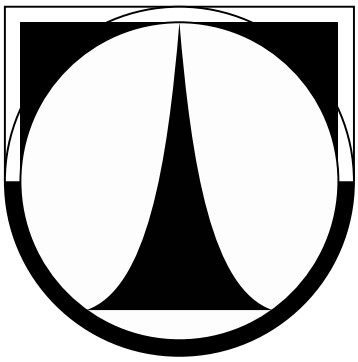
TECHNICKÁ UNIVERZITA V LIBERCI
Fakulta mechatroniky, informatiky a mezioborových studií



Autoreferát disertační práce

Ing. Jiří Hnídek

Liberec 2010

TECHNICKÁ UNIVERZITA V LIBERCI
Fakulta mechatroniky, informatiky a mezioborových studií

Studijní program: P2612 Elektrotechnika a informatika
Studijní obor: 2612V045 Technická kybernetika

Síťový protokol pro grafické aplikace
Network protocol for graphics application

Ing. Jiří Hnídek

Školitel: doc. RNDr. Pavel Satrapa PhD.
Pracoviště: Ústav nových technologií a aplikované informatiky

Rozsah práce:
Počet stran: 171
Počet obrázků: 82
Počet tabulek: 4

30.12.2010

Anotace Tato disertační práce se zabývá návrhem síťového protokolu pro real-timovou výměnu 3D dat mezi aplikacemi sdílené virtuální reality, kdy jsou na navrhovaný protokol kladeny dva protichůdné požadavky. Protokol musí zaručovat úplnou nebo částečnou spolehlivost přenášených dat. Zároveň musí protokol přenášet data s nízkými latencemi. Při návrhu protokolu se částečně vycházelo z konceptu již existujícího protokolu Verse, jehož původní návrh byl zcela přepracován s ohledem na vyšší bezpečnost, spolehlivost a efektivnost.

Práce se zabývám především návrhem nového resend mechanismu, který díky přeposílání aktuálních dat dosahuje nízkých latencí. Korektnost důležitých částí návrhu protokolu - autentizace, handshake a samotný resend mechanismus - byla ověřena programem Spin pomocí verifikačních modelů vytvořených v programovacím jazyku PROMELA.

Efektivita a spolehlivost implementace navrženého protokolu byla ověřena v experimentálním prostředí. V tomto prostředí zároveň došlo k testování vybraných transportních protokolů včetně původního protokolu Verse. Výsledky jednotlivých experimentů prokázaly, že navržený protokol může být efektivně použit pro sdílení rozsáhlých scén virtuální reality.

Klíčová slova: síťový protokol, grafická aplikace, 3D, virtuální realita

Annotation This disertation thesis deals with the design of network protocol for exchange of 3D data between application of distributed virtual environment. Two antithetical requirements are claimed for such protocol. Protocol has to be partially or completely reliable. Neither delay jitter nor too high delay are acceptable. The draft of the protocol si partially based on the concept of existing protocol called Verse. Original draft was completly rewritten with a respect to greater security, realibility and efficiency.

The paper deals with the design of a new resend mechanism. This resend mechanism resends only actual data, thus low latency is achieved. Correctness of the important parts of the draft protocol - authentication, handshake and resend mechanism - was verified using Spin program. Verification models were created in the PROMELA programming language.

Efficiency and reliability of the proposed implementation of the protocol was tested in an experimental environment. Several transport protocols and the original Verse protocol were also tested in this environment. Results of experiments showed that the proposed protocol can be effectively used to share large scenes of virtual reality.

Keywords: network protocol, graphics application, 3D, virtual reality

Obsah

1 Úvod

V oblasti počítačové grafiky se čím dál častěji setkáváme s požadavkem na přenos dat mezi grafickými aplikacemi. V grafických studiích je to dáno tím, že se používají aplikace se specifickým zaměřením na daný úkol (fyzikální simulace, vizualizace, animace postav, apod.). Zároveň dochází k úzké profesní specializaci grafiků pracujících s těmito aplikacemi. V situaci, kdy spolupracuje několik lidí v jednom týmu nebo uživatel používá více různých aplikací pro svoji práci, tak narážíme na problém s konverzí a přenosem dat. Obecně platí, že výstupní soubor jedné aplikace je vstupem pro jinou aplikaci. Většinou se data v tomto případě ukládají do textových souborů, kdy při velkém objemu dat neúměrně narůstá velikost výsledných souborů i doba pro jejich ukládání a načítání. V pracovních postupech uživatelů se stále setkáváme s formátem OBJ Wavefront nebo nověji s formátem COLLADA postaveném na technologii XML, který má ambice stát se průmyslovým standardem v dané oblasti. Obě technologie ovšem neřeší problém nezbytnosti ukládání dat do souboru a jeho opětovné načítání v jiné aplikaci i při sebemenší změně dat.

Další oblast počítačové grafiky, kdy se přenášejí data mezi aplikacemi, jsou aplikace sdílené virtuální reality (SVR). Data jsou v tomto případě přenášena pomocí síťového protokolu. Nejrozšířenějšími aplikacemi SVR jsou bezpochyby herní aplikace typu first-person shooter (FPS), které většinou nevyžadují sdílení geometrie a topologie 3D objektů. Vystačí si se sdílením polohy jednotlivých avatarů a jejich stavů. Pokud se uživatelé ASVR mají setkávat a společně měnit prostředí virtuální reality, tak se protokoly herních aplikací jeví jako nedostatečné. V takovém případě je potřeba odlišný přístup návrhu síťového protokolu. Nároky na něj jsou ovšem velmi protichůdné. Jednak jsou vyžadovány nízké latence přenášených dat, jak to poskytuje například transportní protokol UDP. Zároveň je potřeba zaručit spolehlivost přenášených dat, ale nikoliv úplnou spolehlivost přenášených dat doručených ve správném pořadí, tak jak to zaručuje například transportní protokol TCP.

Síťový protokol, který měl ambice být univerzálním síťovým protokolem pro komunikaci mezi grafickými aplikacemi, je protokol Verse. Verse byl od počátku navrhován speciálně pro efektivní real-timové sdílení dat. Na jeho vývoji se podílelo Uni-Verse konsorcium v rámci 6. rámcového program Evropské unie. Členy konsorcia bylo několik významných evropských univerzit a výzkumných institucí (KTH, Fraunhoufer Institut, Helsinky University of Technology, Interactive Institute, Blender Foundation, a další). Tento projekt měl za cíl i vývoj podpůrného aplikačního vybavení nebo integraci protokolu do vybraných grafických aplikací. Bohužel se možná až příliš zaměřil spíše na vývoj aplikací používající protokol Verse než na vývoj samotného protokolu. Po ukončení financování z Evropské unie vývoj protokolu Verse bohužel víceméně ustal a protokol Verse se nakonec z mnoha důvodů příliš nerozšířil. Cílem této disertace bylo navrhnout lepší náhradu původního protokolu.

2 Protokol Verse

Vývoj původního protokolu Verse započal Eskil Steenberg a Emil Brink v roce 1999 na KTH. Tento protokol byl navrhován poměrně živelně. V konečném důsledku má specifikace [9] i implementace původního protokolu Verse

mnoho nedostatků. Na druhou stranu v jeho specifikaci lze nalézt několik zajímavých myšlenek, které byly inspirací pro nový návrh specifikace protokolu Verse.

Rozhodnutí vytvořit zcela nový protokol mělo mnoho důvodů. Jeho původní ad-hoc návrh, způsob implementace a nekompletní specifikace neumožňovaly návrh změnit, aby výsledný protokol byl robustní, spolehlivý a zároveň zpětně kompatibilní. Dalším důvodem pro návrh nového protokolu byla zkušenost s implementací starého protokolu do programu 3D modelovacího a animačního programu Blender, která byla velmi komplikovaná a v konečném důsledku nekompletní, pomalá a nestabilní. Výsledná implementace byla prezentována na konferenci BCONF 2005 [4].

2.1 Požadavky na nový protokol

Všechny podmínky na navrhovaný protokol vycházejí z myšlenky, že protokol by měl umožňovat v reálném čase sdílet data mezi grafickými aplikacemi pomocí sítě Internet. Z tohoto lze odvodit spoustu dílčích požadavků:

- Data musí být přenášena s minimálními latencemi
- Pro přenos dat je vyžadována částečná spolehlivost
- Protokol musí mít mechanismus pro navázání spojení a přátelské ukončení spojení
- Bude kladen velký důraz na zabezpečení celého protokolu
- Bude možné dohadovat vlastnosti spojení
- Protokol bude implementovat vlastní sadu Congestion Control mechanismů
- Protokol bude podporovat obecný datový model
- Přístup k sdíleným datům bude možné omezit pomocí přístupových práv

2.2 Kódování dat

Všechny vícebytové celočíselné hodnoty jsou přenášeny jako big-endian (nejvíce významný byte je první). Protokol umožňuje přenášet i číselné hodnoty v plovoucí desetinné čárce. Konkrétně jsou podporovány formáty s jednoduchou (real32) a dvojitou přesností (real64) jak je definuje standard IEEE 745. Pro přenos řetězců jsou definovány dvě struktury *strin8* a *string16*. První položka struktury obsahuje vždy délku řetězce v bytech. Položka *Length* se nepočítá do délky řetězce. Řetězec by neměl být ukončen znakem 0x00, jak to vyžaduje například programovací jazyk C/C++. Takový znak by měl být z řetězce odstraněn. Pokud není řečeno jinak, tak řetězec by měl být kódován pomocí UTF-8.

2.3 Transportní protokol

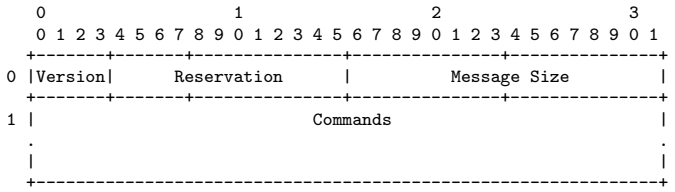
Před rozhodnutím jaký použít transportní protokol byla provedena sada testů s cílem porovnat vhodnost jednotlivých transportních protokolů pro aplikace sdílené virtuální reality (ASVR). Výsledky experimentů jsou uvedeny na straně 22.

Výsledky měření ukázaly, že na transportní vrstvě by nebylo vhodné nasazovat TCP ani spolehlivou variantu SCTP, protože tyto protokoly vedou při real-timovému sdílení dat k příliš vysokým latencím. Při použití částečné spolehlivé varianty protokolu SCTP by se sice předešlo velkým latencím, ale timout omezující přeposílání by komplikoval návrh vlastního resend mechanismu, který má zajišťovat specifické vlastnosti protokolu Verse. DCCP se může zdát jako vhodný kandidát, ale má několik vlastností, které by vedly k neefektivnímu využívání přenosových cest. V jeho neprospěch hraje i fakt, že není široce podporován na koncových zařízeních ani síťových prvcích. Jeho možné budoucí nasazení na transportní vrstvě se ovšem úplně nevylučuje. Jako nejvhodnějším kandidátem se nakonec ukázal protokol UDP [12], protože umožňuje přenos s nízkými latencemi a zároveň je široce rozšířený. Při dalším návrhu se ovšem uvažovalo i s alternativou, že na transportní vrstvě může být v budoucnu použit i jiný vhodný datagramový transportní protokol než je UDP.

Přestože se protokol TCP [13] ukázal jako nevhodný pro real-timový přenos dat, v návrhu protokolu se s ním počítá pro zahájení spojení. Na zabezpečeném TCP spojení nejprve proběhne autentizace uživatele a následně si klient se serverem dohodnout typ datagramového transportního protokolu a další vlastnosti spojení pro real-timové sdílení dat. Použití TCP protokolu pro navázání spojení může mít i tu výhodu, že pokud není možné dohodnout použití UDP, protože to například nedovoluje mobilní operátor, tak spojení pro real-timovou výměnu dat může být degradováno na původní TCP spojení.

2.4 Struktura zpráv a příkazů

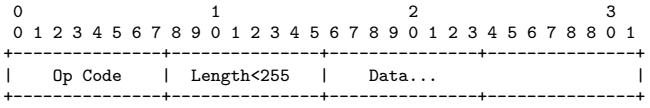
Na TCP spojení jsou data přenášena pomocí zpráv, jež mají strukturu popsanou na obrázku 1.



Obrázek 1: Struktura zprávy posílaná přes zabezpečené TCP spojení

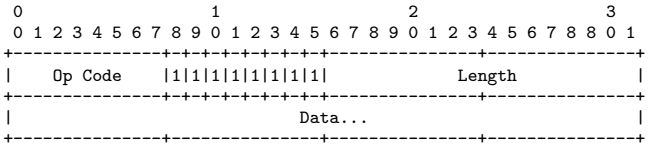
Každá zpráva má jednoduchou hlavičku, která obsahuje *Version*: verzi protokolu, *Reservation*: rezervaci pro další položky a *Message Size*: velikost zprávy včetně hlavičky v bytech. Za hlavičkou následují příkazy, které mají strukturu popsanou na obrázku 2.

Každý typ příkazu má svůj jedinečný *Op Code*, který specifikuje další strukturu příkazu. Za položkou *Op Code* následuje *Length*: délka příkazu (zahrnující i položky *Op Code* a *Length*), která je udávána v bytech a může nabývat hodnot



Obrázek 2: Struktura krátké varianty příkazu

v rozsahu $\langle 2, 254 \rangle$. Pokud je nutné poslat příkaz delší jak 254 bytů, má strukturu popsanou na obrázku 3



Obrázek 3: Struktura dlouhé varianty příkazu

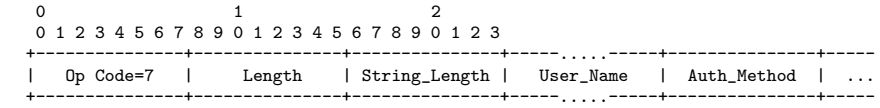
V tomto příkazu může být přeneseno až 65531 bytů dat. Položka s délkou příkazu není záměrně volena větší. Hlavním důvodem je maximální velikost zprávy, která je 65535 bytů. Navíc stejná struktura příkazu se používá i pro realtimový přenos dat a UDP ani DCCP neumožňují přenést v jednom datagramu více jak 65535 bytů.

2.5 Navázání spojení

Komunikaci zahajuje klient na TCP spojení, které musí být šifrované pomocí protokolu Transport Layer Security (TLS) [**?**], protože během této části jsou přenášeny citlivé informace jako je například uživatelské jméno a heslo, které nesmí být odposlechnuty útočníkem. Po autentizaci uživatele si klient se serverem mohou dohodnout, jak bude probíhat komunikace pomocí datagramového spojení.

2.5.1 Autentizace

Po ukončení TLS handshaku začíná vlastní komunikace pomocí protokolu Verse a klient má 30 sekund na to, aby poslal zprávu obsahující příkaz *UserAuthRequest*. Jeho struktura je popsána na obrázku 4. Pokud server tento příkaz do 30 sekund neobdrží, server by měl spojení s klientem automaticky ukončit.



Obrázek 4: Struktura příkazu UserAauthRequest

Op Code tohoto příkazu je 7. Za délkou příkazu následuje délka řetězce v němž je uloženo uživatelské jméno. Za uživatelským jménem následuje identifikátor autentizační metody. Celý příkaz je ukončen vlastními autentizačními daty. V současné době je podporovaná autentizace pouze pomocí uživatelského

jména a hesla. Celý autentizační mechanismus byl ovšem navržen flexibilně, aby bylo možné v budoucnu přidat další autentizační metody.

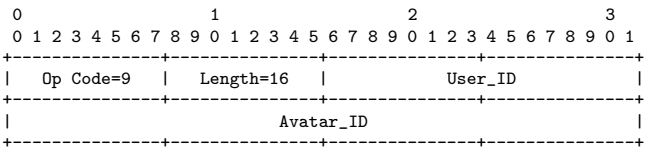
Přestože není doporučeno, aby server podporoval ověřování uživatelů pomocí metody NONE, klient by měl poslat první příkaz *UserAuthRequest* právě s touto metodou. Server musí odpovědět na první příkaz *UserAuthRequest* obsahující nepodporovanou metodu příkazem *UserAuthFailure* se seznamem autentizačních metod, které server podporuje. Tento seznam musí vždy obsahovat metodu PASSWORD a jak server tak klient musí tuto metodu podporovat. Klient by si měl ze seznamu obdržených metod jednu vybrat a použít ji k odeslání platné kombinace uživatelského jména a autentizačních dat.

Když klient pošle neplatnou kombinaci uživatelského jména a autentizačních dat, tak záleží na tom, kolik neúspěšných příkazů *UserAuthRequest* již klient poslal. Pokud tento počet překročil stanovený limit, tak by měl server odpovědět příkazem *UserAuthFailure*, který neobsahuje žádné navrhované metody. V opačném případě by měl server odpovědět původním příkazem *UserAuthFailure* a umožnit klientovi další přihlašovací pokus. Zprávu s tímto příkazem by server neměl posílat ihned, ale odpověď by měl pozdržet. Časový rozestup by měl s každým neplatným pokusem narůstat, aby se co nejvíce zkomplikovalo hádání kombinací uživatelských jmen a autentizačních dat hrubou silou.

Server by měl po odeslání ukončujícího příkazu *UserAuthFailure* ukončit i TLS a TCP spojení. Server by měl spojení automaticky ukončit i v tom případě, že neobdrží další příkaz *UserAuthRequest* do 30 sekund po odeslání příkazu *UserAuthFailure* s navrhovanými metodami.

Pokud server obdrží od klienta první příkaz *UserAuthRequest* obsahující uživatelské jméno, jež není obsaženo v jeho databázi uživatelů, tak by server neměl ukončit spojení, protože útočník zjišťující platné kombinace uživatelských jmen a autentizačních dat by měl ulehčenou práci. Takový přístup by umožňoval případnému útočníkovi nejprve zjistit platná uživatelská jména a tím výrazně ulehčil zjištění platné kombinace uživatelského jména a autentizačních dat.

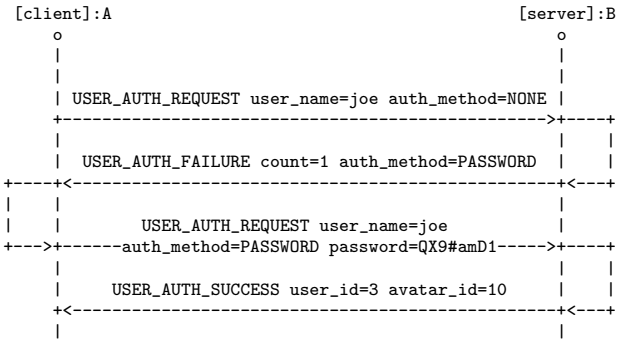
Když se uživatelské jméno a autentizační data (heslo) shodují s databází na straně serveru, server odešle klientu zprávu s příkazem *UserAuthSuccess* jehož struktura je uvedena na obrázku 5.



Obrázek 5: Struktura příkazu UserAauthSuccess

Tento příkaz obsahuje jedinečný identifikátor uživatele: *User_ID* a avatara *Avatar_ID*. *User_ID* se používá například pro nastavení vlastnictví a přístupových práv ke sdíleným datům. Po úspěšném ověření uživatele server vytvoří speciální uzel, který reprezentuje avatara daného Verse klienta. Identifikátor tohoto uzlu je roven *Avatar_ID* z příkazu *UserAuthSuccess*. Uzly reprezentující avatary budou detailně popsány v části 3.

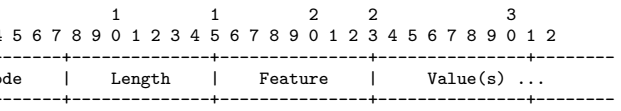
Příklad výměny zpráv a příkazů během úspěšné je uveden na obrázku 6.



Obrázek 6: Příklad úspěšné autentizace

2.5.2 Dohadování vlastností spojení

Po ukončení autentizace může začít dohadování o novém datagramovém spojení. Pro dohadování nového spojení jsou použity systémové příkazy pro dohadování vlastností spojení, které jsou celkem čtyři: *Change_L* (odesílatel navrhuje nastavení vlastnosti na své straně), *Confirm_L* (odesílatel navrhuje nastavení vlastnosti na straně příjemce), *Change_R* (Příjemce potvrzuje odesílateli, že si může nastavit danou vlastnost) a *Confirm_R* (Příjemce potvrzuje, že si nastavil danou vlastnost). Struktura těchto příkazů je uvedena na obrázku 7.



Obrázek 7: Struktura příkazů pro dohadování vlastností spojení

Za délkou příkazu, která nepřímo specifikuje kolik hodnot je v příkazu uvedeno, následuje identifikátor dohadované vlastnosti. Identifikátor vlastnosti je následován seznamem navrhovaných hodnot. Pořadí hodnot v příkazu určuje jejich prioritu. Všechny čtyři příkazy se liší pouze v *Op Code*. Příkazy *Confirm_L* a *Confirm_R* smějí obsahovat buď jednu hodnotu, kterou potvrzují, nebo žádnou v případě, že si příjemce nevybral žádnou z navrhovaných variant.

2.5.3 Dohadování o URL

Uniform Resource Locator (URL) nového datagramového spojení si mohou klient a server dohodnout pouze na TCP spojení po autentizaci klienta. Z vlastního URL by měl klient a server využívat pouze schéma, adresu ve formě doménového jména nebo IP adresy a port. Speciální význam má schéma URL, protože v něm se nastavují základní vlastnosti nového datagramového spojení. Schéma URL musí začínat řetězcem *verse*. Za ním následuje řetězec identifikující datagramový transportní protokol. Na konci schématu musí být uveden řetězec identifikující protokol používaný pro zabezpečení datagramového spojení. Všechny tři části schématu musí být odděleny znakem pomlčky. Podporovaný transportní protokol je v současné době pouze UDP, který je identifi-

kován řetězcem *upd*. Pokud není vyžadováno zabezpečení datagramového spojení, pak je na konci schématu řetězec *none*. Při požadavku na zabezpečení datagramového spojení je možné použít protokol DTLS [16], který je identifikován řetězcem *dtls*. Celé schéma je dle specifikace ukončeno řetězcem *://*. V současné verzi protokolu je tedy možné použít pouze dvě varianty schématu:

- `verse-udp-none://`
- `verse-udp-dtls://`

2.5.4 Dohadování o Cookie

Po dohodnutí nového UDP spojení začne server poslouchat na dohodnutém portu. Na tento port může kdokoliv odeslat paket a server by měl mít možnost ověřit, že daný paket pochází od klienta, který byl autentizován na TCP spojení. Stejně tak klient by měl mít možnost ověřit, že server je ten za koho se vydává. K tomuto účelu slouží náhodně vygenerovaná hodnota Cookie. Dohadovaná hodnota Cookie je přenášena jako string16. Příkazy *Change* a *Confirm* pro dohadování Cookie lze používat jak na TCP, tak na UDP spojení. Na TCP spojení slouží k dohadování nových hodnot Cookie a na UDP spojení slouží k prokázání identity klienta a serveru.

2.5.5 Dohadování o DED

Dohadování o DED (Data Exchange Definition) by mělo sloužit k dohadování pravidel, které musí klient dodržovat při posílání sdílených dat. Tyto pravidla by měla být uložena v externím XML dokumentu.

2.5.6 Dohadování o datagramovém spojení

Vlastní dohadování o novém datagramovém spojení začne server tím, že do zprávy obsahující příkaz *UserAuthSuccess* přidá i příkaz *Change_L* obsahující jeden návrh Cookie a příkaz *Change_R* obsahující jeden návrh DED. Server zasláním těchto dvou zpráv říká klientovi, jakou bude vyžadovat Cookie při poslání prvního prvního paketu na datagramovém spojení a jaká jsou další pravidla při sdílení dat na serveru.

Když server přijme zprávu s potvrzením DED a Cookie, tak náhodně vybere volný UDP port a začne na něm poslouchat. Server nemusí vyslyšet požadavek klienta na vytvoření zabezpečeného nebo nezabezpečeného datagramového spojení a může se řídit vlastní bezpečnostní politikou. Jediné, co je pro server z návrhu URL směrodatné, je navrhovaná IP adresa, protože tu server doplní o platné schéma a použitý port a pošle klientovi v novém návrhu.

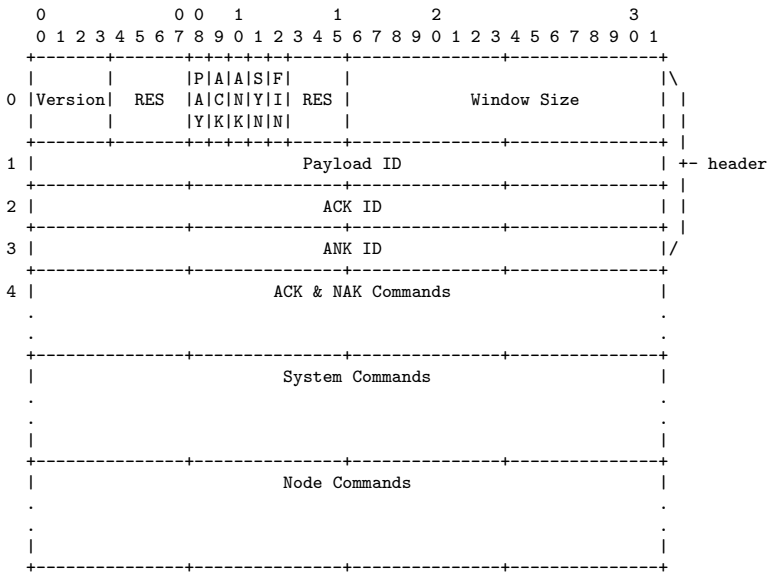
Server tedy odešle klientovi zprávu s příkazem *Confirm_R(URL())*, který klientovi signalizuje, že neakceptuje jeho návrh URL. Server do paketu přidá příkaz *Change_L* s vlastním návrhem platného URL a pokud to klient vyžadoval, tak i potvrzení Cookie. Na tuto odpověď má server limit 30 sekund. Při jeho nedodržení by měl klient spojení se serverem ukončit.

Klient se po přijmutí zprávy obsahující návrh platného URL pokusí provést handshake na UDP spojení a teprve když uspěje, tak serveru navrhované URL

potvrdí. Kdyby handshake na UDP spojení z nějakého důvodu selhal, tak klient odpoví serveru na jeho návrh URL zprávou obsahující prázdný příkaz *Confirm_L(URL())*. Server a klient by měly následně uzavřít spojení. Handshake na UDP spojení bude podrobně popsán v části 2.6.

2.6 Datagramové spojení

Protože v se v současném návrhu protokolu počítá na transportní vrstvě pouze s protokolem UDP, tak se bude nadále uvažovat na datagramovém spojení pouze protokol UDP. Veškeré příkazy přenášené na UDP spojení je potřeba přenášet v paketech, které mají strukturu uvedenou na obrázku 8.



Obrázek 8: Struktura Verse paketu

Paket se skládá z hlavičky za níž musí následovat potvrzovací příkazy. Za potvrzovacími příkazy následují systémové příkazy a teprve po nich tzv. uzlové příkazy přenášející užitečná data. Mezi potvrzovacími příkazy, systémovými příkazy a ani uzlovými příkazy nedochází k žádnému vyplňování. Na obrázku 8 je zarovnání na délku 4 bytů pouze pro lepší přehlednost.

Hlavička paketu začíná verzí protokolu *Version*. Současná verze protokolu má *Version = 1*. Následují 4 bity, které slouží buď jako rezerva pro příliš velká čísla verze protokolu nebo případně pro další příznaky. Za rezervou následují příznaky:

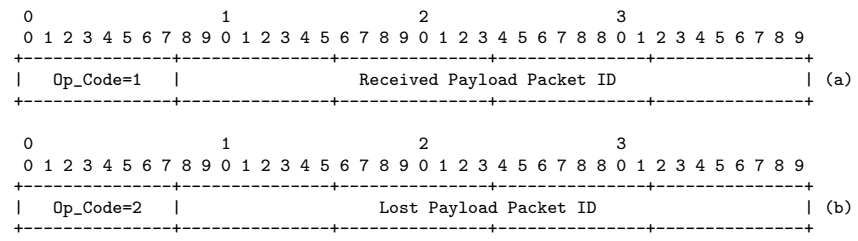
- *PAY* značí, že v paketu jsou přenášena užitečná data (payload data) a položka *Payload ID* obsahuje platnou hodnotu. V opačném případě by měla být vyplněna nulami.
- *ACK* udává, že paket obsahuje speciální *ACK* nebo *NAK* příkazy a položka *ACK ID* obsahuje platnou hodnotu. V opačném případě by měla být vyplněna nulami.

- *ANK* je nastavený, když položka *ANK ID* obsahuje platnou hodnotu. V opačném případě by měla být vyplněna nulami.
- *SYN* se používá během handshaku
- *FIN* se používá během ukončení spojení.

Když si klient a server během handshaku na UDP spojení dohodnou nějaký algoritmus pro Flow Control, tak se pro signalizaci velikosti okénka používá položka *Windows Size*. Přenášená hodnota může udávat velikost v bytech nebo jejich násobcích podle dohodnutého algoritmu.

Payload ID se používá jako jedinečný identifikátor payload paketů. S každým odeslaným payload paketem se inkrementuje příslušný čítač. Když jeho hodnota dosáhne maxima: $2^{32} - 1$, tak se hodnota čítače nastaví na nulu. Stejné pravidlo platí i pro *ACK ID*, které funguje jako jedinečný identifikátor potvrzovacích paketů. Položka *ANK ID* obsahuje identifikátor naposledy potvrzeného payload paketu.

Pokud příjemce potřebuje potvrdit přijetí nebo ztrátu payload paketu, tak za hlavičkou vloží potvrzovací příkazy, které se od ostatních příkazů liší v tom, že se za jejich *Op_Code* nenachází délka příkazu. Potvrzovací příkazy mají konstantní délku a k jejich případné komprimaci dochází jiným způsobem než u ostatních příkazů. Struktura potvrzovacích příkazů je uvedena na obrázku 9. Jejich funkce a použití budou podrobněji popsány v následujících částech.



Obrázek 9: Struktura příkazu ACK (a) a NAK (b)

2.7 Handshake datagramového spojení

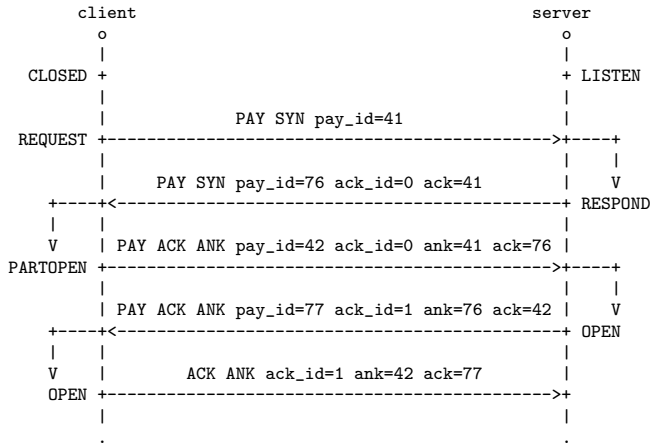
Než si mohou klient a server vyměnit první data, tak mezi nimi musí proběhnout čtyřcestný handshake, jehož návrh byl inspirován handshakem protokolu DCCP [14]. Handshake na datagramovém spojení umožňuje klientovi i serveru prověřit průchodnost přenosových cest před tím než si alokují nezbytné systémové prostředky pro zajištění částečné spolehlivosti datagramového spojení. Během handshaku si také mohou dohodnout další vlastnosti spojení jako jsou použité algoritmy pro FC a CC.

Klient může začít handshake až po té, co od serveru obdrží na TCP spojení návrh URL datagramového spojení. Vlastnímu handshaku musí předcházet DTLS handshake, pokud je schéma serverem navrženého URL zakončeno řetězcem *dtls://*. DTLS protokol i jeho handshake jsou detailně popsány v [16].

Součástí DTLS handshaku je i objevování PMTU, které umožňuje zjistit maximální MTU na obou linkách mezi klientem a serverem. V případě, že je

použita nezabezpečená varianta datagramového spojení, tak je vhodné, aby klient i server použili vlastní PMTU. Získání PMTU umožňuje posílání velkých paketů bez rizika fragmentace paketů na dané lince.

Příklad výměny paketů během handshaku je uveden na obrázku 10. V tomto příkladě nejsou pro přehlednost uvedeny žádné další systémové příkazy pro dohadování vlastnosti spojení (Cookie, Flow Control, Congestion Control).



Obrázek 10: Příklad handshaku na datagramovém spojení

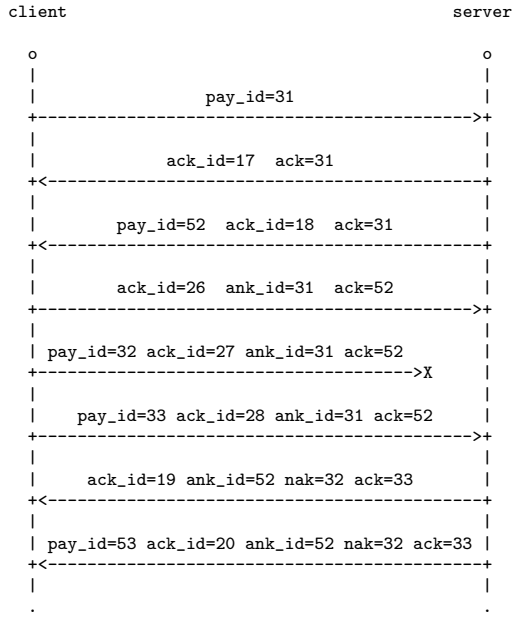
2.8 Resend mechanismus

Jelikož se na transportní vrstvě používá nespolehlivý datagramový protokol UDP a pro real-timové sdílení dat je vyžadována částečná spolehlivost přenášených dat, je potřeba implementovat vlastní resend mechanismus na aplikační vrstvě. Resend mechanismu přejímá některé myšlenky z původního protokolu Verse, ale celkově byl od základu přepracován s důrazem na spolehlivost a robustnost.

2.8.1 Pozitivní potvrzování

Na obrázku 11 je zjednodušený příklad komunikace mezi klientem a serverem ve stavu *OPEN*. Na tomto příkladu budou popsány a vysvětleny základní principy a mechanismy pozitivního a negativního potvrzování paketů pomocí příkazu *ACK*.

Klient si po odeslání payload paketu (obsahuje příznak PAY) s $ID = 31$ uloží jeho obsah do historie odeslaných paketů, protože při jeho případné ztrátě musí být schopný přeposlat aktuální data ze ztraceného paketu. Server po přijetí payload paketu s $ID = 31$ okamžitě odešle klientovi potvrzovací paket (obsahuje příznak ACK) s $ACK\_ID = 17$. Příjemce by měl potvrzovací paket odeslat ihned, aby mohl odesílatel spočítat korektní hodnotu RTT. Po nějaké době pošle server klientovi data v payload paketu s $ID = 52$. Do tohoto paketu je přibalen i obsah předchozího potvrzovacího paketu (obsahuje příznaky PAY i ACK), protože je potřeba maximalizovat pravděpodobnost doručení potvrzovacích příkazů. Toto přibalování paketů se nazývá *Piggybacking*.



Obrázek 11: Příklad pozitivního a negativního potvrzování paketů

Když klient přijme potvrzení o doručení payload paketu s $ID = 31$, obsah tohoto paketu odstraní z historie odeslaných paketů. Klient na příjetí payload paketu s $ID = 52$ zareaguje taktéž odesláním potvrzovacího paketu. Jeho hlavička bude navíc obsahovat platnou položku s $ANK\_ID = 31$ (příznak ANK je nastaven), kterou příjemce (v tomto případě klient) oznamuje odesílateli, že největší potvrzené *Payload_ID* má hodnotu *31* a již není nadále nutné potvrzovat přijetí nebo ztrátu tohoto a předchozích payload paketů.

Příjemce při přijetí payload paketu nemusí posílat potvrzovací paket v samostatném paketu. V případě, že má sám v daný okamžik k odeslání nějaká data, tak potvrzovací paket může být rovnou přibalen k payload paketu.

2.8.2 Negativní potvrzování

Detekce ztráty paketů se v novém protokolu Verse provádí dvěma způsoby. První způsob vychází z původního protokolu Verse, kdy ztráta paketu je detekována příjemcem. Druhý způsob používá pro detekci ztráty paketu Retransmit Timeout (RTO) interval na straně odesílatele paketu. K výpočtu se používá poměrně konzervativní metoda popsaná v [10] zamezující zbytečnému přeposílání dat. *RTO* se vypočítá ze Smoothed Round Trip Time (SRTT):

$$SRTT \leftarrow \begin{cases} RTT, & SRTT = 0 \\ \alpha \cdot SRTT + (1 - \alpha) \cdot RTT, & SRTT > 0 \end{cases} \quad (1)$$

kde RTT představuje Round Trip Time, nebo-li aktuální čas v sekundách, který potřeboval paket na vykonání cesty k příjemci a zpět. Výsledná hodnota $SRTT$ je výsledkem filtrace posledních hodnot RTT. Hodnota α je konstanta v rozsahu $(0, 1)$, která kontroluje, jak rychle se $SRTT$ adaptuje na změny RTT.

Doporučená hodnota pro α je hodnota *0,9*. Časovač pro přeposílání ztracených paketů s spočítá podle vztahu:

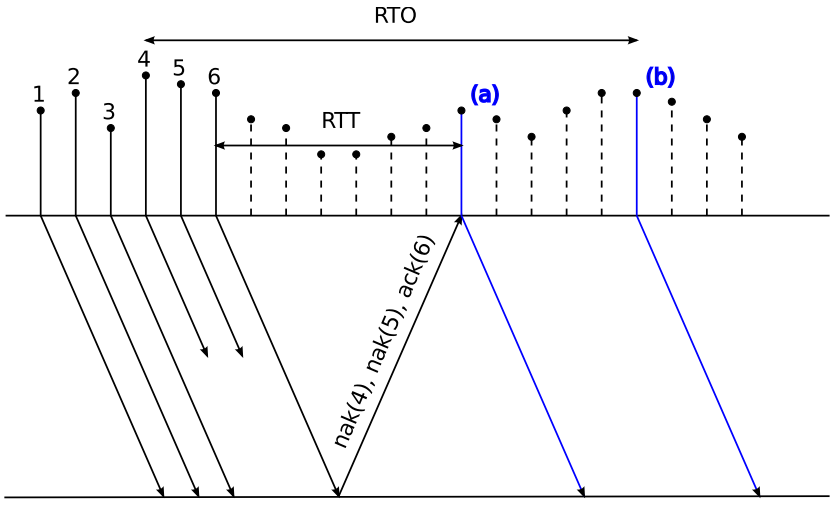
$$RTO = \beta \cdot SRTT \tag{2}$$

kde β je opět konstanta a její doporučená hodnota je 2. K přeposlání paketu dojde v případě, že paket není potvrzen v čase kratším jak *RTO*. Bude ukázáno, že detekce ztráty paketu na základě vypršení časovače není pro aplikace sdílené virtuální reality (ASVR) v některých případech efektivní.

Pro ASVR je někdy efektivnější použít detekci ztráty paketu na straně příjemce, když není doručen paket s očekávaným *Payload_ID*. Tento případ detekce se aplikuje v případě, že $RTT \gg t_{FPS}$, jak je vidět na obrázku 12a. t_{FPS} je doba mezi dvěma snímky. Detekce ztráty paketu na základě časovače se naopak aplikuje v případě, že $RTT \ll t_{FPS}$, jak je vidět na obrázku 12b.

Příklad detekce ztráty paketu na straně příjemce je opět uveden na obrázku 11. Příjemce (v tomto případě server) detekuje ztrátu payload paketu s $ID = 32$, když od odesílatele (v tomto případě klienta) obdrží místo očekávaného paketu s $ID = 32$ paket s $ID = 33$. Změnu pořadí paketu prozatím neuvažujeme. Server musí okamžitě informovat klienta o ztrátě paketu zasláním potvrzovacího paketu s příkazy $Ack = 33$ a $Nak = 32$. Tyto příkazy musí být přibalovány do všech následující paketů, dokud odesílatel nepotvrdí přijetí potvrzovacích příkazů pomocí platné hodnoty ANK v nějakém následujícím paketu.

Když klient obdrží informaci o ztrátě paketu, tak by měl ztracený paket vyvolat z historie odeslaných paketů a ze ztraceného paketu přeposlat pouze aktuální informace. Klient by neměl přeposílat například zastaralou informaci o pozici objektu z paketu $ID = 32$, když pozice objektu byla úspěšně přenesena v paketu $ID = 33$. Prosté přeposílání celého ztraceného paketu by vedlo k nekonzistenci dat na straně klienta a serveru.



Obrázek 12: Porovnání metod detekce ztráty paketu pro velké RTT

Když by byla použita pouze detekce ztráty paketu pomocí časovače na datovém okruhu jehož $SRTT = X\ s$, tak by odesílatel detekoval ztrátu pa-

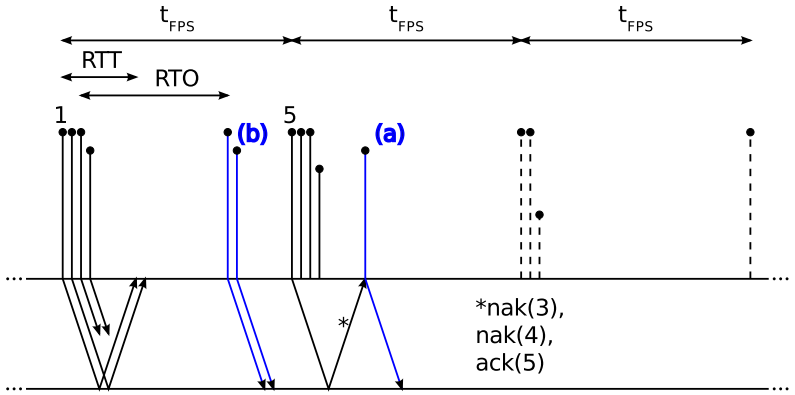
ketu za $t_0 = RTO = 2 \cdot SRTT$. Za předpokladu, že by nedošlo ke ztrátě přeposlaného paketu, by příjemce obdržel paket za $t_{c1} = 2 \cdot SRTT + RTT/2$. Když je $SRTT \sim RTT$, tak výsledný čas doručení muže být aproximován pomocí:

$$t_{c1} \sim 2,5 \cdot RTT$$

Navrhovaná detekce ztráty paketu počítá s tím, že pakety se ztrácejí nejčastěji, když odesílatel posílá velké množství dat a dojde k zahlcení přenosových cest. Odesílatel musí posílat pakety v pravidelných intervalech a časový rozestup t_p mezi jednotlivými pakety musí být $t_p \ll RTT$, jak je uvedeno na obrázku 12. Za tohoto předpokladu příjemce detekuje ztrátu paketu za $t_0 = RTT/2 + t_p$. Jelikož příjemce musí okamžitě odeslat potvrzení o ztrátě paketu a odesílatel ztracený paket, tak zpoždění vzniklé zpracováním lze zanedbat a lze tvrdit, že příjemce obdrží přeposlaný paket za $t_{c2} = RTT/2 + t_p + RTT$. Výsledný čas doručení může být aproximován pomocí:

$$t_{c2} \sim 1,5 \cdot RTT$$

Z obrázku 12 je patrné, že detekce na straně příjemce *(a)* umožňuje za daných podmínek detekovat ztrátu rychleji než pomocí RTO *(b)*. Navíc ztráta všech paketů je detekovaná najednou a nikoliv postupně jako je tomu v případě detekce pomocí RTO.



Obrázek 13: Porovnání metod detekce ztráty paketu pro malé RTT

Na druhou stranu může být i $RTT \ll t_{FPS}$, což je uvedeno v příkladě na obrázku 13. V takovém případě může být značně neefektivní detekovat ztrátu paketu na straně příjemce, protože odesílatel může detekovat ztrátu paketu pomocí RTO *(b)* mnohem dříve než to umožňuje původní metoda *(a)*. Klientská aplikace totiž v mnoha případech nezasílá pakety v pravidelných intervalech, ale pakety jsou zasílány v krátkých shlucích a časové rozestupy mezi shluky t_{FPS} odpovídají FPS používané danou aplikací. Klientská aplikace navíc často nemá potřebu posílat více jak jeden paket za t_{FPS}.

Obě navrhované metody mohou fungovat současně a vždy se uplatní metoda, která detekuje ztrátu paketu dříve. Duplicitnímu přeposlání zabraňuje skutečnost, že při detekci ztráty paketu je ztracený paket odstraněn z historie.

Když odesílatel obdrží příkaz *Nak* obsahující *ID*, které se nenachází v historii odeslaných paketů (ztráta paketu byla detekována časovačem, nebo odesílatel již obdržel tento příkaz *Nak*), tak tento příkaz musí ignorovat.

2.8.3 Keep-alive pakety

Když mezi klientem a serverem není potřeba přenášet žádná data, tak obě strany spojení musí každé 2 sekundy posílat keep-alive pakety, což jsou prázdné payload pakety. Když odesílatel neobdrží potvrzení o doručení Keep-alive paketu do RTO, tak ho nesmí přeposílat, protože prázdný Keep-alive paket neobsahoval žádná užitečná data, která by bylo nutné přeposílat. Když odesílatel neobdrží žádný paket od příjemce po dobu 30 sekund, tak je spojení s příjemcem považováno za ztracené a odesílatel by měl spojení okamžitě zavřít bez pokusu o přátelské ukončení spojení.

2.8.4 Komprese potvrzovacích příkazů

Přestože odesílatel informuje příjemce, že obdržel potvrzení o přijetí paketu pomocí *ANK_ID*, tak při velkém zpoždění na dané lince a velké frekvenci odesílání payload paketů může dojít k brzkému zaplnění velké části paketu potvrzovacími příkazy. Uvažujme následující příklad, kdy klientská aplikace odesílá každých $t_{FPS} = 16,7\ ms$ ($\sim$ 60 FPS) 10 paketů na datovém okruhu s $SRTT = 100\ ms$, by potvrzovací příkazy zabraly v paketu $(SRTT/t_{FPS}) * 10 * 5\ B \sim 300\ B$.

Libovolnou sekvenci potvrzovacích příkazů

$$\begin{aligned} &Ack(N), Ack(N+1), \dots Ack(N+n_1), \\ &Nak(N+n_1+1), Nak(N+n_1+2), \dots Nak(N+n_2), \\ &\dots \\ &Ack(N+n_{m-1}+1), Ack(N+n_{m-1}+2), \dots Ack(N+n_m) \end{aligned} \tag{3}$$

lze zkomprimovat na sekvenci:

$$\begin{aligned} &Ack_0(N_0), Nak_0(N_1), Ack_0(N_2), \dots \\ &\dots Ack_0(N_{m-1}), Ack_{n_{m-1}}(N_{m-1}+n_{m-1}), \end{aligned} \tag{4}$$

kdy v komprimované sekvenci je z každé subsekvence pouze první příkaz *Ack* nebo *Nak*. Celá komprimovaná sekvence musí být ukončena posledním *Ack* příkazem z poslední subsekvence.

2.9 Ukončení spojení

Přátelské ukončení datagramového spojení může být vyvoláno klientem i serverem. Když ukončení spojení vyvolá klient, tak je vhodné, aby se před vlastním ukončením spojení odhlásil od všech sdílených dat a teprve potom se pokusil spojení ukončit. Při ukončení spojení se využívá příznaku *FIN* v hlavičce.

2.10 Bezpečnostní rizika

Některé části navržený protokolu mohou být zranitelné vůči DoS a DDoS útoku. Jedná se především o autentizační mechanismus a handshake datagramového spojení. Při návrhu protokolu se počítalo, že tento problém bude přene-

chán různým packet filtrům jako je například iptables, které umožňují efektivně filtrovat počet pokusů o nové spojení podle různých kritérií.

3 Datový model

Aby byla možná komunikace mezi servere a různými klienty, tak všichni musí vycházet ze stejného datového modelu. Vlastní uložení dat na straně klienta specifikace neřeší. Data jsou strukturována do uzlů podobně jako tomu bylo v původním protokolu Verse. Hlavní rozdíl proti původnímu datovému modelu spočívá v tom, že existuje pouze jeden obecný typ uzlu a uzly musí být umístěny do stromové struktury. Nový protokol dále obsahuje podporu pro přístupová práva na úrovni uzlů, možnost dočasně zamykat uzly a každému uzlu je možné přiřadit určitou prioritu. Podobně jako v původním protokolu může každý uzel obsahovat skupinu tagů, které by měly sloužit především pro popis dat uložených v tzv. vrstvách daného uzlu. Tagy mohou kromě popisu vrstev sloužit i k uložení dat. Každý uzel může reprezentovat objekt, geometrii objektu, materiál, texturu, animační křivku, parametrickou plochu, atd. Výsledná sdílená 3D scéna může být vytvořena pomocí vhodných vazeb a referencí mezi jednotlivými uzly.

3.1 Uživatelé a uživatelské účty

V novém protokolu má každý uživatel přiřazen kromě jedinečného uživatelského jména i jedinečný číselný identifikátor (User ID), který může nabývat hodnot v rozmezí $\langle 100, 65534\rangle$. Rozsah $\langle 100, 999\rangle$ je určen privilegovaným uživatelům. Server by měl autentizovat pouze uživatele z rozsahu $\langle 1000, 65534\rangle$. Hodnota 65535 má speciální význam pro nastavování přístupových práv a nikdy by neměla být přiřazena konkrétnímu uživateli. Uživatel reprezentující samotný server má *User ID* rovný hodnotě 100. Další hodnoty v rozsahu $\langle 100, 999\rangle$ jsou rezervovány pro privilegované uživatele slave serverů distribuované variantu verse serveru.

Verse klient vždy zná své *User ID*, pod kterým je přihlášený, z příkazu *UserAuthSucces*, jež je uvedený na straně 9.

3.2 Uzly

Aby bylo možné jednoznačně identifikovat kterýkoliv sdílený uzel, tak každý uzel musí mít jedinečný identifikátor (Node ID). Musí to být server a nikoliv klienti, kdo přiděluje novým uzlům jejich *Node ID*, protože pouze server má možnost zajistit jejich jedinečnost. *Node ID* může nabývat hodnot od $\langle 0, 2^{32}-2\rangle$. Rozsah možných hodnot je poměrně široký, což dává možnost přiřadit některým hodnotám a rozsahům speciální význam.

Všechny uživatelské účty mají například vytvořený speciální uzel jehož *Node ID* odpovídá *User ID* daného uživatele. Tento uzel může obsahovat další informace o daném uživateli a Verse klienti mají k dispozici jednoduchý způsob jak zjistit seznam platných uživatelů a jejich *User ID*.

3.2.1 Přístupová práva

Přístupová práva by měla umožňovat omezit přístup k libovolným sdíleným datům na serveru. Přístupová práva jsou navržena tak, aby je mohli jednoduše nastavovat uživatelé virtuální reality. Z tohoto důvodu návrh přístupových práv neobsahuje koncept skupin uživatelů, jak je používají UNIXové souborové systémy. Koncept přístupových práv použitý v novém protokolu Verse více vychází z Access Control List (ACL).

Vlastník Každý uzel má svého vlastníka, který má právo daný uzel číst a zapisovat do něj, což znamená, že se může přihlásit k přijímání změn provedených s daty uvnitř tohoto uzlu a sám může měnit data v tomto uzlu. Vlastník uzlu se může svého vlastnictví vzdát ve prospěch jiného uživatele. Tuto změnu může provést buď vlastník nebo privilegovaný uživatel.

Ostatní uživatelé Každý uzel má vždy nastaveno, co s ním mohou dělat všichni ostatní uživatelé. Vlastník souboru může vybranému uživateli explicitně nastavit dodatečná přístupová práva. Může mu povolit číst a zapisovat do daného uzlu.

3.2.2 Stromová struktura

Uspořádání uzlů do obecného grafu bylo v původním protokolu volitelné. Vazby mezi jednotlivými uzly bylo možné vytvořit pouze na základě explicitního požadavku klienta. V novém protokolu je vyžadováno, aby byly uzly uspořádány do stromové struktury. Uspořádání uzlů do stromové struktury zjednodušuje některé operace jako je přihlášení k datům, nastavení priorit apod. Není například nutné explicitně nastavit prioritu pro každý uzel, ale stačí nastavit prioritu pro jeden uzel a tato priorita se dědí na všechny uzly v dané větvi.

Stromová struktura musí vždy sledovat určitá pravidla. Prvním z nich je, že kořenem stromové struktury musí být uzel s ID=0. Jeho vlastníkem musí být server a nikdo další by neměl mít právo do tohoto uzlu zapisovat, pouze ho číst. Kořenový uzel musí mít vždy tři potomky:

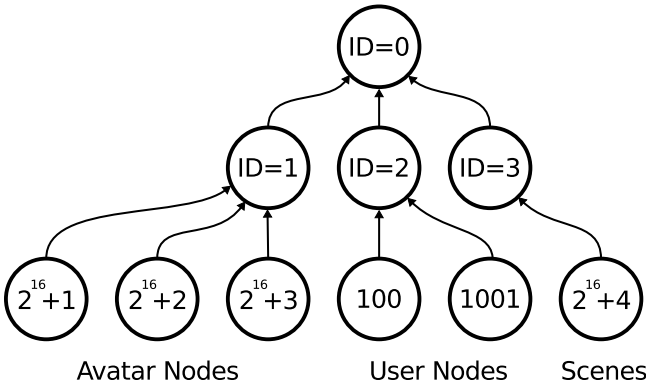
- *Node ID = 1* je předek všech uzlů reprezentujících avatary
- *Node ID = 2* je předek všech uzlů reprezentujících uživatele
- *Node ID = 3* je předek pro kořenové uzly jednotlivých 3D scén

Vlastníkem předka všech avatarů je opět server a nikdo další do něj nesmí mít právo zapisovat. Ostatní uživatelé musí mít právo tento uzel číst. Server musí vytvářet uzly reprezentující avatary jako potomky uzlu s ID=1 a zároveň by zde neměl vytvářet jiné uzly. Díky tomu jsou klienti schopni určit seznam klientů přihlášených k danému serveru.

Podobné vlastnosti platí i pro předka všech uzlů reprezentujících uživatele. Tento uzel je ve vlastnictví serveru a ostatní uživatelé tento uzel smí pouze číst. Potomky tohoto uzlu smí být pouze uzly reprezentující platné uživatele, nejenom aktuálně přihlášené uživatele. Ostatní klienti musí mít šanci určit *User ID* i uživatelů, kteří nejsou momentálně přihlášení. Vlastníkem uzlů reprezentujících

uživatele musí být opět server. Uživatel, jehož *User ID* je rovno danému *Node ID*, by měl mít právo do daného uzlu zapisovat. Ostatní uživatelé by měli mít pouze právo číst obsah tohoto uzlu.

Vlastníkem posledního potomka kořenového uzlu s ID=3 je opět server. Tento uzel smí číst a zapisovat do něj všichni uživatelé. Ve větvích vycházejících z uzlu s ID=3 by mělo docházet k vlastnímu sdílení dat mezi jednotlivými klienty. Na obrázku 14 je uveden příklad uspořádání uzlů do jednoduché stromové struktury.



Obrázek 14: Příklad stromové struktury

Uživatel by měl mít možnost změnit uzlu jeho rodiče. Změna rodiče je uživateli dovolena, pokud mu uzel patří a zároveň musí mít právo zapisovat do rodičovského uzlu.

Při změně rodičovského uzlu musí server hlídat, aby nedošlo k rozdělení stromové struktury na dva nesouvislé grafy (jeden nový strom a graf obsahující kružnicí). Tuto podmínku může server jednoduše zajistit tím, že nedovolí, aby se novým rodičem uzlu stal jakýkoliv jeho následovník.

3.2.3 Avatar

Každá 3D aplikace používá koncept tzv. avatara. Avatar je reprezentant uživatele ve virtuální realitě. Má vždy přiřazenou minimálně jednu virtuální kameru, kterou uživatel sleduje obsah virtuální reality. Dále může mít avatar definované rozměry zabraňující průchod příliš malými otvory, hmotnost, koeficient tření apod.

Každý Verse klient musí mít svého avatara. Je vhodné, aby Verse klient sdílel na serveru informace o svém avataru. U každého typu aplikace SVR je vhodné sdílet o avatarovi rozdílnou sadu informací. Některé aplikace avatara buď vůbec nevyžadují zobrazovat (např. 3D modelační aplikace) nebo zobrazují pouze jeho kameru, aby měli ostatní účastníci ponětí o tom, kde se daný uživatel nachází a na čem pravděpodobně pracuje. Při propojení CAVE [11] pracovišť je zase žádoucí, aby byl avatar pro ostatní účastníky viditelný pomocí animované postavy. Ve všech případech je ovšem žádoucí, aby měli účastníci možnost jednoznačně určit, kolik Verse klientů je k serveru přihlášeno a pod jakými uživateli.

Server tudíž po autentizaci uživatele vytvoří nový uzel označovaný jako *Avatar Node*. Jeho ID se klient dozví v příkazu *UserAuthSucces*, jenž je uvedený na straně 9. Specifikace nic neříká o tom, jakým způsobem se mají sdílet informace o avatarovi v tomto uzlu, protože každý typ aplikace vyžaduje jiný přístup. Toto bude kladeno na bedra DED.

Specifikace ovšem má několik základních pravidel pro *Avatar Node*. Jeho vlastníkem je vždy server a server musí zajistit existenci tohoto uzlu po celou dobu spojení klienta s Verse serverem. Po ukončení spojení by měl server tento uzel smazat včetně všech jeho potomků. Nadřazeným uzlem *Avatar Node* je jak už bylo řečeno uzel s ID=1. Server jako vlastník může daný uzel číst a může do něj zapisovat. Stejné práva by měl mít i uživatel pod kterým je přihlášený daný Verse klient. Ostatní uživatelé by měli mít možnost daný uzel pouze číst.

Uzly pro avatary nemají vyhrazený žádný speciální rozsah. *Node ID* jsou jim přiřazovány jako běžným uzlům z rozsahu $\langle 0, 2^{32} - 2\rangle$.

3.3 Tagy a skupiny tagů

Tagy a skupiny tagů umožňují do uzlů ukládat jednak dodatečné informace jako například textové poznámky, skutečné jméno autora, apod. Tagy a skupiny tagů zároveň umožňují do uzlů ukládat i efektivní informace jako je hmotnost objektu, teplota, atd.

Koncept nového datového modelu počítá s tím, že spousta specializovaných typů objektů je nahrazena jedním obecným typem uzlů. Význam jednotlivých uzlů by měl být popsán právě pomocí tagů.

Každý uzel může tedy obsahovat tagy, které mají (v rámci své skupiny) své jedinečné jméno, jedinečný číselný identifikátor, typ a hodnotu. Tagy navíc musí být uspořádány do skupin, jenž mají (v rámci objektu) své jedinečné jméno a identifikátor. Aby se klient dozvěděl jaké tagy skupina obsahuje a jaké jsou jejich hodnoty, tak se do dané skupiny musí nejprve přihlásit. Od té doby bude od serveru dostávat všechny změny, které ostatní klienti v dané skupině tagů provedou.

3.4 Vrstvy

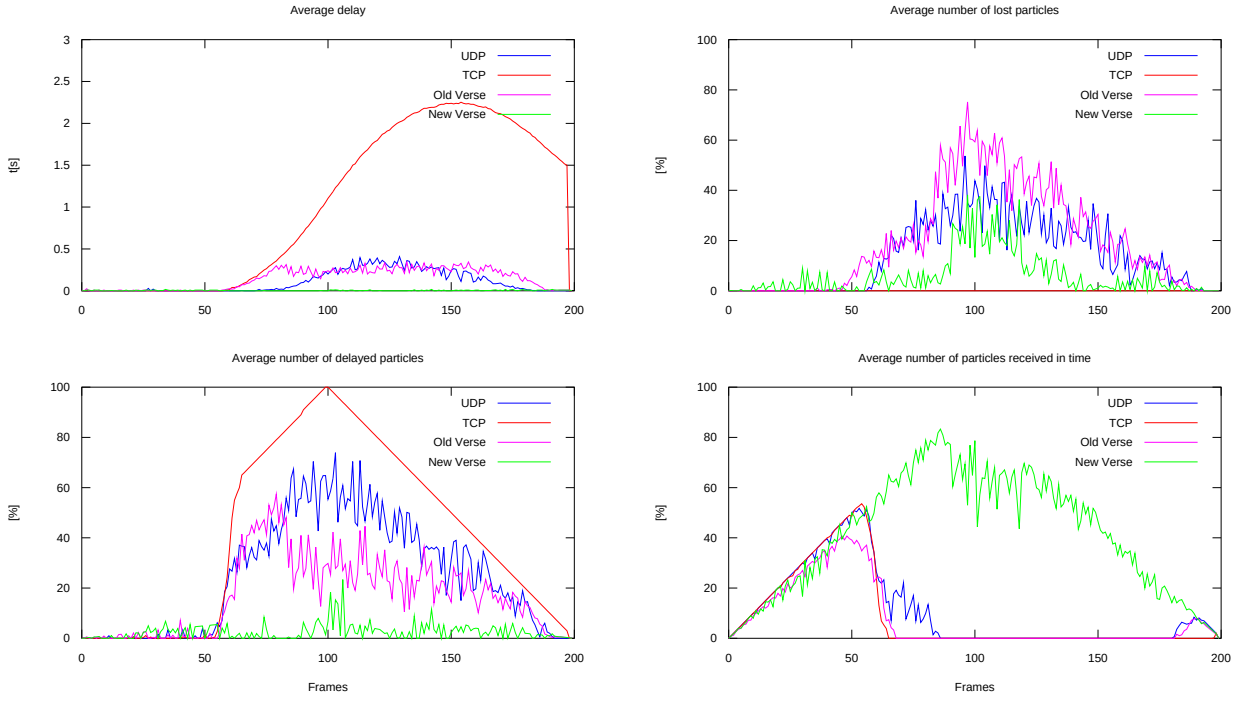
Vrstvy v novém protokolu umožňují sdílet libovolná nestrukturovaná data. Jejich příkazy jsou navrženy pro sdílení objemných dat, jako jsou například souřadnice vertexů, vrcholy plošek, váhy vertexů, apod. Aby klient dostával od serveru změny, které v dané vrstvě provádějí ostatní klienti, tak se musí k této vrstvě přihlásit podobně, jako se přihlašuje ke skupině tagů.

4 Výsledky měření

Pro experimenty byly vygenerovány v programu Blender dva částicové systémy. Pro testování na virtuální lince byl vytvořen částicový systém čítající 100 částic a pro testování ve skutečném síťovém byl vygenerovaný částicový systém obsahující 1000 částic. Oba částicové systémy byly vždy uloženy na stranu klienta i serveru. 100 částic umožňovalo vytvářet při testování na virtuální lince názornou vizualizaci a zároveň bylo možné efektivně omezit šířku pásma pro generovaný datový tok.

Částicový systém byl vygenerován pro animaci s 25 snímky za vteřinu. Všechny testované transportní protokoly používaly na aplikační vrstvě velmi jednoduchý protokol. Klient zahajuje komunikaci se serverem zasláním jednoduché zprávy. Server na tuto žádost reaguje tím, že začne klientovi posílat každých 40 ms pozice všech pohybujících se částic. Prvních 10 příchozích paketů nebo zpráv se použilo k výpočtu průměrného zpoždění na lince. Pozice každé pohybující se částice byla uložena v jednoduché zprávě, která byla totožná pro všechny testované protokoly.

Ve zprávě byla přenášena pouze poloha částic, protože částicový systém měl simulovat stochastický pohyb a další činnosti generované uživateli pracujícími v aplikacích sdílené virtuální reality (ASVR). Pro uživatele virtuální reality je velmi rušivý pohyb objektů, který by měl být spojitý, ale z různých důvodů je tento pohyb přerušován. Jestli se pohyb bude jevit uživateli jako nespojitý ovlivňuje mnoho faktorů: rozlišení zobrazovacího zařízení, vzdálenost uživatele od zobrazovacího zařízení, apod. Z tohoto důvodu byl uvažován nejhorší možný případ, kdy ztrátu jediné polohy částice je schopný uživatel zaznamenat. Zpoždění částice větší jak 40 ms od očekávaného zpoždění bylo tudíž vizualizováno jako problematické zpoždění.



Obrázek 15: Porovnání výsledků experimentálního měření v reálné síťovém provozu

Výsledky experimentů v reálné síťovém prostředí jsou uvedeny na obrázku 15. Spojení, na kterém byly experimenty prováděny, mělo šířku pásma 1,9 Mb/s a průměrné zpoždění na lince bylo 5 ms. K experimentům byl použit částicový systém, který obsahoval 1000 částic. Z výsledků experimentů je patrné, že nový

protokol Verse dává oproti ostatním protokolům mnohem lepší výsledky, které jsou dané především přeposíláním aktuálních dat a jejich efektivní kompresí.

5 Závěr

Tato disertační práce obsahuje popis specifikace nového protokolu Verse určeného pro real-timové sdílení dat v aplikacích sdílené virtuální reality. Specifikace částečně vychází z původního protokolu Verse, ale celý protokol byl od základu přepracován s ohledem na větší bezpečnost, spolehlivost a efektivitu přenosu dat. Specifikace obsahuje popis zahájení spojení, které kromě autentizace uživatele umožňuje provést dohadování o vlastnostech datové komunikace mezi klientem a serverem. Pro dosažení částečné spolehlivosti byl navržen nový robustnější a efektivnější resend mechanismus, který přeposílá pouze aktuální data a zaručuje efektivní využívání přenosových linek. Nové vlastnosti protokolu Verse byly prezentovány na konferenci BCONF 2010 [3].

Součástí specifikace je i popis nového datového modelu, který sice klade na Verse klienty některé nové požadavky, ale zbytečné požadavky původního protokolu ruší. V konečném důsledku je možné provádět sdílení dat, které se starým protokolem nebylo možné. Navíc je možné nastavovat u sdílených dat přístupová práva, což původní protokol také neumožňoval.

Pro všechny důležité části nové specifikace byly vytvořeny verifikační modely v programovacím jazyku PROMELA a provedena jejich verifikace pomocí nástroje Spin. Výsledky verifikace resend mechanismu byly publikovány ve sborníku konference INTED 2009 [2].

Podstatná část specifikace byla implementována v programovacím jazyku C. Popis implementace obsahuje především obecné struktury a postupy, které umožňují efektivně implementovat nový protokol i v dalších programovacích jazycích.

Tato implementace byla použita pro měření v experimentálním prostředí, kdy byla otestovaná vhodnost jednotlivých transportních protokolů pro nový protokol Verse. Zároveň byly v tomto experimentálním prostředí provedeny testy původního a nového protokolu Verse, které ukázaly, že nový protokol dává lepší výsledky. Výsledky těchto experimentů byly přijaty na konfrenci WSCG 2011 [1].

Shrnutí přínosů k rozvoji vědního oboru

V práci je navržený resend mechanismus pro efektivní sdílení dat v aplikacích sdílené virtuální reality, kdy není vyžadován přenos dat s úplnou spolehlivostí, ale jsou kladeny požadavky na nízké latence doručení aktuálních dat. Navržené algoritmy navíc umožňují stanovit priority sdíleným datům a tak efektivně zvýšit šanci jejich včasného doručení.

Shrnutí přínosů pro praxi

Navržený protokol umožňuje sdílet data mezi grafickými aplikacemi bezpečněji, spolehlivěji a hlavně efektivněji než původní protokol Verse. Při programování nových Verse klientů je možné efektivně využít vícevláknové charakteru knihovny a vícevláknová implementace Verse serveru umožňuje lepší škálování.

Další práce a experimenty

Další práce by měla spočívat především v dokončení implementace celé specifikace a opravení chybné implementace DTLS v knihovně OpenSSL, která znemožňuje praktické nasazení zabezpečeného přenosu v praxi. Plnohodnotná implementace DTLS protokolu by měla být následně otestována především s ohledem na velké datové toky a zatížení Verse serveru. Další rozšíření specifikace by se mělo týkat především Congestion Control v kombinaci s dohadováním o FPS, protože posílání paketů v pravidelných intervalech odpovídajících FPS Verse klienta se jeví jako další možný způsob jak zefektivnit přenos dat.

Další rozšíření specifikace by se mělo týkat pravidel sdílení dat na serveru pomocí DED. Grafické aplikace by měly mít jednak možnost definovat si vlastní pravidla, která jsou jim ušita na míru a zároveň by mělo být vytvořena sada pravidel, která by umožňovala sdílet data mezi různými grafickými aplikacemi. V neposlední řadě by měla být provedena opětovná implementace nového protokolu Verse do programu Blender.

Reference

Vlastní publikace

[1] Hnidek, J. Network Protocols for Applications of Shared Virtual Reality. 19th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision Communication Papers Proceedings (2011).

[2] Hnidek, J. Resend Mechanism for Reliable Datagram Protocol. Annual Edition of the International Technology. Education and Development Conference (INTED) (2009).

[3] Hnidek, J. Introduction of New Verse Protocol. Interantional Blender Conference, Stichting Blender Foundation, Amsterdam, the Netherlands, October 2010.

[4] Hnidek, J. Integration of Verse protocol to Blender. Interantional Blender Conference, Blender Foundation, Amsterdam, the Netherlands, October 2005.

Použitá literatura

[5] Holzmann, G. J. Design and validation of computer protocols. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1991.

[6] Keshav, S. An Engineering Approach to Computer Networking: ATM Networks, the Internet, and the Telephone Network. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[7] Bennett, J. C. R., Partridge, C., and Shectman, N. Packet reordering is not pathological network behavior. IEEE/ACM Trans. Netw. 7, 6 (December 1999), 789–798.

[8] Boulanger, J.-S., Kienzle, J., and Verbrugge, C. Comparing interest management algorithms for massively multiplayer games. In Proceedings of 5th ACM SIGCOMM workshop on Network and system support for games (New York, NY, USA, 2006), NetGames '06, ACM.

[9] Brink, E., Steenberg, E., and Svensson, G. The Verse Networked 3D Graphics Platform. In SIGRAD 2006, The Annual SIGRAD Conference (Linkping, Sweden, 2006), H. Gustavsson, Ed., SIGRAD, pp. 44–48.

[10] Jacobson, V. Congestion avoidance and control. SIGCOMM Comput. Commun. Rev. 18, 4 (August 1988), 314–329.

[11] Cruz-Neira, C., Sandin, D. J., DeFanti, T. A., Kenyon, R. V., and Hart, J. C. The CAVE: audio visual experience automatic virtual environment. Commun. ACM 35, 6 (June 1992), 64–72.

[12] Postel, J. RFC 768: User Datagram Protocol. RFC 768, IETF, aug 1980. http://www.ietf.org/rfc/rfc768.txt.

[13] Postel, J. RFC 793: Transmission Control Protocol. RFC 793, IETF, sep 1981. http://www.ietf.org/rfc/rfc793.txt, Updated by RFCs 1122, 3168.

[14] Kohler, E., Handley, M., and Floyd, S. RFC 4340: Datagram Congestion Control Protocol (DCCP). RFC 4340, IETF, mar 2006. http://www.ietf.org/rfc/rfc4340.txt, Updated by RFCs 5595, 5596.

[15] Rescorla, E., and Modadugu, N. Datagram Transport Layer Security. RFC 4347, IETF, apr 2006. http://www.ietf.org/rfc/rfc4347. txt, Updated by RFC 5746.

[16] Rescorla, E., and Modadugu, N. Datagram Transport Layer Security. RFC 4347, IETF, apr 2006. http://www.ietf.org/rfc/rfc4347.txt, Updated by RFC 5746.

Poznámka: tento zkrácený seznam obsahuje pouze publikace důležité vzhledem k obsahu autoreferátu. Kompletní seznam literatury je součástí disertační práce.