Západočeská univerzita v Plzni
Fakulta aplikovaných věd
Katedra informatiky a výpočetní techniky

DIPLOMOVÁ PRÁCE

Plzeň, 2006 Zbyněk Novotný

Západočeská univerzita v Plzni
Fakulta aplikovaných věd
Katedra informatiky a výpočetní techniky

DIPLOMOVÁ PRÁCE

Sdílený virtuální svět

Plzeň, 2006

Zbyněk Novotný

Abstrakt

Thanks to the development in the area of computer network technologies, it is now possible to create distributed three-dimensional virtual environments to fully immerse the user. The environments provide the experience of sharing the world with other people. However, designing such a virtual environment is no easy task. The designers need to make many decisions that influence the system at all levels. Those related to the underlying network subsystem belong to the most important ones.

This thesis proposes the foundation for one such three-dimensional virtual environment system that allows its users to interact with the world using basic geometric objects. The goal of this task is achieved using existing libraries for scene representation (OpenSceneGraph) and virtual reality systems (VRECKO) designed to facilitate setting up and internal representation of the virtual scene, and also interaction with the objects located therein. The RakNet networking library used in this work also makes the handling of network traffic much easier.

Poděkování

Na tomto místě bych rád poděkoval panu Ing. Petru Lobazovi za vedení této diplomové práce a za jeho podnětné dotazy a připomínky. Dále bych chtěl poděkovat panu Ing. Jiřímu Ledvinovi, CSc. a panu Ing. Martinu Šimkovi, PhD. za jejich hodnotné rady a připomínky, jež velkou měrou přispěly k výsledné kvalitě návrhu a řešení této práce. V neposlední řadě bych také rád vyjádřil poděkování panu Mgr. Janu Flasarovi, PhD. z katedry počítačové grafiky a designu Fakulty informatiky Masarykovy univerzity v Brně za jeho neocenitelnou pomoc při řešení řady problémů vyskytnuvších se během tvorby grafické části programu, jenž je produktem této práce.

Obsah

1 Úvod 1

2 Základní pojmy a principy počítačových sítí 2
2.1 Zpoždění . . . 2
2.1.1 Fyzikální důvody vzniku latence . . . 2
2.1.2 Režie při zpracování koncovými body trasy přenosu . . . 2
2.1.3 Zpoždění daná strukturou sítě . . . 2
2.2 Šířka pásma . . . 3
2.3 Spolehlivost sítě . . . 3
2.3.1 Zahození paketu . . . 3
2.3.2 Poškození paketu . . . 4
2.4 Síťový komunikační protokol . . . 4
2.4.1 Formát paketů protokolu . . . 4
2.4.2 Sémantika paketů protokolu . . . 4
2.4.3 Chování při výskytu chyby . . . 4
2.5 Problematika socketů a portů . . . 5
2.5.1 Princip socketů a portů . . . 5
2.6 Základní internetové protokoly . . . 6
2.6.1 Internetový protokol . . . 6
2.6.2 Protokol pro kontrolu přenosu . . . 7
2.6.3 Uživatelský datagramový protokol . . . 8
2.7 Všesměrové vysílání pomocí UDP protokolu . . . 8
2.8 Multicasting pomocí UDP protokolu . . . 9
2.8.1 Směrování zpráv v multicastingu . . . 9
2.9 Výběr vhodného protokolu pro systém virtuální reality . . . 10
2.9.1 Použití TCP/IP . . . 10
2.9.2 Použití UDP/IP . . . 11
2.9.3 Použití všesměrového vysílání . . . 12
2.9.4 Použití IP multicastingu . . . 13

3 Síťové architektury distribuovaných systémů 14
3.0.5 Fyzické spojení . . . 14
3.0.6 Logické spojení . . . 15
3.1 Architektura klient-server . . . 16
3.2 Architektura klient-server s více servery . . . 16
3.3 Architektury typu peer-to-peer . . . 17
3.3.1 Virtuální systémy pro lokální síť . . . 17
3.3.2 Virtuální systémy pro rozsáhlé sítě . . . 18

4 Udržení konzistence stavu v dynamickém systému 19
4.1 Definice problému udržení konzistence v dynamickém virtuálním světě . . . 19
4.2 Souhrn metod a metoda centralizovaného repozitáře . . . 20
4.2.1 Popis a vlastnosti centralizovaných informačních repozitářů . . . 20
4.2.2 Popis a vlastnosti virtuálního repozitáře . . . 21
4.3 Metoda častých aktualizací stavu . . . 22

4.3.1 Vlastnictví objektů . . . 23
4.3.2 Řešení konfliktů . . . 23
4.3.3 Zdokonalení metody . . . 24
4.4 Predikce sdíleného stavu systému . . . 24
4.4.1 Predikce a konvergence . . . 25
4.4.2 Predikce využívající derivační polynomy . . . 25
4.4.3 Hybridní predikce s derivačními polynomy . . . 25
4.4.4 Omezení derivačních polynomů . . . 26
4.4.5 Objektově specializovaná predikce . . . 26
4.4.6 Konvergenční algoritmy . . . 27
4.4.7 Využití predikce pro nepravidelné aktualizace . . . 28

5 Realizace síťové části systému **29**
5.1 Definice úkolu a základní vytyčení cílů . . . 29
5.2 Obecný popis a požadavky na realizaci systému . . . 29
5.3 Návrh a realizace síťové vrstvy systému . . . 29
5.3.1 Použitelné síťové architektury . . . 29
5.3.2 Dostupné protokoly . . . 30
5.3.3 Knihovna RakNet . . . 31
5.3.4 Časové značkování paketů . . . 32
5.4 Popis komunikačního protokolu . . . 32
5.5 Datové struktury pro aktualizační zprávy . . . 33
5.5.1 Obecné informace pro zpracování a adresování zpráv . . . 33
5.5.2 Zpráva požadavku přihlášení do systému . . . 34
5.5.3 Zpráva potvrzující přijetí žádosti o připojení . . . 35
5.5.4 Zpráva zamítající žádost o spojení . . . 35
5.5.5 Zpráva o odhlášení/„vytikání“ uživatele . . . 35
5.5.6 Zpráva o ukončení inicializace . . . 35
5.5.7 Zpráva o zamčení a odemčení kostky . . . 35
5.5.8 Zpráva o vytvoření kostky . . . 36
5.5.9 Zpráva o transformaci kostky . . . 36
5.6 Popis procesu inicializace nově přihlášeného klienta . . . 36
5.7 Spolehlivosti zpráv a řešení problému ztracených zpráv . . . 37

6 Infrastruktura aplikací klienta a serveru **39**
6.1 Společné znaky a komponenty klienta a serveru . . . 39
6.1.1 Vláknová architektura aplikací . . . 39
6.1.2 Vstupní a výstupní fronty zpráv . . . 39
6.1.3 Rozhraní handlerů zpráv . . . 39
6.1.4 Scéna . . . 40
6.1.5 Rozhraní pro spojení grafické a síťové části aplikace . . . 40
6.2 Specifické komponenty aplikace klienta . . . 41
6.2.1 Třída pro zobrazení světa . . . 41
6.3 Specifické komponenty aplikace serveru . . . 41
6.3.1 Registr uživatelů . . . 41
6.4 Princip funkce klienta a serveru . . . 42
6.4.1 Základní inicializace . . . 42

6.4.2 Spuštění podpůrných vláken a hlavních smyček 42
6.4.3 Zpracování zpráv v hlavním vlákně 43

7 Systém VRECKO a jeho propojení se síťovou vrstvou 44
7.1 Obecný popis a koncepce systému 44
7.2 Knihovna OpenSceneGraph a její základní datové typy 44
7.3 Architektura systému VRECKO 45
7.3.1 `BaseClass` – základ komunikace v systému 45
7.3.2 Reprezentace objektů scény 45
7.3.3 Scéna a svět 45
7.3.4 Aktualizace komponent a objektů v systému 46
7.3.5 Mechanismus předávání zpráv mezi objekty systému VRECKO 46
7.3.6 Adresování vstupů a výstupů objektů 46
7.3.7 Definice schopností objektů 47
7.3.8 Správa vstupních zařízení 48
7.4 Propojení se síťovou vrstvou 48
7.4.1 Příprava na komunikaci 48
7.4.2 Zpracování zpráv mezi sítí a systémem VRECKO na straně klienta . . 49
7.4.3 Zpracování zpráv mezi sítí a systémem VRECKO na straně serveru . . 49

8 Testování a zhodnocení realizace 51

9 Závěr 52

Prohlašuji, že jsem diplomovou práci vypracoval samostatně a výhradně s použitím citovaných pramenů.

V Plzni dne 29. srpna 2006, ...

1 Úvod

Účelem této práce je poskytnout detailní popis způsobu řešení při návrhu našeho distribuovaného systému pro virtuální realitu. K řešení tohoto problému je použit již existující systém, nazvaný VRECKO, který však není distribuovaný. Tento systém disponuje mnoha vlastnostmi, umožňujícími jeho výraznou rozšiřitelnost a upravitelnost, což jsou znaky, jež jsou během realizace návrhu řešení s výhodou používány a významně celkový proces řešení zjednodušují.

Jelikož existuje mnoho cest, kterými se lze vydat při řešení tohoto typu problému, je třeba se s nimi nejdříve blíže seznámit a poté si zvolit tu, která bude nejlépe vyhovovat našim požadavkům a zároveň nás co nejrychleji dovede k cíli. První část této práce je proto věnována popisu základních pojmů a principů v oblasti počítačových sítí. Probírány jsou jak základní pojmy a protokoly, jejichž vlastnosti hrají důležitou roli při návrhu distribuovaného systému s důrazem na výkon a rychlost odezvy systému, tak i možné architektury síťových systémů, v dnešní době běžně používaných.

V další části jsou diskutovány možné postupy, sloužící především k udržování konzistentního sdíleného stavu v rámci systému virtuální reality. U každého postupu je probírána jeho charakteristika, výsledky, kterých lze jeho použitím dosáhnout, a shrnuty jsou také jeho výhody či nevýhody. Pokud byl daný postup zajímavým způsobem implementován některým z existujících systému pro virtuální realitu, je tento systém v krátkosti popsán také.

Po skončení popisu principů pro uchování konzistence v systému již následuje popis vlastního řešení našeho problému. Po krátké definici problému a vytyčení základních požadavků, kladených na naše řešení, je diskutována výhodnost specifických síťových architektur a protokolů a poté následuje krátký popis knihovny pro síťovou komunikaci, na které založíme síťovou funkcionalitu našeho systému. V této části jsou také navrženy datové struktury síťových zpráv, které budeme používat, a popsán je i jejich význam a úroveň spolehlivosti, s jakou je každá z těchto zpráv doručována.

Poté je popsána architektura programů klienta a serveru, jejich významné společné i specifické komponenty a princip vzájemné spolupráce těchto komponent. Probírány jsou metody zpracování zpráv, přijatých od vzdáleného hostitele, a způsob, jakým funguje zobrazování scény pro uživatele, a způsob uložení a využití dat připojených uživatelů.

Splnění samotného cíle této práce (tj. rozšíření existujícího systému virtuální reality o možnost síťové komunikace a tím dosažení možnosti spolupráce většího množství uživatelů v rámci tohoto systému) je věnována poslední část práce, která popisuje použitý systém, jeho architekturu a posléze také způsob, jakým je dosaženo komunikace mezi tímto systémem a námi vytvořenou síťovou vrstvou.

V závěru práce jsou zmíněny výsledky, vyplývající z testování vytvořeného systému, a také popis hlavních problémů, s nimiž bylo nutné se během celého procesu realizace řešení vypořádat. Jsou však také shrnuty poznatky a zkušenosti, získané při realizaci, a navrženy další možné směry, kterými se vývoj tohoto systému může ubírat.

2 Základní pojmy a principy počítačových sítí

Počítačové sítě jsou základním pilířem pro stavbu distribuovaných virtuálních prostředí, proto je na místě si nejprve popsat základní pojmy a principy, nejčastěji se vyskytující v oblasti počítačových sítí v souvislosti s distribuovanými systémy virtuálních prostředí.

2.1 Zpoždění

Zpoždění (angl. *network latency*) určuje dobu, jakou potřebuje jeden datový blok (angl. *packet*) k překonání vzdálenosti mezi dvěma uzly na síti. Je to velmi důležitý faktor v návrhu sdíleného virtuálního prostředí, neboť určuje, za jak dlouho po vytvoření a odeslání paketu zdrojovou aplikací virtuálního prostředí tento paket uvidí aplikace na straně příjemce. Tento fakt hraje klíčovou roli při realizaci distribuovaného systému virtuálního prostředí ze dvou důvodů:

- Přímo ovlivňuje kvalitu vjemu a velikost celkového „ponoření“ do virtuálního světa a také určuje, do jaké míry jsou informace přijaté přijímající stranou aktuální.
- Jelikož je zpoždění dáno samotnou strukturou sítě a mnoha dalšími souvisejícími parametry, ve valné většině případů nelze tento faktor (výrazně) redukovat.

Důvodů vzniku zpoždění je několik druhů. Nyní si je stručně popíšeme.

2.1.1 Fyzikální důvody vzniku latence

Na fyzikální úrovni jsou všechna data posílaná přes síťové spoje signály různých forem (elektrický náboj, světelný impuls či elektromagnetické vlnění), avšak všechny se šíří rychlostí světla. Platí samozřejmě, že rychlost světla v různých materiálech je vždy nižší než rychlost světla ve vakuu, z čehož vyplývá, že jednou z příčin vzniku latence je samotný úbytek rychlosti signálu při průchodu určitým prostředím. Ačkoli se může na první pohled zdát, že tyto důvody nehrají příliš významnou roli, je třeba si uvědomit, že se vzrůstající vzdáleností mezi oběma koncovými body trasy přenosu sílí vliv i této složky celkového zpoždění. Jestliže je během přenosu signálu zapotřebí směrování přes orbitální satelit, je vliv na celkové zpoždění ještě markantnější.

2.1.2 Režie při zpracování koncovými body trasy přenosu

Dalším zdrojem latence jsou počítačové systémy, tvořící koncové body spojení, které musejí daný datový blok zpracovat. Jedná se o dobu od chvíle, kdy je aplikací v paměti vytvořen paket, až do okamžiku, kdy síťové rozhraní fyzicky odesílá tento paket do sítě. Délka této doby tedy zcela závisí na tom, jak rychle je systém jako celek schopen daný datový blok zpracovat.

2.1.3 Zpoždění daná strukturou sítě

Třetím typem složky celkového zpoždění je zpoždění způsobené sítí jako takovou. Zde se jedná především o to, že pokud je paket během cesty od zdroje k cíli přesměrován do jiných sítí, musejí jej při přechodu mezi jednotlivými částmi sítě nejprve zpracovat různá síťová zařízení, zejména směrovače ((angl. *routers*)). Doba, kterou router potřebuje k tomu, aby paket zpracoval a předal dál, se samozřejmě taktéž promítá do celkového zpoždění daného datového paketu.

2.2 Šířka pásma

Šířka pásma (angl. *network bandwidth*) je veličina, která určuje, jak velké množství dat dokáže síť přenést za jednotku času. Je přímo závislá na kvalitě síťového hardwaru a propustnosti propojujících médií. V dřívějších dobách modemů a telefonních linek se datová propustnost pohybovala v rozmezí 14 400 – 56 000 bitů za vteřinu (tj. 14,4 – 56 Kbps), přičemž horní hranice šířky pásma pro telefonní linku byla 64 Kbps.

Společně s tím, jak postupoval vývoj v oblasti technologií připojení počítačů k síti, se však zvyšovala i maximální šířka pásma. V dnešní době jsou běžně dostupná připojení rychlostí řádově stovek kilobitů až jednotek megabitů za vteřinu. V případě lokálních ethernetových sítí se maximální propustnost (při použití metalických a optických spojů) pohybuje v řádech desítek až stovek megabitů (Mbps) za vteřinu.

2.3 Spolehlivost sítě

Spolehlivost sítě (angl. *network reliability*) je další vlastností počítačové sítě, kterou je třeba brát v úvahu během návrhu síťové části distribuovaného virtuálního systému. Tato vlastnost určuje, jaké množství z celkového objemu přenesených dat se „ztratilo" během přenosu od zdroje k cíli.

Lze rozeznat dva důvody ztráty paketu:

1. *Zahození* paketu – v tomto případě příjemce data vůbec neobdrží, jelikož paket byl v určitém uzlu sítě zahozen.

2. *Poškození* paketu – v takovémto případě došlo během přenosu paketu k jeho poškození, a tudíž se tento paket stal pro příjemce nepoužitelným.

2.3.1 Zahození paketu

Nejčastější příčinou zahození paketu bývá nadměrná vytíženost zpracujícího routeru na cestě mezi počátkem a cílem cesty paketu. Důvod je ten, že data (pakety) přicházejí na router nerovnoměrně a router je samozřejmě schopen je zpracovat pouze s určitou omezenou maximální rychlostí. Jestliže na router dorazí náhlý větší balík dat, musí je router uložit do fronty ke zpracování. Pokud by však velikost přijatých dat byla příliš velká a paměť určená pro tuto frontu by došla, musel by router všechny pakety, které se nevešly do vyhrazené paměti, zahodit. V takových případech hraje hlavní roli celková vytíženost sítě v danou dobu. Ve špičkách může celková ztrátovost překročit i polovinu celkového objemu zpracovaných paketů, a to i přesto, že v jinou dobu může být naopak míra zahozených paketů minimální.

Odpověď na otázku, jak zjistit, zda příjemce opravdu obdržel poslaná data, leží v *potvrzeních*, což mohou být jednak speciální zprávy k tomu určené, anebo jiné zprávy, které posílá příjemce zpět zdroji a díky kterým je zdrojový systém schopen říci, že původní zpráva dorazila na místo určení v pořádku.

Pokud zdroj neobdrží od příjemce potvrzující zprávu v rámci daného časového intervalu, usoudí, že data do cíle nedorazila, a pošle je znovu. Je ale samozřejmě možné, že data ve skutečnosti do cíle dorazila, avšak potvrzovací zpráva se ztratila při přenosu.

2.3.2 Poškození paketu

Ke ztrátě paketu vlivem *poškození* dochází obvykle velmi zřídka (dle [SZ99] se míra v těchto případech udává řádově jedním bitem z 10^{10} bitů – nebo i méně). Takové poškození je ve většině případů způsobeno nedokonalým propojením síťových prvků nebo vlivem působení vnějšího magnetického nebo elektrického pole. Míra vadných paketů je největší v případě bezdrátových sítí, kdy jsou data převedena na elektromagnetické signály, přenášené vzduchem, neboť může dojít k rušení vlivem jiných bezdrátových přenosů, fyzických překážek na trase mezi zdrojem a cílem přenosu, či vlivem počasí.

Aby bylo možné detekovat poškození paketu, je do paketu obvykle vkládán *kontrolní součet* dat v něm obsažených, obvykle ve formě tzv. *cyklického redundantního kódu* (angl. *cyclic redundancy check, CRC*). Příjemce paketu je pak s použitím tohoto kódu schopen určit, zda daný datový blok dorazil v pořádku či zda je nutné, aby odesílatel poslal data znovu. V některých případech je možno do paketu vložit i informaci, kterou může příjemce v případě nutnosti použít k opravě vadných bitů v datech paketu – tzv. *kódu pro opravu chyb* (angl. *error-correcting code*). S tímto kódem je možné opravit jeden nebo i více vadných bitů a zabránit tak nutnosti přeposlání daného paketu.

2.4 Síťový komunikační protokol

Síťový komunikační protokol (angl. *network protocol*) definuje množinu pravidel určujících, jakým způsobem mohou mezi sebou aplikace komunikovat. Tato pravidla tvoří základ komunikace po síti. Síťový komunikační protokol je určen těmito třemi složkami:

1. *formátem paketů*,
2. *sémantikou paketů*,
3. *chováním při výskytu chyby.*

2.4.1 Formát paketů protokolu

Při vzájemné komunikaci dvou uzlů na síti je nutné zajistit, aby každý uzel rozuměl tomu, co ten druhý říká, a aby oba uzly byly schopny detekovat chybný paket. Formát paketů proto definuje strukturu každého paketu určující, jak daný paket vytvořit a jak z něj číst.

2.4.2 Sémantika paketů protokolu

Tato složka komunikačního protokolu zajišťuje, že příjemce daného paketu chápe jeho smysl a je schopen se na základě přijetí tohoto paketu patřičně zachovat. Nezbytnou podmínkou je, aby přijatá zpráva nebyla chybně interpretována. Vzhledem k tomu, že přijetím určité zprávy se příjemce dostává do specifického stavu, se obvykle sémantika paketů popisuje konečným automatem.

2.4.3 Chování při výskytu chyby

Třetí složka komunikačního protokolu definuje, jak mají obě komunikující strany reagovat v případě chybového stavu. Chybový stav může být způsoben zejména (avšak nejen) těmito událostmi:

- přijetím paketu s chybnou strukturou,
- vnější chybou sofwaru nebo hardwaru, na kterém komunikující program běží,
- chybou, která způsobí, že daný úkol nelze dokončit.

Jelikož sítě jsou svou povahou nespolehlivá přenosová média, je jisté, že k chybám dojde, a proto je nutné, aby v každém případě výskytu chybového stavu existovala dohoda určující další kroky, neboť na tom může záviset použitelnost protokolu jako takového.

Vzhledem k faktu, že reakce na chybové stavy jsou těsně spjaty se samotnou sémantikou paketů, bývá obvykle popis chybových stavů a reakce na ně zahrnuty ve stavovém automatu pro sémantiku paketů.

V dnešní době existují stovky protokolů. Vznik protokolu je vždy řízen účelem, který má navrhovaný protokol plnit. Vysokoúrovňové protokoly mohou například zajišťovat přenos zvukových a obrazových dat či dokumentů ze sítě WWW (řadíme sem protokoly typu HTTP, FTP, MMS apod.), protokoly nižších úrovní mohou naopak řešit komunikaci hostitele s ostatními řídícími prvky sítě, jako jsou například konfigurační servery, servery jmen apod. Do této kategorie spadají například protokoly typu DHCP, DNS, NTP atd.

2.5 Problematika socketů a portů

V další části si vysvětlíme základní principy socketů a využití portů při komunikaci mezi jednotlivými počítači, připojenými k síti. Teorie socketů vychází z těchto základních předpokladů:

- Na každém počítači může v jednu chvíli běžet větší počet procesů, z nichž každý může kdykoliv přistoupit k síti.
- Každý z běžících procesů může komunikovat s jiným vzdáleným hostitelem nebo mohou dva nebo více procesů komunikovat s totožným hostitelem.
- Každý proces může využít více než jedno spojení najednou, přičemž s každým spojením může používat jiný komunikační protokol.

Je třeba zajistit, že každý odeslaný paket dorazí na správné místo určení a každý příchozí paket je přijat tou správnou aplikací. Tyto dva požadavky řeší tzv. *BSD sockety*, původně vyvinuté pro operační systémy typu UNIX, avšak postupně přejaté drtivou většinou ostatních systémů.

2.5.1 Princip socketů a portů

Socket představuje jakýsi bod spojení, skrze který komunikuje lokální proces s jiným (ten může běžet jak na vzdáleném hostiteli, tak i lokálně). Slouží jako identifikátor jednoho komunikačního kanálu a obvykle s sebou nese těchto pět základních typů informace:

1. *Informace o protokolu* – tato informace obvykle zahrnuje typ spolehlivosti přenášených paketů.
2. *Adresu vzdáleného hostitele* – každý paket, poslaný přes tento socket, si s sebou nese informaci o adrese cílového hostitele; pokud socket tuto informaci neobsahuje, je na odesílající aplikaci, aby ji dodala.

3. *Číslo portu vzdáleného hostitele* – toto 16-bitové číslo jednoznačně identifikuje socket na přijímajícím počítači; každý protokol používá pro svou potřebu jiná čísla portů a s touto informací je možné zaručit, že daný datový paket bude na cílovém hostiteli přijat správnou aplikací; pokud socket neudržuje informaci o čísle portu používaného druhou stranou, musí ji dodat aplikace, která chce paket odeslat.

4. *Adresu zdrojového hostitele* – tato informace, ačkoli je zřídkakdy zapotřebí u zdrojového hostitele, může být pro zdroj zajímavá, pokud ma zdrojový hostitel přiděleno více adres.

5. *Číslo portu lokálního hostitele* – stejně jako v případě vzdáleného hostitele se i zde jedná o 16-bitové číslo, jednoznačně identifikující aplikaci, která paket odesílá; informace o čísle portu a adrese zdrojového hostitele, uložená v paketu, umožňuje přijímající aplikaci odpovědět na příchozí zprávy.

Jelikož se jedná o 16bitové číslo, je možné použít maximálně 2^{16} (= 65 536) čísel portů. Každá aplikace si může v podstatě libovolně zvolit číslo svého komunikačního portu, avšak existují jistá pravidla pro přidělování portů. Tato pravidla vznikla z toho důvodu, že za dobu existence počítačových sítí se některé protokoly prosadily natolik, že se staly de facto standardem a bylo zapotřebí zaručit, že dvě aplikace používající ke komunikaci stejný protokol, budou používat také stejný komunikační port. V tabulce níže je stručný souhrn těchto pravidel (rozsahy dle [SZ99]).

Interval čísel portů	**Popis intervalu**
1...1023	Rozsah „rezervovaných“ portů pro určité známé (standardizované) protokoly
1 024...49 151	Rozsah „registrovaných“ portů pro některé známé protokoly
49 152...65 536	Rozsah „nepřiřazených“ portů určených k veřejnému použití

Tabulka 1: Souhrn pravidel pro přidělování portů

Jak již bylo řečeno výše, lze použít pro vlastní potřebu v podstatě libovolné neobsazené číslo portu. Pokud je ale žádoucí, aby náš distribuovaný virtuální systém měl oficiálně přidělená čísla portů, je možné si o ně zažádat u k tomu určené autority s názvem Internet Assigned Numbers Authority (zkráceně IANA).

2.6 Základní internetové protokoly

Výběr vhodného protokolu je jedním ze základních rozhodnutí, která musí designér systému distribuovaného virtuálního prostředí učinit již na samotném začátku procesu návrhu. Každý protokol má totiž vlastní charakteristiku, sadu vlastností, které mohou zásadním způsobem ovlivnit celkový výkon výsledného systému. Při rozhodování, jaký protokol použít, si musíme položit otázku, jaké vlastnosti od protokolu požadujeme (přičemž samozřejmě musíme zvážit všechny vlastnosti daného protokolu, tj. i ty, které mohou mít také negativní vliv na náš systém) a zároveň jaké vlastnosti chceme, aby měl náš systém. Konečné rozhodnutí pak závisí na nalezení „rovnováhy“ mezi těmito dvěma skupinami požadavků.

2.6.1 Internetový protokol

Internetový protokol (angl. *Internet protocol, IP*) tvoří základní úroveň komunikačních protokolů a zajišťuje komunikaci mezi hostitelskými počítači a routery. Tento protokol tvoří

abstrakci nad typem spojující linky a je schopen rozdělovat pakety na menší celky a opět je skládat do původní podoby, pokud přenosová linka není schopna pojmout pakety větších velikostí. V hlavičce paketů tohoto protokolu je také umístěn parametr určující maximální dobu života (angl. *Time To Live, TTL*) každého paketu, čímž je vyloučena možnost, že by v případě nějakého problému paket věčně koloval sítí. Parametr určující dobu života je tvořen čítačem nastaveným na určitou hodnotu[1], přičemž s každým průchodem tohoto paketu routerem z něj router odečte jedničku. Po dosažení nulové hodnoty doby života je paket zahozen.

Ve většině případů není třeba, aby aplikace, komunikující přes síť, používaly tento protokol přímo. Místo toho mohou použít některý z protokolů vyšší úrovně, založených na IP. V závislosti na typu zvoleného protokolu tím získají další služby, které protokol nabízí (podporu potvrzování, portů,...).

2.6.2 Protokol pro kontrolu přenosu

Protokol pro kontrolu přenosu (angl. *Transmission Control Protocol, TCP*) je poměrně složitý, avšak v současné době zdaleka nejpoužívanější protokol pro přenos dat z Internetu. Verze tohoto protokolu založená na Internetovém protokolu se obvykle označuje jako TCP/IP. Protokol TCP bývá označován jako spolehlivý, neboť disponuje vlastnostmi, které zajišťují, že pakety vždy dorazí k příjemci v pořádku a ve správném pořadí. Této spolehlivosti je dosaženo díky následujícím vlastnostem:

- Správného pořadí je dosaženo díky
  - *číslování paketů* – každý paket si nese své sériové číslo a příjemce je tak schopen ověřit, že příchozí pakety skutečně dorazily ve správném pořadí,
  - *mechanismu kontroly toku dat* – protokol TCP je schopen přizpůsobit se rychlosti sítě a hostiteli na opačné straně spojení a díky tomu zaručuje, že pakety nikdy nebudou zasílány rychleji než je síť schopna je přenést nebo příjemce zpracovat.
- Jistoty správného doručení je dosaženo díky
  - *automatickému potvrzování* – protokol TCP implementuje automatické potvrzování přijatých zpráv a díky tomu můžeme mít jistotu, že naše zprávy přijímající strana opravdu obdržela,
  - *automatické kontrole integrity dat* – data každého paketu jsou po svém přijetí podrobena kontrole pomocí kontrolního součtu, jenž je uložen v hlavičce paketu; tím lze ověřit, že data nebyla během přenosu poškozena.

Protokol TCP kromě výše popsaných vlastností také automaticky řeší dělení dat na pakety, jejich opětovné skládání a automatické zahazování duplicitních paketů. Díky všem těmto vlastnostem mohou aplikace, využívající tento protokol, chápat spojení mezi dvěma koncovými body jako spolehlivý proud dat.

Za nevýhodu tohoto protokolu lze považovat fakt, že pro zaručení této úrovně spolehlivosti je zapotřebí jistá režie (zejména nutnost přenosu informace o čísle zprávy a jejím kontrolním součtu), avšak i samotný princip potvrzování zpráv lze chápat jako režii navíc. Pro příjemce také hraje důležitou roli to, že musí opravdu zpracovat všechny příchozí pakety a není možné část paketů přeskočit, třebaže by to z hlediska funkce aplikace nemělo vliv. Vždy je nutné

[1] Jedná se o jednobajtový čítač, tudíž jeho rozsah je tvořen intervalem $\langle 0, 255 \rangle$.

zpracovat celý proud tak, jak byl zdrojem poslán. Jak si později ukážeme, ne vždy je tato spolehlivost zapotřebí (či dokonce nutná).

2.6.3 Uživatelský datagramový protokol

Uživatelský datagramový protokol (angl. *User Datagram Protocol, UDP*) je také založen na IP protokolu, avšak narozdíl od protokolu TCP je to jednoduchý a lehký protokol, který nezná pojem „spojení“, což lze chápat také tak, že UDP pakety vysílané zdrojem jsou v podstatě vysílané naslepo. Každý paket je odeslán s maximální snahou o doručení, avšak neexistuje *žádná* záruka, že k příjemci opravdu dorazí. UDP protokol nepodporuje žádnou detekci vadných či nedoručených zpráv. Ve výsledku to znamená, že pakety mohou, ale nemusejí k příjemci dorazit, a že mohou být doručeny v nesprávném pořadí. Zdroj také může usuzovat, že příjemce „žije“, jen z jeho odpovědí. Narozdíl od TCP protokolu se v UDP protokolu neukládají žádné stavové informace o komunikaci, což znamená, že stavy jsou dány samotným obsahem každého poslaného či přijatého paketu. V případě příliš velkých paketů, které je třeba rozdělit na menší části, také existuje riziko, že některá z těchto částí se může během přenosu po síti ztratit, čímž dojde ke znehodnocení informace daného datového bloku jako celku. Z tohoto důvodu by neměly být pakety posílané UDP protokolem příliš velké.

Výše popsané vlastnosti protokolu UDP lze na jednu stranu chápat jako nevýhody oproti protokolu TCP, avšak v mnoha případech je naopak velmi žádoucí, aby použitý protokol nebyl zatížen režií, nutnou k zajištění korektnosti přenášených dat. Díky těmto zdánlivým nedostatkům jsou pakety, přenášené s využitím UDP protokolu, snáze zpracovatelné jak zdrojem, tak příjemcem. Mimo to lze pakety, připravené k přenesení po síti, odeslat ihned a není třeba je stavět do fronty, aby byla zajištěna správná posloupnost jejich odeslání. Analogický případ platí pro příjemce, kdy naopak není zapotřebí přijaté pakety rovnat do fronty a případně čekat na ztracené, dříve odeslané zprávy, ale je možno přijaté pakety zpracovat ihned po jejich doručení.

Z výše řečeného lze usuzovat, že protokol UDP snadno najde využití všude tam, kde je důležitější rychlost doručení a zpracování než spolehlivost přenosu a kde je potřeba obhospodařovat vícero přípojených uživatelů, aniž bychom chtěli být zatíženi nutností spravovat skutečné spojení s každým z nich. A protože rychlost odezvy a škálovatelnost pro různé počty uživatelů jsou parametry, které při návrhu distribuovaného virtuálního systému hrají velmi významnou roli, dávají návrháři těchto systémů přednost protokolu UDP daleko častěji než protokolu TCP. Většina existujících systémů virtuální reality implementuje část funkcionality protokolu TCP (zejména potvrzování a detekci vadných paketů) a občasná ztráta paketu navíc ve spoustě případů vůbec nevadí.

Pokud je třeba zaslat jednu zprávu většímu počtu příjemců a používáme-li protokol UDP, je možné použít jeden ze tří postupů, jak toho docílit. V dalších sekcích si je popíšeme.

2.7 Všesměrové vysílání pomocí UDP protokolu

První možnost, která se nám nabízí, je poslat tutéž zprávu každému příjemci zvlášť. Tento postup má však dvě základní nevýhody. První nevýhodou je fakt, že spotřebováváme zbytečně velkou šířku pásma, neboť ten samý paket musíme poslat několikrát za sebou. Druhým neduhem je pak to, že všichni účastníci v systému (kteří jsou zároveň zdroji) musejí udržovat aktuální informace o ostatních příjemcích.

Druhá možnost, kterou máme a kterou se budeme zabývat nyní, je *všesměrové vysílání*

(angl. *IP broadcasting*). Tato metoda je založena na zcela slepém rozesílání paketů, kdy všem účastníkům ve virtuálním systému k přijímání i posílání zpráv stačí poslouchat na známém portu a o žádné jiné detaily probíhající komunikace se nemusejí starat. Každý vyslaný paket je doručen všem hostitelům na síti a není tudíž třeba jej posílat zvlášť každému zájemci.

Hlavní nevýhoda tohoto přístupu vyplývá z jeho samotné podstaty. Necílené rozesílání zpráv znamená, že zpráva je doručena *úplně všem* hostitelům připojeným k dané síti, nezávisle na tom, zda mají či nemají zájem o tuto zprávu, přičemž nelze určit, zda je zpráva skutečně určena danému hostiteli, dokud nejsou UDP data zpracována operačním systémem[2]. Broadcasting také nelze použít pro virtuální systémy, které mají fungovat i přes Internet. Správnou funkci všesměrového vysílání lze totiž zaručit jen tehdy, pokud má daný systém pro virtuální realitu fungovat po místní síti (tj. v rámci jednoho rozsahu adres).

2.8 Multicasting pomocí UDP protokolu

IP multicasting je třetí možností, jak doručit zprávu cílovému hostiteli. Výhodou této metody distribuce zpráv je to, že narozdíl od broadcastingu nezatěžuje zprávami hostitele, kteří o ně nemají zájem, a lze jej použít i mimo lokální síť.

2.8.1 Směrování zpráv v multicastingu

Vzhledem k výše popsaným výhodám oproti všesměrovému vysílání je pochopitelné, že směrování zpráv je v tomto případě složitější. Základním předpokladem toho, aby hostitel mohl přijímat zprávy od určitého zdroje, je podmínka, že musí být *zaregistrován v dané multicastové skupině.*

Pro názornost si zkusme představit topologii složitější sítě složené z několika segmentů, které jsou vzájemně propojeny routery. Každý router si uchovává seznam existujících multicastových skupin a pokud dorazí zpráva určená pro některou z těchto skupin, router pošle kopii této zprávy každému členovi této skupiny. Pokud některý z členů skupiny je také router, musí i tento mít seznam relevantních skupin a tudíž i on musí každou zprávu zreplikovat pro všechny zaregistrované členy dané multicastové skupiny. Dá se říci, že celá skupina tvoří strom, kde kořenem je původce zprávy, uzly jsou „distributoři“, kteří tuto zprávu předávají svým zaregistrovaným členům skupiny a listy jsou cíloví hostitelé, pro které je zpráva určena.

Zde je souhrn nejdůležitějších vlastností multicastingu:

- aby se předešlo slepému rozesílání zpráv, každý příjemce, který má zájem přijímat zprávy dané skupiny, se musí zaregistrovat,
- stejná zpráva nikdy neputuje jedním komunikačním kanálem dvakrát,
- zprávy nejsou doručovány nikomu jinému než zaregistrovaným hostitelům,
- žádný distribuční uzel sítě nepotřebuje znát identity všech členů skupiny, pouze „svých“ lokálních.

Multicasting se v dnešní době stále více prosazuje při návrhu distribuovaných systémů virtuální reality, jelikož není tak náročný na šířku pásma a je možné s využitím více skupin

[2] Jelikož je nutné z každého příchozího paketu sejmout hlavičky protokolů, které používá dané síťové rozhraní a typ sítě, není možné určit, zda jsou data pro hostitele zajímavá, dříve než po jejich zpracování operačním systémem. Rozhodující je totiž číslo portu, které je však součástí až UDP dat.

dosáhnout efektivního přenosu více typů dat. Jakmile se připojí nový účastník do systému, může jednoduchým způsobem oznámit svoji přítomnost v systému a multicasting také usnadňuje vyhledávání služeb, kdy hledající aplikace vyšle dotazovací zprávu na předem známou adresu a pokud je dostupný hostitel, který tuto službu poskytuje, odpoví tazateli na jeho zprávu.

Známá omezení tohoto protokolu jsou dána především jeho (zatím) nepříliš širokým použitím. Abychom totiž byli schopni využít služeb multicastingu, je zapotřebí, aby všechny routery na trasách mezi jednotlivými koncovými body komunikace tento protokol podporovaly. Pokud nějaký router v rámci skupiny nepodporuje multicasting, bývá tento problém řešen tak, že multicastové zprávy jsou „tunelovány“ okolními routery za tento, multicastu neschopný, router.

Jelikož Internet samotný není plně schopen multicastu, byla vytvořena tzv. Multicast Bone (zkráceně MBone) síť, která poskytuje aplikacím a členům multicastových skupin dojem plně propojeného Internetu, poskytujícího podporu multicastu (viz [BM94]). Důležitou podmínkou toho, aby byl určitý hostitel schopen připojit se ke skupině, je to, že router, přes který se tento hostitel připojuje, musí podporovat multicasting a musí být připojen k síti MBone.

Více informací o adresování paketů všesměrového a multicastového vysílání lze nalézt v příloze 1 na straně 58.

2.9 Výběr vhodného protokolu pro systém virtuální reality

Výběr protokolu pro daný systém virtuální reality hraje klíčovou roli, protože každý z výše popsaných protokolů má své přísně specifické vlastnosti, výhody a omezení, které je třeba zvážit a v souvislosti s požadovanými vlastnostmi navrhovaného systému se pak rozhodnout, který z protokolů bude sloužit jako základ pro síťovou komunikaci. Poměrně často je také vhodné využít v rámci jednoho systému více druhů protokolů v závislosti na povaze přenášených dat a požadavcích na spolehlivost, rychlost a počet příjemců. Autoři [SZ99] velmi trefně podotýkají, že otázka by neměla znít „Jaký protokol bych měl využít pro svůj systém?“, ale „Jaký protokol bych měl použít pro přenos této informace?“.

Nyní si popíšeme, jak lze využít nebo doplnit vlastnosti jednotlivých protokolů během návrhu síťové vrstvy pro distribuovaný virtuální systém.

2.9.1 Použití TCP/IP

Protokol TCP/IP nabízí možnost spolehlivého spojení mezi dvěma hostiteli. Automaticky řeší situace vzniklé ztrátou nebo poškozením paketu, tudíž programátor se nemusí o tyto věci starat. Režie spojená s nutností udržení spolehlivosti přenosu paketů (detekce vadných/ztracených paketů a jejich přeposílání) a s tím spojená možná zpoždění, aby se dodrželo správné pořadí zasílaných paketů, jsou ale dosti limitující faktory pro širší využití v rámci systému virtuální reality.

Jak vyplývá ze souhrnu výše a podrobnějšího popisu vlastností tohoto protokolu v sekci 2.6.2 na straně 7, tento protokol se dobře hodí pro síťové virtuální systémy určené pro malé počty účastníků. TCP/IP protokol není příliš vhodný pro vytváření spojení typu peer-to-peer mezi každými dvěma účastníky v tomto systému. Z tohoto důvodu bývají systémy, využívající protokol TCP/IP, postaveny na architektuře klient/server. V takovém systému je hlavním úkolem serveru příjem a další distribuce zpráv od připojených klientů. Pokud je to žádoucí (například z důvodu vysokého počtu současně připojených klientů), je možné rozšířit tuto

architekturu tak, že místo jednoho serveru bude existovat serverů několik a ty budou moci spolu komunikovat pomocí TCP/IP a rovnoměrně tak rozložit celkovou zátěž systému.

2.9.2 Použití UDP/IP

Tento protokol, podrobněji popsaný v sekci 2.6.3 na straně 8, není zatížen režií spojení tak jako TCP/IP, avšak neposkytuje takový komfort a pokud programátor potřebuje, aby síťová vrstva jeho virtuálního systému měla alespoň částečné vlastnosti protokolu TCP/IP (detekce vadných zpráv a zaručení správnosti jejich pořadí), ale nemůže použít TCP/IP protokol přímo, musí tyto vlastnosti implementovat sám.

Hlavní využití v rámci oblasti distribuovaných systémů virtuální reality nachází protokol UDP/IP při přenosech informací, u nichž spíše než na spolehlivosti záleží na rychlosti přenosu a počtu potenciálních příjemců. Spolehlivost nemusí být na prvním místě, neboť případné ztracené zprávy budou rychle nahrazeny novými, a protože mezi hostiteli neexistuje žádné spojení charakteru protokolu TCP/IP, nemusí být z hlediska efektivity nezbytně nutné použít architekturu typu klient-server, ale bude postačovat, když vysílající hostitel bude zprávy posílat ostatním účastníkům v systému přímo. Nicméně však ani tento protokol není v tomto ohledu zcela efektivní, jelikož je stále založen na komunikaci mezi dvěma koncovými body, a jeho nasazení pro velké distribuované systémy s kapacitami několik set až tisíc připojených, může být velmi problematické.

Zajištění správného pořadí s UDP/IP

K zajištění správného pořadí přijetí zpráv lze využít metody, kdy s každou zprávou je uloženo i její *jedinečné sériové číslo*. Pokud tato čísla tvoří monotónní rostoucí posloupnost (což lze zajistit zcela jednoduchým čítačem), může každý příjemce zprávy přijaté od jednoho zdroje seřadit do takové posloupnosti, v jaké byly zdrojem odeslány.

Jestliže je třeba, aby byl každý příjemce schopen seřadit do rostoucí posloupnosti zprávy od více než jednoho zdroje, princip sériových čísel v tomto případě nebude fungovat, neboť každý zdrojový hostitel své zprávy čísluje jinak (jinak řečeno – čítač každého zdroje může mít v danou chvíli jinou hodnotu). V takovém případě lze místo čítače použít tzv. *časovou značku*, která určuje přesný čas vytvoření nebo odeslání daného paketu. Příjemce je pak schopen porovnáním časových značek paketů z různých zdrojů seřadit tyto do posloupnosti odpovídající pořadí, v jakém byly ze zdrojů odeslány. Jak si ukážeme později, časové značky mohou mít v distribuovaném systému virtuální reality i další využití. Aby však byla tato metoda efektivní, je třeba zajistit správnou synchronizaci hodin na všech připojených hostitelích.

Detekce ztracených zpráv

Výše popsaná metoda, používající k jedinečnému označení paketů sériová čísla, umožňuje příjemci seřadit zprávy do pořadí, v jakém byly odeslány, avšak není příliš užitečná co se detekce ztracených zpráv týče. Je obvyklé, že zdroj neposílá všechny odchozí pakety všem potenciálním příjemcům, a může se tedy stát, že určitý příjemce vidí „mezery“ v číslech paketů. Tento problém lze řešit kupříkladu tak, že odesílající hostitel si bude uchovávat čítač odeslaných paketů pro každého příjemce zvlášť. Tím bude dosaženo konzistence v pořadí čísel paketů pro každého příjemce, což je základ pro efektivní detekci ztracených zpráv.

Abychom byli schopni detekovat ztracené zprávy, musíme si být jisti, že pokud zpráva na místo určení opravdu dorazí, příjemce nám pošle potvrzení o jejím přijetí. Potvrzování lze realizovat dvojím způsobem:

1. *Pozitivním potvrzováním* – V tomto případě pošle příjemce zdroji potvrzení, kdykoli přijme zprávu. V tomto potvrzení se obvykle nachází identifikační číslo potvrzované zprávy. Je také možno „akumulovat“ potvrzení několika zpráv do jedné, tj. v rámci jednoho potvrzení potvrdit několik přijatých zpráv. Aby bylo možné detekovat ztracenou zprávu, zdroj po odeslání každého paketu nastaví časovač a pokud během určitého intervalu neobdrží patřičné potvrzení, usoudí, že zpráva do cíle nedorazila, a odešle ji znovu, přičemž znovu nastaví tento časovač.

2. *Negativním potvrzováním* – Tato metoda je opakem výše popsané metody. Zde neměří intervaly mezi jednotlivými přijatými zprávami zdroj, ale příjemce. Ten měří dobu od poslední přijaté zprávy a pokud její délka překročí určitou mez, vyšle zdroji zprávu o tom, že již dlouho neobdržel žádnou novou informaci. Jakmile je přijata nová zpráva, příjemce časovač resetuje. Tato metoda je efektivnější, pokud příjemce ví, od koho a jak často může očekávat novou zprávu. Je proto nutné zajistit, aby každý hostitel zasílal nové zprávy s určitou frekvencí, nebo – pokud toto nelze zaručit – aby alespoň jednou za určitou dobu poslal „ping“ zprávu a tím zaručil, že časovač na straně příjemce nevytiká.

Mimo výše popsaných schopností, jinak dostupných pouze v protokolu TCP/IP, je možno také – pokud je to nutné – implementovat podporu pro „zpomalení zdroje“, tzn. že pokud příjemcova fronta přijatých zpráv přesáhne určitou mez, může vyslat speciální zprávu, která způsobí, že zdroj sníží frekvenci, s jakou odesílá nové zprávy, a tím umožnit příjemcovi či příjemcům snažší zpracování těchto zpráv.

Je třeba si uvědomit, že všechna tato vylepšení s sebou přinášejí další a další režii, které jsme na počátku chtěli vyhnout tím, že jsme místo protokolu TCP/IP zvolili UDP/IP protokol. Jestliže by se ukázalo, že je zapotřebí do protokolu založeném na UDP/IP implementovat více schopností, jinak dostupných pouze v TCP/IP protokolu, rozhodně by bylo na místě zvážit, zda se nakonec nevyplatí přechod k TCP/IP, jelikož tam jsou tyto schopnosti již implementovány.

2.9.3 Použití všesměrového vysílání

Síťový subsystém distribuovaného systému virtuální reality, který používá všesměrové vysílání (detailně popsané v sekci 2.7 na straně 8), má v podstatě stejné vlastnosti jako systém používající prostý UDP/IP protokol, avšak oproti jemu má tu výhodu, že každý paket je doručen všem hostitelům bez rozdílu a tudíž není zapotřebí uchovávat čítače sériových čísel paketů pro každého hostitele zvlášť. Jelikož ale nelze použít broadcasting mimo lokální síť, lze takový distribuovaný virtuální systém používat jen v rámci jedné domény. Kromě toho – jak již bylo řečeno výše – jsou zprávy, doručované všesměrovým vysíláním, zasílány všem lokálním hostitelům, nezávisle na tom, zda mají o tyto zprávy zájem či nikoli. To je také důvod, proč nelze rozumně provozovat takový distribuovaný systém na síti s velkým množstvím připojených hostitelů.

Jestliže navrhujeme systém, jehož síťové jádro bude pracovat na bázi všesměrového vysílání, musíme počítat s možností, že účastníci v našem systému mohou (a velmi pravděpodobně budou) přijímat zprávy, které buď vůbec nemusejí souviset s naším systémem, nebo pocházejí z jiné, souběžně běžící instance našeho systému. Je proto nezbytně nutné zajistit, že naše aplikace bude schopna takové zprávy zjistit a nebudeme tak chybně interpretovat přijatou zprávu. Způsobů, jak toho dosáhnout, je několik. My si popíšeme alespoň tyto tři:

- *Celková eliminace problému* – Pokud bude náš systém provozován na spravované síti, je možno přiřadit každé běžící instanci systému a ostatních aplikací, používajících broadcasting, vlastní číslo portu a tím vyloučit možnost konfliktu (ačkoli nikdy není možné vyloučit pravděpodobnost chyby při konfiguraci).

- *Použití speciálních tokenů instancí* – Při startu našeho systému je možno vytvořit speciální token, který bude jednoznačně identifikovat danou běžící instanci systému. Tento token bude posléze součástí každého paketu a příjemci mohou jednoduchým testem tohoto tokenu ověřit, že přijatý paket je skutečně platný. Tato metoda je založena na předpokladu, že různé instance systému nezvolí stejné náhodné číslo a pakety, pocházející od ostatních aplikací, nebudou obsahovat toto číslo na stejném místě, jako pakety našeho systému.

- *Šifrování* – Podobně jako v předchozím případě je i zde vytvořen náhodný token, avšak ten je zde používán jako klíč k šifrování (na straně zdroje) a dešifrování (na straně příjemce) obsahu každé zprávy. Jakmile příjemce po přijetí zprávy zjistí, že její obsah nelze s použitím známého klíče převést na zprávu s platnou strukturou, může s jistotou říci, že se nejedná o zprávu určenou pro tento systém.

2.9.4 Použití IP multicastingu

Použitím IP multicastingu (který jsme si detailně popsali v sekci 2.8 na straně 9 lze dosáhnout největší efektivity přenosů zpráv, co se počtu příjemců a vytížení sítě týče. Pokud je ale žádoucí rozdělit všechny informace mezi několik multicastových skupin, je to z návrhářského hlediska velmi obtížné rozhodování. Lze samozřejmě mít pouze jednu skupinu, ale v převážném množství případů se používá několik multicastových skupin, aby bylo možno docílit určitého dělení a filtrování přenášených zpráv. Příjemci se tak mohou zaregistrovat pouze do těch skupin, o které mají zájem.

3 Síťové architektury distribuovaných systémů

V této části se zaměříme na jednotlivé používané síťové architektury distribuovaných systémů a popíšeme si jejich strukturu, princip a výhody a omezení, které je nutno brát v potaz, pokud takový systém navrhujeme.
Typy spojení rozlišujeme dva pro všechny typy architektur:

- *fyzické* – tento typ představuje vodiče propojující jednotlivé komunikující stanice; tento typ spojení je nutno uvažovat proto, že jeho kvalita (propustnost, šířka pásma) omezuje naše možnosti při návrhu systému,
- *logické* – tento typ reprezentuje směry toku dat mezi účastníky v systému a z těchto dvou typů je uvažován na prvním místě (při návrhu se řídíme především tím, jaké schopnosti od systému požadujeme, a teprve potom musíme najít kompromis mezi těmito požadavky a možnostmi, které nám nabízí dostupné síťové spojení).

3.0.5 Fyzické spojení

Pro metriku připojení lze využít protokol, nazvaný Distributed Interactive Simulation (zkráceně DIS) a popsaný v [IEEE95]. Úkolem tohoto protokolu je číselně popsat množství informací, které lze za jednotku času poslat spojením o určité šířce pásma. Tento protokol používá pro vyjádření základní jednotky dat tzv. *programovou datovou jednotku* (angl. *Program Data Unit, PDU*). Velikost této jednotky činí 144 bajtů a obsahuje informace k tomu, aby (dynamický) stav entit byl udržován v aktuálním stavu pro všechny účastníky v systému. Současně s tím je zapotřebí, aby daná jednotka obsahovala informace o svém původci, tj. jeho pozici a orientaci ve světě, aktuální rychlost a zrychlení jeho pohybu, informace potřebné k přesnému vyjádření stavu původce[3] a případně další informace, které je zapotřebí sdělit ostatním.

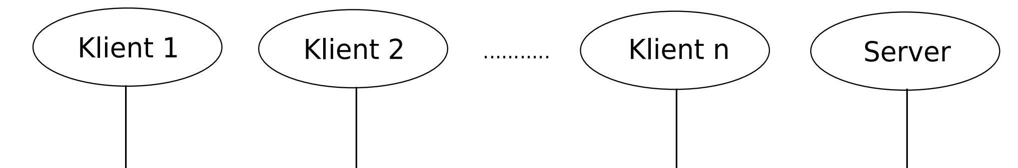
Abychom byli schopni vypočítat, jak velké množství dostupného pásma spojení budeme potřebovat (či co si můžeme dovolit, než šířku pásma vyčerpáme), musíme mít představu, jakou snímkovou frekvenci chceme udržovat v rámci celého systému a kolik PDU za jednotku času (obvykle za vteřinu) budou jednotlivé objekty ve virtuálním světě generovat. Jakmile jsme schopni zodpovědět si tyto otázky, jsme schopni také předpokládat určité počty jednotlivých typů objektů ve světě a s tím i celkové počty generovaných datových jednotek a spotřebovanou šířku pásma použitého spojení.

Současně s tím však musíme vzít v potaz, na jakých typech linek chceme, aby náš systém byl schopen pracovat. Pokud stavíme komplexní systém, v němž bude najednou existovat větší počet dynamických objektů různých typů, a nechceme, aby systém pracoval s „rozumným“ výkonem pouze po lokální síti, kde se šířka pásma počítá v megabitech za vteřinu a kde není až tak silně limitujícím faktorem, bude zřejmě zapotřebí učinit určité úpravy protokolu (komprese paketů nebo přenos menšího množství dat), abychom byli schopni přenést za jednotku času větší množství paketů, nebo přijmout omezení maximálního počtu objektů ve světě, abychom byli schopni zaručit rozumný výkon i na slabších linkách.

[SZ99] používá pro názornost příklad měřítek stanovených pro systém NPSNET-IV, navržený v [MAC94], v němž existuje několik různých typů objektů (letadla, vozidla a plně popsané lidské bytosti), z nichž každý má definovaný počet PDU, které je schopen za jednotku času vytvořit. S touto informací je pak dostupná šířka pásma spojení vyjádřena maximálními počty

[3] Tím jsou myšleny informace, které jsou potřebné k tomu, abychom věděli například přesné pozice částí těla avatara daného hráče apod.

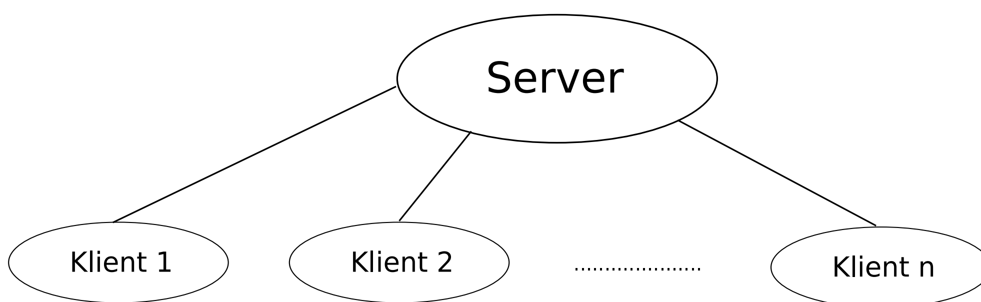
jednotlivých druhů definovaných objektů v rámci dvou typů spojení, tvořících hranice celkového rozsahu šířky pásma dnes používaných spojení – lokální sítě s propustností 10 Mbit/s a vytáčená připojení přes 56 kbit/s modemy. Zároveň s tím je předpokládáno, že veškerá dostupná šířka pásma bude využita pouze navrhovaným systémem, a jsou zanedbány i některé vlastnosti daných připojení či sítí (například že LAN se zahltí při 70% využití).



Obrázek 1: Schéma fyzického spojení klientů a serveru

3.0.6 Logické spojení

Co se týče logického spojení, v tomto případě mezi sebou komunikují dva účastníci. Komunikace může probíhat jak přes protokol TCP/IP, který nám zajistí spolehlivost při přenosu, tak i přes UDP/IP protokol, kde žádné záruky nemáme, avšak data mohou být přenesena a zpracována rychleji. Data lze posílat buď přímo k cílovému hostiteli, nebo všesměrovým vysíláním poslat paket všem hostitelům. Volba způsobu doručení dat obvykle závisí na tom, s jakou spolehlivostí a jakému počtu účastníků je třeba je doručit. Pokud se jedná pouze o dva hráče, není to příliš důležité, avšak pokud by hráčů bylo mnoho, protokol UDP/IP, případně v kombinaci s všesměrovým vysíláním, by byl velmi výhodnou volbou.



Obrázek 2: Schéma logického spojení klientů a serveru

3.1 Architektura klient-server

V systémech, používajících tuto síťovou architekturu, komunikují jednotliví účastníci mezi sebou prostřednictvím serveru. Třebaže použití serveru poněkud zpomaluje celkový přenos zpráv mezi jednotlivými klienty v systému, výhody jsou poměrně zásadní jak pro efektivitu z hlediska využití sítě, tak i – jak si později ukážeme – pro zachování konzistentního stavu v systému.

Jednou z výhod serveru a způsobů, jak odlehčit síti, je zamezení odesílání zbytečných zpráv řešením viditelnosti pro každého připojeného klienta. Pokud server přijme zprávu, popisující určitou událost, může na základě testu pozice a orientace[4] každého připojeného klienta ve světě zjistit, zda je pro daného klienta tato zpráva zajímavá. Pokud není, není ani tomuto klientovi odeslána.

Další výhodou je možnost na straně serveru sdružovat více přijatých zpráv do menšího počtu odchozích zpráv. Jestliže například server přijme několik zpráv rychle za sebou, přičemž jejich vyhodnocením zjistí, že zajímavá je pro ostatní klienty jen poslední z nich, může klientům odeslat pouze poslední přijatou zprávu, čímž je dosaženo rovnoměrnějšího vysílání zpráv ze strany serveru. V případě, že je žádoucí, aby přenosy dat byly spolehlivé (tj. aby existovala detekce ztracených a poškozených zpráv), lze se použitím architektury klient-server vyhnout režii, spojené s případem, kdy všichni komunikují se všemi.

Aby se zvýšil maximální dovolený počet hráčů v systému, používá se v praxi často metoda méně častých aktualizací stavů jednotlivých objektů ve scéně za cenu větší míry chyb, které mohou hráči zpozorovat během interakce v systému. Také princip sdružování zpráv je v tomto případě výhodný.

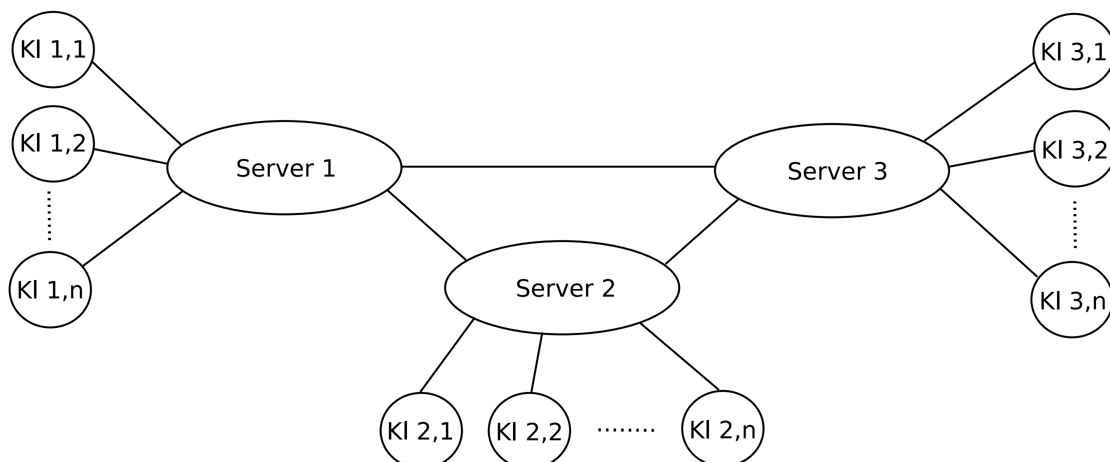
3.2 Architektura klient-server s více servery

V případě architektury s více než jedním serverem jsou servery navzájem propojené a každý obsluhuje určitý počet klientů. Tato architektura je výhodná v případě, že systém virtuální reality pojme takový počet účastníků, který by jediný server nebyl schopen obsloužit jak z hlediska propustnosti linky, kterou je k síti připojen, tak i rychlostí, se kterou je schopen požadavky od klientů zpracovávat. Propojení mezi servery je důležité, neboť lze uvažovat příklad, kdy se setkávají či protínají oblasti zájmu dvou nebo více klientů, přičemž tito klienti jsou připojeni k různým serverům.

Aby byl systém, složený z několika serverů, schopen pracovat dostatečně rychle, je třeba zaručit, že data, putující mezi jednotlivými servery, stráví na této cestě minimum času. (V opačném případě by mohlo docházet k příliš velkým zpožděním, což by silně ovlivňovalo celkový výsledný vjem.) Je proto velmi žádoucí, aby spoje mezi servery a veškerý podpůrný hardware byly co možná nejrychlejší. Na těchto spojích je silně závislá škálovatelnost celého systému pro virtuální realitu, avšak v tomto případě hraje dosti významnou roli také rychlost centrálních procesorových jednotek používaných serverů.

Pro spojení mezi klientem a serverem postačuje k dosažení uspokojivých výsledků jakýkoli typ linky, od modemu až po lokální síť. U klienta je výpočetní síla důležitá zejména pro renderování scény, kterou ze své pozice hráč či uživatel vidí.

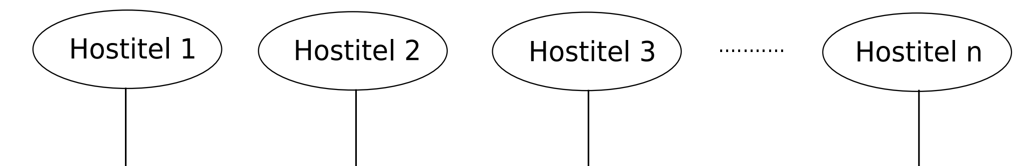
[4] Lze však samozřejmě zohlednit i jiné atributy.



Obrázek 3: Schéma architektury server-klient s více servery

3.3 Architektury typu peer-to-peer

Systémy, postavené na architektuře peer-to-peer, nepoužívají pro svůj provoz žádné servery a veškerá komunikace probíhá přímo mezi připojenými hostiteli. Motivací k použití této architektury je fakt, že i se seberychlejšími servery jsme poměrně silně limitováni v maximálních možných počtech současně připojených klientů. Maximální dovolené počty současně připojených by byly i s těmi nejrychlejšími servery výrazně nižší než kdybychom použili systém, postavený na principech peer-to-peer architektur. Škálovatelnost peer-to-peer systémů je závislá pouze na síle dostupných výpočetních a síťových zdrojů.



Obrázek 4: Spojení peer-to-peer na lokální síti

3.3.1 Virtuální systémy pro lokální síť

V návrhu systému, který zakládáme na peer-to-peer architektuře, musíme zohlednit především maximální zeměpisnou rozlohu tohoto systému. Pokud plánujeme provozovat tento systém pouze na místní síti, je naše situace nejjednodušší a můžeme pro komunikace mezi připojenými s výhodou využít buď všesměrového vysílání nebo – pokud chceme být efektivnější z hlediska zatížení sítě a nechceme slepě vysílat pakety na všechny strany – multicastingu.

Pokud jdeme cestou multicastingu, budeme potřebovat program, jenž bude roztřiďovat zprávy do patřičných multicastových skupin. V případě odchozích zpráv bude paketům přiřazovat adresy dané typem zprávy a cílovou skupinou, u příchozích zpráv pak musí na místním

hostiteli zajistit aktualizaci té části systému, které se tato zpráva týká. Mimo tyto činnosti bude také úkolem tohoto programu sledovat, které skupiny jsou dostupné (k tomu se dá využít separátní komunikační kanál). Tento typ programů se souhrnně nazývá *software pro řízení oblasti zájmu* (angl. *Area Of Interest Management, AOIM*).

3.3.2 Virtuální systémy pro rozsáhlé sítě

Jestliže navrhujeme systém, o němž předpokládáme, že bude provozován i na rozlehlých sítích, nemůžeme pro komunikaci mezi hostiteli použít všesměrové vysílání, neboť zprávy takto vysílané jsou v převážné většině případů routery blokovány. Rovněž se nabízí možnost použít multicasting, neboť ten lze na WAN sítích používat, avšak je třeba mít jistotu, že všechny routery na linkách mezi jednotlivými hostiteli multicasting podporují, což ale nemusí být splněno. Abychom docílili přenosu i po takovéto síti, je nutné multicastové pakety „zabalit“ do běžného UDP/IP paketu, jenž je poté směrován přes tu část sítě, která multicasting nepodporuje, až do místa, kde je multicasting opět dostupný. V tomto uzlu sítě je paket opět rozbalen a rozeslán všem členům patřičné multicastové skupiny na daném lokálním segmentu sítě.

4 Udržení konzistence stavu v dynamickém systému

V minulé části jsme si pověděli o základních architekturách distribuovaných systémů a nyní nastal čas říci si o problematice, týkající se udržování konzistentního stavu mezi jednotlivými účastníky v systému.

4.1 Definice problému udržení konzistence v dynamickém virtuálním světě

Hlavním cílem systému virtuální reality je poskytnout všem připojeným účastníkům iluzi světa, ve kterém se všichni mohou navzájem ovlivňovat, mezi sebou komunikovat a také manipulovat s objekty, přítomnými v tomto světě. Aby tato iluze byla na co nejvyšší úrovni, je zapotřebí zajistit, aby všichni účastníci viděli jeden a tentýž svět. Tím je myšleno především to, že vzájemnou interakcí připojených uživatelů se tento virtuální svět neustále mění a stojíme tedy před problémem, jak co nejpřesněji a nejrychleji poskytovat informace o změnách v tomto světě všem účastníkům, kterých se daná změna týká. Pokud by se nám nedařilo v dostatečné míře splnit tento cíl, každý uživatel by si připadal izolován od ostatních a iluze sdílení našeho virtuálního světa by tím utrpěla.

Množství dynamicky se měnících informací, které je nutno sdílet mezi všemi účastníky, je zcela závislé na složitosti daného systému. V případě jednodušších systémů se může jednat jen o pozice a chování jednotlivých entit uživatelů ve virtuálním světě, avšak pokud chceme uživatelům poskytnout větší možnosti činností v našem virtuálním světě, vzrůstá společně s našimi požadavky i množství a složitost informací, které bude nutné přenášet k jednotlivým připojeným účastníkům, abychom jim poskytli přesný obraz dění ve světě. Tím vzrůstají i požadavky našeho systému na propustnost a rychlost sítě, na které budeme chtít systém provozovat. Každý návrhář distribuovaného systému virtuální reality musí učinit určitá rozhodnutí a najít rovnováhu mezi omezeními danými vlastnostmi sítě a dostupnými výpočetními zdroji na jedné straně a požadovanou úrovní realismu, poskytovaného systémem, na straně druhé.

Autoři [SZ99] v této souvislosti formulují pravidlo vztahu mezi propustností sítě a konzistencí v systému (angl. *Consistency-Throughput Tradeoff*) takto: „Je nemožné, aby se sdílené informace o dynamickém stavu systému často měnily a všichni hostitelé současně měli přístup k těmto informacím.“ To znamená, že systém může být buď dynamický, přičemž se může stát, že ne všichni připojení uživatelé budou mít zcela aktuální informace o současném stavu v systému, nebo může být konzistentní, tj. všichni uživatelé budou znát přesný stav v systému, avšak za cenu nižší četnosti změn. Není možné vytvořit systém, který by byl dynamický a zároveň zaručoval absolutní konzistenci mezi stavy, udržovanými jednotlivými zúčastněnými.

Pokud bychom chtěli dosáhnout bezpodmínečné konzistence, museli bychom především splnit podmínku, že všichni připojení uživatelé obdrží informaci o změně v systému a pošlou potvrzení o přijetí, a teprve poté bychom mohli vyslat zprávu o další změně. Tato prodleva by navíc zahrnovala i latenci danou sítí a v případě ztracených zpráv i přeposlání takových zpráv. Z toho vyplývá, že celý systém by mohl vytvářet zprávy o změnách jen s velmi nízkou frekvencí, neboť s každou další zprávou o změně stavu by musel čekat do té doby, než by obdržel potvrzení o přijetí poslední odeslané zprávy.

Z výše řečeného vyplývá, že pokud navrhujeme systém, jenž bude mít charakteristiku dynamického systému (tj. bude často docházet ke změnám stavu entit v něm obsažených), bude nutné dostatečně často vysílat zprávy o nastalých změnách. Abychom zajistili konzistenci

mezi stavy, sdílenými jednotlivými připojenými hostiteli, je třeba, aby tito hostitelé potvrzovali přijaté zprávy a – pokud to bude nutné – aby případně přeposílali ztracené či poškozené zprávy. Protože však zachování absolutní konzistence stavů v systému vede k systému, jenž není schopen častých aktualizací stavů, je nutné tolerovat určitou míru nekonzistence mezi stavy, známými jednotlivým připojeným hostitelům. Každý hostitel si na základě dosud přijatých zpráv vytvoří svůj vlastní obraz virtuálního světa a je proto pravděpodobné, že mezi hostiteli budou v tomto obraze existovat odchylky. Protože však tyto odchylky mají negativní vliv na celkový vjem, který virtuální systém poskytuje, je žádoucí tento vliv minimalizovat.

4.2 Souhrn metod a metoda centralizovaného repozitáře

Existují tři skupiny metod, sloužících k udržování konzistence mezi stavy připojených hostitelů. Dvě z nich představují opačné konce spektra, definovaného vztahem propustnosti sítě a konzistence systému, třetí je kompromisem mezi nimi. Jedná se o tyto skupiny:

- *Centrální informační repozitáře* umožňují dosažení maximální konzistence mezi stavy jednotlivých připojených uživatelů na úkor dynamiky systému.
- *Predikce na základě posledních známých informací* nabízí aproximaci současného stavu systému za cenu nízké konzistence.
- *Časté aktualizace stavových informací* spočívají v častém vysílání aktuálních informací o stavu v systému u všech připojených uživatelů.

4.2.1 Popis a vlastnosti centralizovaných informačních repozitářů

Metoda centralizovaných repozitářů umožňuje návrhářům distribuovaných systémů pro virtuální realitu dosáhnout stavu, kdy všichni připojení hostitelé vidí svět stejně, neboť je zaručena celková konzistence stavu systému. Repozitář informací je chráněn zámky, aby zápisy do něj probíhaly v určitém pořadí a aby připojení hostitelé viděli tyto změny v tom samém pořadí. Každý hostitel čte informace o současném stavu systému z tohoto repozitáře, avšak pokud hostitel změní stav v repozitáři, musí zajistit aktualizaci stavu ve všech ostatních hostitelích, připojených do systému. Jelikož žádný z připojených hostitelů si neukládá data lokálně, ale vždy je čte z repozitáře, vidí všichni hostitelé zcela aktuální stav systému.

Autoři [SZ99] používají pro demonstraci funkce systému centralizovaného repozitáře souborový server, kde má každý připojený uživatel uložen soubor s informacemi o sobě, a všichni uživatelé navzájem interagují tím, že mění informace o objektech v systému, jež jsou taktéž uloženy v diskových souborech na tomto serveru. Pokud chce některý z uživatelů změnit stav systému změnou informací některého z existujících objektů ve světě, učiní tak prostřednictvím otevření patřičného souboru, zápisu nových dat o objektu a opětovným zavřením souboru. Souběhy požadavků od uživatelů, kdy dva nebo více uživatelů chce změnit data jednoho objektu v systému, jsou řešeny zámky, implementované síťovými souborovými systémy.

Nevýhodou takového systému je především jeho nízký výkon. Vše se odvíjí od nutnosti otevření souboru, zápisu dat a zavření souboru, neboť všechny tyto operace jsou velmi pomalé. To se projevuje především na rychlosti, s jakou jsou připojení hostitelé schopni renderovat snímky světa a s jakou jsou uživatelé schopni se po světě pohybovat, neboť každý pohyb znamená provést cyklus otevření souboru, zápisu a zavření souboru. Tak, jak přibývá uživatelů, se také zvyšuje režie souborového systému, jelikož přibývá souborů s informacemi o uživatelích.

Následkem toho pak trvá déle, než je nalezen a otevřen správný soubor a než jsou sesbírány všechny informace, potřebné k vyrenderování dalšího snímku.

Možným urychlením funkce celého systému je použití entity serveru, simulujícího práci sdíleného souborového systému. Každý hostitel (v tomto případě klient) se může ptát serveru na informace o sdíleném stavu systému a zároveň může tyto informace měnit.

Výhodnější postup, než ten, kdy se klient ptá serveru na aktuální stav, je jeho pravý opak, tj. že server podle situace zasílá klientům informace o změnách stavu systému. Tento postup je výhodnější z toho důvodu, že některé informace, které si klient při každém aktualizačním cyklu od serveru vyžádá, se nemusely od poslední komunikace vůbec změnit a nebylo tudíž zapotřebí je získávat znovu. Klienti si v takovém případě uchovávají lokálně poslední známé informace, které jsou průběžně aktualizovány tak, jak jsou přijímány nové zprávy ze serveru. Takto lze jednak urychlit proces vykreslování nových snímků, protože není zapotřebí ptát se před každým novým snímkem serveru, a také snížit celkové zatížení sítě tím, že zbytečně nepřenášíme již známe informace.

Mezi další výhody využití serveru patří především tyto:

- Významně se zkrátí doby, potřebné k přístupu a změně informací o uživatelích a objektech ve světě, protože server si tyto informace uchovává v operační paměti a nečte je z pevného disku.
- Server je schopen vybírat mezi příchozími požadavky a není proto nezbytně nutné používat zámky pro přístup ke sdíleným datům. Server je také schopen seřadit požadavky do fronty či některé požadavky zahodit v případě, že by byly ihned přepsány jinými.
- Server může podporovat dávkové operace, kdy lze do jedné zprávy umístit více než jeden požadavek. Tím lze dosáhnout menší síťové režie a také jisté atomičnosti operací nad repozitářem.

Nevýhodou je však možnost pádu serveru a pokud nejsou stavové informace ukládány po každé operaci s centralizovaným repozitářem na pevný disk, ztratíme tak veškeré informace o světě. Pokud pro spojení mezi klienty a serverem používáme protokol TCP/IP, znamená to z hlediska sítě (v případě většího množství připojených klientů) poměrně velkou zátěž.

Výhody a jednoduchost implementace takového systém jsou však významnější než nevýhody. Z tohoto důvodu je tato metoda funkce virtuálního systému velmi populární a často používaná pro malé a středně velké systémy.

4.2.2 Popis a vlastnosti virtuálního repozitáře

Ve výše popsaných příkladech bylo dosaženo absolutní konzistence tím, že existovala centrální entita (server nebo systém souborů se zámky), obsahující aktuální informace o stavu systému. V jistém ohledu lze takovou entitu chápat jako „úzké hrdlo“, protože server se může zahltit požadavky od klientů a souborový systém může použitím zámků zpomalovat přístup většího počtu klientů k požadovaným datům. Z tohoto důvodu vznikly systémy, využívající tzv. *virtuální repozitář* (tuto metodu používá například virtuální systém Shastra, popsaný v [AB94]). U těchto systémů neexistuje žádná centrální autorita, skrze kterou by proudily všechny požadavky na aktuální data o stavu systému, přesto však stále existuje požadavek na správné řazení aktualizací stavu, aby bylo zajištěno, že tyto aktualizace obdrží všichni klienti, pro které mají tyto aktualizace smysl.

V případě virtuálních repozitářů zajišťuje konzistenci stavů protokol. Připojení hostitelé si vyměňují informace přímo mezi sebou a použitý protokol zaručuje jak doručení zpráv, tak i jejich celkové řazení, které odpovídá pořadí událostí v systému. Každý připojený hostitel získává informace ze svého lokálního „obrazu“ světa a tento obraz, který je aktualizován zprávami od ostatních hostitelů, přicházejícími v určitém pořadí, se ve výsledku chová jako obraz centralizovaného repozitáře.

Mezi výhody takového systému patří především fakt, že je eliminováno úzké hrdlo, vyplývající z podstaty metody repozitáře, umístěného na jednom hostiteli. Toto úzké hrdlo je způsobeno dvěma, navzájem souvisejícími faktory:

- *Omezeným výpočetním výkonem* hostitele s centralizovaným repozitářem. Pokud je tento hostitel příliš vytížen zpracováním požadavků od ostatních připojených uživatelů, dojde ke zpomalení celého systému.

- *Omezenou šířkou pásma sítě*, ke které je hostitel s centralizovaným repozitářem umístěn. Jestliže musí veškerý síťový provoz systému projít přes tohoto hostitele, je zřejmé, že šířka pásma sítě může být při velkém zatížení vyčerpána.

Pokud dojde k neočekávanému selhání jednoho nebo více připojených hostitelů, nemusí to díky rozdělení úloh centralizovaného repozitáře mezi větší počet hostitelů znamenat pád celého systému, neboť každý hostitel ví přesně, jak vypadá obraz světa na všech ostatních hostitelích, a je tedy vyšší šance, že systém jako celek neselže a obnova systémů po selhání proběhne v pořádku.

Příkladem systému, využívajícího metodu virtuálního repozitáře, je systém DIVE, popsaný v [CH93]. Tento systém umožňuje připojeným klientům interakci s ostatními klienty a všemi objekty. Pokud chce uživatel manipulovat s určitým objektem, jeho počítač získá sdílený zámek na tento objekt a uživatel může změnit jeho pozici. Po této změně vyšle jeho počítač informaci o ní (s využitím spolehlivého protokolu) všem ostatním připojeným klientům a uvolní se zámek na tento objekt. Virtuální repozitář je reprezentován obrazem světa, jehož kopii si každý hostitel uchovává a spravuje.

4.3 Metoda častých aktualizací stavu

Výše jsme si popsali systémy, které udržují absolutní konzistenci sdílených stavů využitím centralizovaného repozitáře. Hlavními nevýhodami systémů, založených na tomto principu, je různě velká doba, potřebná k získání aktuálních dat, a také vysoká režie použitého protokolu, neboť všechny přenosy dat musejí být spolehlivé.

Poměrně často je výhodnější spíše než na konzistenci stavů spoléhat na aktuálnost přijatých dat[5]. Pokud je to vhodné, lze nahradit systém, využívající absolutní konzistenci stavů na všech připojených hostitelích, systémem, v němž žádný hostitel nebere v potaz, jaké informace o aktuálním stavu systému jsou známy ostatním hostitelům, ale používá metodu *časté aktualizace stavu*, aby dodával ostatním připojeným nejnovější informace o celkovém stavu určité entity (či entit) v systému (nezávisle na tom, zda se tyto informace změnily či nikoliv). Ostatní připojení uživatelé tak mají aktuální přehled o současném stavu dané entity.

Všechny tyto informace jsou vysílány naslepo v určitých intervalech a příjemci neposílají žádná potvrzení o přijetí té určité zprávy. Protokol také neprovádí žádné řazení paketů.

[5]Tím je myšleno stáří dat, tj. doba od vzniku události až do chvíle obdržení informace o ní.

Všichni příjemci si poslední získané informace ukládají do své lokální paměti a používají je k vytvoření obrazu světa.

Protože všechny vyslané informace jsou vysílány nespolehlivým protokolem, nemusejí ke svému cíli dorazit, nebo mohou dorazit s velkým zpožděním, avšak díky častým aktualizacím můžeme předpokládat, že některé vyskytnuvší se nesrovnalosti, způsobené ztrátou určitého paketu, se velmi rychle napraví přijetím nových dat. Díky nižší síťové režii je možné i na síti se střední mírou ztrátovosti paketů dosáhnout častějších aktualizací než u systémů, využívajících centralizovaný repozitář.

4.3.1 Vlastnictví objektů

Princip častých aktualizací stavu s sebou přináší nutnost zajistit, že více klientů najednou se nepokusí přistoupit ke stejnému objektu. U centralizovaných repozitářů jsme měli vždy záruku, že události jsou doručeny všem a zpracovány ve stejném pořadí na všech připojených hostitelích. Zde však nic takového zaručeno nemáme. Může se stát, že dva klienti začnou manipulovat se stejným objektem ve stejnou chvíli. Oba vyšlou informaci o aktuálním stavu objektu, avšak může se snadno stát, že jeden klient přijme a zobrazí nejprve aktualizaci od prvního vysílajícího hostitele a jiný klient přijme a zobrazí jako první aktualizaci od druhého vysílajícího. Výsledkem potom je, že objekt na obrazovce „poskakuje" tak, jak jsou přijímány aktualizace od obou vysílajících klientů.

Abychom zamezili tomuto chování, musíme zavést pojem *vlastnictví objektu* a tím zaručit, že s každým objektem může v jednu chvíli manipulovat pouze jeden uživatel. Ten je také zodpovědný za vysílání informací o změnách aktuálního stavu daného objektu. Nutnou podmínkou tohoto vysílání je však vlastnictví onoho objektu. Princip vlastnictví je podobný mechanismu zámků u centralizovaného repozitáře, avšak narozdíl od něj má zde uživatel možnost provést libovolný počet změn (v případě centralizovaného úložiště byla dovolena vždy pouze jedna změna), který je omezenou pouze dobou, po kterou je onen klient vlastníkem daného objektu.

Pro získání vlastnictví nad určitým objektem musejí klienti zpravidla kontaktovat určitou autoritu, která „přiděluje práva" k vlastnictví objektů (většinou se jedná o server nebo nějakého jiného specializovaného hostitele v rámci distribuovaného virtuálního systému). Tento hostitel zajišťuje, že žádný objekt ve světě nemá nikdy více jak jednoho vlastníka. Tuto autoritu klienti kontaktují také v případě, že se chtějí vzdát vlastnictví nějakého drženého objektu. Po odejmutí práv klientovi se pak vlastníkem takového objektu stává ona autorita.

Při návrhu systému je třeba počítat také s tím, že může dojít k odpojení klienta ve chvíli, kdy je vlastníkem nějakého objektu. Abychom se vyhnuli stavu, kdy daný objekt zůstává uzamčen i přesto, že jeho vlastník již není připojen do systému, je vhodné stanovit dobu, po kterou může být objekt vlastněn jedním klientem. Použitím této techniky lze definovat politiku vlastnictví objektů, tj. jak dlouho může být ten který objekt vlastněn apod.

4.3.2 Řešení konfliktů

Pokud nastane případ, kdy dva klienti chtějí manipulovat s objektem, přičemž jeden z nich je v danou chvíli vlastníkem objektu, má druhý hostitel dvě možnosti:

1. Použít metodu *aktualizace přes prostředníka* (angl. *proxy update*)

2. Získat objekt pod svou kontrolu *předáním vlastnictví* (angl. *ownership transfer*)

V prvním případě musí klient, jenž není vlastníkem onoho objektu, poslat soukromou zprávu vlastníkovi objektu, v níž mu sdělí, jakou operaci chce s objektem provést. Vlastník, pokud žádost akceptuje, pak podle informací, uložených v žádosti, provede danou operaci a vyšle aktualizaci ostatním připojeným hostitelům. Vlastník objektu se tak stává prostředníkem pro všechny změny, provedené na objektu nezávisle na tom, od jakého hostitele tyto požadavky ve skutečnosti pocházejí.

Ve druhém případě dochází k fyzickém přesunu vlastnictví objektu z jednoho hostitele na druhého. Hostitel, přejímající vlastnictví, se tak stává zodpovědným za veškeré aktualizace stavu získaného objektu. Operace, které je zapotřebí za tímto účelem provést, jsou závislé na dané implementaci autorizačního podsystému virtuálního systému. Autoři [SZ99] popisují dva postupy. Dle prvního musí klient, žádající o vlastnictví daného objektu, poslat tuto žádost současnému vlastníkovi objektu. Pokud vlastník tuto žádost přijme, vyšle informaci o předání autoritě, spravující zámky k objektům, a poté žadateli potvrdí předání vlastnictví nad objektem. Pokud se použije druhý postup, musí žádající klient informovat autoritu a ta rozhodne o předání vlastnictví. Autorita, rozhodující o vlastnictví objektů, však vždy uchovává informace o tom, kdo je v danou chvíli vlastníkem daného objektu.

4.3.3 Zdokonalení metody

Jako možné zdokonalení této metody se nabízí snížení množství vysílaných aktualizací tím, že s každou zprávou se určí, pro které klienty je „zajímavá“. Základním principem metody je vysílání aktualizací všem bez rozdílu, což znamená, že danou zprávu musí přijmout a zpracovat i klienti, pro které není nijak užitečná, čímž dochází k plýtvání jak šířkou pásma sítě, tak i výpočetních zdrojů každého klienta. Vylepšení takového chování je proto velmi žádoucí.

Jednou ze základních metod zdokonalení je filtrace zpráv. Tím lze dosáhnout stavu, kdy zprávy s aktálními informacemi o stavu objektů ve světě jsou odesílány jen těm klientům, pro které mají význam. Je však třeba si zodpovědět otázku, kde se toto rozhodování bude dít. Jestliže budeme předpokládat, že každý klient bude filtrovat svá data, musí pak také mít dostatek informací k tomu, rozhodnout, kdo musí být informován o změnách stavu entit v systému a kdo nikoliv.

Představitelem systému, který řeší filtrování aktualizací, je systém RING [Funk95], vyvinutý v Bellových laboratořích. Tento systém používá pro filtrování zpráv server, kam jsou zasílány všechny aktualizace, a který sleduje pozice a orientace všech účastníků a s každou zprávou řeší viditelnost entity, které se zpráva týká, z hlediska pozice a orientace každého připojeného klienta. Klientům, kteří danou entitu nevidí, není aktualizace předána.

4.4 Predikce sdíleného stavu systému

Metody založené na *predikci sdíleného stavu* (angl. *dead reckoning of shared state*) patří zřejmě k nejpokročilejším z výše uvedeného seznamu metod. Postupy, popsané v předchozích sekcích, pouze převedly na obraz poslední známé informace o stavu entit v systému. V této sekci si popíšeme metody, které jdou o krok dál. Tyto metody používají méně časté přenosy aktualizací a z posledních známých informací se snaží aproximovat skutečný stav entit v systému.

Tyto techniky představují opačný konec spektra, definovaného vztahem mezi propustností sítě a konzistencí v systému, o němž jsme si řekli v sekci 4.1 na straně 19. Využití predikce nám umožňuje snížit množství zpráv potřebných k udržení aktuálního sdíleného stavu mezi připojenými hostiteli (a tím zvýšit limit počtu připojených, které je systém schopen pojmout)

za cenu snížení přesnosti sdíleného stavu, jenž je znám každému z připojených hostitelů.

4.4.1 Predikce a konvergence

Každý protokol, založený na predikci stavu, se skládá ze dvou základních a úzce spolu souvisejících částí:

- *Predikce*
- *Konvergence*

Predikce slouží k vypočtu aktuální pozice objektu z posledních známých dat. Jakmile jsou přijaty nové informace o aktuální pozici daného objektu, jsou původní data nahrazena a další výpočty odhadu jsou založeny na těchto nových datech. Jelikož však odhad, který jsme prováděli, než jsme přijali nové informace, může být (a v závislosti na kvalitě predikčního algoritmu s určitou pravděpodobností je) nepřesný, musíme přesunout objekt z predikované pozice do nové. K tomuto účelu slouží druhá komponenta protokolu – *konvergence*. Její algoritmus určuje, jakým způsobem bude během příštích několika snímků objekt přesunut z původního umístění do nového.

4.4.2 Predikce využívající derivační polynomy

Derivační polynomy bývají nejčastěji základním stavebním kamenem predikčních algoritmů. Tyto polynomy jsou tvořeny výrazem, obsahujícím různé stupně derivace pozice daného objektu[6]. Je zřejmé, že platí, že se zvětšujícím se řádem polynomu se zvyšuje i kvalita výsledků predikce, které s jeho pomocí získáváme.

Nejjednodušší variantou derivačního polynomu je *polynom nultého řádu.* Zde je použita pouze informace o současné pozici objektu a nejsou z ní odvozovány žádné další informace. Tím se nám problém de facto zúžil na pouhé aktualizace stavů, popsané v předcházející sekci.

Algoritmus, využívající pro predikci *polynom prvního řádu*, bere v potaz jak současnou pozici objektu, tak i jeho rychlost, což je informace, která nám již dává poměrně dobrou představu o chování objektu v čase a díky tomu jsme schopni dosáhnout lepších výsledků při odhadu dalšího chování objektu. Abychom však byli schopni s postupem času odhadnout novou pozici objektu, musí aktualizační paket obsahovat informace jak o pozici objektu, tak i o jeho rychlosti.

V dnešní době nejpoužívanější derivační polynom pro predikci stavu je *polynom druhého řádu.* Tento polynom používá kromě informace o pozici a rychlosti pohybu objektu i druhou derivaci, tj. zrychlení objektu (informace o něm musí být samozřejmě také obsažena v aktualizačních paketech). Popularita tohoto polynomu pramení z jeho jednoduchosti, rychlosti výpočtu a poměrně dobré kvalitě výsledků, které s ním lze dosáhnout.

4.4.3 Hybridní predikce s derivačními polynomy

Jelikož každý z výše popsaných polynomů nabízí různou kvalitu predikce a má odlišné nároky na výpočetní výkon, je občas vhodné zamyslet se, zda je v dané situaci lepší použít polynom nižšího řádu, který ačkoli nenabízí příliš kvalitní výsledky, je výpočetně méně náročný, než polynom vyššího řádu. Pokud má například určitý objekt jen minimální zrychlení, můžeme

[6] Vzpomeňte si, že derivace prvního řádu určuje rychlost objektu, derivace druhého řádu pak jeho zrychlení.

použít polynom prvního řádu a přesto získat dostačující výsledky. Lepší predikce dosáhneme s polynomem prvního řádu i v případě, že zrychlení daného objektu se mění ve velkých skocích. Tehdy je lepší zanedbat informaci o akceleraci objektu a zaměřit se na případně (časově) stabilnější rychlost objektu. O protokolu, který v závislosti na situaci střídá polynom prvního a druhého stupně, aby dosáhl přijatelných výsledků s co nejnižšími nároky na výpočetní výkon, říkáme, že používá tzv. *hybridní predikci*.

V tomto směru je velmi zajímavý protokol nazvaný Position History-Based Dead Reckoning (PHBDR), vyvinutý pro stanfordský systém PARADISE a popsaný v [SC95]. Tento protokol nejen, že dynamicky rozhoduje, zda použít polynom prvního či druhého řádu, ale také místo vyhodnocení pouze nejaktuálnější informace používá informace získané ze tří posledních aktualizací. Pokud zjistí, že během sledované doby byly změny zrychlení příliš malé nebo naopak příliš velké, použije polynom prvního řádu. Mezní hodnoty zrychlení jsou definovány zvlášť pro každou entitu a změny zrychlení mají vliv také na výběr metody konvergence.

4.4.4 Omezení derivačních polynomů

Pokud bychom chtěli dále zvyšovat stupně derivačních polynomů přidáváním dalších a dalších derivací, do určité míry bychom dosáhli lepších výsledků, avšak také si musíme uvědomit, že se zvyšujícím se řádem predikčních derivačních polynomů se dosti snižuje zisk, který z toho máme, a naopak se silně zvyšuje náročnost jak na síťovou šířku pásma, tak i na výpočetní výkon počítače.

Jelikož pro výpočty derivací vyšších řádů potřebujeme více informací, které je třeba přenést s každým aktualizačním paketem, zvyšujeme tak náročnost protokolu na propustnost sítě, což – dá se říci – je kontraproduktivní, neboť hlavní výhodou predikčních protokolů měla být právě nízká náročnost na propustnost sítě. Pokud bychom přidávali další stupně derivace, neúměrně bychom zatěžovali síť a vliv těchto derivací vyšších řádů by byl stále menší.

Zároveň s požadavky na šířku pásma by s dalšími stupni rostla také výpočetní náročnost predikce. Je nutné mít na paměti, že predikci budeme používat pro všechny entity v systému[7] při výpočtu každého obrazového snímku. Pokud by počet těchto entit byl příliš vysoký, mohli bychom vyčerpat výpočetní zdroje počítače.

Jedním z méně zřejmých důvodů, proč se vyhnout používání derivací vyšších řádů, je fakt, že derivace vyšších řádů mohou ovlivnit výsledek daleko více než derivace nižších řádů a je proto nutné mít jistotu, že hodnoty derivací vyšších řádů jsou dostatečně přesné. Nepřesnost těchto hodnot může drasticky snížit získané výsledky a celkovou přesnost predikce. Praxe však ukazuje, že odhadovat derivace vyšších řádů a zaručit tak jejich přesnost je těžké především proto, že tyto hodnoty mají tendenci měnit se mnohem více než hodnoty derivací nižších řádů (mohou zde působit různé vlivy např. okolního prostředí, míra stažení svalů člověka[8] apod.). Z tohoto důvodu je lepší vyhnout se používání derivací vyšších řádů.

4.4.5 Objektově specializovaná predikce

Zvýšení kvality výsledků můžeme dosáhnout, pokud v našich odhadech začneme zohledňovat také informace o současné aktivitě objektu, o množině aktivit, kterých je daný objekt schopen, a o tom, kdo momentálně daný objekt ovládá. Aby však byly splněny tyto požadavky, musí být pro každý typ objektu zaveden příslušný protokol, což může vést k vysoké složitosti a velikosti

[7] Přesněji řečeno ty, jejichž stavové informace se s časem mění.

[8] V případě, že tento člověk ovládá například zrychlení automobilu nebo letadla.

zdrojového kódu systému. Nicméně ale pokud je žádoucí vysoká přesnost odhadů, je vhodné tyto optimalizace zavést[9].

V případě některých systémů není zapotřebí, aby vzdálený hostitel znal přesné chování určitého objektu nebo jevu, pokud je sám schopen je dostatečně přesně určit. Pokud například hostitel obdrží v aktualizačním paketu zprávu o tom, že v určitém místě ve světě došlo k explozi, může sám tento jev nasimulovat, aniž by potřeboval znát přesný počet nebo směr odletujících částic. Autoři [SZ99] uvádějí také příklad, kdy určitý objekt začne tancovat nebo hořet. V těchto případech také není tak důležitá přesná znalost práce nohou či pohyb jednotlivých plamenů. Mnohem důležitější je samotný jev tance či hoření. Proto je v takových případech postačující, když hostitel přijme zprávu o události a poté provede simulaci této události, aniž by znal její přesné podrobnosti.

4.4.6 Konvergenční algoritmy

Jak jsme si již řekli na samém počátku sekce o predikci sdíleného stavu systému, konvergenční algoritmy slouží ke korekci odhadovaného stavu. Od kvalitního algoritmu požadujeme rychlou, avšak co nejvíce plynulou korekci. Podobně jako v případě derivací u predikčních algoritmů i zde rozlišujeme algoritmy různých řádů, přičemž i v tomto případě platí, že se zvyšujícím se řádem roste jak kvalita konvergence, tak i výpočetní náročnost algoritmu.

Nejjednodušší formou korekce je *konvergence nultého řádu*. Ta spočívá v prostém přesunutí objektu na nové místo. Ačkoli je tato metoda jednoduchá, poskytuje nejhorší kvalitu vizuální korekce, neboť uživatel vidí náhlý skok objektu z původní předpokládané pozice na nové místo. Lepší by bylo, kdyby objekt provedl plynulejší přesun na nové místo.

Lineární konvergence (tj. *konvergence prvního řádu*), byť také není dokonalá, již nabízí výrazně lepší výsledky. S jejím použitím jsme schopni dosáhnout stavu, kdy se objekt, jehož korekci právě provádíme, plynule přesouvá z původního predikovaného místa do *bodu konvergence* na nové předpokládané trase pohybu. Uživatel tak vidí objekt přesouvat se po přímce mezi starou a novou pozicí. Doba přesunu mezi starou a novou pozicí se nazývá *doba konvergence*.

Ačkoli metoda lineární konvergence již nabízí lepší vizuální výsledky při korekci, pohyb objektu stále není plynulý a jakmile je zahájena korekce, je patrné, jak objekt náhle změní směr pohybu při přechodu z původní predikované trasy na trasu konvergence a z trasy konvergence na novou předpokládanou trasu pohybu. Abychom docílili plynulejšího přechodu mezi trasami pohybu, musíme použít některou z metod prokládání křivek.

Pokud použijeme pro určení trasy konvergence *kvadratickou křivku*, dosáhneme výrazně lepších výsledků a plynulejšího přechodu objektu z původní predikované trasy na trasu konvergence. Aby byl přechod z bodu na původní trase plynulý, musíme zahájit výpočet křivky v bodě, v němž se objekt nacházel před určitou stanovenou dobou. S tímto typem křivky však nedosáhneme plynulosti přechodu na novou predikovanou trasu v bodě konvergence. Z tohoto důvodu je vhodné použít *kubickou křivku*.

Pokud chceme dosáhnout plynulého přechodu mezi trasou konvergence a novou trasou předpokládaného pohybu objektu, musíme sestrojit křivku třetího řádu, která bude spojovat oba koncové body trasy konvergence. Obdobně jako v případě kvadratické křivky je i zde nutné zahájit výpočet křivky v bodě za současnou pozicí objektu na původní předpokládané trajektorii, abychom docílili hladkého přesunu z původní trasy pohybu na trasu konvergence.

[9] V dnešní době se tyto specializace používají zejména ve vojenských simulátorech.

Narozdíl od kvadratické křivky však musíme křivku ukončit v bodě *před* cílovou pozicí na nové předpokládané trase pohybu[10]. Tím dosáhneme hladkého přechodu na novou trajektorii pohybu (byť za cenu vyšších výpočetních nároků). Pokud bychom definovali více bodů, jimiž bychom posléze proložili danou křivku trasy konvergence, dosáhli bychom ještě hladšího pohybu.

Protokol PHBDR, zmíněný v sekci 4.4.3 na straně 25, dynamicky rozhoduje mezi použitím lineární a kvadratické konvergence v závislosti na tom, co je v danou chvíli vhodnější použít. Při malé odchylce v predikci používá ke korekci lineární konvergenci, jelikož kvadratická by poskytla jen malé zlepšení vjemu a vzhledem k množství nutných operací by byla příliš drahá.

4.4.7 Využití predikce pro nepravidelné aktualizace

Protokol, využívající predikci, lze výhodně využít ke snížení množství přenášených aktualizací. Pokud víme, jakým způsobem vzdálení hostitelé provádějí výpočty předpokládaných stavů objektů, umožňuje nám to snížit frekvenci, s jakou jsou vysílány aktualizace stavů objektů.

Hlavní myšlenka této metody je jednoduchá. Jestliže jsme schopni říci, s jakou přesností ostatní hostitelé předpovídají stav entit v systému, můžeme také odhadnout, kdy se jejich předpoklady začnou příliš odlišovat od skutečného stavu a tehdy jim zašleme informaci o skutečném stavu objektu, abychom je zkorigovali. Jinými slovy – pokud se odhady stavů na vzdálených hostitelích pohybují v rámci určitých, předem stanovených mezí, není třeba, abychom zasahovali. Aktualizační paket tak vyšleme pouze tehdy, pokud jejich odhad překročí danou mez chyby.

Tento postup nabízí několik výhod. První z nich je skutečnost, že při použití dobrého predikčního algoritmu lze výrazně snížit množství aktualizací, potřebných k udržení rozumné úrovně přesnosti odhadů u všech hostitelů. Se znalostí způsobu výpočtu odhadu na vzdálených hostitelích jsme také schopni zaručit celkovou přesnost odhadů (narozdíl od ostatních typů systémů, postavených na principu pravidelných aktualizací, kde nic takového zaručeno nemáme). Jelikož aktualizace zasíláme pouze tehdy, když míra nepřesnosti překročí danou mez, máme jistotu, že žádný odhad stavu objektů v systému se nestane příliš nepřesným.

V závislosti na podmínkách, panujících v danou chvíli v systému, a zatížení sítě, na které systém provozujeme, máme také možnost dynamicky upravovat prahy chyb, a tím přímo ovlivňovat množství přenášených aktualizací. Pokud je na síti velký provoz, zvýšením prahu můžeme dočasně snížit míru vysílaných aktualizací a tím se přizpůsobit stávajícím podmínkám.

Je však třeba mít na paměti, že pokud je predikční algoritmus opravdu kvalitní a přesný, nebo pokud se stav objektu či objektů nijak výrazně nemění, může dojít k situaci, kdy zdroj, provádějící korekci ostatních hostitelů, dlouho nevyšle žádnou aktualizaci. Za takovýchto podmínek pak nově se připojivší hostitelé nezískají od zdroje aktuální informace o stavu systému a ostatní připojení nebudou schopni říci, zda to, že již dlouho neobdrželi žádnou aktualizační zprávu, je způsobeno tím, že ve svých odhadech nepřekročili míru chyby, nebo nějakým problémem na síti (či případně tím, že daný objekt již není v systému).

Aby se předešlo tomuto stavu, je vhodné, pokud již dlouho nebyla vyslána žádná zpráva, alespoň jednou za určitou dobu vyslat informační paket ostatním hostitelům. Tak budou ostatní hostitelé schopni říci, kdy je vše v pořádku (tj. obdrží jednou za určitý interval patřičný informační paket) a kdy došlo k selhání sítě či zdroje (tj. již delší dobu nebyla přijata žádná zpráva), a dle toho se zachovat.

[10] To znamená v bodě, do kterého se objekt na nové trase teprve dostane.

5 Realizace síťové části systému

5.1 Definice úkolu a základní vytyčení cílů

Nyní již přikročíme k popisu realizace hlavního úkolu této práce. Cílem práce je navrhnout a vytvořit dynamický distribuovaný virtuální systém, v němž budou uživatelé schopni společně manipulovat s objekty, které se budou v tomto systému nacházet.

Funkce tohoto demonstračního systému je pojata jako virtuální hřiště, na němž bude umístěna hromada kostek, která bude kostky dynamicky generovat a ze které si uživatelé budou moci kostky vybírat a použít je ke stavbě složitějších objektů. Uživatel svým výběrem tuto kostku uzamkne a dokud ji drží, není možné, aby ji jiný uživatel získal pod svoji kontrolu. Během držení kostky s ní bude moci libovolně manipulovat. Kostka bude uvolněna, jakmile ji daný uživatel pustí.

Navrhovaný systém by měl splňovat alespoň tyto základní požadavky:

- jednoznačné řešení kolizních stavů
- synchronizace událostí
- podpora „rozumně“ velkého počtu uživatelů
- možnost funkce na WAN sítích

5.2 Obecný popis a požadavky na realizaci systému

Systém bude realizován s využitím již existujících knihoven pro síťovou komunikaci (knihovna RakNet), management a reprezentaci trojrozměrné scény (knihovna OpenSceneGraph) a systému pro interakci s objekty ve virtuální realitě s názvem VRECKO. Jelikož všechny použité knihovny jsou napsány v jazyce C++ a vzhledem k vlastnostem tohoto jazyka (zejména jeho multiplatformitě a nezávislosti na prostředí, v němž jsou programy v něm psané spouštěny), je vhodné použít tento jazyk i pro realizaci navrhovaného systému. Multiplatformita našeho systému je taktéž jedním z cílů, kterého chceme dosáhnout. Tomu je zapotřebí podřídit jak výběr knihoven, jejichž vlastností chceme využít, tak i stylu programování, abychom nevyužívali případných rozšíření jazyka, která jsou dostupná v překladačích pouze na určité platformě. Zároveň s tím se budeme snažit využít některých možností knihovny šablon tříd STL (Standard Template Library), abychom se vyhnuli případným problémům, souvisejícím s používáním ukazatelů, a usnadnili si tak práci.

5.3 Návrh a realizace síťové vrstvy systému

Abychom mohli položit základy komunikační vrstvy našeho systému, musíme si nejprve ujasnit, na jaké *síťové architektuře* bude náš systém založen a jaký *komunikační protokol* bude používat jako základ pro vlastní komunikaci mezi hostiteli.

5.3.1 Použitelné síťové architektury

Nejdříve prodiskutujme možnosti v oblasti síťových architektur. Možnosti se nabízejí dvě:

- Architektura *peer-to-peer*

- Architektura *klient-server*[11]

Architektura *peer-to-peer* je založena na přímé komunikaci připojených hostitelů. Jako jistou výhodu této architektury lze chápat absenci entity serveru, jenž může být – zejména při vysokém zatížení – úzkým hrdlem celého systému. Abychom ale byli schopni doručovat data všem připojeným hostitelům, museli bychom pro jejich distribuci použít metodu všesměrového vysílání (broadcastingu) nebo multicastingu. Výše jsme si jako jeden ze základních požadavků, které na náš systém klademe, uvedli, že musí být schopen pracovat i na rozsáhlých sítích, tudíž je vyloučena možnost použití broadcastingu (ten lze využít pouze na místní síti). Pro použití multicastingu bychom potřebovali mít jistotu, že všechny routery na trasách připojení mezi jednotlivými cílovými hostiteli budou mít podporu multicastového přenosu. Jelikož však tuto záruku nemáme, není možné nadále uvažovat multicasting jako využitelnou možnost. S architekturou peer-to-peer by také bylo složitější řešení kolizních stavů (potřebovali bychom silný protokol, kterým bychom zaručili správnost pořadí zpracování zpráv na všech hostitelích, čehož lze dosáhnout v podstatě pouze s protokolem TCP/IP) a co se týče nároků na šířku pásma sítě, při větším počtu připojených uživatelů bychom síť zřejmě příliš zatěžovali. Z těchto důvodů pro nás bude zřejmě výhodnější použít architekturu *klient-server*.

S architekturou klient-server můžeme dosáhnout nižší zátěže sítě, jelikož každý klient nebude více či méně slepě rozesílat data, ale bude je zasílat serveru, který je bude zpracovávat a aktualizace stavů entit v systému přeposílat ostatním připojeným klientům. Server může zároveň sloužit jako arbitr, který má poslední slovo, pokud dojde k situaci, kdy se setkají dva požadavky o zamčení jedné kostky (nebo nastanou jiné kolizní stavy). Použitím serveru do systému zavádíme entitu, která může při větším počtu uživatelů snížit výkon celého systému, proto zřejmě náš systém bude moci pojmout relativně malé počty uživatelů (řádově desítky). Výhody, které však použitím serveru získáváme, jsou významné a dle mého názoru dostatečně vyvažují nevýhody.

5.3.2 Dostupné protokoly

Nyní nastal správný čas si uvést možnosti v oblasti protokolů. Nabízí se jak protokol TCP/IP, tak i protokol UDP/IP. Mezi hlavní přednosti protokolu TCP/IP patří naprostá spolehlivost přenosu dat, čímž bychom měli zajištěno, že všechny zprávy dorazí na svá místa určení v pořádku a ve stejném pořadí, v jakém byly odeslány. S protokolem TCP/IP je však také spojena vysoká režie přenosu, nutná k zaručení spolehlivosti a pořadí, a je třeba zvážit, zda je pro nás opravdu výhodná.

Protokol UDP/IP není zatížen stejnou režií jako TCP/IP protokol, avšak nemáme žádné záruky, že naše data do cíle skutečně dorazí (či případně, že dorazí ve správném pořadí). Pokud tedy budeme potřebovat tyto záruky, budeme muset použít takovou knihovnu pro síťovou komunikaci, která nám je bude schopna poskytnout.

Abychom se mohli rozhodnout, který z protokolů je pro nás nejvhodnější, musíme mít představu o tom, jakým způsobem se bude systém z hlediska síťových přenosů chovat. Shrňme si proto nyní předpoklady a požadavky, související se síťovým provozem našeho systému:

- Každý klient bude komunikovat pouze se serverem.

[11] Náš systém nebude natolik rozsáhlý, abychom jako možnost uvažovali i architekturu klient-server s více než jedním serverem.

- Každý klient si bude lokálně uchovávat svou vlastní kopii světa, uloženého na serveru, a nad touto kopií bude operovat. Informace o všech objektech v tomto světě budou klientovi zaslány při jeho přihlášení do systému.
- K zasílání aktualizačních zpráv bude docházet pouze tehdy, pokud se změní stavové informace daného objektu či světa jako takového. Protože každý klient má svou cache paměť, ve které si uchovává poslední známé informace o všech objektech ve světě, není zapotřebí přenášet znovu a znovu stavová data všech entit v systému.
- Zprávy týkající se posunu kostek uživateli budou vždy generovány v poměrně velkém množství a nebude nutné, aby byly zpracovávány všechny. Jistá ztrátovost je tedy dovolena. Na druhou stranu však bude žádoucí, aby u některých „citlivějších“ typů zpráv bylo zaručeno, že do svého cíle dorazí. Budeme proto potřebovat podporu pro přeposílání ztracených a detekci vadných zpráv.
- Velmi žádoucí je, aby zprávy byly na místo určení doručeny co nejrychleji.

Z toho, co jsme si zatím řekli o vlastnostech našeho systému, vyplývá, že svou povahou bude poměrně blízký systémům, založeným na metodě centralizovaného repozitáře s existující entitou serveru. Server bude mít rozhodující slovo a veškerá komunikace bude proudit přes něj. Kromě toho bude také přidělovat práva na vlastnictví objektů a bude hlídat, aby žádný objekt neměl více jak jednoho vlastníka.

Zároveň se dá říci, že ač by některé z našich požadavků byly uspokojeny použitím TCP/IP protokolu, neměli bychom zaručeno, že zprávy se někde nezdrží z důvodu čekání na zpracování daným hostitelem, což může mít dosti negativní vliv na výkon celého systému.

Z tohoto důvodu bude pro nás lepší založit komunikační protokol navrhovaného systému na protokolu UDP/IP s tím, že základní síťová vrstva musí být schopna v případě potřeby detekovat ztracenou zprávu a přeposlat ji. Abychom však znovu nevynalézali ono příslovečné kolo, můžeme s výhodou použít některou z existujících open-source knihoven. Knihovna RakNet, popsaná níže, je pro nás (nejen) z tohoto hlediska ideální.

5.3.3 Knihovna RakNet

Knihovna RakNet [RakNet] je multiplatformní síťová knihovna, jejímž autorem je Kevin Jenkins. Je založena na UDP/IP protokolu a navržena k využití zejména v oblasti počítačových her, kde rychlost přenosu hraje důležitější roli než záruka spolehlivého přenosu a kde záleží na každém ušetřeném bajtu šířky pásma sítě. Přestože je primárně určena ke komerčnímu použití s placenou licencí, její autor ji vydal také pod GNU/GPL licencí k nekomerčnímu účelům.

Jelikož se jedná o profesionální knihovnu, vyvinutou především pro využití v počítačových hrách, obsahuje velké množství tříd a nástrojů, užitečných při tvorbě a implementaci síťových rozhraní jak pro serverové, tak i pro klientské aplikace, podporu pro zápis datových struktur, často používaných ve hrách s trojrozměrnou grafikou (matice, vektory a kvaterniony)[12], podporu komprese a šifrování posílaných dat, vzdálené volání procedur a mnoho dalších funkcí.

[12] Je však třeba podotknout, že tyto funkce obvykle pro zápis dat používají ztrátovou kompresi na určitý počet desetinných míst, a proto nejsou příliš vhodné tam, kde na přesnosti záleží.

Ačkoli je založena na UDP/IP protokolu, podporuje tato knihovna spolehlivé přenosy různé úrovně (nespolehlivé, spolehlivé, ale mimo pořadí atd.) a v tomto ohledu je velmi významným rysem fakt, že spolehlivost, s jakou chceme zprávu doručit, lze určit při odesílání každé zprávy. Je tudíž možné použít spolehlivý přenos u těch sekvencí zpráv, kde hraje pořadí důležitou roli, a ostatním, méně důležitým zprávám lze nastavit nižší úroveň spolehlivosti, abychom systém zbytečně nebrzdili, pokud se některá z těchto zpráv ztratí.

Vytváření paketů

K vytváření paketů lze použít dva postupy (resp. typy úložiště), každý se svými výhodami a nevýhodami:

1. struktury, konverzí převedené na pole znaků (v C/C++ typ `char*`) o délce odpovídající velikosti struktury,
2. objekty *proudu bitů* (tzv. `BitStream` objekty)

Výhodou použití struktur je možnost jednoduše změnit definici struktury a vidět, jaká data opravdu posíláme. Máme také jistotu, že posílaná data jsou zapisována a čtena ve stejném pořadí. Nevýhodou je pak to, že vytvoření a změna struktury se může týkat mnoha souborů a není také možné použít funkce pro zápis dat s kompresí velikosti.

Pokud použijeme `BitStream` objekty pro vytváření paketů, stačí nám vytvořit objekt tohoto typu, zapsat do něj data ve zvoleném pořadí a objekt předat funkci, která jej zabalí do paketu a odešle příjemci. S objekty tohoto typu máme také možnost použít funkce pro zápis a čtení kompresovaných dat, což je vhodné zejména při posílání řetězců znaků. Musíme si však dát pozor, abychom data zapisovali a četli ve stejném pořadí, jinak bude docházet k problémům.

V obou případech je však třeba dát si pozor a správně dereferencovat ukazatele, protože nejsou dereferencovány automaticky a může se stát, že místo zamýšlených dat do paketu zapíšeme pouze adresu, na které se v paměti daného hostitele nacházejí.

5.3.4 Časové značkování paketů

Za obecných podmínek platí, že každý hostitel, který v rámci našeho systému odesílá a přijímá aktualizační zprávy, má jinou informaci o aktuálním čase a pokud bychom posílali informace o absolutním čase, docházelo by na každém z hostitelů k odlišné interpretaci momentu výskytu dané události. Jelikož však je časová synchronizace velmi důležitá pro správnou interpretaci událostí, které přijímáme v aktualizačních paketech, máme výhodu, že knihovna RakNet tento problém řeší, přičemž bere v úvahu jak časové rozdíly mezi hostiteli, tak i zpoždění zpráv od nich. Aplikace, využívající tuto knihovnu, může být nastavena tak, že bude posílat všem známým hostitelům v určitých intervalech PING zprávu a z odezvy na tuto zprávu určí všechny potřebné proměnné. Zprávy, nesoucí v sobě časovou značku, však tuto značku (společně s informací, že ji skutečně obsahují) musejí mít zapsánu na samotném začátku datového bloku paketu.

5.4 Popis komunikačního protokolu

Nyní, když už jsme si ujasnili, jaký základní protokol a síťovou architekturu použijeme pro náš systém, si musíme promyslet, jak bude vypadat komunikační protokol. Abychom měli

přehled, jaké typy zpráv budeme potřebovat, potřebujeme mít představu, jaké události mohou v systému nastat. Poté bude nutné si definovat, jaké informace budeme s každým typem zprávy potřebovat znát. Zde je souhrn událostí, o kterých budeme muset prostřednictvím aktualizačních paketů informovat ostatní hostitele během jejich přítomnosti v systému:

- události spojené se změnou stavu uživatele
  - přihlášení uživatele,
  - odhlášení uživatele,
  - „vytikání“ uživatele (tj. absence odezvy na kontrolní zprávu v určitém intervalu),
- události spojené se stavem kostky
  - vytvoření nové kostky,
  - zamčení kostky uživatelem,
  - odemčení kostky uživatelem,
  - posun (či případně jiná transformace) kostky jejím majitelem.

Všechny tyto události budou muset být oznámeny všem patřičným hostitelům, aby byla zachována konzistence sdíleného stavu v systému.

Z výše uvedeného seznamu již můžeme usuzovat, jaké typy zpráv budeme muset uvažovat a jaká data s nimi budeme muset posílat.

5.5 Datové struktury pro aktualizační zprávy

Nyní si podrobněji popíšeme, jak budeme přistupovat k tvorbě datových struktur pro síťové události a co do nich budeme ukládat.

Jazyk C++, který používáme k implementaci systému, je objektovým jazykem, tudíž je vhodné využít postupy objektově orientovaného programování. Při návrhu struktur pro naše zprávy pak vhodnou dekompozicí můžeme dospět k optimálnímu rozdělení problému a vyhnout se případným duplicitám kódu při implementaci.

5.5.1 Obecné informace pro zpracování a adresování zpráv

Abychom byli schopni správně zpracovat přijatou zprávu a případně po zpracování správně adresovat odpověď, musíme znát tyto informace:

- O jaký typ zprávy se jedná
- Kdo je odesílatelem a příjemcem zprávy
- Zda bude nutné na serveru tuto zprávu zreplikovat a zaslat také ostatním klientům

Toto jsou informace, které budou muset být přítomny v každé zprávě. Umístěny budou v bázové třídě pro naši hierarchii tříd zpráv. Tuto třídu nazveme `Message` a na ní založíme všechny ostatní třídy zpráv.

Typ zprávy lze reprezentovat výčtovým typem (nazvěme si jej `message_type`), který bude obsahovat výčet všech existujících typů zpráv. Tento parametr musíme definovat při konstrukci objektu každé zprávy, aby byl jasně a přesně určen kontext dat zprávy.

Odesílatele (a příjemce) zprávy lze specifikovat identifikačním číslem daného uživatele. Toto číslo, přidělované serverem, slouží k jedinečné identifikaci klienta. Server bude uchovávat seznam uživatelů včetně informací o nich a tento identifikátor bude používat k přístupu k datům každého klienta.

Některé zprávy, které server zpracuje, bude muset přeposlat také ostatním uživatelům. K vyjádření, zda je zpráva určena pro všechny uživatele nebo jen původce žádosti, nám bude postačovat booleovský příznak. Hodnotu tohoto příznaku bude každá třída určovat z množiny možných hodnot proměnné typu zprávy `message_type`. Při zpracovávání zprávy na straně serveru pak můžeme před odesláním snadno zjistit, zda tato zpráva poputuje ke všem klientům či jen původci žádosti.

Co se týče implementovaných metod, mimo metod pro přístup k členským proměnným, jejichž sada je specifická pro každou třídu zprávy, stojí za zmínku následující metody. Všechny jsou označeny jako virtuální, aby mohla být v dceřinných třídách upravena jejich implementace:

- Metody `packSelf()/unpackSelf()` – tyto metody mají jako parametr referenci na objekt třídy `BitStream` knihovny RakNet. Jejich úkolem je serializace a deserializace objektů zpráv do a z poskytnutého objektu třídy `BitStream`. Implementace těchto metod provádí na každé úrovni hierarchie tříd zápis a čtení dat, které jsou v dané třídě definovány. Voláním rodičovské implementace těchto metod před zápisem dat, vlastních odvozené třídě, docílíme jistoty, že data jsou vždy zapsána a čtena v takovém pořadí, v jakém mají být.

- Metoda `clone()` – účelem této metody je vytvořit hlubokou kopii (angl. *deep copy*) objektu, nad kterým je tato metoda volána. Tato metoda vznikla proto, že fronty zpráv, které jsou detailněji popsané v sekci 6.1 na straně 39, jsou implementovány s použitím STL šablony pro datovou strukturu fronta a voláním metod pro vyjmutí objektu z této fronty dojde ke smazání vyjímaného objektu, což nám brání v jeho dalším zpracování. Pokud si před vyjmutím tento objekt naklonujeme, můžeme jej dále bezpečně používat a přitom zajistit, že již nezabírá místo ve frontě.

Za poznamenání stojí, že některé typy zpráv obsahují totožná data. Je proto možné využít již existující třídu a dát jí nový význam tím, že při vytváření objektu dané zprávy určíme její specifický typ některou z hodnot typu `message_type`.

5.5.2 Zpráva požadavku přihlášení do systému

První zpráva, odvozená od třídy `Message`, kterou si popíšeme, je zároveň první zpráva, kterou klient zašle při svém spouštění serveru. Třída, která bude představovat tuto žádost, se jmenuje `LoginRequestMessage`. Abychom zamezili možnosti, že klient a server budou ke komunikaci používat rozdílné verze protokolu, bude tato zpráva obsahovat proměnnou typu typu `unsigned short`, obsahující číslo verze protokolu, kterou klient používá. Porovnáním tohoto čísla s číslem své verze protokolu rozhodne server, zda tuto žádost povolí či zamítne. Zároveň s informací o verzi protokolu bude zpráva obsahovat řetězec, obsahující přezdívku, pod kterou bude uživatel znám ostatním.

5.5.3 Zpráva potvrzující přijetí žádosti o připojení

Tato zpráva představuje kladnou odpověď na klientův požadavek o připojení. Třída, kterou pro tento účel použijeme, se nazývá `LoginAcceptMessage`. Jelikož jediným úkolem této zprávy je předat nově připojenému klientovi jeho identifikační číslo, kterým bude posléze označovat všechny své zprávy pro server (a server zprávy pro tohoto klienta), bude jediná členská proměnná, specifická pro tuto třídu, celočíselná hodnota obsahující klientův identifikátor.

5.5.4 Zpráva zamítající žádost o spojení

Druhou možností, jak může server reagovat na klientovu žádost, je její odmítnutí, pokud je verze klientova komunikačního protokolu jiná, než ta, kterou používá server. Pokud dojde k této situaci, je nutné sdělit klientovi, z jakého důvodu byla jeho žádost zamítnuta. Proto je třeba, aby tato zpráva obsahovala řetězec, jenž bude reprezentovat popis tohoto důvodu.

Třída, reprezentující tento typ zprávy, se nazývá `KickMessage`. Jelikož je možné, že v budoucnu budeme chtít uživatele „vyhodit“ ze systému i z jiného důvodu, než jen kvůli nevyhovující verzi protokolu, je lepší pojmenovat ji obecněji.

5.5.5 Zpráva o odhlášení/„vytikání“ uživatele

Pokud uživatel vypíná program klienta, klient automaticky před ukončením a zrušením objektu síťového rozhraní zašle serveru odhlašovací zprávu, v níž uvede důvod ukončení. Ten bude opět reprezentován řetězcem znaků.

Třída, jejíž objekty budou utvářet zprávy tohoto typu, ponese název `ClientConnectionStatusChangeMessage`, jelikož bude využita pro všechny zprávy, obecně se týkající změny stavu klienta.

5.5.6 Zpráva o ukončení inicializace

Protože inicializační proces nově připojeného klienta se sestává z několika kroků, je třeba klientovi říci, kdy je inicializace dokončena a klient bude moci přejít do interaktivního stavu[13]. K tomu účelu poslouží objekt třídy `InitCompleteMessage`. Jelikož při oznámení ukončení inicializace není třeba klientovi sdělovat žádné další informace, nemá tato třída žádné specifické členské proměnné.

5.5.7 Zpráva o zamčení a odemčení kostky

Třída, představující tyto zprávy o těchto událostech, bude obsahovat celočíselnou členskou proměnnou, která bude obsahovat identifikátor kostky, které se událost týká. Název třídy je `BrickLockUnlockMessage`, neboť tato třída bude použita pro oznámení jak události zamčení, tak i události odemčení kostky. Tato zpráva bude serverem zreplikována a přeposlána také ostatním klientům.

[13] Ačkoli by bylo možné zahrnout příznak konce inicializace do některé z ostatních zpráv, museli bychom při rozšiřování procesu inicializace o další zprávy tuto zprávu upravit. Z tohoto důvodu je použití samostatné zprávy výhodnější.

5.5.8 Zpráva o vytvoření kostky

Tato zpráva, signalizující vznik kostky v hromadě na serveru, bude tvořena objekty třídy `BrickCreatedMessage`. Protože s touto zprávou budeme muset poskytnout kompletní informace o nově vzniklé kostce, bude muset tato zpráva mimo identifikátor kostky obsahovat také kompletní informace o pozici kostky (vektor), natočení kostky (matici) a měřítku (vektor). Systémy OpenSceneGraph a VRECKO (o nichž bude řeč později) používají pro reprezentaci souřadnic objektů typy `float` a `double`, tudíž i data v této zprávě budou mít tento typ.

Je třeba podotknout, že výše uvedený popis se vztahuje na data, jež budou uložena do objektu `BitStream` třídy. Samotný objekt třídy `BrickCreatedMessage` bude uchovávat vektor ukazatelů na objekty kostek, pomocí nichž budou při serializaci objektu do paketu získána patřičná data. Velikost tohoto pole bude omezena veřejně přístupnou statickou konstantou, definovanou v též třídě. Záměrem je, aby výsledný paket nebyl příliš veliký.

Jelikož se jedná o důležitou událost, týkající se celkového stavu světa, je tato zpráva rozesílána všem připojeným klientům.

5.5.9 Zpráva o transformaci kostky

Pokud některý z připojených klientů provede transformaci určité kostky, je vygenerován objekt třídy `BrickTransformMessage`. Objekt této třídy obsahuje identifikátor transformované kostky, proměnnou výčtového typu, určující typ provedené transformace – translace či rotace. Tím zabráníme nutnosti posílat s každou zprávou všechny geometrické informace a můžeme tak posílat pouze ty informace, které se opravdu změnily. Samozřejmě společně s typem transformace jsou v objektu zprávy uložena také aktuální data po transformaci.

Abychom uchovali konzistentní stav u všech klientů, je tato zpráva po přijetí serverem zreplikována k odeslání všem ostatním klientům.

5.6 Popis procesu inicializace nově přihlášeného klienta

Po spuštění klienta je k jeho inicializaci zapotřebí ze strany serveru provést několik kroků. Nyní si popíšeme, co se děje během tohoto procesu jak na straně klienta, tak na straně serveru.

1. V prvním kroku je serveru poslána samotná žádost o připojení. Jak již bylo řečeno u popisu zprávy `LoginRequestMessage` v sekci 5.5.2, v této žádosti je obsažena informace o verzi protokolu, jenž klient používá, a přezdívka uživatele tohoto klienta.

2. Dalším krokem je akceptování žádosti serverem. Tehdy je klientovi přiřazeno první volné identifikační číslo a je mu odesláno ve zprávě typu `LoginAcceptMessage`. Server si ještě před odesláním potvrzovací zprávy vytvoří instanci třídy `User`, kam si uloží všechny potřebné informace o tomto klientovi (identifikátor, přezdívku a jeho síťovou adresu) a tento objekt si zařadí do svého registru uživatelů. Klient si toto přidělené číslo taktéž uloží do své lokální instance třídy `User` a bude jej používat při každé komunikaci se serverem.

3. Po přidělení identifikátoru a odeslání zprávy jsou klient i server připraveni k inicializaci dat světa. Server získá seznam kostek ze své lokální mapy objektů ve světě a v dávkách je pošle klientovi, který si vytvoří své lokální objekty a nastaví jim data dle informací uložených v každé zprávě[14].

[14] Jelikož informace, obsažené v každé zprávě, odpovídají informacím, které je třeba posílat při vytvoření

4. Po odeslání poslední zprávy s inicializačními údaji světa je server hotov s inicializací a klientovi tuto skutečnost signalizuje odesláním zprávy typu `InitCompleteMessage`. Jakmile ji klient přijme, je schopen přejít do interaktivního stavu a umožnit uživateli interakci se světem.

5.7 Spolehlivosti zpráv a řešení problému ztracených zpráv

Abychom zajistili konzistenci v systému, musíme určit, jaké zprávy si můžeme dovolit ztratit a jaké zprávy naopak chceme, aby byly skutečně doručeny (či případně doručeny ve správném pořadí).

Jelikož náš systém nepoužívá metodu častých aktualizací a zprávy posílá pouze tehdy, pokud opravdu dojde ke změně stavu, za situace, kdy by všechny naše zprávy byly odesílány nespolehlivě, síť by byla vytížená a docházelo by k zahazování paketů na routerech, by se mohlo velmi často stávat, že naše zprávy by nedorazily do svých cílů a vznikala by tak neúnosná míra nekonzistence. Proto musíme opravdu důležité zprávy odesílat tak, aby existovala jistota, že do svého cíle opravdu dorazí.

Zpráva se žádostí o připojení do systému je odeslána s úrovní spolehlivosti, která nám zaručuje, že zpráva do cíle dorazí. Jelikož tato zpráva není ze strany klienta členem určité sekvence zpráv, není zapotřebí, aby bylo zaručeno pořadí (ačkoli ze stejného důvodu lze usuzovat, že by ničemu nevadilo, pokud by tuto úroveň spolehlivosti měla).

Zprávy, tvořící odpověď serveru na klientovu žádost o připojení, však mají nastavenu spolehlivost, zaručující doručení ve správném pořadí, protože se jedná o sekvenci zpráv, zakončenou speciální zprávou o konci inicializace a je žádoucí, aby tato zpráva byla opravdu poslední zprávou z této sekvence. Dalším důvodem je také fakt, že pokud by se například ztratila některá ze zpráv o inicializaci objektů světa, mohl by pak již inicializovaný klient obdržet zprávu, týkající se kostky, o které by neměl žádný záznam. Jelikož k odesílání této sekvence zpráv dochází pouze zřídka (z hlediska celkové doby, po kterou je možné systém provozovat), nemá toto vliv na celkovou interaktivitu a dobu odezvy systému.

Zprávy, týkající se uzamčení či odemčení kostky, mají taktéž nastavenu úroveň spolehlivosti, umožňující doručení v určitém pořadí. Důvody jsou dva:

1. Zpráva uzamčení kostky může předcházet sekvenci zpráv o manipulaci s touto kostkou. Aplikace klienta v zájmu zachování konzistence může kontrolovat, zda identifikační číslo uživatele, nacházející se ve zprávě o transformaci kostky, skutečně odpovídá číslu uživatele, o kterém daná kostka hlásí, že je jejím majitelem. Pokud by tato podmínka nebyla splněna, znamenalo by to, že z pohledu daného klienta s kostkou manipuluje někdo, komu dle všech informací tato kostka nepatří, což je chybový stav.

2. Při vysoké zátěži sítě by se mohlo stát, že některá z tohoto páru zpráv by byla zahozena a mohlo by dojít k situaci, kdy pro určitého klienta zůstává některá z kostek dlouhodobě zamčená proto, že zpráva o jejím odemčení k tomuto klientovi nedorazila, nebo by mohlo dojít ke stavu, kdy klient přijme zprávu odemčení kostky uživatelem, o němž by nevěděl, že kostku zamknul (což by ovšem nemusel být takový problém).

Abychom předešli těmto nekonzistencím, nastavíme úroveň spolehlivosti přenosu pro tuto zprávu na dostatečnou k tomu, aby tyto zprávy byly doručeny ve správném pořadí jak serveru, tak i potvrzení těchto zpráv všem klientům.

nové kostky, je k tomuto účelu využita třída `BrickCreatedMessage`, avšak se svým vlastním typem zprávy (hodnotou typu `message_type`).

Zprávy o transformaci kostky jsou jediným typem zpráv, kde případná ztráta jedné nebo několika zpráv nevadí, protože informace o pohybu kostky jsou aktualizovány v krátkých intervalech. Případná ztracená zpráva je rychle nahrazena novou (v tomto ohledu je zde podobnost se systémy, využívající časté aktualizace stavových informací entit ve světě). Tyto zprávy jsou proto odesílány bez záruk na doručení (bylo by zbytečné přeposílat – a tím i čekat na – zprávu, obsahující informace, které už navíc mohou být v příštích několika okamžicích zastaralé).

Zprávy, týkající se změny stavu klienta, mají v závislosti na své důležitosti nastaveny různé úrovně spolehlivosti. Zpráva od klienta, týkající se jeho odhlášení, má nastavenu spolehlivost přenosu na takovou úroveň, aby k serveru dorazila v pořádku a co nejdříve po svém odeslání. Zprávy serveru, oznamující událost přihlášení či odhlášení uživatele však již tak důležité nejsou a proto nám postačuje, že ke svému cíli opravdu dorazí. Pokud měl odhlášený klient ve svém vlastnictví nějakou kostku, mohou ostatní klienti po obdržení oznámení použít číslo kostky, v něm uložené, k jejímu odemčení ve svých vlastních kopiích světa.

Nyní, když jsme si popsali základní datové struktury, prodiskutovali výhody a nevýhody možných síťových architektur a komunikačních protokolů a popsali si úrovně spolehlivosti jednotlivých zpráv, si detailněji popíšeme, jakým způsobem svážeme síťovou komunikaci se zbytkem aplikace klienta a serveru.

6 Infrastruktura aplikací klienta a serveru

V této sekci se zaměříme především na popis struktury programu klienta a serveru. Podrobněji si popíšeme, jaké třídy tvoří strukturu těchto programů a jakým způsobem mezi sebou objekty těchto tříd komunikují.

6.1 Společné znaky a komponenty klienta a serveru

Co se týče základní struktury aplikací klienta a serveru, jsou zde jisté společné základní znaky. Značná část komponent obou aplikací má svůj základ ve společných bázových třídách, některé třídy jsou pak používány zcela beze změn. Bez zabíhání do příliš podrobných implementačních detailů si nyní popišme základní stavební kameny obou aplikací.

6.1.1 Vláknová architektura aplikací

Pravděpodobně hlavním společným znakem je fakt, že v rámci obou aplikací běží dvě vlákna. První z nich, hlavní vlákno, je zodpovědné za vytvoření objektu druhého vlákna, objektů dalších podpůrných komponent a dále za zpracování zpráv, přijatých od vzdáleného hostitele. Druhé vlákno, založené na sdílené bázové třídě `ConnectionThread` (která je odvozena od třídy `Thread`, pocházející z multiplatformní open-source knihovny OpenThreads), je zodpovědné za zpracování síťového provozu (tj. přijímání a odesílání zpráv). Samotné odesílání a přijímání zpráv je zprostředkováno přes rozhraní knihovny RakNet, jež je v této knihovně implementováno jak pro stranu serveru (rozhraní `RakServerInterface`, implementované třídou `RakServer`), tak pro stranu klienta (rozhraní `RakClientInterface`, implementované třídou `RakClient`).

6.1.2 Vstupní a výstupní fronty zpráv

Hlavní a vedlejší vlákno sdílí dvě fronty zpráv – *vstupní* frontu a *výstupní* frontu. Obě fronty jsou instancemi třídy `MessageQueue` a jak již název napovídá, vstupní fronta shromažďuje zprávy, přijaté od vzdáleného hostitele, a výstupní fronta uchovává zprávy, určené k odeslání. Třída `MessageQueue` je v podstatě obalovou třídou pro kontejnerovou třídu STL knihovny s adaptérem fronty (angl. *queue*). Protože k ní však může přistupovat více vláken, je nutné zajistit, že v jednu chvíli bude její obsah měnit pouze jedno z nich. Z tohoto důvodu je jednou z členských proměnných této třídy i mutex, což je objekt třídy `Mutex` z knihovny OpenThreads. Ten nám zaručuje bezpečnost operací s frontou z hlediska přístupu více vláken tím, že pro kritické, obsah fronty měnící operace, ji nejdříve zamkne a po dokončení operace opět odemkne.

6.1.3 Rozhraní handlerů zpráv

Dalším společným rysem obou aplikací je rozhraní pro zpracování přijatých zpráv. Jeho základ je definován třídou `MessageHandlerInterface`, obsahující ryze virtuální funkci nazvanou `processInfoMessage()`, kterou budeme volat v implementujících třídách za účelem zpracování přijaté zprávy.

Funkce objektů tříd, odvozených od tohoto rozhraní, je dvojí:

- Zpracování příchozích zpráv

- Generování odchozích zpráv a reakcí na příchozí zprávy

Serverová implementace tohoto rozhraní je definovaná třídou `ServerMessageHandler`, klientská implementace je pak obsažena ve třídě `ClientMessageHandler`. Obě tyto implementace mají jako svůj vstupní bod funkci `processInfoMessage()`, které je předáván odkaz na objekt zprávy, čekající na zpracování.

Každá z těchto implementací samozřejmě obsahuje vlastní specifickou sadu metod, přičemž všechny metody lze klasifikovat do dvou skupin:

- *handle*- metody sloužící ke zpracování příchozí zprávy
- *post*- metody sloužící ke generování odchozí zprávy

Již na úrovni bázové třídy je také předáván pointer na výstupní frontu zpráv a objekt scény, na úrovni jednotlivých implementací pak další podpůrné objekty, k nimž je nutné mít během zpracovávání zpráv přístup.

6.1.4 Scéna

Sdílená třída `Scene`, která slouží coby základ pro klientskou třídu `ClientScene` a serverovou třídu `ServerScene`, je odvozena od stejnojmenné třídy systému VRECKO (viz sekce 7.3.3 na straně 45). Ve své bázové implementaci obsahuje tato třída obalové funkce pro již existující metody rodičovské třídy pro získávání objektů kostek ze scény na základě jejich identifikátoru. Přidané jsou však také metody pro management kostek (zamykání a odemykání).

Na úrovni jednotlivých implementací jsou dodatečně implementovány metody pro přidávání objektů kostek do scény za specifických podmínek (při počáteční inicializaci na straně serveru a po přijetí zprávy o přidání kostky na straně klienta) a jejich opětovné získávání ve formě vektoru.

6.1.5 Rozhraní pro spojení grafické a síťové části aplikace

Poslední třídou, která představuje společný základ pro specifické implementace na obou stranách, je třída, tvořící komunikační rozhraní mezi síťovou a grafickou částí obou aplikací[15].

Ačkoli si princip funkce a základní komponenty systému VRECKO popíšeme později, v tomto bodě bych rád uvedl alespoň tu skutečnost, že celý systém je postaven na vnitřním posílání zpráv a pokud tedy bude uživatel během své práce v systému vytvářet události, z nichž bude zapotřebí syntetizovat síťové zprávy pro server, je zapotřebí mít k dispozici objekt, jenž bude vstupy klienta zpracovávat na síťové zprávy a naopak informace o akcích ostatních připojených uživatelů dle potřeby převádět na události systému VRECKO.

Třída tohoto rozhraní, společná oběma aplikacím, se nazývá `VreckoNetworkInterface`. Je odvozena od třídy, tvořící základ hierarchie tříd systému VRECKO – `BaseClass`. Je tomu tak proto, že chceme být schopni jednoduše spojit funkcionalitu této třídy se zbytkem používaných komponent VRECKO.

Třída `VreckoNetworkInterface` obsahuje ryze virtuální metody, jež budou sloužit ke zpracování událostí systému VRECKO a registraci spojení pro správnou funkci předávání událostí mezi tímto systémem a síťovou částí aplikací. Její dceřinné třídy jsou nazvány

[15] Ačkoli server nevytváří během své funkce žádné zobrazovací grafické okno, používá některé komponenty systému VRECKO, úzce související se zpracováním událostí, jinak běžně vznikajících při práci v „grafickém módu“ tohoto systému.

ClientVreckoNetworkInterface a ServerVreckoNetworkInterface a obsahují specifickou sadu privátních a chráněných funkcí pro zpracování zpráv systému VRECKO.

Tímto jsme vyčerpali seznam komponent obou aplikací, které mají společný základ ve sdílené sadě tříd, a nyní si v krátkosti řekneme něco o komponentách, specifických pro každou stranu.

6.2 Specifické komponenty aplikace klienta

6.2.1 Třída pro zobrazení světa

Tato třída je v rámci aplikace klienta jedinou třídou, která nemá svůj ekvivalent na straně serveru. Jejím posláním je vytvoření a správa okna, jež bude uživateli zprostředkovávat pohled na dění ve virtuálním světě našeho systému.

Tato třída nese název View a podobně jako třída VreckoNetworkInterface je odvozena od třídy BaseClass, abychom docílili požadovaného stupně integrace se systémem VRECKO, jehož funkce budeme používat pro zobrazení světa a zpracování uživatelských vstupů.

Během tvorby objektu této třídy jsou prováděny inicializace v oblasti tvorby okna jako takového a renderování (nastavení materiálů, osvětlení,...). Jelikož systém VRECKO ve své stávající implementaci vyžaduje, aby posloupnost kroků pro inicializaci zobrazení, správy scény a hardwarových zařízení pro manipulaci s objekty ve světě, byla provedena v určitém pořadí, je inicializace objektu této třídy poměrně těsně svázána s inicializacemi v rámci celé aplikace. Z tohoto důvodu jsou některé inicializační operace, související se zobrazováním a manipulačními zařízeními, prováděny v hlavním programu, kde je také načítán konfigurační soubor, jenž nám pomůže základní objekty a vazby nastavit bez našeho zásahu.

Po vytvoření tohoto objektu je volána jeho metoda draw(), která se stará o překreslování scény a zpracování operací nad touto scénou.

6.3 Specifické komponenty aplikace serveru

6.3.1 Registr uživatelů

Registr uživatelů, jenž je implementován třídou UserRegistry, slouží k ukládání informací o uživatelích. Jedná se o třídu, obalující objekt šablony třídy map z knihovny STL, jenž slouží jako fyzické úložiště objektů třídy User, z nichž každý představuje jednoho připojeného uživatele. Jako klíč ke každému objektu User je používán unikátní číselný identifikátor, který je každému uživateli přidělen po jeho připojení do systému.

Kromě mapy objektů uživatelů obsahuje objekt třídy UserRegistry také členské proměnné, sloužící k řízení přidělování identifikátorů. K tomuto účelu slouží dvě proměnné, definované v této třídě:

- Proměnná typu user_id (což je ekvivalent typu unsigned short), sloužící jako čítač připojených uživatelů.
- Instance STL kontejnerové třídy s adaptérem stack, do níž jsou ukládány identifikátory odhlášených uživatelů (tj. identifikátory volné ke znovupoužití).

Při odstraňování každého odhlášeného uživatele server zneplatní jeho identifikátor, tím, že jej uloží do zásobníku „opuštěných“ identifikátorů. Pokud přijde požadavek na připojení od nového klienta, při volání funkce pro přidělení volného identifikátoru je nejdříve zkontrolován

zásobník volných identifikátorů. Pokud se v něm nacházejí volná identifikační čísla, jsou ze zásobníku vyjmuta a přidělena znovu. Pokud je tento zásobník prázdný, je jako nový identifikátor použita aktuální hodnota čítače uživatelů a tento čítač je zvětšen o jedničku. Tím je zabráněno plýtvání identifikátory.

6.4 Princip funkce klienta a serveru

Předtím, než si popíšeme vlastnosti a hlavní komponenty systému VRECKO, si ještě v několika odstavcích vyložme princip funkcev programů klienta a serveru.

Když jsme si v sekci 6 vysvětlovali způsob implementace jednotlivých komponent těchto aplikací, řekli jsme si, že převážná část tříd, jež tyto komponenty definují, má společný, sdílený základ. Z toho lze usuzovat, že i rámcová funkce obou aplikací bude zhruba stejná. Nicméně vzhledem ke specifikám úkolů, které mají tyto aplikace plnit, jsou zde pochopitelně jisté rozdíly. Věnujme tedy několik následujících odstavců popisu toho, jak spolu jednotlivé komponenty aplikace klienta a serveru spolupracují.

6.4.1 Základní inicializace

Proces inicializace (tj. vytvoření sady objektů, tvořících základní stavební kameny obou aplikací) je v podstatě totožný. Vytvořeny jsou objekty:

- síťového vlákna, zpracovávajícího tok dat z a do sítě
- vstupní a výstupní fronty zpráv
- objekty scény a světa, shromažďující objekty ve světě našeho virtuálního systému
- handler zpráv
- rozhraní mezi systémem VRECKO a síťovou vrstvou obou aplikací
- objekty, uchovávající informace o uživateli (či případně uživatelích)
- na straně klienta také objekt pro zobrazení scény v okně

6.4.2 Spuštění podpůrných vláken a hlavních smyček

Po vytvoření objektů, popsaných výše, je spuštěno síťové vlákno, zpracovávající síťový provoz. To provede inicializaci síťové vrstvy knihovny RakNet a v případě klienta odešle na zadanou adresu serveru žádost o připojení. V obou případech (tj. jak v případě klienta, tak i serveru) se v síťovém i hlavním vlákně čeká ve smyčce na zprávy ke zpracování. Síťové vlákno s každým průběhem smyčky kontroluje, zda byl přijat paket a zda jsou ve výstupní frontě zprávy, čekající na odeslání.

Jestliže byl přijat paket (a jedná se o paket, obsahující zprávu z našeho protokolu), je „rozbalen“, převeden na instanci některé z tříd, odvozených od třídy `Message` (viz sekce 5.5.1 na straně 33) a uložen do fronty příchozích zpráv.

Pokud se ve frontě odchozích zpráv nacházejí čekající zprávy, jsou z fronty vyjmuty, zabaleny do instance třídy `BitStream` a předány knihovně RakNet k odeslání vzdálené straně.

6.4.3 Zpracování zpráv v hlavním vlákně

Smyčka v hlavním vlákně kontroluje přítomnost zpráv ve frontě příchozích událostí. Pokud je tato fronta neprázdná, je zpráva, čekající na čele fronty, z této fronty vyjmuta a předána ke zpracování objektu handleru zpráv. Ten je schopen dle typu zprávy vyvolat patřičnou metodu, která bude schopna korektně interpretovat její obsah. Objekt handleru zpráv má díky svým členským proměnným (což jsou pointery na ostatní podpůrné objekty, tvořící kostru aplikace) přístup do ostatních částí programu a je tak schopen dle potřeby provádět operace v souladu s požadavky právě zpracovávané zprávy.

Jakmile je příchozí zpráva zpracována a je nutné na ni reagovat odesláním jiné zprávy, je na konci metody, která zprávu zpracovala, s patřičnými argumenty volána metoda, která vytvoří odpověď na předchozí zprávu a vloží ji do fronty odchozích zpráv. Po ukončení zpracování je objekt zpracované zprávy zrušen.

Objekt handleru zpráv je svázán také s objektem rozhraní síť – VRECKO. Způsob spolupráce mezi těmito komponentami bude podrobně popsán v následující sekci.

7 Systém **VRECKO** a jeho propojení se síťovou vrstvou

V poslední části této práce se budeme věnovat popisu systému VRECKO z hlediska principu jeho funkce a architektury a poté si popíšeme způsob, jakým je dosaženo komunikace mezi tímto systémem a naší síťovou vrstvou.

7.1 Obecný popis a koncepce systému

Systém VRECKO, vyvinutý pracovníky a studenty Katedry počítačové grafiky a designu Fakulty informatiky Masarykovy univerzity v Brně, je komplexní systém pro virtuální realitu, navržený pro maximální rozšiřitelnost. Lze jej používat s mnoha typy haptických zařízení a pomocí přídavných modulů je možno libovolně měnit a rozšiřovat jeho vlastnosti.

VRECKO je založeno na poměrně velkém počtu podpůrných knihovnách, z nichž pravděpodobně nejdůležitější je knihovna OpenSceneGraph, kterou si nyní stručně popíšeme včetně jejích základních datových typů.

7.2 Knihovna **OpenSceneGraph** a její základní datové typy

Knihovna OpenSceneGraph [OSG], původně vyvíjená jako soukromý projekt bývalými pracovníky firmy Silicon Graphics, kteří později její kód otevřeli pro všechny, používá grafovou strukturu pro reprezentaci scény. Grafová reprezentace scény je výhodná především z důvodu efektivity při renderování, jelikož zjednodušuje a urychluje procesy jako je určování viditelnosti, řazení apod.

Graf scény má obvykle formu acyklického orientovaného grafu, přesněji řečeno obráceného stromu, v němž nejsvrchnější uzel je *kořen* celého grafu a nejspodnější uzly jsou jeho *listy*. V listech se obvykle nacházejí informace o geometrii a materiálech, které tyto objekty používají, zatímco vnitřní uzly stromu mohou definovat prostorové a logické skupiny těchto objektů.

Jako základní datový typ pro všechny typy uzlů v rámci grafu scény slouží třída `Node`, která obsahuje seznam pointerů na své rodičovské uzly (kterých může být více než jeden) a přístupové metody pro různá uživatelská data. Současně s těmito daty je s každou instancí `Node` spojena instance třídy `BoundingSphere`, představující ohraničující kouli tohoto uzlu a všech jeho potomků.

Datový typ pro vnitřní uzly grafu má název `Group` a instance tohoto typu obsahují seznam ukazatelů na své potomky. Díky své flexibilitě lze objektům a třídám, odvozených od `Group` vtisknout nejrůznější vlastnosti. Kromě seskupování uzlů dle určitých pravidel lze uzly tohoto typu použít například pro sestrojení různých typů stromů (quadtree, octree), definici různých úrovní detailů, umísťování podstromů pomocí matic apod.

Listové uzly jsou tvořeny speciálními uzly, pro něž je používán typ `Geode` (**Geo**metry **Node**). Tyto uzly obsahují seznam objektů typu `Drawable`, reprezentující zobrazitelná data. Každý z těchto objektů musí být schopen se sám vykreslit a vypočítat svůj ohraničující kvádr.

Je vhodné podotknout, že třída `Drawable` není odvozena od `Node` a tudíž nepředstavuje uzel grafu, stejně jako třída `StateSet`, která se používá pro nastavení některých stavových proměnných knihovny OpenGL, což je knihovna, kterou OpenSceneGraph používá pro vykreslování scény. `StateSet` zároveň slouží jako bázová třída pro další typy, sloužící především k uchovávání stavových informací uzlu.

7.3 Architektura systému VRECKO

Základem jádra systému je množina tříd, které dohromady tvoří infrastrukturu, pohánějící celý systém, a umožňují rozšiřitelnost systému. Jelikož naším cílem je propojit tento systém s námi navrženou síťovou vrstvou, je zapotřebí znát, jakým způsobem do sebe jednotlivé komponenty jádra systému VRECKO zapadají a jak probíhá komunikace mezi nimi. Několik příštích odstavců bude proto věnováno jejich popisu.

7.3.1 `BaseClass` – základ komunikace v systému

Třída `BaseClass` tvoří základ pro velké množství tříd systému. Používá metodu vstupů a výstupů (z nichž každý nese určité jméno), jichž může být libovolné množství a které všem dceřinným třídám `BaseClass` poskytují funkcionalitu pro vytváření vazeb, nutných k předávání událostí mezi jednotlivými objekty v systému. Všechny vazby jsou registrovány v objektu třídy `EventDispatcher`, která – jak napovídá její název – slouží k předávání těchto zpráv jejich adresátům (podrobněji je tento mechanismus popsán v sekci 7.3.5).

7.3.2 Reprezentace objektů scény

Pro reprezentaci objektů ve světě slouží jako základ třída `EnvironmentObject`, odděděná od třídy `BaseClass` a `MatrixTransform` z knihovny OpenSceneGraph, která představuje uzel grafu pro maticové transformace jeho potomků. Každý objekt této třídy si nese své identifikační číslo, skrze které se lze na tento objekt odkázat, a obsahuje také dva vektory tří souřadnic a matici, určující pozici, měřítko a natočení tohoto objektu v prostoru. Poskytovány jsou také obalovací metody pro již existující metody bázové třídy `MatrixTransform` pro správu uzlů grafu, tvořících potomky daného uzlu.

Pro podporu rozšiřitelnosti je možno každému objektu třídy `EnvironmentObject` definovat sadu uživatelských dat, přičemž typ těchto dat je dán hodnotou proměnné výčtového typu `UserData_Type`, určující seznam možných datových typů. Podporován je v podstatě jakýkoli typ dat – základní typy `int`, `float` apod., vektory různých velikostí, matice i uživatelem definované typy.

Současně s tím můžeme každému objektu této třídy definovat zcela individuální schopnosti a chování. Toho lze docílit tím, že objektu přiřadíme jeden nebo více objektů typu `Ability` (podrobnější informace viz sekce 7.3.7).

7.3.3 Scéna a svět

Objekty prostředí (třídy `EnvironmentObject`) jsou shromažďovány v objektu typu `Scene`. Ten obsahuje mapu, do níž jsou tyto objekty ukládány a identifikátor každého objektu je použit jako klíč pro přístup k němu. Mimo správu těchto objektů řeší scéna kolize mezi objekty a obsahuje také metody pro správu grafu scény.

Objekt třídy `World` se pak používá pro reprezentaci světa jako takového. „Obaluje“ objekt scény a současně s tím v sobě udržuje informaci o současné pozici pozorovatele (či pozorovatelů) ve světě a řídí přidělování identifikátorů objektům, které do něj (resp. do scény) vkládáme.

7.3.4 Aktualizace komponent a objektů v systému

Aby se každý objekt nemusel starat o své vlastní aktualizace a zároveň bylo dosaženo jisté úrovně synchronizace, je v systému používán *plánovač* (angl. *scheduler*). Ten je reprezentován objektem třídy `Scheduler` a je používán pro aktualizaci objektů, přičemž u každého objektu lze při jeho registraci do plánovače definovat, s jakou frekvencí (tj. počet aktualizací za vteřinu) má být tento objekt aktualizován.

Aktualizace určitého objektu je prováděna voláním jeho virtuální metody `update()`, kterou objekt dědí od třídy `BaseClass` a jejíž implementace je specifická vzhledem k povaze každého objektu.

Ukazatel na objekt plánovače v sobě obsahují všechny instance třídy `BaseClass` a jelikož se jedná o statickou členskou proměnnou, všechny její instance mají tento ukazatel společný.

7.3.5 Mechanismus předávání zpráv mezi objekty systému VRECKO

Jak jsme již krátce zmínili v sekci 7.3.1, popisující základní třídu `BaseClass` jádra systému VRECKO, třída `EventDispatcher` (resp. její objekt) slouží k předávání zpráv mezi jednotlivými objekty jak systému jako takového, tak i objekty scény. Princip jeho funkce lze připodobnit k poštovní službě, kde každý příchozí balíček, pocházející od určitého adresáta, je předán určitému příjemci.

Adresaci zpráv lze provést dvojím způsobem (přičemž k oběma slouží přetížená funkce `reportEvent()` této třídy):

1. Lze vytvořit vazbu mezi výstupem jednoho objektu a vstupem druhého objektu[16]. Vazby tohoto typu jsou uchovány přímo v objektu dispečeru událostí a rozhodující je zde přesná adresa výstupu objektu, který událost vysílá.

2. Lze odeslat zprávu přímo adresátovi. K tomuto účelu je třeba znát (resp. mít ukazatel na) objekt adresáta a název vstupu, na který chceme událost poslat.

Jelikož zpracování událostí a jejich předávání adresátům probíhá v metodě `update()` dispečeru událostí, je výhodné objekt dispečeru zaregistrovat v plánovači. Tím lze snadno dosáhnout zpracování zpráv systému s požadovanou frekvencí.

Při doručení zprávy je vyvolána metoda `processEvent()` adresáta (zděděná z `BaseClass`), v níž se dle názvu události rozhoduje, jaká vnitřní metoda objektu tuto událost zpracuje.

7.3.6 Adresování vstupů a výstupů objektů

Jestliže je použit první způsob adresování, popsaný v předchozí sekci, je nutno přesně uvést tyto informace:

1. o jaký typ objektu se jedná – systém VRECKO v tomto směru rozlišuje několik typů objektů jak odesílatele, tak adresáta

   - *objekt ve světě* (tj. objekt třídy nebo potomka třídy `EnvironmentObject`) – ten je označován zkratkou *EO*,
   - *objekt zařízení* (angl. *device*) (instance třídy `Device`) – označován zkratkou *De*,

[16] K úspěšnému vytvoření takové vazby je nezbytné, aby oba objekty ve chvíli vytváření této vazby již v systému existovaly.

- *objekt scény* (tj. objekt třídy `Scene`) – tento typ je označen zkratkou CS[17],
- *vnější objekt* – tj. jakýkoli objekt třídy, odvozené od `BaseClass`, který nespadá do žádné z výše uvedených kategorií objektů, je označován zkratkou OO (angl. *outer object*).[18]

2. identifikátor objektu (číselný nebo jiný),
3. název vstupu/výstupu daného objektu (pokud se jedná o abilitu, je označena zkratkou Ab a popsána jak svým názvem, tak také názvem modulu, pod který spadá).

Pro větší názornost si uveďme příklad adresování objektů pro vytvoření spojení mezi systémem VRECKO a síťovou vrstvou, pomocí něhož budeme odchytávat události uchopení kostky, abychom z nich vytvořili zprávu o uzamčení kostky. Řetězec, označující odesílatele, vypadá takto:

```
EO|999#Ab|base::MouseHand#GrabEvent
```

Řetězec, identifikující adresáta, má pak tento tvar:

```
OO|NETWORK_INTERFACE#MouseGrab
```

Adresa odesílatele říká, že odesílatelem je abilita objektu scény typu `EnvironmentObject` s identifikátorem 999, název této ability je *MouseHand* a pochází ze zásuvného modulu *base*. Název výstupu ability, z něhož událost vychází, je pak *GrabEvent*.

Analogicky u příjemce – příjemcem je vnější objekt, zaregistrovaný pod řetězcem *Network-Interface* a vstup pro tuto událost má název *MouseGrab*. Pokud by se jednalo o vstup, na němž by měla být určitá hodnota, lze ji taktéž definovat.

Znak '|' vždy odděluje typ objektu a jeho identifikátor (či název vstupu a hodnotu na něm), znak '#' je oddělovačem jednotlivých částí adresy.

7.3.7 Definice schopností objektů

Schopnost (angl. *ability*) je vlastnost objektu provádět nějakou činnost nebo mít určité chování. Protože třída `Ability`, která představuje entitu jedné schopnosti objektu, je odděděna od třídy `BaseClass` (z toho vyplývá, že může mít libovolný počet vstupů a výstupů) a protože jeden objekt může mít vícero abilit, je možné docílit toho, aby jeden objekt měl mnoho schopností a byl schopen zpracovat nejrůznější typy událostí. Z tohoto důvodu je často výhodnější při implementaci schopností různých objektů posílat události nikoli na vstupy objektů `EnvironmentObject`, ale na vstupy jejich abilit.

Jádrem funkcionality abilit je metoda `processEvent()`. Zde jsou – stejně jako v případě všech podtříd `BaseClass` – zpracovávány příchozí události. Jelikož ability se obvykle používají při implementaci rozšiřujících modulů systému VRECKO, obsahuje třída `Ability` také proměnné pro uložení názvu modulu, jehož je daná instance této třídy členem. Ten je pak používán pro vytváření instancí abilit a adresování událostí pro ně (viz předchozí sekce).

[17] Celý název konstanty, označující v objektu třídy `EventDispatcher` tento typ objektu, je *CLIENTSCENE*, odtud zkratka *CS*.

[18] Takovým objektem je jak třída `View` (viz sekce 6.2.1 na straně 41), tak i objekty potomků třídy `VreckoNetworkInterface` (viz sekce 6.1.5 na straně 40).

7.3.8 Správa vstupních zařízení

Jak již bylo řečeno v obecném popisu systému VRECKO, podporuje tento systém mnoho typů zařízení, používaných pro interakci s virtuálním světem. Podpora pro zařízení je implementována prostřednictvím zásuvných modulů a jejich správu má na starosti objekt třídy `DeviceManager`. Ten řídí především přidělování přístupu k zařízením. Je schopen poskytnout ukazatel na objekt požadovaného zařízení (v případě, že toto zařízení bylo již „otevřeno“ a správce má k dispozici odkaz na ně) a pokud zařízení nebylo dosud použito a dorazí požadavek o ně, je schopen je otevřít a ukazatel poskytnout.

Na zařízení se lze odkázat jak jeho celočíselným identifikátorem, tak i názvem jeho typu. Jakmile je získán ukazatel na objekt daného zařízení, je možno je využívat. Komunikace se zařízením probíhá přes předem určený port, na kterém ovladač tohoto zařízení poslouchá a skrze který je schopen vysílat a přijímat zprávy o událostech.

7.4 Propojení se síťovou vrstvou

Nyní se již budeme věnovat popisu způsobu, jakým je dosaženo získávání zpráv o událostech v systému VRECKO a jejich přetváření na síťové zprávy.

7.4.1 Příprava na komunikaci

V sekci 6.1.5 na straně 40 jsme si krátce popsali třídu `VreckoNetworkInterface`, která slouží jako základ pro převod zpráv systému VRECKO na zprávy pro síťovou komunikaci. Ačkoli na každé straně spojení zpracovávají objekty jejích specifických implementací odlišné zprávy, princip vytvoření a funkce těchto objektů je totožný.

Při vytváření objektů tříd `Client-` a `ServerVreckoNetworkInterface` jsou konstruktorům obou tříd předány ukazatele na důležité objekty – scénu, dispečera událostí a handler zpráv, kterému budeme předávat hodnoty pro syntézu zpráv daných typů a který naopak bude volat metody rozhraní, aby vyvolal událost v systému VRECKO. V konstruktoru obou objektů jsou vytvořeny vstupy a výstupy, skrze které budeme komunikovat s ostatními objekty systému VRECKO. Při definici každého vstupu a výstupu musíme určit jeho název a typ hodnoty, který je s ním svázán.

Vytvoření vstupů a výstupů však k zajištění komunikace nestačí. Je nutné ještě vytvořit a zaregistrovat vazby mezi komunikujícími objekty do objektu dispečeru událostí. Toto je úkolem metody `registerInterfaceInterconnections()`, jež je definována jako ryze virtuální v bázové třídě obou tříd, aby byla vynucena její implementace v podtřídách. V této metodě je před každým voláním metody `insertEventInterconnection()` třídy `EventDispatcher` vytvořena přesná adresa odesílatele a adresáta zprávy a při registraci vazby je definován typ této vazby. Ten může být jednoho z těchto dvou typů:

- Typ *předání výstupu* je reprezentován konstantou *FORWARD_OUTPUT* a používá se v případě, kdy chceme předat na vstup volaného objektu určitou, předem neznámou hodnotu.

- Typ *aktivace vstupu* je reprezentován konstantou *ACTIVATE_INPUT* a používá se tehdy, kdy chceme jen aktivovat vstup adresáta s nějakou předem danou hodnotou.

Jak již bylo řečeno výše, musejí ve chvíli registrace vazby existovat oba objekty. Metoda rozhraní `registerInterfaceInterconnections()` je proto volána samostatně až po načtení

konfiguračního souboru a nikoliv v konstruktoru objektu rozhraní, neboť některé další objekty jsou vytvářeny až po načtení tohoto souboru a pokud by chyběly, vazba by se nevytvořila. Jakmile jsou vazby registrovány, je objekt připraven k přijímání a vysílání zpráv.

7.4.2 Zpracování zpráv mezi sítí a systémem VRECKO na straně klienta

Co se týče typů zpracovávaných zpráv mezi sítí a systémem VRECKO, zajímají nás především tyto události:

- událost *uchopení (uzamčení) kostky*,
- událost *puštění (odemčení) kostky*,
- událost *posunu kostky*,
- událost *vynucení puštění kostky.*

První tři typy událostí vznikají při interakci klienta se scénou v systému VRECKO a my budeme chtít ze zpráv o nich vytvořit určité síťové zprávy. Abychom však byli schopni v případě výskytu těchto událostí (vyjma čtvrté) vytvořit patřičnou síťovou zprávu, musíme vědět, kterého objektu kostky se zpráva týká. Z tohoto důvodu je hodnotou, kterou společně s oznámením o události obdržíme, identifikační číslo této kostky. S jeho znalostí jsme schopni získat ukazatel na patřičný objekt ve scéně a z něj si posléze vzít všechny potřebné informace. Tvorba síťových zpráv se provádí v handleru zpráv, kde voláním patřičných metod, kterým předáme ukazatel na objekt, jehož se změna týká, dojde k vytvoření síťové zprávy a její uložení do fronty odchozích zpráv.

Zpráva o výskytu poslední události má opačný směr než předchozí tři. Týká se situace, kdy klient obdrží zprávu o uzamčení kostky, kterou již uživatel drží. To znamená, že server na základě žádosti jiného klienta danou kostku uzamkl dříve, než k němu dorazil náš požadavek o uzamčení této kostky. Abychom udrželi konzistenci v systému, je třeba uživateli tuto kostku „vzít“. Toho lze docílit tak, že objektu myši ve scéně, který je zodpovědný za uchopování a uvolňování kostek na základě stisku tlačítka myši, vyšleme zprávu o uvolnění tlačítka, čímž dojde k uvolnění kostky, kterou můžeme následně zamknout pro správného uživatele a znemožnit tak původnímu držiteli další pokusy o její uzamčení.

Tohoto je dosaženo voláním metody `forceBrickGrabRelease()`, implementované ve třídě `ClientVreckoNetworkInterface`. Tato metoda předává toto oznámení objektu myši *přímo* (nikoliv přes vazbu). Proto od scény získá nejdříve ukazatel na tento objekt a na daný vstup pošle parametr o hodnotě, odpovídající stavu uvolnění tlačítka myši.

7.4.3 Zpracování zpráv mezi sítí a systémem VRECKO na straně serveru

Na straně serveru jsou pro nás zajímavé dvě události:

- Událost *vyjmutí kostky z hromady*
- Událost *vytvoření nové kostky v hromadě*

Tyto dvě události spolu úzce souvisejí, neboť kostka je v hromadě vytvořena ihned potom, co je z hromady vyjmuta kostka jiná.

Událost vyjmutí kostky je svázána se vstupy *StartUse* a *StopUse* objektu hromady, jenž je umístěn ve scéně. Jakmile na server dorazí požadavek o uzamčení kostky v hromadě a nic

nebrání vyhovění tomuto požadavku, je kostka uzamčena a hromadě je na vstup *StartUse* odeslána zpráva s identifikačním číslem kostky. Hromada pak vybere ze scény kostku s tímto číslem, získá ukazatel na její abilitu[19], která má definován vstup s názvem *Use*, a tento vstup aktivuje. Vyvolání tohoto vstupu způsobí zpětné vyvolání vstupu objektu hromady, který způsobí vytvoření nové kostky na hromadě.

Analogicky jako v předchozím případě je zpracována zpráva o odemčení kostky, která vyvolá vstup *StopUse* hromady, avšak v současné implementaci není tento vstup využíván.

Po vytvoření nové kostky na hromadě je obratem vytvořena událost, oznamující tuto skutečnost, a poslána na vstup *BrickCreated* rozhraní systému VRECKO a sítě. Jakmile objekt tohoto rozhraní přijme zprávu o události, vyvolá metodu, která ze scény vybere nově vytvořenou kostku a pointer na ni předá metodě handleru zpráv, který pomocí ní získá informace o pozici, měřítku a natočení kostky, vytvoří zprávu o vytvoření kostky a uloží ji do fronty odchozích zpráv.

[19] Pro upřesnění – název této ability je *Opoustec* a má ji definovánu každá kostka v hromadě.

8 Testování a zhodnocení realizace

Popisem předávání zpráv jsme vyčerpali vše, co se dalo říci k popisu řešení jako takového. Na tomto místě bych proto zhodnotil celkový výsledek práce a také se rád vyjádřil k některým problémům, jimž bylo potřeba během realizace čelit. Především bych chtěl říci, že si myslím, že systém, který jsme zrealizovali v rámci této práce, tvoří dobrý základ pro další rozšiřování jeho vlastností a možností. Současná implementace síťové vrstvy dává dobrou představu, jakým způsobem lze využít principy systému VRECKO k dosažení schopnosti síťové komunikace.

Při testování realizovaného systému nebyly zjištěny žádné zásadní nedostatky z hlediska síťové komunikace. Systém byl schopen rychlé odezvy na uživatelské vstupy a choval se stabilně. Systém VRECKO, který jsme použili, je ve své současné podobě solidním systémem pro virtuální realitu s velkými možnostmi úprav a vzhledem k jeho architektuře je přidání podpory pro další zprávy, přenositelné po síti, v podstatě jednoduchou záležitostí. Důležité je, aby v tomto systému existovala podpora pro věci, které budeme chtít v budoucnu svázat se síťovou vrstvou. Převod zpráv o událostech s nimi souvisejících je pak víceméně přímočarý.

Bohužel však musím podotknout, že ačkoli jsem přesvědčen, že realizovaný systém je dobrým základem pro další rozvoj, jsem toho názoru, že mohl být ještě lepší. Hlavním problémem, se kterým jsem během práce velmi často a urputně bojoval, byl někdy až zoufalý nedostatek kvalitní dokumentace k použitým knihovnám, samotný systém VRECKO nevyjímaje. Některé, z hlediska našich požadovaných cílů významné knihovny (zejména OpenSceneGraph) totiž mimo dokumentaci, vytvářenou zpracováním komentářů v souborech se zdrojovým kódem, nabízejí jen velmi omezené zdroje informací. Samotní tvůrci projektu OpenSceneGraph doporučují svým uživatelům pro seznámení se s knihovnou a pochopení funkce jejích komponent studium zdrojového kódu. To je dle mého názoru nevhodný postup pro tento účel, jelikož má vlastní zkušenost praví, že u tak složitých systémů, jako je OpenSceneGraph a částečně také VRECKO se studiem zdrojového kódu získává celkový obraz funkce (a tedy i použití systému) velmi pomalu a je dosti časově náročný.

Já sám jsem studiem zdrojového kódu spotřeboval ohromné množství času a jsem přesvědčen, že s kvalitnější dokumentací klíčových knihoven a systému VRECKO by rychlost celého procesu realizace řešení byla o mnoho rychlejší a zcela jistě by zbyl čas také na další vlastnosti, které jsme na počátku projektu zamýšleli (například detekce kolizí mezi kostkami a s ní související určité optimalizace toku síťových dat, podpora audiohovorů apod.).

Nicméně však si myslím, že navržený systém plní základní požadavky, které jsme na něj kladli, a cíle, které jsme si vytyčili.

9 Závěr

Cílem této práce bylo provést návrh a realizaci distribuovaného virtuálního systému, přičemž jako základ byl použit systém pro virtuální realitu nazvaný VRECKO. Tento systém byl částečně rozšířen o nové zásuvné moduly, aby podporoval vlastnosti v souladu s požadavky této práce, nicméně jeho architektura umožňuje celkem bezproblémové rozšíření o schopnost komunikace přes síť, zejména díky mechanismu předávání zpráv mezi jeho komponentami.

Přestože jsme nedosáhli implementace všech původně zamýšlených vlastností, musím uznat, že tato práce byla pro mne velmi zajímavá hned z několika důvodů. Tím hlavní je pravděpodobně to, že se poměrně živě zajímám o technologie, používané v moderních počítačových hrách a povaha této práce a v ní řešené problémy mají k těmto technologiím poměrně blízko. Další důvod, který bych jmenoval, je, že všechny programy, vytvořené v rámci této práce, byly napsány s maximální snahou o přenositelnost, což znamenalo věnovat speciální péči jak výběru knihoven, tak stylu programování, aby byly dodrženy všechna pravidla a nepoužívaly se vlastnosti, které nejsou platné pro všechny zamýšlené platformy.

Mimo tyto důvody jsem měl díky této práci také možnost oživit si principy programování síťových aplikací a celou problematiku této oblasti informačních technologií, což je v praxi také výhodné.

Jelikož navržený systém je pouze základem pro další rozvoj, směrů, jakými se může ubírat je velmi mnoho. Za všechny jmenujme výše zmíněnou detekci kolizí, dále kupříkladu podporu pro složitější objekty a také složitější svět jako celek (včetně detekce kolizí mezi objekty světa), vizuální reprezentaci virtuálních postav, případně přenosy videa a zvuku apod. Možností je opravdu mnoho.

Reference

[AB94] V. Anupam, C. Bajaj: *Distributed and Collaborative Visualization*, IEEE Multimedia 1 (2), Summer 1994

[BM94] D. P. Bruntzman, M. R. Macedonia: *MBone provides audio and video across the Internet*, IEEE Computer 27(4), April 1994

[CH93] C. Carlsson, O. Hagsand: *DIVE – Platform for multi-user virtual environments*, Computers & Graphics 17, November – December 1993

[Funk95] T. Funkhouser: *RING: A client-server system for multi-user virtual environments*, Proceedings of the 1995 Virtual Reality Annual International Symposium, ACM SIGGRAPH, Seattle, April 1995

[IEEE95] Institute for Electrical And Electronics Engineers: *IEEE Standard for Distributed Interactive Simulation – Application Protocols*, IEEE Standard 1278.1, IEEE Standards Press, 1995

[MAC94] M.R. Macedonia, M.J. Zyda, D.R. Pratt, P.T. Barham, S. Zeswitz: *NPSNET: A Network Software Architecture for Large Scale Virtual Environments*, PRESENCE: Teleoperators and Virtual Environments 3 (4), Fall 1994

[OSG] http://www.openscenegraph.org – domovská stránka projektu OpenSceneGraph

[RakNet] http://www.rakkarsoft.com – domovská stránka multiplatformní knihovny RakNet pro síťovou komunikaci

[SC95] S. Singhal, D. Cheriton: *Exploiting position history for efficient remote rendering in networked virtual reality*, PRESENCE: Teleoperators and Virtual Environments, Spring 1995

[SZ99] S. Singhal, M. Zyda: *Networked Virtual Environments: Design And Implementation*, ACM Press, 1999

Příloha 1

Návod na testování a uživatelská dokumentace

Návod na testování a uživatelská dokumentace

V prvním kroku je zapotřebí připravit si prostředí pro spuštění systému. K tomu bude zapotřebí USB klíčenka, neboť live CD, které použijeme k nabootování systému, nelze za běhu vyjmout. Na ni nahrajte obsah adresáře *test* z doprovodného CD.

Nyní nabootujte z CD Knoppix Linux. Je nezbytně nutné, aby počítač, na kterém budete systém zkoušet, získal z DHCP serveru nastavení, jinak byste si museli síťové rozhraní nakonfigurovat ručně.

Jakmile bude systém spuštěn a připraven, zapněte terminál klikem na ikonu obrazovky v pruhu dole. Nyní použijte tuto sekvenci příkazů:

```
# Přepneme na uživatele root
sudo su

# Připojíme klíčenku
# sda1 by mělo být zařízení, odpovídající vaší klíčence.
mount /mnt/sda1

# Zkopírujeme archív s knihovnami do cílového umístění
cp /mnt/sda1/lib/libs.tar /usr/local/lib

# Přepneme se do adresáře s archívem knihoven
cd /usr/local/lib

# Rozbalíme archiv
tar xvpf libs.tar

# Nastavíme proměnnou LD_LIBRARY_PATH, aby ld našlo naše knihovny
export LD_LIBRARY_PATH=‘‘/usr/local/lib’’

# Pokud budeme muset vyjmout USB klicenku, radeji si data zkopirujeme jinam
mkdir /home/test
cp -r /mnt/sda1/dp_test/* /home/test

# Nyní jsme připraveni ke spuštění programů
cd /home/test
```

Dle toho, zda chceme spustit aplikaci klienta či serveru zvolíme jeden z následujících příkazů:

```
cd netclient    # Přepnutí do adresáře klienta
cd netserver    # Přepnutí do adresáře serveru
```

Pokud spouštíme server, stačí nám zadat tento příkaz:

```
./netserver
```

Tím se spustí aplikace serveru a začne čekat na příchozí připojení. Pokud budeme chtít nyní spustit klienta, budeme si muset vytvořit novou instanci terminálu. Pro to stačí v okně

stávajícího terminálu stisknout klávesy Ctrl + Alt + N, což otevře druhou instanci ve stejném okně, nebo můžeme opět kliknout na ikonu, což vytvoří samostatné okno.

Každopádně však budeme muset před spuštěním klienta provést dva kroky z předchozího postupu – přepnout na uživatele *root* a exportovat proměnnou *LD_LIBRARY_PATH* stejným způsobem jako v předchozím případě:

```
sudo su
export LD_LIBRARY_PATH=‘‘/usr/local/lib’’
```

Nyní se přesuneme do adresáře klienta a spustíme jej

```
cd /mnt/sda1/dp_test/netclient
./netclient <ip_adresa_serveru> <přezdívka_uživatele>
```

Pozor! Toto pořadí parametrů je *povinné*. Adresu zadáváme bez čísla portu.

Nyní by již měly běžet obě aplikace a mělo by být zobrazeno okno se scénou. V něm pomocí myši můžete přesouvat kostky z hromady na libovolné místo a na hromadě by se s každou odebranou kostkou měla vytvořit nová.

Příloha 2

Adresování zpráv v broadcastingu a multicastingu

Adresování zpráv v broadcastingu a multicastingu

Adresování zpráv v broadcastingu

Princip adresování paketů pro všesměrové vysílání spočívá v tom, že vysílající hostitel nejprve vytvoří masku všesměrové adresy, která vymezuje určitou část sítě (tj. určitou skupinu potenciálních příjemců). Pokud by kupříkladu chtěl nějaký hostitel vyslat zprávu všem počítačům připojeným do domény s IP adresou 147.228.52.*, musel by vytvořit masku ve tvaru 147.228.52.255. Číslo 255 definuje všesměrovou adresu pro danou podsíť. Pro doménu 147.*.*.* by pak všesměrová adresa měla tvar 147.255.255.255. Pokud vysílající hostitel nechce vymezovat podsíť příjemců, může použít všesměrovou adresu ve tvaru 255.255.255.255. Zprávy takto adresované budou pak doručeny všem příjemcům připojeným k lokální síti.

Adresování zpráv v multicastingu

Každý strom v multicastingu má přiřazenu speciální pseudo IP adres, tzv. *multicastovou IP adresu* nebo také *adresu třídy D*. Pro multicastové adresy jsou vyhrazeny tyto specifické rozsahy (dle [SZ99]):

Typ rozsahu	**Rozsah(y)**
Celkový rozsah multicastových adres	224.0.0.0 – 239.255.255.255
Rezervované rozsahy přiřazené IANA	224.*.*.*, 225.*.*.*, 232.*.*.*
Rozsah obvyklý pro jednu instituci	239.*.*.*
Volné rozsahy	226.*.*.* – 231.*.*.*, 233.*.*.* – 238.*.*.*

Tabulka 2: Seznam typů rozsahů multicastových adres

Zdroj použije multicastovou adresu určité skupiny k odeslání zprávy a tato zpráva bude pak postupně doručena všem členům této skupiny.

Definice rozsahu v multicastingu

Pokud někdy vyvstane potřeba omezit rozsah potenciálních příjemců zprávy na určitou podmnožinu celkové množiny členů dané multicastové skupiny, lze k tomuto účelu použít parametr doby života paketu, používaný v IP protokolu, popsaného na straně 6. Pro potřeby multicastingu je jeho rozsah hodnot rozdělen do několika intervalů a každý interval rozšiřuje předchozí skupinu o nový rozsah příjemců. V tabulce 3 jsou vypsány jednotlivé rozsahy hodnot TTL a s nimi související skupiny příjemců.

Hodnota TTL	**Rozsah příjemců**
0	Lokální hostitel
1	Členové skupiny na místním segmentu LAN
2 – 31	Celá místní síť
32 – 63	Místní region
64 – 127	Místní kontinent
128 – 255	Všem bez omezení

Tabulka 3: Rozsahy hodnot TTL pro určování množiny členů skupiny

Na tomto místě by bylo vhodné podotknout, že rozmezí hodnot TTL, uvedené v tabulce 3 na řádcích 4 a 5, nejsou v současné době implementovány příliš spolehlivě a chování v dané situaci závisí na vyhodnocení paketu routerem (časté jsou případy, kdy paket s TTL o hodnotě 32 a výše je vyhodnocen jako „globální“, tj. určen pro všechny členy dané multicastové skupiny, nezávisle na jejich geografickém umístění).

Není také možné jakýmkoli způsobem omezit přístup aplikací k určité multicastové adrese. Z tohoto důvodu se může stát, že aplikace, využívající jednu určitou adresu, může přijmout paket, který byl vytvořen nějakou jinou, naprosto neznámou aplikací. Je proto důležité, aby navrhovaný systém byl schopen rozlišovat mezi „svými“ a „cizími“ zprávami (přičemž mezi cizí lze počítat i zprávy pocházející od zcela nezávisle běžících instancí systému, provozovaných jinou skupinou uživatelů).

Je dobré ujistit se, že daná adresa není používána jinou aplikací. Konflikt v používání jedné adresy by měl za následek to, že jedna aplikace by přijímala zprávy druhé, čímž by docházelo jednak ke zbytečnému zahlcování sítě a pak také k plýtvání výpočetních sil každého účastníka skupiny (musel by příchozí zprávu zpracovat). Ve výsledku by tak byl snížen celkový výkon systému.

Příloha 3

Výčet typů zpráv a jejich tříd

Výčet typů zpráv a jejich tříd

Níže jsou vypsány všechny hodnoty, definující typ zpráv v rámci jejich tříd.

- MSG_LOGIN_REQUEST (použita ve třídě LoginRequestMessage) – označuje zprávu jako požadavek na přihlášení do systému (odesílána klientem),
- MSG_LOGIN_ACCEPT (použita ve třídě LoginAcceptMessage) – označuje zprávu jako kladnou odpověď na klientův požadavek na přihlášení (odesíláná serverem pouze danému klientovi),
- MSG_LOGIN_REJECT (použita ve třídě KickMessage) – označuje zprávu jako negativní odpověď na klientův požadavek na přihlášení (odesílána serverem pouze danému klientovi),
- MSG_WORLD_DATA (použita ve třídě BrickCreatedMessage) – označuje zprávu jako informaci pro počáteční inicializaci klienta při jeho přihlášení do systému (odesílána serverem pouze danému klientovi),
- MSG_INIT_END (použita ve třídě InitCompleteMessage) – označuje konec procesu počáteční inicializace (odesílána serverem pouze danému klientovi),
- MSG_BRICK_LOCK (použita ve třídě BrickLockUnlockMessage) – označuje zprávu jako požadavek o uzamčení kostky (odesílána klientem),
- MSG_BRICK_LOCKED (použita ve třídě BrickLockUnlockMessage) – označuje zprávu jako potvrzení/oznámení o uzamčení kostky (odesílána serverem všem klientům),
- MSG_BRICK_UNLOCK (použita ve třídě BrickLockUnlockMessage) – označuje zprávu jako požadavek o odemčení kostky (odesílána klientem),
- MSG_BRICK_UNLOCKED (použita ve třídě BrickLockUnlockMessage) – označuje zprávu jako potvrzení/oznámení o odemčení kostky (odesílána serverem všem klientům),
- MSG_BRICK_TRANSFORM (použita ve třídě BrickTransformMessage) – označuje zprávu jako požadavek o transformaci kostky (odesílána klientem),
- MSG_BRICK_TRANSFORMED (použita ve třídě BrickTransformMessage) – označuje zprávu jako potvrzení/oznámení o transformaci kostky (odesílána serverem všem klientům),
- MSG_BRICK_CREATED (použita ve třídě BrickCreatedMessage) – označuje zprávu jako oznámení o vytvoření nové kostky na hromadě (odesílána všem klientům),
- MSG_CLIENT_LOGGED_IN (použita ve třídě ClientConnectionStatusChangeMessage) – oznamuje přihlášení nového uživatele do systému (odesílána serverem všem klientům),
- MSG_CLIENT_LOGOUT (použita ve třídě ClientConnectionStatusChangeMessage) – oznamuje odhlášení klienta (odesílána klientem),
- MSG_CLIENT_LOGGED_OUT (použita ve třídě ClientConnectionStatusChangeMessage) – označuje zprávu jako oznámení o odhlášení klienta ze systému (odesílána serverem všem zbylým klientům),

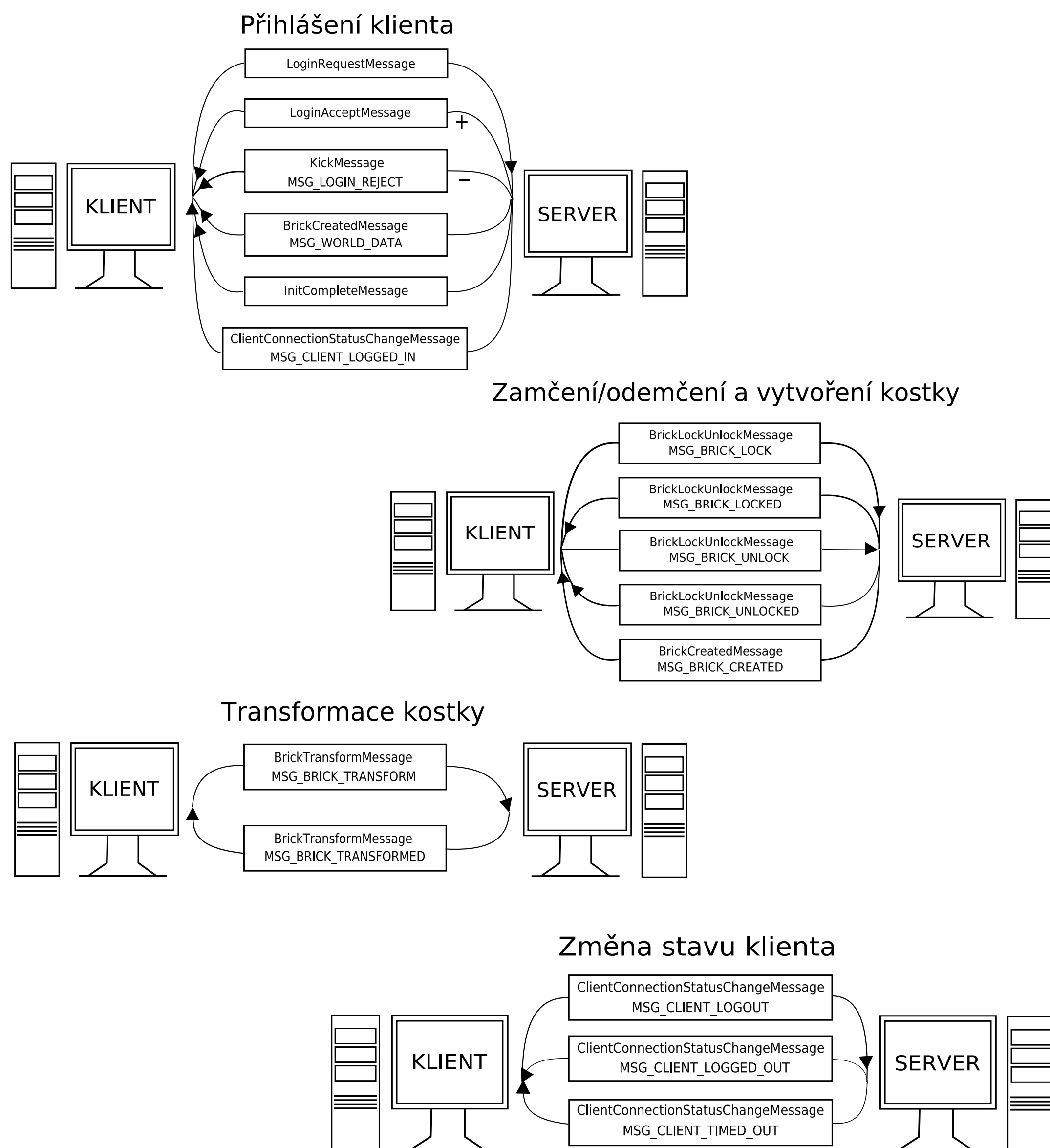
- `MSG_CLIENT_TIMED_OUT` (použita ve třídě `ClientConnectionStatusChangeMessage`) – oznamuje vytikání uživatele (odesílána serverem zbylým klientům),
- `MSG_DUMMY` – speciální hodnota, tento typ nesmí být nikdy použit pro odeslání zprávy.

Příloha 4

Schéma typů zpráv a směr jejich pohybu

Schéma typů zpráv a směr jejich cesty

V této příloze jsou graficky předvedeny typy zpráv a směr jejich cesty. V každém případě jsou zprávy seřazeny odshora dolů dle posloupnosti jejich možného výskytu.



Obrázek 5: Schéma směrů zpráv

Seznam tabulek

1 Souhrn pravidel pro přidělování portů 6
2 Seznam typů rozsahů multicastových adres 58
3 Rozsahy hodnot TTL pro určování množiny členů skupiny 58

Seznam obrázků

1 Schéma fyzického spojení klientů a serveru 15
2 Schéma logického spojení klientů a serveru 15
3 Schéma architektury server-klient s více servery 17
4 Spojení peer-to-peer na lokální síti 17
5 Schéma směrů zpráv 65

Souhlasím s tím, aby tato diplomová práce byla k dispozici k prezenčnímu půjčování v Univerzitní knihovně.

V Plzni dne 29. srpna 2006 ....