutné vytvořit nového uživatele, databázové schéma a přidělit uživateli patřičná oprávnění. Detailní postup jak nastavit veškerá oprávnění najdeme v dokumentaci výrobce [42].

Po nainstalování databází na všechny použité jednotky přejdeme k instalaci replikačního nástroje Symmetric DS. Ten je nabízen v základní verzi zcela zdarma a nabízí nepřeberné množství způsobů, jakým lze replikovat data mezi uzly. Zároveň podporuje i celou řadu SQL i NoSQL databází. Dále umožňuje šifrované HTTPS spojení pro replikaci záznamů. Ze stránek výrobce stáhneme balíček s aktuální verzí (v našem případě 3.7.29) a rozbalíme do paměťového uložiště. Následně nastavíme prametry uzlů. V souboru jméno_uzlu.properties umístěném v adresáři /engines specifikujeme všechny nezbytné parametry jako použitou databázi, údaje pro připojení a způsob jakým jsou data

replikována mezi uzly. Pro představu, jak by měl soubor vypadat jsou připraveny ve složce */samples* ukázkové soubory pro několik základních způsobů replikace. Pro kompletní přehled poskytuje výrobce přehlednou dokumentaci [43]. V našem případě jsme zvolili MultiMaster replikaci dat.

V případě vytvoření libovolného záznamu je tak tento záznam replikován na všechny připojené uzly. To může být velmi praktické z hlediska dotazování, ale zároveň může přestavovat určité bezpečnostní riziko v případech, kdy dochází k replikaci citlivých dat v rámci všech zařízení ve společnosti. Princip jakým jsou data replikována zobrazuje následující schéma na Obrázku 23.



Obrázek 23: Schéma replikace dat mezi ulzy pro architekturu klient-server; převzato z [43]

Po prvním spuštění se v databázi vytvoří pomocné záznamy tzv „SYM_TABLES“ poznáme je podle předpony SYM_ v názvech tabulek. Zde je možné specifikovat, jaké tabulky je nutné replikovat.

Jako poslední budeme potřebovat nástroj pro zobrazení naměřených údajů. V našem případě bude webová aplikace programována v Javě v kombinaci s Frameworkem Spring MVC. Ten je použit zejména z důvodu snadné udržitelnosti kódu a s ohledem na pozdější zozšiřitelnost webové aplikace. Jako webový server bude použit Apache Tomcat 8.0 [44]. Jeho výhodou je snadná konfigurace ale i dobrá optimalizace i pro slabší hardware. Instalace samotného balíku je velice jednoduchá. Po stažení balíčku a jeho rozbalení příkazy:

```
wget http://apache.com/tomcat/tomcat/v8.0.23/bin/apache-tomcat-
8.0.23.tar.gz

tar xzf apache-tomcat-8.0.23.tar.gz
```

nebo příkazem:

```
apt-get install tomcat8
```

nyní již stačí jen nakonfigurovat administrátorský účet pro správu webového serveru v souboru *conf/tomcat-users.xml*. První start serveru provedeme pomocí příkazu:

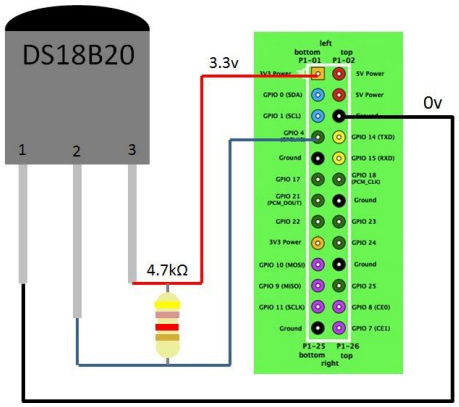
```
sudo bash /bin/startup.sh
```

Apache Tomcat startuje standardně na portu 8080.

Použité senzory jsou DS18b20 a DHT11. První ze jmenovaných je senzor teploty, který se připojuje přímo k GPIO sběrnici. Mezi datovou sběrnici a napájení +3.3V se umisťuje Pull-up rezistor 3,7kΩ. Schéma zapojení je uvedeno na Obrázku 24.

Senzor se připojuje k napájecímu pinu +3.3V, datovému digitálnímu IO pinu (v našem případě GPIO4 – pin7) a GND – pin6.

!!! Připojovat jakékoliv senzory ke GPIO sběrnici se doporučuje vždy ve vypnutém stavu a s odpojeným napájením !!!



Obrázek 24: Schéma zapojení teplotního čidla DS18b20; zdroj autor

Po opětovném startu zařízení se senzor zobrazí v adresáři */sys/devices/w1_bus_master1/* jako nový adresář s názvem unikátního sériového čísla, které přiděluje výrobce. To má formát:

$$28 - XXXXXXXXXXXX,$$

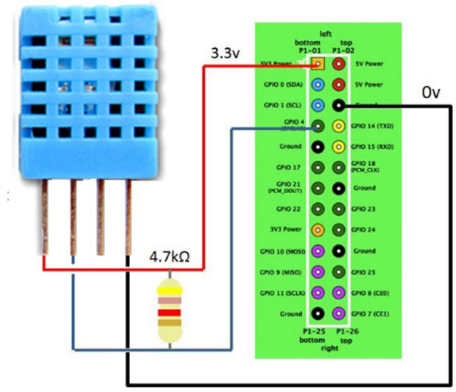
kde X zastupují sériové číslo v hexadecimální soustavě. Je tak vyloučeno, aby dvě zařízení měla stejná sériová čísla, a můžeme toto číslo bez obav použít pro pozdější jednoznačné identifikaci konkrétního senzoru v rámci aplikace. Soubor w1_slave obsahuje konkrétní informace o teplotě a další meta informace sloužící k ověření správnosti naměřené hodnoty (bude vysvětleno níže).

Hodnotu tak lze velmi jednoduše periodicky číst obdobně jako by se nacházela na paměťové kartě. Stačí nám znát pouze jednoznačný identifikátor zařízení. Rovněž lze sbírat data ze všech bezprostředně připojených senzorů v rámci jednoho cyklu za pomoci regulérního výrazu „28*“. Výsledná podoba souboru může vypadat následovně:

```
73 01 4b 46 7f ff 0d 10 41 : crc=41 YES

73 01 4b 46 7f ff 0d 10 41 t=23187
```

Políčko „YES“ u CRC značí, že při přenosu informace nedošlo k porušení informace a uvedenou teplotu lze považovat za relevantní. Položka t=23187 pak označuje samotnou naměřenou teplotu ve °C. Před jejím použitím je však nutné vydělit toto číslo 1000. V našem případě je naměřená teplota 23.187°C. Vzhledem k udívané přesnosti senzoru



Obrázek 25: Schéma zapojení senzoru DHT11; zdroj autor

±0.5°C, je však vhodné ještě získanou hodnotu vhodně zaokrouhlit. Takto zpracovanou hodnotu již lze použít v rámci aplikace. Způsob zapojení senzoru DHT11 je znázorněn schématem na Obrázku 25.

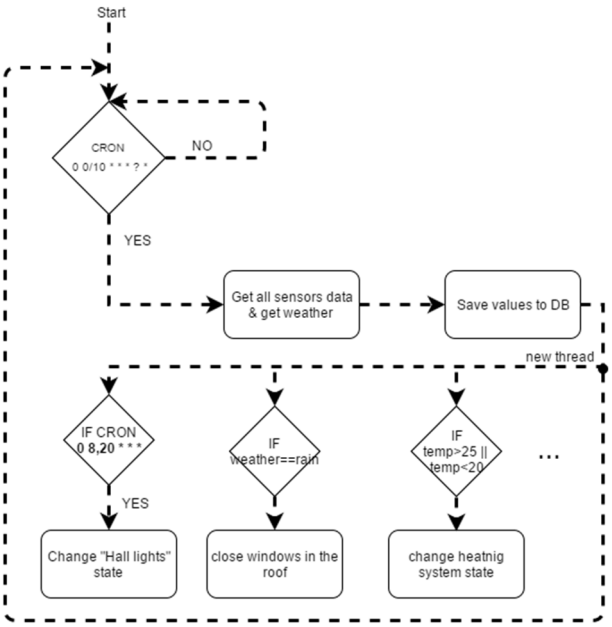
Pro získání dat ze senzoru uvedeme datovou sběrnici do stavu LOW a po opětovném uvedení do stavu HIGH by do 20-40µs měl senzor zahájit přenos dat po sériové datové lince. Pro čtení hodnot použijeme knihovnu Adafruit library [45]. Výsledkem volání funkce update() je pole hodnot s teplotou a vlhkostí. Pro ovládání relay board použijeme knihovnu Pi4J [46].

Aplikace v jednotlivých uzlech je napsána v Javě. Kromě periodického ukládání hodnot do databáze má za úkol stahovat informace o počasí a síle větru. Data budou průběžně aktualizována každých 10 minut. K tomu nám poslouží knihovna Quartz Scheduler. Tento doplněk umožňuje opakované spouštění periodicky se opakujících procesů v přesně zadaných časech. Po každé aktualizaci naměřených hodnot automaticky vyhodnotí, zda je potřeba vykonat určitou akci. Příklad interní logiky v jednom z uzlů znázorňuje Obrázek 26.

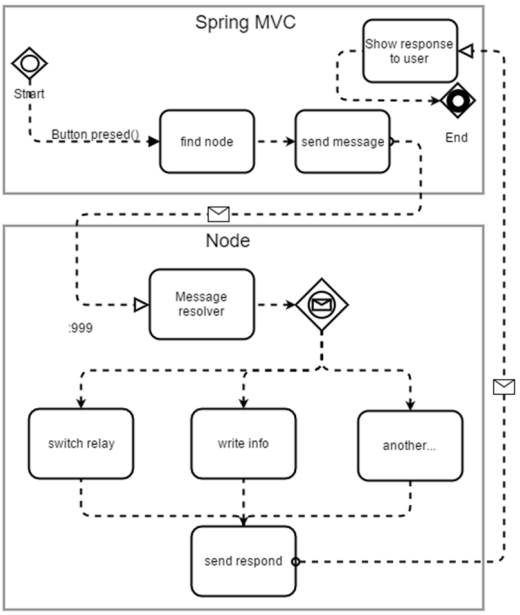
Zároveň musí být schopna reagovat na požadavky od uživatele. Ty budou mít podobu zpráv zasílaných pomocí interní knihovny Java.Net.Socket. Za tímto účelem ja na každém

zařízení otevřený port 9999, přes který komunikace probíhá. Princip zasílání zpráv a způsob vyhodnocení zobrazuje Obrázek 27.

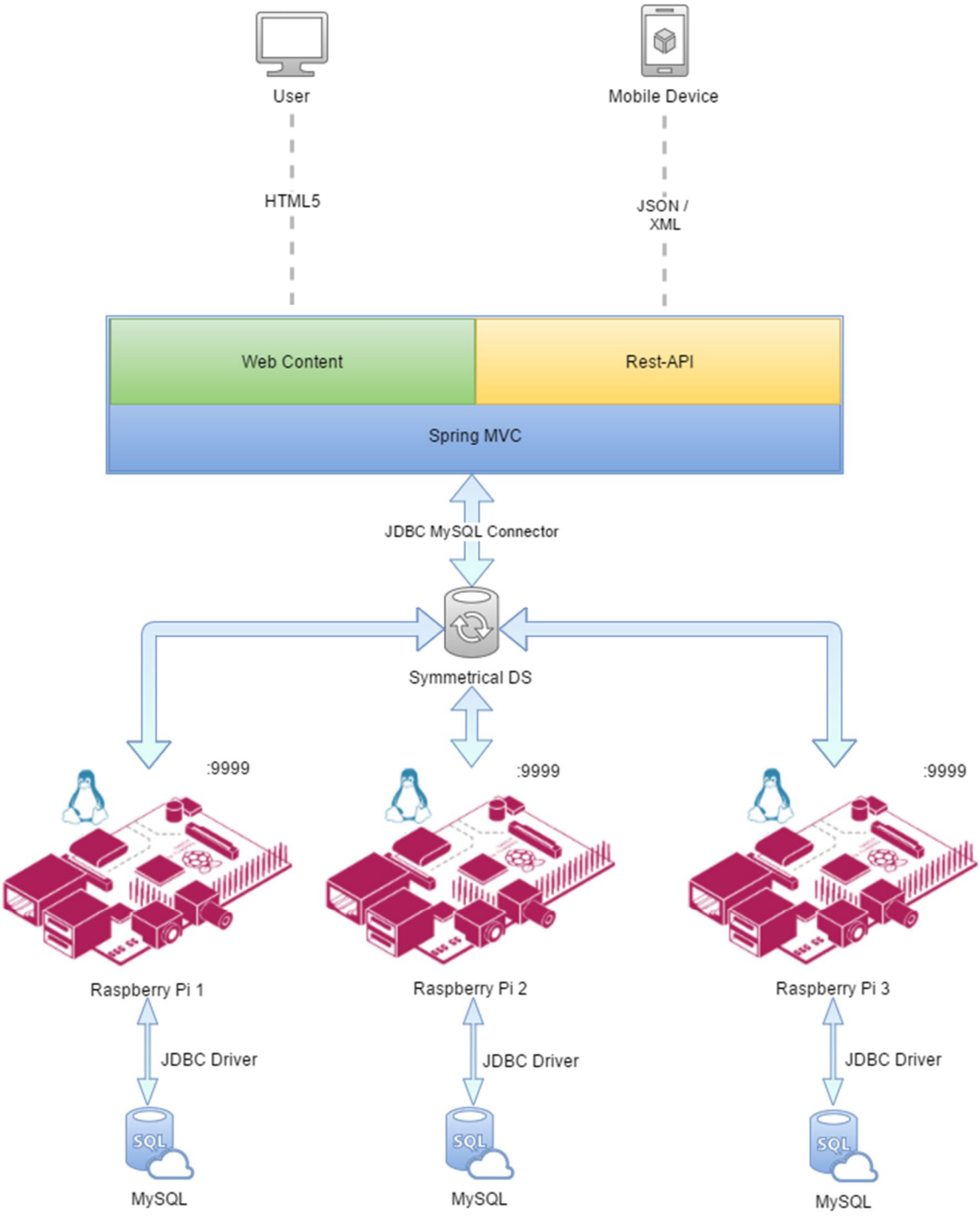
Jako aplikace pro zprostředkování dat uživateli a dalším aplikacím za pomoci REST API poslouží webová Java aplikace spolu s frameworkem Spring MVC. Detailní podobu systému poskytuje schéma na Obrázku 28.



Obrázek 26: Schéma interní logiky systému; zdroj autor



Obrázek 27: Princip socket komunikace; zdroj autor



Obrázek 28: Znázornění komunikace v rámci systému;
zdroj autor

6.7 Testování

6.7.1 Synchronizace dat

V následující části se zaměřím na dobu potřebnou pro synchronizaci dat mezi jednotlivými uzly. Mějme modelový případ, kdy uložíme určité množství dat do databáze uzlu 3, přičemž měříme dobu, za kterou se data rozšíří mezi všechny zařízení v síti. Standardní doba mezi jednotlivými PUSH/ PULL požadavky je 60000ms. Každých 60s tedy probíhá kontrola nových souborů a případná výměna dat. Služba Push Job může být rovněž nakonfigurována tak, aby umožňovala více paralelních PUSH požadavků naráz. Každá změna tak může být paralelně distribuována mezi všechny ostatní uzly. Maximální čas potřebný pro distribuci dat mezi libovolnými dvěma uzly je tedy maximálně 60 sekund.

Oproti MESH sítím, kde se frekvence komunikace mezi zařízeními pohybuje v řádech desítek minut, se jeví toto řešení pro domácí použití jako daleko vhodnější.

V případě potřeby lze interval mezi jednotlivými PUSH/PULL požadavky libovolně přizpůsobit potřebám aplikace. Prodloužením periody mezi jednotlivými požadavky můžeme snížit vytížení sítě, zároveň však prodloužíme dobu potřebnou pro přenos informací mezi uzly. V extrémních případech může být perioda PUSH/PULL požadavků vyšší, než frekvence s kterou jsou pořizovány záznamy z čidel. V takových případech je však nutné počítat se tím, že u senzorů. Které nejsou bezprostředně připojené k danému uzlu, bude možné zpracovat pouze data pocházející z poslední synchronizace.

6.7.2 Socket komunikace

Socket komunikace je navržena tak, aby dokázala zprostředkovat komunikaci mezi aplikacemi v reálném čase. Doba mezi vysláním požadavku a vyvoláním patřičné reakce je tak z velké části ovlivněna síťovou odezvou. Pro zjištění odezvy provedeme test, kdy budeme měřit čas, který uplyne mezi vyvoláním akce (stisk tlačítka) a obdržením odpovědi o úspěšném provedení dané operace. K tomu použijeme jednoduchý java-skriptový kód, ve kterém budeme porovnávat systémový čas v okamžiku vyslání požadavku a čas v okamžiku obdržení odpovědi. Výsledná odchylka bude značit dobu potřebnou pro zpracování požadavku. V měření bude zohledněn i druh použitého připojení, proto v testu použijeme hned několik běžně používaných druhů připojení, od klasické domácí LAN sítě, až po mobilní 3G připojení. Výsledky měření v milisekundách zobrazuje následující Tabulka 7.

	LAN [ms]	*WIFI [ms]*	*3G [ms]*
	42	16	54
	20	17	62
	18	17	58
	24	20	56
	18	24	61
	19	16	59
Průměr	**23,5**	**18,34**	**58,34**

Tabulka 7: Naměřené hodnoty odezvy na manuální vyvolání akce v ms

Všechny naměřené hodnoty se pohybovaly v řádech ms. I když se potvrdilo, že reakční doba systému je ve velké míře závislá na odezvě sítě, ve všech případech však došlo k vykonání akce během několika milisekund bez zbytečných prodlev systému. I když se mi nepodařilo zjistit, jakou technologii používá systém Loxone, předpokládám, že reakční doba obou systémů bude obdobná.

6.7.3 Vizualizace dat

V rámci vizualizace dat jsme se zaměřily na výkonové schopnosti zařízení Raspberry Pi 2. Zde budeme zkoumat časy potřebné k načtení stanoveného počtu záznamů, jejich zpracování a vizualizaci dat uživateli. Pro srovnání nám poslouží běžné domácí PC. I když je předem jasné, které zařízení na tom bude s výkonem lépe, zajímají nás především rozdíly mezi oběma zařízeními. Bude webový server běžící na jednom z uzlů dostatečně rychlý, aby poskytl uživateli data v přijatelném čase, nebo bude třeba výkonnější zařízení pro obsluhu webového serveru.

Tabulka 8 zobrazuje průměrný čas v sekundách potřebný k načtení zadaných dat. Udávaná hodnota vznikla zprůměrováním vzorku 10 měření.

	PC [s]	**Raspberry Pi 2 [s]**
Statická webová stránka bez čtení z DB	*0,19*	*0,38*
Webová stránka s čtením záznamů z DB	*0,63*	*1,38*
Načtení a zobrazení grafu se 144 hodnotami z DB	*1,25*	*2,8*
JSON výstup s názvy senzorů a poslední naměřenou hodnotou	*0,90*	*1,65*

Tabulka 8: Měření rychlosti načítání obsahu

I když se potvrdil hypotéza, že Raspberry Pi 2 bude potřebovat více času k zobrazení požadovaných výstupů, nejsou časy potřebné pro zobrazení výstupu nějak výrazně nepřiměřené. U žádných z testovaných dotazů nepřekročila doba čekání na výstup 3 sekundy. To je pro domácí použití přijatelná hodnota. Proto jsme přistoupili k umístění webového serveru Tomcat přímo na jeden z uzlů distribuovaného systému.

V případě potřeby je však rovněž možné provozovat webový server na samostatné stanici, případně použít pro zobrazení dat existující firemní infrastrukturu.

6.7.4 Shrnutí

Navržený systém dokáže monitorovat široké spektrum veličin, v závislosti na použitých sensorech. Oproti komerčním systémům umožňuje připojit takřka libovolný senzor, nebo aktuátor pro ovládání libovolného zařízení. Systém rovněž umožňuje uživateli definovat si vlastní logiku pro ovládání zařízení.

Systém je navržen tak, aby mohl být nasazen jak v domácnostech, tak i pro použití v korporátní scéně, například v oblastech IT, vodohospodářství, meteorologii, dopravě a řadě jiných. Zde může být implementován do budov, ale i pro monitoring demograficky vzdálených oblastí, či dopravních prostředků.

7 Závěr

V teoreticé části byly podrobně vysvětleny všechny nezbytné pojmy. Představeny vlastnosti distribuovaných systémů, principy fungování meziprocesové komunikace a synchronizace včetně detailního popisu vybraných algoritmů pro synchronizaci hodin. Následně byly podrobně představeny principy distribuovaných databází. Způsoby fragmentace dat, základní principy Transakcí a zamykání objektů. Vybrané principy a algoritmy byly následně využity při implementaci v praktické části.

Oproti stávajícím systémům se podařilo dosáhnot takové distribuce dat, aby nebylo nutné všechny senzory připojovat bezprostředně k jedné konkrétní jednotce. Zároveň byla zachována okamžitá reakci na požadavky uživatele. Rovněž se podařilo vytvořit API pro integraci dalších aplikací využívajících dat nasbíraných z tohoto systému.

Pro případný další vývoj by bylo vhodné integrovat HTTPS spolu s SSL šifrování spojení pro Socket komunikaci, dále přidat autentifikaci uživatelů včetně pravidel pro získání dat a ovládání aktivních prvků. Vhodným prvkem by rovněž bylo rozšíření stávající logiky pro autoamtické ovládání systému o určitý model predikce a strojového učení, nebo rozšířené správy pravidel bez nutnosti opětovné kompilace zdrojového kódu.

Dále by bylo vhodné zajistit automatickou detekci a konfiguraci nově připojených zařízení, případně implementace algoritmů detekujících selhání některého z uzlů.

Těmto a mnoha dalším bodům bych se však rád věnoval ve svém dalším studiu.

8 Citovaná literatura

[1] A. S. Tanenbaum a M. V. Steen, Distributed Systems: Principles and Paradigms (2nd Edition), 2. editor, Upper Saddle River, NJ: Pearson Education. Inc., 2007, p. 704.

[2] C. Klimeš, „Distribuované systémy," 2007. [Online]. Available: http://www1.osu.cz/~prochazka/ds/SkriptaKlimes.pdf. [Přístup získán 2015].

[3] J. Ledvina, „Zápoadočeská Univerzita v Plzni," 2002. [Online]. Available: http://zcu.arcao.com/kiv/ds/ds.pdf. [Přístup získán 26 květen 2016].

[4] B. C. Neuman, „Scale in Distributed Systems," v *Readings in Distributed Computing Systems*, Los Alamitos, 1994.

[5] J. Peterka, „Přednáška: Počítačové sítě, verze 3.0, lekce č. 12, slide č. 2," 16 říjen 2005. [Online]. Available: http://www.earchiv.cz/l212/slide.php3?&l=12&me=2. [Přístup získán 21 květen 2016].

[6] S. M. Thampi, „Introduction to Distributed Systems," *arXiv preprint arXiv:0911.4395,* 2009.

[7] G. Coulouris , J. Dollimore, T. Kindberg a G. Blair, Distributed Systems: Concepts and Design, 5. editor, Boston, Massachusetts: Pearson, 2011, pp. 185-228.

[8] P. Burian, „Internet inteligentních aktivit," Grada Publishing a.s., 2014, p. 94.

[9] I. Holubová , J. Kosek, K. Minařík a D. Novák, „Big Data a NoSQL databáze," Grada Publishing a.s., 2015, pp. 72-76.

[10] B. Sosinsky, „Mistrovství – počítačové sítě," v *Mistrovství – počítačové sítě*, Computer Press, Albatros Media a.s., 2011, pp. 43-83.

[11] J. Peterka, „Přednáška: Rodina protokolů TCP/IP, verze 2.7," 2011. [Online]. Available:

http://www.earchiv.cz/l223/slide.php3?l=2&me=29. [Přístup získán 2 červen 2016].

[12] P. Krčmář, „Velké trable s malým MTU,“ Root.cz, 7 leden 2013. [Online]. Available: http://www.root.cz/clanky/velke-trable-s-malym-mtu/. [Přístup získán 13 květen 2016].

[13] J. Peterka, „Linková vrstva - II.,“ eArchiv, 12 září 1996. [Online]. Available: http://www.earchiv.cz/a92/a219c110.php3. [Přístup získán 22 červen 2016].

[14] A. S. Tanenbaum, „Implementing the Client-Server Model,“ v *Distributed Operating Systems*, Boston, Pearson Education, 1995, pp. 68-80.

[15] A. Puder, K. Römer a F. Pilhofer, „Client/Server Model,“ v *Distributed Systems Architecture: A Middleware Approach*, San Francisco, Elsevier, 2005, pp. 12-16.

[16] B. Sosinsky, Mistrovství – počítačové sítě, Brno: Computer Press, 2011.

[17] O. Filip, „Úvod do IP multicastu,“ Internet Info, s.r.o., 10 září 2014. [Online]. Available: http://www.lupa.cz/clanky/uvod-do-ip-multicastu/. [Přístup získán 06 červen 2016].

[18] G. Hosszú, „Mediacommunication Based on Application-Level multicast,“ v *Encyclopedia of Virtual Communities and Technologies*, Hungary: IGI Global, 2005, pp. 302-308.

[19] G. Sukumar , Distributed Systems: An Algorithmic Approach, Second Edition, 2. editor, Iowa City: CRC Press, 2014, p. 224.

[20] T. M. Özsu a P. Valduriez, Principles of Distributed Database Systems, New York: Springer Science & Business Media, 2011.

[21] R. Chhanda, „Distributed Database Concepts,“ v *Distributed Database Systems*, Kolkata, West Bengal: Pearson Education, 2009, pp. 31-42.

[22] H. Theo a R. Andreas, „Principles of Transaction-oriented Database Recover,“ *ACM Comput. Surv.,* sv. 15, č. 4, pp. 287-317, Prosinec 1983.

[23] I. Holubová , J. Kosek, K. Minařík a D. Novák, „Transakce v distribuoaném prostředí,“ v *Big Data a NoSQL databáze*, Praha, Grada Publishing a.s, 2015, pp. 205-216.

[24] D. a. H. M. a. S. A. Georgakopoulos, „An overview of workflow management: From process modeling to workflow automation infrastructure,“ *Distributed and parallel Databases,* sv. 3, č. 2, pp. 119-153, 1995.

[25] R. Yoav, „The Dynamic Two Phase Commitment (D2PC) protocol,“ *Database Theory — ICDT '95: 5th International Conference Prague, Czech Republic, January 11--13, 1995 Proceedings,* pp. 162-176, 1995.

[26] J. G. Brookshear, „Databázové systémy,“ v *Informatika*, Praha, Computer Press, 2016, pp. 345-386.

[27] Sreality.cz, „Domy budoucnosti budou úsporné, praktické i inteligentní,“ 23 10 2015. [Online]. Available: http://www.novinky.cz/bydleni/reality-a-finance/196359-domy-budoucnosti-budou-usporne-prakticke-i-inteligentni.html.

[28] Wikipedia, the free encyclopedia, „Mesh networking,“ Wikipedia, 18 12 2015. [Online]. Available: https://en.wikipedia.org/wiki/Mesh_networking. [Přístup získán 10 1 2016].

[29] M. Sudip, C. M. Subhas a I. Woungang, Guide to Wireless Mesh Networks, United Kingdom: Springer Science & Business Media, 2009, p. 528.

[30] M. M. Vladimir Vujović, „Raspberry Pi as a Sensor Web node for home automation,“ 2015.

[31] X. L. Sheikh Ferdoush, „Wireless Sensor Network System Design Using Raspberry Pi and Arduino for Environmental Monitoring Applications,“ Procedia Computer Science, 2014.

[32] Google, „Google Visualization API Reference,“ Google Inc., 5 1 2016. [Online]. Available: https://developers.google.com/chart/interactive/docs/reference. [Přístup získán 11 1 2016].

[33] R. Cimler, J. Matyska a V. Sobeslav, „Cloud Based Solution for Mobile Healthcare Application,“ v *Proceedings of the 18th International Database Engineering & Applications Symposium*, Porto, Portugal: ACM, 2014, pp. 298-301.

[34] A. Drabek, O. Krejcar, A. Selamat a K. Kuca, „A Smart Arduino Alarm Clock Using Hypnagogia Detection During Night,“ v *Trends in Applied Knowledge-Based Systems and Data Science: 29th International Conference on Industrial Engineering and Other Applications of Applied Intelligent Systems, IEA/AIE 2016, Morioka, Japan, August 2-4, 2016, Proceedings*, Cham, Springer International Publishing, 2016, pp. 514-526.

[35] J. Horálek, J. Matyska, J. Stepan, M. Vancl, R. Cimler a V. Soběslav, „Lower Layers of a Cloud Driven Smart Home System,“ v *New Trends in Intelligent Information and Database Systems*, Springer International Publishing, 2015, pp. 219-228.

[36] J. Horalek a V. Sobeslav, „Intelligent Heating Regulation,“ v *Advanced Computer and Communication Engineering Technology: Proceedings of ICOCOE 2015*, Hradec Králové, Springer International Publishing, 2016, pp. 807-817.

[37] Maxim Integrated, „DS18B20 1-Wire Digital Thermometer Datasheet,“ 2008. [Online]. Available: http://www.ges.cz/sheets/d/ds18b20.pdf. [Přístup získán 10 1 2016].

[38] SUNROM technologies, „DHT11 - Humidity and Temperature Sensor Datasheet,“ 12 8 2012. [Online]. Available: https://www.adafruit.com/datasheets/DHT22.pdf. [Přístup získán 11 1 2016].

[39] Openweathermap, „Openweathermap API,“ Openweathermap, 2015. [Online]. Available: http://openweathermap.org/api. [Přístup získán 2016].

[40] Wikipedia, the free encyclopedia, „Berkeley sockets,“ WikiMedia, New York.

[41] Raspberry PI;, „Raspberry Pi - Teach, Learn, and Make with Raspberry Pi,“ Raspberry Pi, 15 2 2015. [Online]. Available: https://www.raspberrypi.org/. [Přístup získán 19 2 2016].

[42] MySQL, „MySQL Documentation,“ Oracle Corporation, 19 1 2016. [Online]. Available: http://dev.mysql.com/doc/. [Přístup získán 19 2 2016].

[43] JumpMind, Inc, „Symmetric DS Tutorials,“ JumpMind, Inc, 2015. [Online]. Available: http://www.symmetricds.org/doc/3.7/html/tutorials.html. [Přístup získán 19 2 2016].

[44] Apache Software Foundation, „Appache Home page,“ 11 02 2016. [Online]. Available: http://tomcat.apache.org/. [Přístup získán 19 02 2016].

[45] Github, „adafruit/Adafruit_Python_DHT: Python library to read the DHT series of humidity and temperature sensors on a Raspberry Pi or Beaglebone Black.,“ 27 08 2015. [Online]. Available: https://github.com/adafruit/Adafruit_Python_DHT. [Přístup získán 2015 03 20].

[46] Pi4J, „The Pi4J Project,“ 18 04 2015. [Online]. Available: http://pi4j.com/. [Přístup získán 20 03 2016].

[47] M. Herlihy, D. Kozlov a S. Rajsbaum, Distributed Computing Through Combinatorial Topology, 1. editor, Waltham, MA: Elsevier Inc., 2014, p. 336.

[48] G. Coulouris, J. Dollimore, T. Kindberg a G. Blair, Distributed Systems: Concepts and Design, 5th Edition, 5. editor, Addison-Wesley, United States of America: Pearson Education. Inc., 2011, p. 1008.

[49] Loxone, „Centrála digitální domácnosti - Loxone Miniserver,“ 10 1 2016. [Online]. Available: http://www.loxone.com/cscz/produkty/miniserver/miniserver.html.

[50] A. Pereira, M. Atri, P. Rogalla, T. Huynh a M. E. O'Malley, „Assessment of feasibility of running RSNA's MIRC on a Raspberry Pi: a cost-effective solution for teaching files in radiology," Springer Berlin Heidelberg, Berlin , 2015.

[51] G. Alonso, „Remote Procedure Call (RPC)," Swiss Federal Institute of Technology, 16 06 2015. [Online]. Available: http://slideplayer.com/slide/5211289/. [Přístup získán 08 06 2016].

Příloha A

Struktura přiloženého CD:

Node [DIR] – zdrojové kódy uzlů

MasterNodeServer [DIR] – Zdrojové kódy uživatelského rozhraní

Sec_David_DP.pdf – Diplomová práce ve formátu PDF

Dokumentace.pdf – Dokumentace ke zdrojovým kódům ve formátu PDF

Univerzita Hradec Králové
Faculty of Informatics and Management
Akademický rok: 2014/2015

Studijní program: Applied Informatics
Forma: Full-time
Obor/komb.: Aplikovaná informatika (ai2-p)

Podklad pro zadání DIPLOMOVÉ práce studenta

PŘEDKLÁDÁ:	ADRESA	OSOBNÍ ČÍSLO
Šec David	Pravy 35, Pravy	I1477

TÉMA ČESKY:

Distribuované systémy

TÉMA ANGLICKY:

Distributed systems

VEDOUCÍ PRÁCE:

Mgr. Josef Horálek, Ph.D. - KIT

ZÁSADY PRO VYPRACOVÁNÍ:

Cílem práce je podrobně představit principy distribuovaných systémů s důrazem na meziprocesorovou komunikaci. Dále představit principy distribuovaných databází. V praktické části implementuje řešení distribuovaného systému s několika fyzickými uzly s cílem sbírání dat z okolního prostředí a na jejich základě vykonávat zadané akce.

OSNOVA:
1.ÚVOD
3.Meziprocesorová komunikace v DS
4.Synchronizace
5.Topologie DS
6.Distribuované databáze
7.Model návrhu využití Distribuovanho systému

SEZNAM DOPORUČENÉ LITERATURY:

COULOURIS, George F. Distributed systems: concepts and design. 5th ed. Boston: Addison-Wesley, c2012, xvi, 1047 p. ISBN 01-321-4301-1.

S. TANENBAUM, Andrew a Maarten VAN STEEN. Distributed Systems: Principles and Paradigms. 2007. United States od America: Prentice Hall, 2007. 2nd edition. ISBN 0-13-239227-5.

Podpis studenta: ..... **Datum:** 13.10.2015

Podpis vedoucího práce: ..... **Datum:** 13.10.2015