

Katedra informatiky
Přírodovědecká fakulta
Univerzita Palackého v Olomouci

DIPLOMOVÁ PRÁCE

Metody optimalizace přenosu dat v distribuovaném
systému



2026

Vedoucí práce:
Mgr. Tomáš Urbanec, Ph.D.

Martin Šlachta

Studijní program: Aplikovaná informatika,
Specializace: Vývoj software

Bibliografické údaje

Autor:	Martin Šlachta
Název práce:	Metody optimalizace přenosu dat v distribuovaném systému
Typ práce:	diplomová práce
Pracoviště:	Katedra informatiky, Přírodovědecká fakulta, Univerzita Palackého v Olomouci
Rok obhajoby:	2026
Studijní program:	Aplikovaná informatika, Specializace: Vývoj software
Vedoucí práce:	Mgr. Tomáš Urbanec, Ph.D.
Počet stran:	54
Přílohy:	elektronická data v úložišti katedry informatiky
Jazyk práce:	český

Bibliographic info

Author:	Martin Šlachta
Title:	Methods of data transfer optimizations in distributed systems
Thesis type:	master thesis
Department:	Department of Computer Science, Faculty of Science, Palacký University Olomouc
Year of defense:	2026
Study program:	Applied Computer Science, Specialization: Software Development
Supervisor:	Mgr. Tomáš Urbanec, Ph.D.
Page count:	54
Supplements:	electronic data in the storage of department of computer science
Thesis language:	Czech

Anotace

Ukázkový text závěrečné práce na Katedře informatiky Přírodovědecké fakulty Univerzity Palackého v Olomouci, který je zároveň dokumentací stylu pro text práce v $\text{\LaTeX}$. Zdrojový text v $\text{\LaTeX}$ je doporučeno použít jako šablonu pro text skutečné závěrečné práce studenta.

Synopsis

Sample text of thesis at the Department of Computer Science, Faculty of Science, Palacký University Olomouc and, at the same time, documentation of the $\text{\LaTeX}$ style for the text. The source text in $\text{\LaTeX}$ is recommended to be used as a template for real student's thesis text.

Klíčová slova: styl textu; závěrečná práce; dokumentace; ukázkový text

Keywords: text style; thesis; documentation; sample text

Děkuji, děkuji, děkuji.

Odevzdáním tohoto textu jeho autor/ka místopřísežně prohlašuje, že celou práci včetně příloh vypracoval/a samostatně a za použití pouze zdrojů citovaných v textu práce a uvedených v seznamu literatury.

Obsah

1	Úvod	9
2	Distribuované systémy	10
2.1	Počítačové sítě	10
2.2	Komunikace v síti	11
2.3	Rodina protokolů TCP/IP	11
2.4	Protokoly v aplikační vrstvě	12
2.4.1	Middleware	13
2.4.2	HTTP	13
2.4.3	HTTP/3	13
2.4.4	gRPC	14
2.4.5	Internet protocol	14
2.5	Protokoly v transportní vrstvě	14
2.5.1	TCP	14
2.5.2	Spolehlivost	14
2.5.3	UDP	15
2.5.4	QUIC	15
2.6	Asynchroní IO	17
2.6.1	ZeroMQ	17
2.6.2	Work stealing	17
2.6.3	Async/Await	17
2.6.4	Boost Asio	18
2.6.5	Stack pointer	18
2.7	Architektury	20
2.7.1	Klient-Server	20
2.7.2	Peer-to-Peer	20
3	Hra pro více hráčů	20
3.1	Hry	20
3.1.1	Entt	21
3.2	Svět	22
3.2.1	Ovladač pro hru jednoho hráče	22
3.2.2	Ovladač pro hru více hráčů	22
3.3	Klient-Server architektura	22
3.4	Posílání zpráv	23
3.4.1	Serializace	23
3.4.2	Kódování zpráv	25
3.4.3	Implementace	25
3.4.4	Detekce a náprava chyb	25
3.5	Server	26
3.5.1	Registr spojení	26
3.5.2	Replikátor	26
3.5.3	Správa zájmů	26

3.6	Klient	27
3.6.1	Replicator controller	27
3.6.2	Interpolace	27
3.6.3	Rollback a simulace vlastního hráče	27
3.7	Metriky	28
3.7.1	TimescaleDB	28
3.7.2	Grafana	28
3.7.3	ImGui	28
3.7.4	Tracy	29
3.7.5	Konkrétní metriky	29
3.7.6	Sbírání v reálném čase	29
3.8	Protokol QUICr	29
3.8.1	Frame	30
3.8.2	Handshake	31
3.8.3	Spolehlivost	32
3.8.4	Enkodér	33
3.8.5	Testování	33
3.8.6	Měření	33
3.9	Optimalizace replikátoru	34
3.10	Optimalizace manažera zájmů	38
3.11	Optimalizace QUICr	38
3.11.1	Šifrování	38
3.11.2	SPR-6	38
3.11.3	Lockstep	39
3.11.4	Congestion control	39
3.11.5	Bandwidth control	39
3.12	Sharding	39
3.13	Komprese	39
3.13.1	Bit Buffer	39
3.13.2	Komprese QUICr	39
3.13.3	Komprese protokolu replikátoru	39
4	Styly pro psaní bakalářských a diplomových prací	40
4.1	Požadavky a podprovaná prostředí	40
4.2	Přepínače	40
4.3	Geometrie stránky	43
5	Sazba částí dokumentu	43
5.1	Sazba úvodní strany či obsahu	43
5.2	Závěry	43
5.3	Matematika	43
5.4	Sazba literatury	44
5.4.1	Sazba bibliografie přes BIB _Λ TEX	44
5.4.2	Manuální sazba bibliografie	44
5.5	Drobná makra	44

5.6	Sazba rejstříku	45
5.7	Sazba zdrojových kódů	45
Závěr		49
Conclusions		50
A První příloha		51
B Druhá příloha		51
C Obsah elektronických dat		51
Seznam zkratk		53
Literatura		54
Rejstřík		54

Seznam tabulek

1	Rozhraní třídy QuicrEndpoint	30
2	Seznam přepínačů	41
3	Odstavce v tabulkách	46

1 Úvod

2 Distribuované systémy

Počítačový systém se skládá ze služeb, každá implementována jako kolekce procesů. Pokud jsou počítače propojeny v síti a procesy systémů jsou rozprostřeny přes více komunikujících počítačů, dostaneme „síťový systém“. Ty mohou mít různé topologie podle toho, jak v systému komunikují. Příkladem takového systému je World Wide Web, který poskytuje dokumenty podle adresy „URL“, tedy Unique Resource Location.

Distribuovaný systém se snaží skrýt svou distribuovanost pomocí „middleware“. Ten slouží jako vrstva mezi sítí a problémy s ní spjaté a samotnou aplikací. Představíme dva přístupy: Message Oriented Middleware (MOM) a Remote Procedure Call (RPC).

RPC je přirozenější přístup, který umožňuje zavolat proceduru lokálně, která se zabalí do zprávy a odešle na jiný počítač. Ten zprávu přečte a podle instrukcí začne vykonávat. Výsledek pak odešle zpět jako zprávu. Odesílatel tedy implementačně nemusí vědět, že se vykonala na jiném počítači. Problémem tohoto přístupu je jednak to, že strana která odesílá zprávu musí znát adresu druhé strany. Druhá nevýhoda je, že obě strany musí běžet současně.

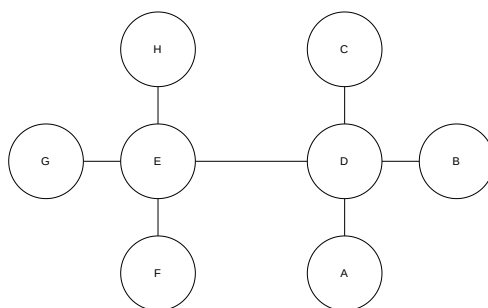
Druhým přístupem je posílání zpráv. V tomto přístupu aplikace posílají zprávy na čistě logický cíl. Např. identifikovaný typem zprávy. Aplikace pak může projevit zájem o různé typy zpráv a middleware se postará o to, že tyto zprávy aplikace dostane. Přístupu se říká „publish-subscribe“.

2.1 Počítačové sítě

Nedílnou součástí distribuovaného systému je síť, přes kterou je schopný komunikovat. Počítačová síť je skupina propojených počítačů, které si mezi sebou přenášejí data. Například lokální síť (LAN) propojují až tisíce počítačů, které jsou geograficky blízko, například v rámci jedné budovy. Rozsáhlé sítě (WAN) propojují miliony různých zařízení po celém světě. Příkladem je síť Internet. Můžeme díky ní snadno posílat zprávy z jednoho kontinentu na druhý. Kvůli této rozmanitosti musí síť podporovat různá přenosová média, jako měděný kabel pro rychlé ale krátké vzdálenosti, nebo bezdrátové připojení pomocí satelitu pro pomalejší, ale na delší vzdálenost apod.

Počítačovou síť si lze představit jako graf, ve kterém počítače představují vrcholy a fyzická média mezi nimi jsou hrany. Vrcholy dále dělíme na „koncové body“ a „propojovací prvky“. Koncové body jsou počítače, mobilní telefony a jiná zařízení, na kterých běží procesy. Dva procesy na dvou různých počítačích si mezi sebou mohou posílat zprávy. Propojovací prvky jsou přepínače, rozbočovače a opakovače. Ty naopak slouží pouze k směřování zpráv po síti. Využití je propojení skupiny počítačů do sítě přes jedno spojení jako vidíme na obrázku 1. Propojovací body jsou E a D. Není potřeba připojovat každý počítač s každým.

Dva počítače, přímo propojené fyzickým médiem, komunikují posíláním n-tic bytů zvané „rámce“. Variantou jsou „přepínané“ sítě, které nemusejí mít mezi všemi dvojicemi počítačů přímé spojení, ale využívají „přepínače“. Proces, kdy



Obrázek 1: Příklad využití propojovacích bodů

doručujeme zprávu na cílový počítač přes vícero přepínačů se nazývá „směrování“. K tomu je třeba, aby každé zařízení mělo přiřazenou unikátní adresu a posílali se formátované rámce zvané „pakety“. Ty se skládají z hlavičky, ve které najdeme informace ke směrování, a tělu obsahujícím samotnou přenášenou zprávu.

Do přepínače můžeme připojovat počítače. Na obrázku TODO vidíme, jak tato topologie vypadá, říká se jí hvězda. Tato skupina propojených zařízení přes přepínače tvoří síť. Dva počítače v ní mezi sebou nemusejí mít přímé spojení, protože si pakety posílají přes přepínač, který ho nasměruje na správný počítač. Tato skupina počítačů propojených přepínačem tvoří síť. Můžeme propojovat i přepínače a vytvořit tak síť sítí. Bežným příkladem je Internet, největší síť sítí na světě.

2.2 Komunikace v síti

V této části představíme jak probíhá komunikace v síti. Aby si dva počítače navzájem rozuměly, používají tzv. „komunikační protokol“: soubor pravidel pro výměnu informací mezi počítači. Definují syntaxi, sémantiku a synchronizaci zpráv. Mají různé charakteristiky, např. jestli potřebují navázat spojení, které obě strany odsouhlasí, aby mohla začít komunikace mezi procesy. Další vlastností je způsob řešení spolehlivosti doručení rámců.

Nejčastěji se strany domlouvají na tajném klíči pro symetrické šifrování zpráv mezi sebou tak, aby nemohli být odposlouchávány. Dále si mohou vyměnit informace o verzi protokolu a omezeních, jako rychlost přenosu apod. Pokud protokol nevyžaduje navázání spojení, může přes něj na proces přijít zpráva z libovolného počítače bez předešlého souhlasu. Protokol často definuje, co se má v takových situacích stát.

2.3 Rodina protokolů TCP/IP

Představíme rodinu protokolů pro komunikaci v síti Internet, která se nazývá TCP/IP. Jedná se o celou soustavu protokolů, které fungují ruku v ruce jako vrstvy. Název se skládá ze dvou důležitých protokolů: IP (Internet Protocol) a

TCP (Transmission Control Protocol). IP slouží pro směrování paketů a TCP pro řízení přenosu. IP umožňuje komunikaci libovolných dvou uzlů počítačů v propojených sítích. TCP zajišťuje spolehlivý obousměrný přenos dat mezi procesy na dvou počítačích (ne nutně různých).

Rodina je tvořena vrstvami. Vrstva čte a zapisuje do vrstvy pod ní. Konkrétně architektura TCP/IP je rozdělena do čtyř vrstev: aplikační, transportní, síťová a síťové rozhraní. Každá vrstva obsahuje množinu protokolů a pro každou situaci lze sestavit ideální čtveřice. Vrstvy podrobněji představíme.

Aplikační

Nejvyšší je „aplikační“ vrstva, ve které jsou aplikace. Příkladem protokolů jsou HTTP nebo gRPC, které spadají do kategorie tzv. „middleware“. To znamená, že protokol není nutně specifický pro konkrétní aplikaci. Narozdíl od protokolu pro hru World of Warcraft nebo SMTP.

Transportní

Druhá vrstva je „transportní“. Tato vrstva je rozbalena až na koncových bodech, jak vidíme na obrázku 2, nikoliv po cestě. Není totiž důležitá pro směrování paketů. Stará se o to, jaké pakety přišli, v jakém pořadí a jestli nejsou poškozené. Příkladem protokolů jsou TCP, UDP nebo QUIC.

Síťová

Pro adresaci máme vrstvu síťovou. Ta směruje a předává jednotlivé datagramy. Příkladem protokolů jsou IP, ARP, RARP nebo IPSEC. Je nutné, aby stejně jako vrstva síťového rozhraní, byla implementována ve všech vrcholech na cestě mezi dvěma počítači, které chtějí v internetu komunikovat.

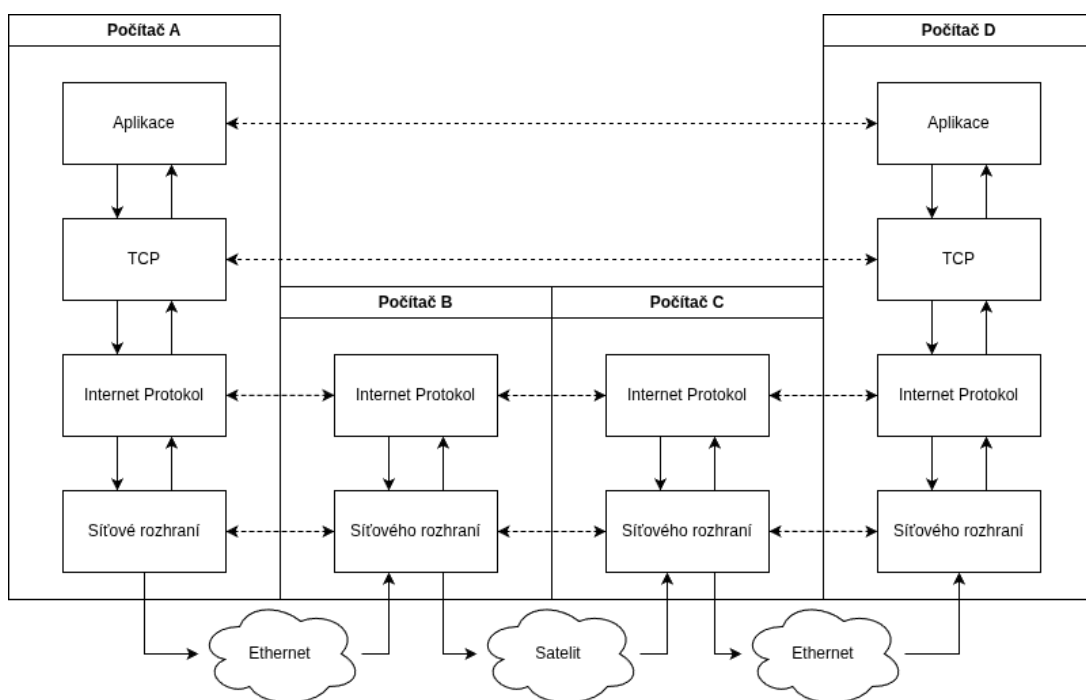
Síťové rozhraní

Nejnižší vrstva, která se stará o přenos přes fyzické médium. Pro různé média bude vyžadovat různé protokoly. Příkladem jsou Ethernet, Token ring, apod.

Na obrázku 2 vidíme, jak mezi sebou jednotlivé vrstvy komunikují. Aplikace na počítači A odešle zprávu, kterou počítač D uvidí ve své instanci aplikace. Vrstva zná pouze vrstvu pod sebou. Plnou čarou vidíme datový tok zprávy, jak putuje přes jednotlivé vrstvy. Šrafovaná čára reprezentuje abstrahovaný datový tok tak, jak ho vidí vrstva. Počítače B a C jsou propojovací body. Všimněme si, že se nepoužívají transportní nebo aplikační vrstvy, pouze si rozbalí IP paket a zjistí, kam mají pakety posílat dál. Vrstva síťového rozhraní pracuje s různými médii, jak také vidíme na obrázku.

2.4 Protokoly v aplikační vrstvě

Nejvyšší v rodině protokolů TCP/IP je aplikační vrstva s protokoly jako HTTP, gRPC nebo SMTP. Aplikačním protokolům, které ale nejsou specifické pro konkrétní aplikaci, se říká middleware. Mezi takové řadíme právě HTTP nebo gRPC,



Obrázek 2: Komunikace vrstev TCP/IP

kteřé jsou pro obecné posílání zpráv. V aplikační vrstvě děláme autentizaci a autorizaci. Další příkladem protokolu je API pro službu STAG.

2.4.1 Middleware

Speciálním případem protokolů v aplikační vrstvě je „middleware“. Je to takový protokol, který není specifický pro žádnou konkrétní aplikaci. Příkladem jsou protokoly gRPC, HTTP, OAuth apod. Existuje celá řada různých aplikací, které je mohou využívat právě díky své obecnosti. Typické jsou různé typy rozhraní pro posílání zpráv. Pokročilejším případem jsou autentizační a autorizační middleware. Autentizace je ověření, že klient je opravdu ten, za koho se vydává. Autorizace je ověření, že klient má právo na to, co se snaží udělat. Je jasné, že tato problematika bude trápit více různých aplikací s ne nutně jinak společnou částí.

2.4.2 HTTP

2.4.3 HTTP/3

Poměrně zajímavou novinkou je HTTP/3, která běží na protokolu QUIC. Jinak je podobný protokolu HTTP/2.

2.4.4 gRPC

2.4.5 Internet protocol

V síťové vrstvě, která se stará o směrování zpráv, se v síti Internet používá tzv. Internet Protocol.

2.5 Protokoly v transportní vrstvě

Nejčastější protokoly transportní vrstvy jsou TCP a UDP. V roce 2012 představila společnost Google ještě protokol QUIC, který využívá UDP. Tyto protokoly si představíme, protože jejich vlastností budeme využívat při optimalizacích.

2.5.1 TCP

Nejběžnější protokol je TCP. Při použití mezi sebou dva koncové body vytvoří spojení, přes které mohou posílat zprávy obousměrně. Protokol garantuje spolehlivé doručení dat v pořadí, ve kterém byla odeslány. Zároveň za Vás řeší zahlcení sítě. Internet protokol říká, že propojovací prvky mohou v případě, že jsou zahlcené, pakety zahodit. Příkladem může být omezení rychlosti stahování poskytovatelem internetového připojení. Pokud platíte za 300Mb/s a začnou Vám chodit data rychlostí 500Mb/s, přebytečné pakety se prostě zahodí.

Spojení v TCP musí odsouhlasit obě strany. Jedna strana se připojuje a druhá pomocí volání C++accept spojení přijme.

Pokud se na náš socket někdo pokusí připojit, operační systém vykoná tzv. handshake a přidá nové připojení do fronty. Z fronty můžeme vytahovat právě pomocí 'accept'. Zjistit, jestli ve frontě už nějaké připojení máme, můžeme pomocí funkce 'listen', která blokuje, dokud nějaké připojení nepříbyde.

Nové přijaté připojení bude mít svůj vlastní socket s vlastním portem. Když zavoláme 'write' a do argumentu dáme ukazatel na data a jejich délku, operační systém je odešle na druhou stranu, která s námi navázala spojení. Zaručí to, že na druhou stranu dojdou všechna data, nebo žádná. Druhá strana může pomocí 'read' do alokované paměti přečíst to, co mu do socketu přišlo. Je nutné si uvědomit, že pokud zavoláme 'write' pro data o délce řekněme 1000 bytů, tak druhá strana nemusí dostat všechna data pomocí jednoho 'read'. Tento příkaz vrací počet bytů, které přečetl. Pokud tak přečte méně, než očekáváme, musíme provést volání znova, dokud nepřečteme vše co očekáváme. Mohlo by nás zarazit: jak může druhá strana vědět, kolik má očekávat dat? Pro to slouží tzv. komunikační protokol, který rozeberu v kapitole...

2.5.2 Spolehlivost

Popíšeme si, jak TCP zajišťuje spolehlivost pomocí ACK zpráv. Pokaždé, co dostane packet, odešle druhému koncovému bodu zprávu ACK s číslem packetu. V případě, že odesílatel ACK, neboli potvrzení o přijetí, nedostane, odešle packet znova. Příjemce si u sebe postupně skládá seřazený proud bytů. Jakmile je na

socketu seřazená posloupnost bytů, umožní nám ji přečíst pomocí ‘read’. Pokud odešle jedna strana pakety A, B a C, a příjemci přijde nejprve C a B, ‘read’ nebude nic vracet, dokud nepříjde i packet A. Zmíním konkrétní případ, kdy to nemusí být žádoucí: Představme si, že A tvoří jeden příkaz a C a B tvoří druhý příkaz. Oba příkazy jsou na sobě nezávislé a příjemce je dokáže vykonat v libovolném pořadí. Stále potřebujeme spolehlivost doručení, ale stačí nám jen částečné uspořádání.

Mezi aplikační protokoly využívající TCP patří HTTP/2, email, nebo SSH.

Mezi nevýhodu bych zařadil to, že přenáší proud dat. Jinými slovy, v protokolu není proud rozdělen na jednotlivé, řekněme, "zprávy". Tuto logiku si musíme implementovat sami. Později si ukážeme, jak toho lze docílit.

2.5.3 UDP

Méně spolehlivá alternativa je UDP, neboli Unreliable Datagram Protocol. Ten komunikuje pomocí tzv. datagramů, na rozdíl od proudu bytů, jako je tomu v případě TCP. Znamená to, že jedním systémovým voláním ‘write’ odešleme právě jeden datagram, a na příjemci právě jedním ‘read’ přečteme celý datagram.

Zde již nebudeme volat ‘listen’ ani ‘accept’. Výhoda je, že pokud zavoláme ‘write’ a odešleme data o velikosti 100 bytů, je zaručeno, že druhá strana dostane celý tento blok právě jedním zavoláním ‘read’, ve kterém nebude nic dalšího. UDP totiž rozděluje data na datagramy, které v podstatě reprezentují pakety. To usnadňuje práci s komunikačním protokolem. Protokol je nespolehlivý: datagramy mohou dorazit v jiném pořadí nebo vůbec. Jsou ale způsoby, jak spolehlivost, alespoň částečnou, implementovat.

2.5.4 QUIC

QUIC je spolehlivý protokol transportní vrstvy od společnosti Google. Jeho cílem je být rychlejší varianta TCP, která využívá protokol UDP. Mezi dvěma koncovými body vytváří spolehlivé spojení. Na rozdíl od TCP umožňuje otevřít více proudů dat v jednom spojení a modelovat tak částečné uspořádání. Když se čeká na paket v TCP, tak se pozastaví celý tok dat, ale v případě QUIC se zastaví jen konkrétní proud.

V protokolu se nachází pakety, rámce a čísla koncových bodů. Paketů máme více druhů a liší se především hlavičkou, ale také typem šifrování a její silou. Pro otevření spojení odešle klient serveru paket typu Initial, který má v hlavičce číslo lokálního spojení (LCID - Local connection ID) a číslo cílového spojení (DCID - Destination connection ID). Když server obdrží paket typu Initial, s číslem cílového spojení takovým, které nemá v registru, vytvoří úplně nové spojení s jiným číslem. Zpět klientovi pošle opět paket typu Initial a v hlavičce bude jako lokální číslo spojení mít nově vygenerované ID a jako cílové bude mít to, co klient poslal jako lokální. Touto výměnou se dva koncové body shodnou na svých číslech. Podotknu že čísla na obou koncích se liší, a prvotní cílové ID, které posílá klient, tak server nepoužívá. Jeho využití zmíním později.

Paket v sobě obsahuje různé rámce, každý nějakého typu. Mezi typy patří třeba: ACK, CRYPTO, PADDING, PING, STREAM apod. Postupně je rozeberu. Rámec typu ACK funguje podobně jako v TCP a obsahuje čísla paketů, které chce druhé straně oznámit jako zpracované. Tento rámec najdeme právě v paketu typu Initial od serveru na klienta, kdy server chce oznámit klientovi, že paket přijal a zpracoval. Druhý rámec který už při handshake uvidíme je CRYPTO. Ten je v Initial paketu a obsahuje klient secret resp. server secret. Server je v moment, kdy mu přijde první Initial paket odvodit tzv. session key z klient secret a svého vygenerovaného server secret. Na klienta odešle paket Initial s CRYPTO rámcem obsahujícím server secret. Hned za ním odešle Handshake paket, který už je zašifrovaný pomocí session klíče a obsahuje certifikát serveru. Jakmile klient obdrží od serveru Initial a Handshake, může začít bezpečná komunikace. Druhá strana musí na všechny zprávy odpovědět pomocí ACK rámce.

QUIC se tak shodl na tajném klíči i navázal spojení dvěma roundtripy, namísto čtyřmi, jako je tomu klasicky u TCP s TLS. Důvod je, že pakety může spojovat do jednoho datagramu. Takže server odpovídá na první Initial paket dvěma pakety: Initial a Handshake, ale v jednom datagramu. Dalším typem paketu, který už v tento moment může použít, jsou RTT-0 a RTT-1. RTT-0 lze poslat už v datagramu, ve kterém klient posílá na server Initial, a může obsahovat STREAM rámce. To znamená, že obsahuje užitečná aplikační data. Pokud nám tak v datagramu zbývá místo, nebo nechceme čekat na odpověď, můžeme okamžitě poslat RTT-0. Tento paket je sice šifrovaný, ale slabě a není odolný proti replay útoku, kdy útočník paket odposlechne a pošle znova. RTT-1 už je plně šifrovaný pomocí session klíče a nelze použít replay útok.

Jak už jsem zmínil, užitečná aplikační data jsou ve STREAM rámcích. Pro začátek proudu ho musí jedna ze stran otevřít pomocí specifického rámce. Kdykoliv ho pak může zase zavřít. Dokonce je možné v jednom datagramu proud otevřít, poslat data a zase zavřít. Vidíme, že lze snadno modelovat částečné uspořádání a snížit blokování, které třeba v TCP běžně nastává.

Čísla spojení, které mají dva koncové body spojení vybrané, se využívají pro identifikaci odesílatele. Normálně by TCP využilo adresu z IP hlavičky, ale ta se může třeba při přepnutí z mobilních dat na Wi-Fi, změnit. V ten moment nastává zdoluhavý TCP timeout, kdy se socket nakonec zavře a aplikace se pokusí navázat spojení znovu. V případě QUIC je přepnutí okamžité, protože druhý koncový bod se podívá jen na číslo cílového spojení v hlavičce a ví, od koho mu datagram přišel.

Jak QUIC řeší spolehlivost už jsem trochu vysvětlil při handshake. Obecně používá ACK rámce, které posílá druhé straně, a který obsahuje úseky čísel paketů, které obdržel. Číslo paketu je v hlavičce a je postupně se inkrementující. Pokud druhá strana neobdrží ACK rámec, například pro paket n, pak odešle identický paket n, ale s novým číslem, které mu přiřadí čítač. Pokud druhá strana ACK odeslala, ale ten zrovna nepřišel, musí nějak řešit duplicity. Každý STREAM rámec obsahuje úsek dat proudu, který obsahuje, například od x do y. Díky tomu může koncový bod zjistit, že tuto část proudu už má a rámec zahodit.

2.6 Asynchroní IO

Čtení a zápis na socket je běžně blokující operace. To znamená, že při zavolání ‘read’ pro čtení 10 bytů, bude volání blokovat, dokud na socket nepříjde 10 bytů. V případě serveru pro online hru je to nevhodné, protože pokud nám nic nepřišlo, můžeme mezitím vykonávat ostatní logiku hry. Socket lze nastavit jako neblokující tak, že pokud zavolám ‘read’ a budu chtít číst 10 bytů, a ty tam ještě nebudou, vrátí mi ty co zatím přečetl (pokud přišlo jen 5 bytů, vrátí mi jen 5 bytů). Tuto menší část si můžu u klienta ukládat do bufferu a postupně zkoušet dekodovat jako zprávu.

Podobnému přístupu se říká **asynchronní sockety**. Jinými slovy to znamená oddělení volání a samotného vykonání. V tomto případě sice zavoláme ‘read’, ale toto volání by se uložilo někam do fronty, a jiné samostatné vlákno by ho mohlo vykonat později. Protějšek z reálného světa je hodit dopis do schránky a pokračovat ve svém klasickém programu dne. Doručení se stane v nejbližších dnech a to někým jiným, takže my už se o dopis nemusíme starat. Pro lepší představu si pár existujících implementací představíme:

2.6.1 ZeroMQ

Při použití knihovny ZeroMQ vytváříme objekt ‘ctx = zmq_context(num_threads)’ . Ten vytvoří námi specifikovaný počet vláken ‘num_thread’. Když následně zavoláme ‘zmq_send(ctx, x)’, protějšek klasického ‘send(x)’, tak nedojde k blokování. Místo toho se zpráva ‘x’ umístí do fronty, ze které pak vlákna z ‘zmq_context’ tzv. "kradou"(work-stealing), a odesílají v pozadí.

To samé platí o ‘zmq_receive’, které si v pozadí bufferuje celou zprávu. TCP totiž nemá žádnou strukturu zprávy, a proto když zavoláme ‘recv’, můžeme dostat jen část. ZeroMQ místo toho v pozadí bufferuje, dokud neobdrží celou zprávu, a až poté vrací ‘true’.

2.6.2 Work stealing

Běžnou strategií pro dynamický multithreading, je tzv. work stealing. Jiné strategie jsou třeba work sharing. Implementuje se thread-safe fronta. Při asynchroním volání úkolu se vloží do této fronty. Následně máme X vláken, které se ve smyčce dotazují na frontu. Pokud v ní něco je, vytáhnou to a začnou na tom pracovat. Těmto vláknům se někdy říká "Worker". Většinou jich je právě tolik, kolik je fyzických jader v procesoru, protože cílem je paralelismus, a ten s vyšším počtem vláken nezlepšíme. Zajímavou vlastností je, že součástí úkolu může být právě vložení dalších úkolů do fronty.

2.6.3 Async/Await

Populární implementací, kterou vidíme například v Rust nebo C#, je async/await. Async už jsme si vysvětlili. Await slouží pro volání asynchroních funkcí. Místo ‘queue.push(make_task(...))’ tak úkol může být právě volání funkce. Práce je tak

daleko intuitivnější. Zde pozor, že pouhým voláním `await` na asynchroní funkci v C# nebo Rust paralerismu nedocílíme! Na to je třeba v jeden moment `await` více úkolů.

Příklad z reálného světa je poslat dopis s otázkou. Opět stačí hodit dopis do schránky. Až si ho eventuálně někdo přečte, může nám poslat odpověď. Když dostaneme odpověď, vzpomeneme si, kde jsme skončili, když jsem práci přerušili a odeslali dopis s otázkou, a když už máme odpověď, tak pokračujeme.

2.6.4 Boost Asio

Velmi podobná ZeroMQ je Boost ASIO. Ta ale nechává vytváření vláken na nás. Pouze vytvoříme objekt `asio_context`, který má metodu `run`. Když z vlákna zavoláme metodu `run`, stane se vlákno workerem, který krade práci z fronty, která je taky `context`.

Tento přístup je intuitivnější, protože se jedná přesně o to, jak worker thready fungují. Pouze spuštěná funkce, která se opakovaně dotazuje na frontu. Pokud něco najde, vytáhne to, a začne na tom pracovat. Jakmile práci dokončí, opět se dotazuje na frontu. Zde si také všimněme, že se nejedná o klasický rychlý context switching, který dodává iluzi více jader v operačních systémech, ale worker opravdu práci nejprve dokončí, než změní kontext a začne pracovat na něčem jiném.

Otázka by mohla být, proč nevytvořit thread pro každou zprávu, kterou chceme odeslat. Důvodem je, že vytváření vláken je sice rychlejší, než celých procesů, ale pořád vyžaduje systémové volání, které může způsobit zpomalení.

Proto se často používají tzv. green thready, fibery, user thready, ... což jsou thready, ale spravované kompletně v user-space. Není třeba žádné systémové volání.

Na začátku se spustí několik workerů, což jsou thready, které právě kradou práci z fronty a vykonávají je. Většinou mají dlouhou životnost.

2.6.5 Stack pointer

Problém může být definovat práci pro thready. Jak vypadá takový objekt `Job`? Intuitivně by nás napadlo, že by obsahoval ukazatel na funkci. Nebo v případě, že máme jen velmi omezený počet různých implementací, může obsahovat jen typ, a worker už si dohledá, co má dělat.

Druhým způsobem, který vidíme při použití `await/async` v jazycích jako Rust nebo C#, je prostě zavolat funkci, která reprezentuje práci, pomocí `await`. Funkce `await` si uloží aktuální ukazatel na stack, vytvoří nový stack speciálně pro job, a objekt `Job` tak vypadá tak, že obsahuje tento nový stack pointer, ukazatel na funkci, a všechny potřebné registry.

Všimněme si, že můžeme job vytvářet i z jiného jobu, a tím v podstatě větvit. Zároveň můžeme kód paralerizovat tak, že najednou budeme `awaitovat` více jobů. Představme si, že máme fyzický systém s metodou `step`. Ta zavolá `awaitAll(jobs)`, kde `jobs` je list jobů. Automaticky se všechny dostanou do fronty, a

různé workery je mohou začít vykonávat paralelně.

2.7 Architektury

Distribuované systémy mají různé architektury. Liší se např. v tom, kdo má v systému jak velké slovo a kdo komunikuje s kým. My zmíníme dvě: klient-server a peer-to-peer.

2.7.1 Klient-Server

První architekturu, kterou zmíníme, je klient-server. Vrcholy v síti jsou rozděleny na

2.7.2 Peer-to-Peer

3 Hra pro více hráčů

Pro zkoušku různých technik jsme se rozhodli vytvořit hru pro více hráčů. Nejprve představíme, jak vypadá architektura hry jednoho hráče. Podporu pro více hráčů jsme implementovali jako rozšiřující vrstvu, která příliš neovlivňuje klasickou architekturu her. Ve hře se jednotlivý hráči uvidí, budou se moct pohybovat a interagovat s vybranými objekty.

Představíme komponenty, které jsme do programu implementovali a proč. Pro komunikaci jsme vytvořili nový protokol QUICr, který je podobný QUIC, ale upouští od spolehlivosti a nabízí tak lepší odezvu.

3.1 Hry

Program hry většinou obsahuje smyčku, kde jedné iteraci říkáme „snímek“. Moderní hry iterují od 60 do 240 snímků za sekundu, někdy více. V našem případě jsme se rozhodli pro jednoduchou 3D hru s vykreslováním 60 snímků za sekundu, která pro naše účely stačí. Zajímají nás různé metody pro optimalizaci her více hráčů. Hra si také udržuje svůj stav, jako entity ve světě, počasí, terén, herní pravidla apod.

Každý snímek nejprve posbírání vstupy od hráče, převede je na jednotlivé akce, podle toho, co pro hru vstupy znamenají, a aplikuje je na stav hry. Například stisknutí tlačítka W převede na akci: pohyb dopředu. Tyto akce budou tvořit derivaci stavu hry. Tu následně aplikuje na aktuální stav hry, čímž ho integrujeme a dostaneme se do dalšího stavu. Takto postupně iterujeme z počátečního stavu hry.

Architektura programů her je specifická tím, že tolik nevyužívá dědičnost a virtuální funkce. V případě milionu entity, pro které voláme funkci 60 krát za sekundu, se i virtualizované volání negativně projeví ve výkonu. Druhý důvod je čistě praktický. Pokud chceme mít obrovský svět s různými postavami, kde každá má různé schopnosti a vlastnosti, špatně by se nám modeloval strom dědičnosti. Daleko lépe se různé entity modelují skládáním jejich vlastností.

<https://cowboyprogramming.com/2007/01/05/evolve-your-heirachy/>

Logika vlastností a schopností, které různé entity mohou mít, se rozdělí do tzv. „komponent“. Příkladem je komponenta pro obchodníka, stráž, kouzelníka, truhlu, dialog apod. Ty skládáme do tzv. „entit“, která je pouhým identifikátor se svým seznamem komponent. Tento styl je podobný kompozici objektů, ale s tím rozdílem, že komponenty skládáme do entity až za běhu programu. Nedefinujeme třídu pro každý typ, který může ve hře být.

Seznam všech entit a jejich komponent je stav hry. Pro integraci stavu z derivace máme tzv. „systémy“. Ty iterují přes seznam všech komponent konkrétního typu a provádí jejich transformaci. Tomuto přístupu se říká Entity, Component, System, zkráceně ECS.

3.1.1 Entt

Pro ECS využíváme knihovnu Entt. Příkladem her, které ji využívají taky je třeba Minecraft pro Windows. Nejprve vytvoříme registr, což je objekt typu ‘entt::registry’, který spravuje své entity a jejich komponenty. Entita je v tomto případě pouze unikátně identifikační číslo, komponenta je libovolný typ a systém je lambda. Pro vytvoření systému musíme definovat pohled, tedy n-tici komponent. Tento pohled pak bude iterátor přes všechny entity, které mají každou z těchto komponent. Můžeme pak podle stavu jedné aktualizovat jinou a dělat různé interakce mezi nimi.

```
1  entt::registry registry;
2
3  entt::entity entity = registry.create();
4
5  registry.emplace<Transform>(entity, Transform::from_position(
6      position));
7
8  registry.view<Transform, CharacterBody>()
9      .each([&](Transform& ts, CharacterBody& rb) {
10          auto character = rb.m_character;
11
12          auto transform = character->GetWorldTransform();
13          memcpy(&ts.transform, &transform, sizeof(glm::mat4));
14      });
```

Zdrojový kód 1: Příklad použití knihovny Entt

Představíme komponenty, které jsme pro hru definovali a proč:

Transform

Obsahuje transformační matici. Entity, které chceme zobrazit ve světě, tuto komponentu potřebují. Příkladem entity, která ji mít nebude, je zpráva v chatu.

Mesh

Obsahuje mesh, který má hra využít pro vykreslení entity. Mesh je n-tici bodů, které dohromady dávají 3D povrch objektu. Systém, který entity vykresluje si vytvoří pohled na Mesh a Transform, protože je potřeba vědět kam entitu vykreslit.

Rigidbody

Slouží pro fyzikální informace o entitě, jako hmotnost a sílu, kterou na entitu budeme aplikovat. Systém pro tuto komponentu bude fyzický engine.

3.2 Svět

Stav hry reprezentujeme kompletně ve třídě Svět, neboli 'World'. Obsahuje registr všech entit a jejich komponent. Objekt zvaný ovladač, pak bude sbírat vstupy a stav světa postupně aktualizovat. V případě hry pro jednoho hráče získá ovladač vstupy z klávesnice a myši a vhodně aktualizuju svět. Například při zmáčknutí klávesy W pro posun dopředu, nastavím rychlost Rigidbody entity pro hráče tak, aby při zavolání integrace posunula Transform hráče. Oddělím tak logiku která drží stav a tu která stav mění.

3.2.1 Ovladač pro hru jednoho hráče

Definujeme, jak bude vypadat ovladač světa, tedy vrstva, která transformuje vstupy od hráče na akce ve hře, a podle nich postupně mění stav světa. Pro tento přístup jsme vytvořili 'LocalWorldController'.

3.2.2 Ovladač pro hru více hráčů

V případě hry pro více hráčů jsme vytvořil `\ClientWorldController`. Ve jméně je zmíněný `*Client*`, protože pro peer-to-peer architekturu by ovladač vypadal jinak. Nejedná se tedy o univerzální řešení pro hru více hráčů. Další součástí systému je 'WorldServer'. Ten má v případě klient-server architektury zodpovědnost přijímat vstupy od klientů a odesílat zpátky stav světa. Architektura je podobná té pro hru jednoho hráče, pouze se změnilo kdo integruje.

Protějškem je hra šachů, kterou ovládáme hlasem a někdo jiný za nás táhne. Zvoláme "pěšec XY na ZW", tahač táhne a my vidíme, jak se našim vstupem změnil stav hry. Stejně tak to vidí i druhý hráč. Všimněme si, že druhý hráč vůbec nemusí vědět, co byl náš vstup (ten může dostat derivací), ale stačí mu pouze stav.

3.3 Klient-Server architektura

Hry budeme programovat s architekturou klient-server. To znamená, že hráči budou mít klienta a někde v síti musí být dostupný server, na který se klienti budou připojovat. Server bude mít absolutní autoritu. Klienti budou na server

posílat pouze vstupy, server se k nim bude chovat stejně, jako by se jednalo o hru jednoho hráče a on je přečetl z klávesnice a myši. Server bude na klienty posílat co nejčastěji stav světa. Tento stav se bude postupně měnit. Zodpovědnost programu klienta je tento stav správně zobrazit. Například vykreslit všechny entity na správných pozicích apod.

Alternativou je peer-to-peer architektura, kdy každý uzel je zároveň server i klient, neboli peers. Na rozdíl od klient-server ale nemá nikdo autoritu. Uzly si mezi sebou posílají opět jen vstupy. Hra se simuluje na každém peerovi zvlášť. Zde nastává problém synchronizace, protože simulace se nám mohou snadno rozejít. Musíme tedy definovat, jak zjistit, že se desynchronizace stala a co s ní dělat. Musíme si dát pozor na pseudo-náhodné generátory, aby náhodné prvky hry se vyhodnotili všude stejně. Další problém může nastat v přesnosti čísel s plovoucí desetinou čárkou. Některé procesory mohou udělat operaci jinak a způsobit desynchronizaci.

Detekovat desynchronizaci můžeme tak, že si jednou za čas peers pošlou snapshot svého stavu.

3.4 Posílání zpráv

V této sekci se budu soustředit na middleware pro komunikaci mezi procesy pomocí zpráv. Rozeberu jak lze zprávy kódovat a strukturovaná data serializovat. Komunikovat pomocí zpráv sice neschovává síťovou komunikaci, jako třeba RPC.

3.4.1 Serializace

V systému jsem chtěl zprávu reprezentovat jako instanci konkrétního objektu. To znamená pro zprávu, která říká, že hráč napsal něco do chatu, bude existovat třída `NewChatMessage` a konkrétní hodnoty budou v instanci této třídy. Díky tomu jsem schopný snadno vytvářet zprávy v systému. Alternativou je implementovat dynamický strom, který bych následně kódoval jako JSON. Uzly by byly JSON objekty a listy JSON hodnoty. Podmínkou je, že každý list musí být převeditelný na hodnotu v JSONu a musí mít klíč. Druhý přístup ale zbytečně obchází vlastnosti jazyka C++, jako dědičnost, kontrolu typů (i když ta je v C++ slabá) a další. Na druhou stranu je tento přístup hodně dynamický, proto se hodí například, kdyby schéma zprávy definoval sám uživatel, až po kompilaci programu.

Pokud jsem chtěl posílat instance objektů, musel jsem instance převést na n-tici bytů. Tomuto procesu se říká serializace (resp. deserializace při převedení n-tice bytů na instance objektu). Velmi častým příkladem je serializace do JSON formátu, hojně využívaná v aplikacích s REST API rozhraním, nebo do XML v případě SOAP API rozhraní. To jsou tzv. textové formáty, které jsou pro člověka čitelné. Díky tomu jsou vhodné pro webové aplikace, kde rozhraní musí být srozumitelné pro snadnou práci. Nevýhoda je paměťová náročnost. Například číslo 1 000 000 se v JSON zakóduje do 7 bytů, i když binární reprezentace stejného čísla se vleze do 4 bytů. Rozdíl je směšný, ale téměř dvojnásobný. Druhým kamenem

úrazu je struktura. Každý atribut objektu se v JSONu reprezentuje řetězcem, namísto třeba čísla o 2 bytech, které by unikátně atribut identifikovalo.

Všechna tato snížení prostorové náročnosti řeší binární formáty. Známým příkladem je ProtoBuf, FlatBuffers, Parquet nebo Cap'n'Proto. Postupně je rozeberu jejich specializace. ProtoBuf je volně dostupný protokol od Googlu pro serializaci strukturovaných dat. Pro použití je třeba definovat v univerzálním jazyce to, jakou má zpráva strukturu. Jazyk je podobný definici struktur v jazyce C. Pro přestavu si ukážeme:

```
1 message EntitySpawnMessage {
2     uint32 entity_id = 1;
3     bool is_player = 2;
4     string name = 3;
5 }
```

Zdrojový kód 2: cpp

Vidíme, že místo **struct** píšeme **message** a atributy musí mít unikátní ID. Důvodem je kompatibilita verzí definice. Pokud bychom měli systém o dvou službách, a v jedné bychom přidali atribut 'position', jak vidíme v příkladě, mohla by být druhá služba stále schopná serializovaná data první službou deserializovat.

```
1 message EntitySpawnMessage {
2     uint32 entity_id = 1;
3     Vector3 position = 4;
4     bool is_player = 2;
5     string name = 3;
6 }
```

Zdrojový kód 3: cpp

Stejnou vlastnost má i JSON, kdy unikátní ID je právě řetězec pro klíč. Serializující služba může přidávat atributy, ale deserializaci to neovlivní. Z .proto souboru se vygenerují přímo třídy do potřebného jazyka. V mém případě C++. Výhoda definovat zprávy v univerzálním jazyce je čitelnost, možnost převést do libovolného programovacího jazyka a především reflexe.

FlatBuffers je opět protokol od Googlu, který se vyvíjel pro použití v herních enginech pro hry více hráčů. Jeho výhoda je, že není třeba zprávu "deserializovat". Instanci třídy čte atributy přímo z n-tice bytů. V případě 60 zpráv za vteřinu zprávy od stovek hráčů, je rozdíl mezi deserializací a FlatBuffers znát.

Parquet je formát pro sloupcová data, který umožňuje v serializovaných datech vyhledávat a má menší paměťovou náročnost, takže je vhodný i pro přenos.

Posledním příkladem je Cap'n'Proto, čteno Captain Proto. Je to volně dostupný protokol, který nemá deserializaci, a je tak velmi rychlý. Jeho tvůrce navíc pracoval právě na ProtoBuf.

3.4.2 Kódování zpráv

Při čtení dat ze socketu pomocí TCP protokolu dostáváme proud bytů. Je potřeba z proudu zprávy dekodovat a naopak je být schopni do proudu zakódovat. Proto jsem definoval kódování zpráv a komponentu Messenger. TCP je spolehlivý protokol, proto se málokdy stane, že zprávy nebudou následovat hned po sobě. Pro zakódování zprávy zakóduju nejprve tzv. hlavičku. To je prvních pár bytů, které ukládají informaci o zprávě, jako třeba délku nebo typ. Za hlavičkou následuje serializovaná zpráva. V hlavičce budou první 4 byty tzv. magické číslo, které označuje začátek kódování a slouží k synchronizaci čtecí hlavy v případě, že došlo při deserializaci k chybě. Dalšími 4 byty je n pro n -tici bytů pro serializovanou zprávu.

Při zavolání peek na instanci Messenger, se do bufferu v instanci, který má velikost 64KB, přečte funkcí recv z TCP socketu. Volání není nutné opakovat, protože peek budeme opakovat hned další snímek za pár milisekund. Po zavolání 'recv' skenujeme vstupní buffer a hledáme magické číslo. Jakmile ho najdeme, čteme další 4 byty jako délku n . Pokud v bufferu zbývá ještě alespoň n bytů, vložíme je do fronty zpráv a odebereme z bufferu pro uvolnění místa. Nakonec funkce 'peek' ověří, jestli něco ve frontě je, pokud ano, vrátí typ nejstarší zprávy ve frontě.

Druhou funkcí je 'pop', která má template s argumentem T pro datový typ zprávy. Uživatel instance 'Messenger' musí umět z čísla, které reprezentuje typ, dostat datový typ. To jsem udělal pomocí tzv. template specialization. Uživatel musí zavolat 'peek' a podle vrácené hodnoty zavolat 'pop' se správným template argumentem. Funkce 'pop' zavolá deserializační funkci generovanou ProtoBuf.

3.4.3 Implementace

Bylo potřeba zadefinovat, jak se zpráva bude posílat přes síť. Cílem nebyla univerzálnost a čitelnost, jako je například protokol HTTP, ale optimalizace velikosti. Je třeba definovat, jak zpráva v toku dat vypadá, a jak ji dekodovat.

První čtyři byty obsahují tzv. 'MAGIC' číslo. Může to být libovolná konstanta, protože jediný její význam je synchronizace toho, kde zpráva začíná. Důvod si ukážeme v části o nápravě chyb. Další byte je typ zprávy. Další 2 byty je délka zprávy. Průměrná délka packetu je 1200-1500 bytů, nemá proto smysl posílat delší zprávy. K tomu tento protokol není. Nakonec následuje samotná serializovaná zpráva. Těmto prvním 6 bytům se říká hlavička.

3.4.4 Detekce a náprava chyb

Občas mohou nastat v proudu chyby, kdy zprávu nelze dekodovat, nebo nastane chyba při deserializaci. Proto jsem do kódování implementoval nápravu chyb. Definoval jsem maximální délku bytů, kterou lze přeskočit při hledání magického čísla. To nestačí, pokud jsou data náhodná, protože se může stát, že číslo objeví i tak (i když je pravděpodobnost $1/(256^4)$). Proto si dekodér pro socket počítá

chyby. Pokud se objeví 3 po sobě jdoucí chyby, tak spojení ukončí. Příkladem takové chyby je selhání deserializace nebo ‘recv’ vrátí chybu.

3.5 Server

Rozebereme si jak vypadá program pro server. Ukážeme komponenty, které jsme pro podporu více hráčů museli přidat. Architektura se skládá z replikátoru a manažera zájmu. Stav světa, který server odesílá na klienty, se skládá z entity a komponent, které chceme synchronizovat. Replikátor pro každého hráče zakóduje entity a jejich komponenty a udržuje je synchronizované. Pomocná komponenta je manažer zájmu. Ta říká, které entity se mají synchronizovat kterým. Základní implementace by pro hráče *A* mohla omezit entity, které mu server bude synchronizovat, na ty v maximální vzdálenosti d metrů.

3.5.1 Registr spojení

Vytvořili jsme registr, ve kterém jsou všichni připojení hráči. Tato komponenta slouží jako sifon pro všechny zprávy, které klienti odesílají. Zároveň z ní lze zjistit kdo se právě připojil a kdo odpojil. Třída na to má dvě metody: `\popDisconnectedPlayers` a `\popConnectedPlayers`. Třída si udržuje seznam nových připojení od posledního zavolání `\popConnectedPlayers`.

Také zde udržujeme počet selhání, které u klienta nastali a jakmile počet překročí určitou hranici, např. 5 chyb, relaci s klientem ukončíme.

3.5.2 Replikátor

Systém hry pro více hráčů se snaží mít na všech zařízeních stejný stav. O to se stará replikátor. Pro každého klienta si drží seznam entit, které musí danému klientovi synchronizovat, aby svůj úkol splnil. V případě klient-server architektury odesílá server svůj stav klientům. Naopak klienti posílají pouze své vstupy. Základní implementace by posílala každý snímek kompletní stav entity se všemi komponentami. Lepší varianta je posílat pouze změny. Pokud se například pozice nezmění, není třeba posílat stejnou pozici jako minule. Na druhé straně u klienta máme replicator receiver. Ten se stará o dekodování zpráv a správné aktualizaci komponent entit.

3.5.3 Správa zájmů

Zájem, neboli seznam entit, které se musí klientovi *A* synchronizovat, vybírá komponenta manažer zájmů.

Systém, který se bude starat o to, které entity se musejí klientům synchronizovat, se nazývá **Interest manager**. Nejprve klientovi, podle jeho ID, přiřadíme tzv. delegáta ve světě. Bude to entita s transformací. Pro každého delegáta bude v pravidelných intervalech kontrolovat a přiřazovat do zájmu ty entity, které jsou v dostatečné blízkosti. Nejprve jsem udělal základní ground-truth implementaci,

která funguje na principu brute-force a nevyužívá akceleračních struktur. Jednoduše pro každou jinou entitu pomocí Pythagorovy věty zjistí, jestli je dostatečně blízko.

Dotazu na všechny body, které jsou v určité vzdálenosti, se říká ‘range-query’. Datové struktury, které mi pomohli tento proces zrychlit jsou spatial grid, quad stromy a nebo kd-stromy. Techniky představím a porovnám:

3.6 Klient

V klientské aplikaci mám komponenty svět a ovladač. ClientWorldController, která obstarává vše kolem síťování. Třída World funguje stejně jako na serveru a stará se o klasický běh hry.

3.6.1 Replicator controller

Abychom oddělili to, jak replikátor funguje, vytvořili jsme pro něj na straně klienta ovladač. To nám později umožnilo měnit protokol mezi klientem a serverem podle potřeby.

3.6.2 Interpolace

Když jsme začali systém měřit, zjistili jsme, že až příliš zatěžuje síť. Graf s průměry pro počet hráčů vidíme na grafu. Prvně jsme snížili počet snímků za sekundu, které server na klienty posílá, na 20 za sekundu. To snížilo zátěž na třetinu. Problém ale byl neplynulost hry. Klient sice vykresloval 60 krát za vteřinu, ale méně časté aktualizace způsobily neplynulý pohyb. Řešení je na klientovi interpolovat spojitě proměnné, jako pozice nebo rotace. Do ovladače pro replikátor jsme přidali komponentu interpolátor. Do něj registrujeme jednotlivé komponenty, které chceme interpolovat. Interpolátor si je ukládá i s časovou značkou do bufferu. Na rozhraní jsem vytvořili funkci s argumentem čas a typ komponenty.

Technika měla nepříjemný důsledek zvýšení odezvy. To vadí především pro entity, které hráč ovládá, jako např. hráčova postava. Problém je, že když na interpolátor přijde snapshot n , může teprve začít interpolovat z pozice $n - 1$. Proto

3.6.3 Rollback a simulace vlastního hráče

Interpolovat sice pomohlo pro plynulost, ale zvýšilo odezvu, což je nepříjemné. Způsob, jakým jsem se rozhodl tento problém řešit, je integrovat hráče pro daného klienta přímo na klientovi. Nejen že odezva bude v podstatě nulová, ale navíc bude ještě nižší než před interpolací. Musel jsem ale vytvořit mechanismus, který v případě, že se simulace neshodnou, tak na klientovi stav hráče opraví.

Do bufferu jsem si ukládal pro konkrétní snímky vstup od hráče, který pošlu na server, a aktuální stav, ve kterém hráč je. Následně když ve snímku x přijde snapshot pro snímek $x - t$, který má až moc velkou odchylku, tak vrátíme stav světa na $x - t$ a podle vstupů z bufferu znova integrujeme. Tím dostaneme novou a opravenou pozici.

3.7 Metriky

Abych byl schopný správně optimalizovat tok dat, potřeboval jsem sbírat metriky. Proto jsou třeba dvě komponenty. První je úložiště, kde budu metriky ukládat. Může se jednat o CSV soubor nebo databázi. Druhá je rozhraní, ve kterém budu metriky vizualizovat. Představím dvě služby, které pro to využívám. Následně si představíme samotné metriky, které budu měřit a proč. Všechny metriky budu měřit po vteřinách, některé jako sumu, jako např. množství odchozích dat, a jiné jako průměr, jako např. odezvu.

3.7.1 TimescaleDB

Jako úložiště jsem zvolil TimescaleDB. Jedná se o rozšíření do PostgreSQL, takže není třeba spouštět a starat se o další službu. Stačí mít spuštěnou PostgreSQL databázi. TimescaleDB umožňuje vytvářet tzv. hypertables, které jsou optimalizované pro rychlý insert. Vytvořil jsem komponentu ‘NetworkBandwidthReporter’, která má na rozhraní ‘report_outbound’ a ‘report_inbound’. Tyto údaje posílá do komponenty ‘TimescaleDBReportOutputter’, která už zajišťuje připojení do databáze a umožňuje ‘report(metric_name, value)’. V rámci optimalizace přidávám hodnoty do bufferu a posílám je pouze jednou za vteřinu jako sumu.

3.7.2 Grafana

Grafana je služba, která skrze webovou stránku poskytuje rozhraní pro vizualizaci metrik. Lze nastavit svůj dashboard a v něm různé grafy. V mém případě graf pro množství odchozích a množství příchozích dat, zatížení procesoru a nebo zatížení paměti RAM.

3.7.3 ImGui

Chtěl jsem mít i možnost se na metriky podívat přímo ve hře při běhu. Pro uživatelské rozhraní používám knihovnu ImGui a ImPlot. Tyto knihovny využívají pro vykreslování grafickou kartu a snadno jsem je mohl integrovat do svého vykreslovacího enginu. Knihovna funguje procedurálně, nikoliv objektově. Každý snímek, který vykresluju, musím celé rozhraní definovat. To udělám pomocí volání funkcí. Nejprve zavolám funkci ‘Begin("Název okna")’, které zobrazí okno, do kterého můžu vložit další prvky, jako texty, textové vstupy, nebo tlačítka. Tento přístup je pro herní enginy a podobné kreativní aplikace ideální, protože umožňuje velmi snadno a rychle dělat dynamické uživatelské rozhraní.

3.7.4 Tracy

Pro měření výkonu programu serveru používám výborný profiler Tracy. Je zdarma, open-source, napsaný v C++ a využívající ImGui. Poskytuje přesnost na nano-sekundy, což se u serverů, na kterém může být miliony různých entity, hodí. Navíc umožňuje se k programu, který měří, připojit vzdáleně, případně sbírat metriky do souboru, ten si ze serveru stáhnout a analyzovat později.

3.7.5 Konkrétní metriky

Ukážu různé metriky, které lze měřit, jak a hlavně proč. Je dobré zmínit, že místo, ze kterého budu metriky sbírat, je server. To ale nevadí, protože jak uvidíte, vše potřebné na něm měřit jde, a to co ne, tak nás nemusí zajímat.

Inbound/Outbound bandwidth

Nejpřirozenější metrika je příchozí a odchozí množství dat v bytech. Už zde pro mě bylo nutné si uvědomit důležitost oddělit vrstvu, která kóduje a dekóduje zprávy, od zbytku. Přesně na tomto místě totiž budu metriky dělat. Metriku budu počítat.

Odezva

Budu měřit odezvu mezi odesláním vstupu od klienta pro snímek x a odpovědí od serveru pro snímek x. Tuto metriku budu průměrovat.

Počet snímků za vteřinu

Ze začátku se mi stávalo, že úzké hrdlo nebyla síť, ale síťová vrstva v aplikaci, která třeba dlouho skládala zprávy. Proto si udržuju i přehled o počtu snímků, které server stíhá. Podle toho se lze orientovat při výběru hardware pro server.

3.7.6 Sbírání v reálném čase

Přímo v hráčově klientské aplikaci jsme chtěli mít možnost vidět metriky v reálném čase. Do síťové vrstvy jsme přidali komponentu pro reportování událostí a jejich metrik zvanou `\NetworkMetricReporter`, která obsahuje metody jako `\reportInbound` a `\reportOutbound`. První metoda hlásí příchozí byty a druhá odchozí.

3.8 Protokol QUICr

Naše hra používala ke komunikaci mezi vrcholy protokol TCP. Zmínili jsme ale, že ten pro hry nemusí být vhodný. Vytvořil jsme proto protokol inspirovaný QUIC, který využívá UDP, ale upustil jsem podmínky pro spolehlivost, konkrétně jistotu doručení a uspořádání. Nakonec jsme TPC a QUICr porovnali, abychom se o optimalizaci přesvědčili.

Protokol využívá UDP a přenáší tak datagramy. Ty se skládají z balíku a ten z jednoho nebo více rámců. Rámce můžeme rozdělit na dva typy: datové a

ovládací. Ovládací slouží pro handshake nebo ukončení spojení a budou vždy spolehlivě doručené. Datové obsahují aplikační data a mohou definovat svou spolehlivost. Ty, které budou obsahovat pozice hráčů, tak budeme moct odesílat nespolehlivě, zatímco změny stavu, jako výměna animace nebo modelu, budu posílat spolehlivě. Pro spojení budu používat dva objekty: „koncový bod“ a „spojení“.

Objekt pro koncový bod se stará o soket: čtení a zápis na něj, a udržuje si objekty spojení, které ho využívají. Když přijde na soket datagram, správně ho přiřadí spojení. Jeho chování je definované jako tik. V něm posbírání zprávy, které se spojení snaží poslat a pošle je, následně přečte data ze soketu a roztrídí je do navázaných spojení.

Název metody

bind(port)

write(adress, payload) Odešle 'payload' přes socket na adresu 'address'

poll

Přečte ze socketu všechny datagramy, které

Objekt spojení je navenek stavový stroj a fronta zpráv, podobně jako TCP spojení. Stav se mění v reakci na příchozí rámce. Když na koncový bod přijde datagram, odešle ho do správného objektu připojení. Tento objekt si datagram přečte a přeloží na pakety a rámce. Ty postupně prochází. Stavový stroj bude důležitý hlavně v procesu zvaném "handshake", který definuju níže.

Zároveň jsem se rozhodl, že na spojení nebude zdánlivě nekonečný proud bytů, jako je tomu například v TCP, ale fronta zpráv. Výhoda je, že když začnu číst datagram, tak mám jistotu, že nenarazím na konec bufferu a budu si muset načíst další část. V mém případě jedno čtení znamená čitelná zpráva, tak jak jsme byla odeslána. Zároveň mám z aplikace kontrolu nad datagramy. Replikátor může sám od sebe vytvořit namísto jedné velké zprávy více malých a odeslat je. Klient nemusí čekat, až by mu přišla celá zpráva a protokol by mu ji poskládal.

3.8.1 Frame

Inspiroval jsem se u protokolu QUIC, který datagram rozděluje na tzv. frame neboli „rámce“. Tyto rámce jsou vhodná reprezentace pro ovládání protokolu, protože je můžeme kódovat proudem bytů v TCP i datagramem v UDP. Využili jsme nespolehlivosti a rychlosti UDP a u každého rámce definujeme jeho spolehlivost, tedy jestli musí rámec vždy dojít. Objekt pro připojení obsahuje jednotku pro spolehlivost, kterou si představíme v kapitole o spolehlivosti.

U UDP je další problém, a to ten, že když implementuju připojení pro UDP, tak všechny datagramy budou chodit na socket, na kterém poslouchám. Když budu ze socketu číst, dostanu libovolný datagram, který zrovna přišel, společně

s adresou, ze které přišel. Tuto adresu přečtu z IP hlavičky, která není zašifrovaná, a tudíž ji kdokoliv může zfalšovat. Tomu se říká tzv. IP spoofing. Tento útok je poměrně slabý, protože v případě odpovědi budu stále odesílat data zpět na správnou adresu. Útok může sloužit například pro posílání požadavku na odpojení, nebo posílat jiné vstupy. Jedná se tedy spíše o neškodný ale otravný charakter útoku.

3.8.2 Handshake

Po vytvoření instance připojení z koncového bodu je připojení ve stavu ‘Closed’, neboli uzavřené. Pro navázání spojení jsem potřeboval, aby se dva objekty domluvili na svých ID, tajném klíči a verzi. Proto jsem definoval proces zvaný „handshake“. Mějme dva koncové body A a B , které se mezi sebou budou snažit navázat spojení. Začne koncový bod A , a to tím, že odešle na koncový bod B paket typu Initial, který obsahuje rámec typu ‘Hello’. Jakmile na koncový bod dorazí QUICr paket, který má v hlavičce jako Destination ID číslo, které nemá v registru, tak si ho přidá a přiřadí novou instanci připojení. Nová instance se chová stejně jako na straně A a vygeneruje si unikátní identifikátor. Po vytvoření instance nechá koncový bod toto připojení tento paket zpracovat celý.

Nové připojení projde všechny rámce v paketu. Měl by narazit na rámec Hello, který když najde, a je ve stavu ‘Initial’, tak z hlavičky zjistí unikátní ID druhé strany a uloží si ho. Reaguje na rámec tak, že druhé straně odešle přes koncový bod paket, který obsahuje: rámec ACK o tom, že paket s číslem přijal, a rámec Hello. ACK si pro začátek představme zakódovaný jako: první 4 byty reprezentují kladné celé číslo pro počet rámců, řekněme n , které chceme označit jako přijaté. Následně přečteme $4n$ bytů. Každé 4 byty představují číslo paketu, který označíme jako ACKed. Jednotce, která udržuje stav o tom, jaké pakety jsou potřeba oznámit jako ACK a které je třeba poslat znovu, budu říkat ‘ReliabilityUnit’ a podrobněji ji rozeberu později.

Přeskočme zpět na klienta, který dostává paket typu Initial, ve kterém je rámec Hello a ACK. Pro rámec Hello se chová stejně jako druhá strana, a pokud je ve stavu ‘SendHello’, nastaví si Destination ID. Všechna čísla paketů z ACK rámce předá do jednotky pro spolehlivost. Jelikož víme, že tam je číslo paketu, který obsahuje právě prvotní Hello, tak si jednotka tento rámec odebere z fronty pro opakované odeslání.

Closed

Počátečním stavem je ‘Closed’. Zároveň se jedná o stav, do kterého se spojení dostane, pokud dlouho s druhou stranou nekomunikuje.

SentHello

Po odeslání Hello packetu se spojení dostane do stavu ‘SentHello’. V tomto stavu čeká na ‘HelloAck’ zprávu.

ReceivedHello

Jakmile server dostane zprávu Hello, dostane se do stavu ‘ReceivedHello’.

Established

Po odeslání HandshakeDone zprávy se klient dostane do stavu Established. Jakmile server přijme zprávu HandshakeDone, taky se dostává do stavu Established.

Klient následně odpovídá Handshake done, protože už má všechny informace, které potřebuje o spojení. Upustil jsem od packetů a nechal pouze rámce.

3.8.3 Spolehlivost

Pro spolehlivost se protokol stará podobně jako QUIC, s tím rozdílem, že ji rozlišuje nad jednotlivými rámci. Všimněme si, že něco podobného má i samotný QUIC. Pokud odešleme paket, ve kterém je pouze ACK rámec, tak nedostaneme nikdy zpátky ACK. Rámec ACK se tedy chová stejně, jako by byl nespolehlivý. Rámce tak můžeme dělit na dvě kategorie: spolehlivé a nespolehlivé. Nespolehlivé rámce spojení odešle a hned je zapomene, tzv. fire and forget. O spolehlivé rámce se stará třída zvaná ‘QuicReliabilityUnit’. Každá instance připojení na koncovém bodě má svou instanci, kterou ke spolehlivosti využívá. Její rozhraní tvoří 4 hlavními metodami:

push_reliable_frame_to_send(deadline, frame)

Uloží si zakódovaný rámec a nastaví si u něj čas deadline. Jakmile bude po deadline, jednotka se bude snažit odeslat rámec znovu.

peek/pop_reliable_frames_to_send(packet_number)

Vrátí rámce, které je potřeba v tomto okamžiku odeslat. Číslo paketu je důležité proto, aby jednotka věděla, které všechny rámce může ACK pro určitý paket označit jako spolehlivě doručené.

push_ack(packet_number)

Uloží si do seznamu číslo paketu a v další iteraci ho odešle jako ACK.

peek/pop_acks_to_send()

Vrátí seznam paketů, které je třeba odeslat jako ACK. Zároveň si je ze seznamu odebere.

Jednotka pro spolehlivost si každý vložený rámec uloží, společně s časem, do kdy je potřeba ho odeslat znova, tzv. ‘deadline’. Tento přístup mimo jiné umožňuje vložit spolehlivý rámec rovnou do jednotky s deadline aktuálního času. Rámec se odešle co nejdříve a zajistí se jeho spolehlivost.

Dále si jednotka ukládá čísla paketů, které jsou v oběhu. Každý takový paket v oběhu ukazuje na seznam odkazů na spolehlivé rámce, které obsahuje. Když přijde číslo paketu jako ACKed, jednotka odstraní všechny rámce, na které paket ukazuje, a nakonec zapomene celý paket. Mohlo by se stát, že pro jisté pakety se nedoručí ACK nikdy, ale tyto pakety můžeme jednoduše vyčistit kontrolou, jestli obsahují alespoň jeden rámec, který nebyl takto označen.

Do jednotky si ukládám už zakódované rámce, protože tak můžu snadno odhadnout jejich velikost. Když bude

Možná jste si všimli, že nezaručujeme spolehlivost pořadí. Tuto záruku jsme do protokolu zatím nepřidávali, protože si stejně do paketu ukládáme číslo snímku, které je potřebné pro správnou aktualizaci světa. Toto číslo bychom z čísla paketu nebyly schopni vyčíst. Zároveň nás v našem případě nezajímá, ve kterém pořadí pakety pro jeden snímek přijdou. Kdybychom měli úplné uspořádání, a odeslali bychom z replikátoru datagramy A a B , které by přišli na klienta v pořadí B , A , musel by klient čekat na datagram A , i když by mohl zpracovávat datagram B .

3.8.4 Enkodér

Pro kódování jsme vytvořili komponentu `\QuicrEncoder`, která přes konstruktor získá cílový buffer, do kterého má kódovat a na rozhraní má různé pomocné metody pro kódování hlaviček a rámců. Specifická vlastnost, kterou jsme potřebovali, je kódovat, dokud je v bufferu místo. Pokud zbývá r bytů místa a replikátor by chtěl zakódovat rámec, který má $> r$ bytů, musí metoda vrátit informaci o neúspěchu, ale nevyhazovat vyjímku, ani neposunovat kurzor. Důvodem je, že UDP je nad protokolem IP, který může začít fragmentovat datagramy. Proto se většina protokolů snaží držet velikost datagramů kolem 1200 bytů. Podobný přístup jsme adaptovali my. Dále musel umožňovat zakódovat hodnotu zpětně, jako například délku, kterou kvůli omezení bufferu víme až později.

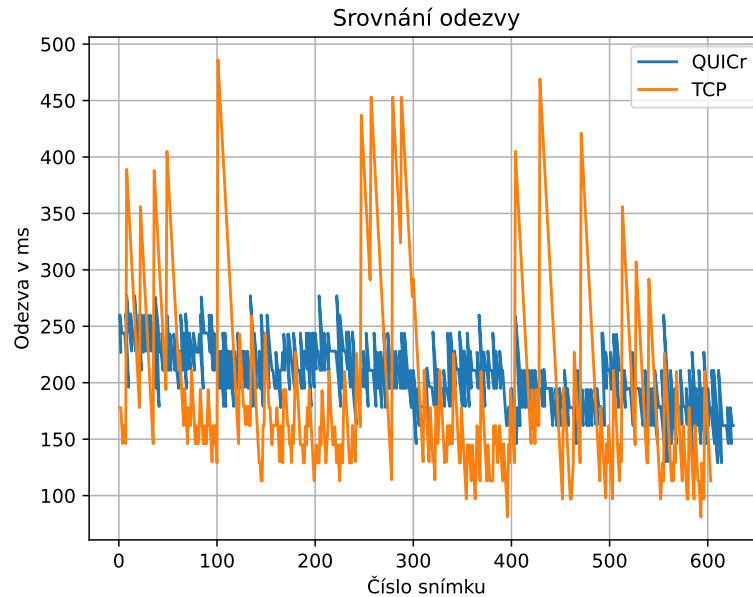
3.8.5 Testování

Při implementaci jsme zjistili, že je dobré začít testy, které definují jednotlivé vlastnosti, jako například: "Spojení začne komunikaci rámcem Hello", "Po rámci Hello odpoví spojení rámcem Hello a ACK", "Na pakety obsahující spolehlivé rámce odpoví spojení ACK rámcem" apod. Tomuto přístupu se říká test driven development.

3.8.6 Měření

Protokol měl za cíl snížit odezvu na nespolehlivé síti. Vytvořili jsme test, který simuluje program hry. V testu jsou dvě vlákna, jedno pro klienta a druhé pro server. Oba mají svou vlastní smyčku, ve kterém iterují snímky. Klient odesílá hodnotu derivace typu `double`. Server si udržuje stav hodnoty, nazvěme ji „pozice“. Při obdržení derivace ji přičte k derivaci. Každý snímek server odesílá na klienta stav pozice, kterou si klient aplikuje.

Pro nasimulování nespolehlivosti sítě jsme využili Linuxový nástroj „tc“. Ten obsahuje network emulator, který umožňuje například nastavit, kolik procent paketů má zahodit, jaká má být odezva nebo jak moc se budou prohazovat pakety. Nástroj a jeho vlastnosti, které budeme využívat, si představíme. Nastavili jsme odezvu na 20-100ms a 2% paketů se zahodí.



Obrázek 3: Porovnání odezvy TCP a QUICr

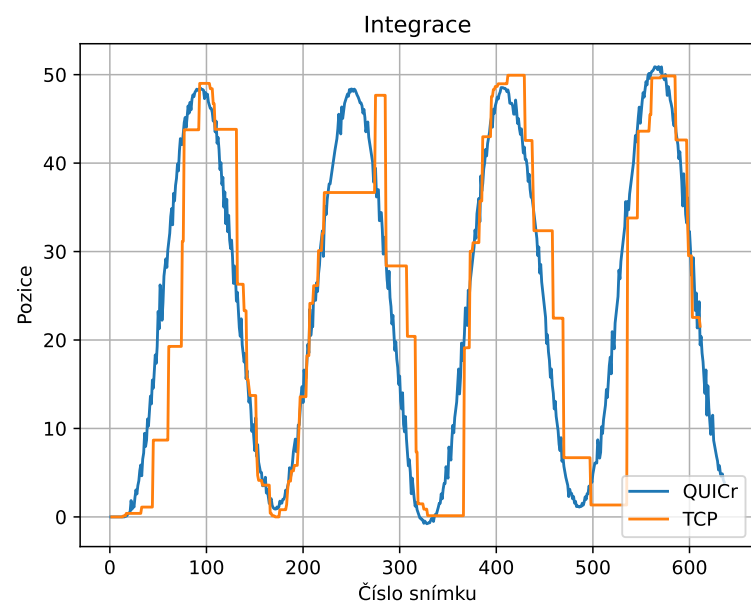
V prvním testu jsme pouze měřili odezvu mezi odesláním vstupu od klienta pro snímek n a získáním jeho integrovaného stavu ze serveru. Na obrázku 3 vidíme, že TCP obsahuje skoky a QUICr je stabilnější.

Následně jsme zkusili, jak se nespolehlivost sítě a skoky odezvy z prvního měření projeví v integraci proměnné. Pro derivaci jsme použili sinusoidu, abychom ji mohli zreplikovat snadno v TCP i QUICr. Na obrázku 3.8.6 vidíme, že TCP při ztrátě paketů způsobuje skoky, které nemusí být pro hráče příjemné. Naopak náš protokol má integraci plynulejší.

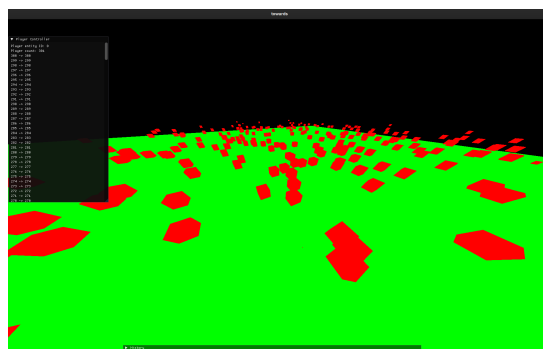
Nakonec jsme integrovali protokol přímo do replikátoru. Vytvořili jsme simulovaného klienta, který na server posílá náhodné vstupy a způsobuje na serveru svůj pohyb. Těchto simulovaných klientů můžeme spouštět libovolný počet. Každý naváže se serverem spojení a komunikuje s ním klasicky přes replikátor a ovladač. Můžeme tak snadno měřit, jak se zatížení projeví v síti, a zároveň můžeme využít nástroje jako tc pro různé podmínky. Na obrázku 3.8.6 vidíme 301 hráčů, kde 300 je simulovaných individuálních připojení a jeden je náš hráč. Server počítal každý snímek dlouho, asi 250ms, ale zátěž zvládl. Dalším krokem bylo zvýšit počet snímku za sekundu pro stovky hráčů.

3.9 Optimalizace replikátoru

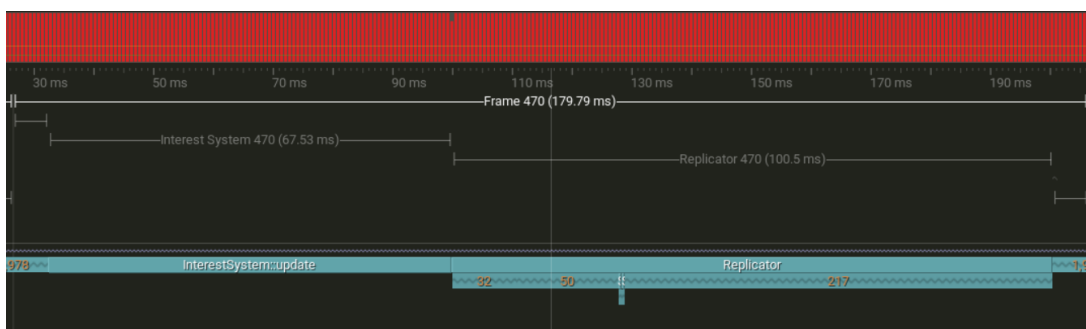
Test s 301 hráči odhalil nedostatečný výkon serveru. Pomocí Tracy jsme začali program analyzovat. Na obrázku 3.9 vidíme, že pouze aktualizace replikátoru trvá každý snímek kolem 100ms. Pro představu, pokud bychom chtěli, aby server



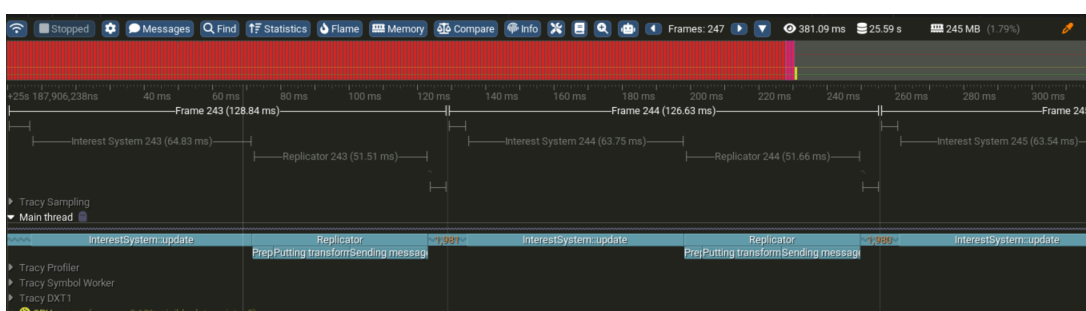
Obrázek 4: Porovnání integrace TCP a QUICr



Obrázek 5: 301 připojených hráčů



Obrázek 6: Analýza replikátoru



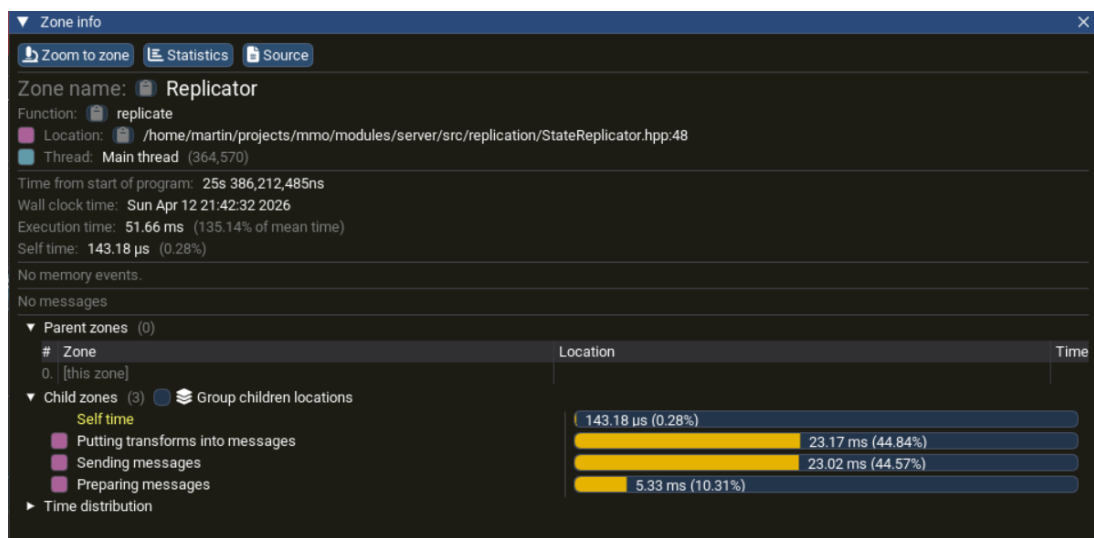
Obrázek 7: Analýza optimalizace replikátoru pro ECS

integroval 20 krát za sekundu, museli bychom všechno, nejen aktualizaci replikátoru, stihnout do 50 milisekund. Proto je aktuální stav nepříjemně dlouhá doba. Úzké hrdlo najednou nebyla síť, ale příprava smysluplných dat, kterými síť zatížíme. Soustředili jsme další optimalizace právě na replikátor.

Z analýzy vyšlo najevo, že problém je skládání zprávy. Tedy vkládání pozic entit, o které se klient zajímá, do objektu. Je to právě tento objekt, který se serializuje přes ProtoBuf a odesílá skrze soket na klienta.

Replikátor pro každého klienta zjistil jeho zájem a pro každou entitu v něm vyhledával jeho komponenty. Vyhledávání je v případě Entt v $O(1)$. Začali jsme měřit i jiná řešení. Rozhodli jsme optimalizovat pro ECS architekturu. Vytvořili jsme buffer instancí zpráv pro každého klienta. Přes komponentu, kterou jsme chtěli replikovat, jsme vytvořili Entt pohled a iterovali. Pro každý prvek jsem komponentu zapsali do zprávy pro každého klienta, kterého zajímala. Výslednou analýzu vidíme na obrázku 3.9. Optimalizace snížila dobu, kterou trvá aktualizace replikátoru, na polovinu.

Replikátor jsme začali analyzovat do větších detailů. Na obrázku 3.9 vidíme, které jeho části zabrali jakou dobu. Zjistili jsme, že problém byl samotný ProtoBuf. Formát je podobně jako JSON orientovaný na strukturu objektů. V našem případě je ale zpráva pro snapshot světa primitivní. Rozhodli jsme udělat porovnání s FlatBuffers. V testu jsme simulovali 100 snímků, tak, abychom si mohli



Obrázek 8: Analýza optimalizace replikátoru pro ECS

	ProtoBuf	FlatBuffers	Memcpy
Čas (s)	9.455691762	10.345692894	0.754350866

Obrázek 9: Porovnání serializátorů

dopředu alokovat všechnu potřebnou paměť, kterou můžeme mezi snímky využívat. Dopředu si vytvoříme vektor pozic, které v testu budeme serializovat. Stejně jako v replikátoru používáme architekturu orientovanou na ECS a procházíme všechny pozice, ty pak umísťujeme do alokovaných bloků paměti pro každého klienta a simulujeme tak skládání rámců, které můžeme odesílat. Všimli jsme si, že v případě primitivní zprávy, jako je snapshot stavu světa, by nám stačila klasická funkce ze standartní C knihovny zvaná `memcpy` a do testu jsme ji zařadili. Výsledky vidíme v tabulce 3.9.

Zjistili jsme, že problémem byl `ProtoBuf` a `FlatBuffers` by nám nepomohl. Nikoho nepřekvapí, že `memcpy` byl nejrychlejší. Rozhodli jsme se tedy, že budeme pro serializaci a deserializaci snapshotů světa používat vlastní enkodér a dekodér.

Pro kódování jsme využili už existující `ByteBuffer`, který umožňuje snadné kódování po bytech do vektoru. Třída `WorldStateWriter` obsahuje konkrétní kódování pro tyto zprávy a přes konstruktor je nutné předat instanci `ByteBuffer`.

Zpráva obsahuje hlavičku, ve které je počet aktualizovaných entit a počet nových entit. Počet odebraných entit můžeme vynechat, ten jsme schopni zjistit z délky zprávy a zbylých počtů.

Název testu	Celkový čas	Čas vkládání	Čas dotazování
Naivní	38770	0	38650 (99.96%)
Fixní mřížka	467	301 (64.56%)	17 (3.74%)
Hashovací mřížka	1070	736.63 (68.76%)	42 (3.93%)
Quad tree	1830	1560 (85.45%)	30.29 (1.66%)

Obrázek 10: Porovnání algoritmů

3.10 Optimalizace manažera zájmů

Další na řadě byla optimalizace manažera zájmů, který trval každý snímek kolem 60 milisekund. Nejprve jsme se rozhodli přidat datové struktury. Druhým cílem bylo zlepšit API, kterým se replikátor dotazuje, jestli je entita pro klienta zajímavá.

Jedna s datových struktur, která má nízkou složitost pro dotaz na seznam entit ve vzdálenosti maximálně, je klasická mřížka. Charakteristika hráčů ve hrách je, že se hodně pohybují. Mřížka nepotřebuje žádný přepočít, pouze vložit do předem alokovaného bufferu, nebo z něj naopak odebrat.

Testovali jsme 4 implementace, z toho 3 různé datové struktury a jednou naivní přístup. Simulovali jsme pohyb 100 000 různých entit po dobu 100 snímků. Všechny možnosti představíme. Naivní přístup využívá pouze Pythagorovu větu, aby získal vzdálenost dvou bodů. Jediná optimalizace je vynechat odmocninu a umocnit místo toho vzdálenost. Druhá a třetí implementace využívá mřížku a entity rozděluje do svých polí. Fixní mřížka je vhodná pro světy, které nemění svou velikost, jako např. World of Warcraft. Naopak hashovací mřížka přiřazuje entity do polí pomocí hashe jejich pozice. Jsou tak ideální pro dynamické a potenciálně nekonečné světy, jako např. Minecraft. Poslední je quad tree. Jedná se o obdobu binárního stromu rozšířenou o rozměr. V tabulce 3.10 vidíme výsledky testu. Quad tree implementace se ukázala jako nevhodná. S přibývajícím hloubkou je navíc pomalejší. Ideální se ukázali varianty s fixní a hashovací mřížkou.

3.11 Optimalizace QUICr

3.11.1 Šifrování

Jednotlivé datagramy a zprávy je třeba šifrovat a pro to je třeba, aby si dvě strany vyměnili tajný klíč. Představím metodu, kterou pro výměnu budu využívat.

3.11.2 SPR-6

Metodu, kterou jsem viděl například u síťového protokolu pro hru World Of Warcraft je SRP-6. Jedná se o protokol, který nevyžaduje, aby server znal heslo

klienta, ale měl u sebe pouze validátor, a klient mu díky němu dokáže, že heslo zná. Pro tento přístup je třeba udělat handshake.

3.11.3 Lockstep

Aby tato technika fungovala správně, musí být intergrace ve fyzickém enginu stejné na klientovi a serveru.

3.11.4 Congestion control

3.11.5 Bandwidth control

Entity, které jsou od hráče daleko, není třeba aktualizovat tak často jako ty, které jsou k němu blízko. Proto jsem do zájmu ke každé entitě přidal důležitost. Replikátor se při odesílání snaží nejprve nacpat do packetu entity s vyšší prioritou. Aby nedošlo k vyhladovění, tak odeslané entitě trochu klesne důležitost. Tuto informaci už si řeší replikátor.

3.12 Sharding

Proces, při kterém hráče rozdělíme do skupin, které se navzájem nevidí, i když jsou na stejném serveru blízko sebe, se říká sharding. Podobný princip funguje i v databázích. Tato technika složitá udělat správně. Vývojář musí dobře definovat logiku toho, že dva hráči mohou skončit na jiném shardu.

3.13 Komprese

Poslední optimalizací, kterou jsem pro přenos informací použili, byla komprese. Nejprve jsme nahradili byte buffer, který pracuje pouze s byty, bit bufferem, který nám umožnil pracovat s jednotlivými bity.

3.13.1 Bit Buffer

Snadným zmenšením packetů je zaměřit se na bity namísto bytů. Původně jsem nově příchozí datagramy obalil pomocí ByteBuffer, který umožnil dekodovat postupně příchozí byty. Často ale potřebujeme mít možnost zakódovat informaci pomocí jednotlivých bitů.

3.13.2 Komprese QUICr

Protokol QUICr kóduje pakety a rámce. Toto kódování jsme se rozhodli zkompresovat.

3.13.3 Komprese protokolu replikátoru

Komprese, která lze provést na úrovni QUICr protokolu, je značně omezená. Komprese lze ale provést i na samotných aplikačních zprávách. Pro

4 Styly pro psaní bakalářských a diplomových prací

Toto jsou styly pro psaní bakalářských a diplomových prací přes typografický systém L^AT_EX, tedy **kistyles**.

4.1 Požadavky a podpořovaná prostředí

Sada balíku **kistyles** podporuje následující distribuce systému L^AT_EX:

- T_EX Live.

Jsou podporovány všechny výstupní ovladače, tedy jak **dvi**, tak **pdf** i **ps**. Funkčnost zmiňovaných distribucí byla ověřena na několika operačních systémech, mezi které patří:

1. Windows 8.1,
2. Archlinux,
3. Debian GNU/Linux.

Důrazně se doporučuje používat aktuální verzi dané distribuce systému L^AT_EX.

4.2 Přepínače

Styl kidiplom je z hlediska uživatele zastoupen ekvivalentně nazvanou třídou, kterou je třeba volat na začátku dokumentu:

```
1 \documentclass[
2   master,
3   program=ainfvs,
4   printversion,
5   biblatex,
6   language=english,
7   font=sans,
8   figures=false,
9   tables=false,
10  theorems,
11  sourcecodes,
12  joinlists,
13  glossaries,
14  index,
15  encoding=utf8,
16  bibencoding=utf8
17 ]{kidiplom}
```

Zdrojový kód 4: Volání třídy **kidiplom**

Následuje přehled přepínačů, je vždy uvedeno jméno přepínač, včetně výchozí hodnoty. Přepínače uvádí tabulka 2.

Tabulka 2: Seznam přepínačů

Přepínač	Výchozí hodnota	Popis
master	false	Povolí nebo zakáže režim diplomové práce. Výchozí režim je tedy bakalářská práce.
printversion	false	Je-li zapnuto, pak budou odkazy vysázeny optimalizovaně pro knižní sazbu. Tuto volbu je nutno použít pro tisk práce.
biblatex	false	Zapne sazbu bibliografie přes balík BIBL _A T _E X.
language	czech	Jazyk textu práce. Možné hodnoty jsou czech , english a slovak .
font	serif	Zapne či vypne podporu pěkného bezpatkového fontu. Možné hodnoty jsou: serif Patkové písmo (Computer Modern).
figures	true	Je-li zapnuto, pak seznam (tvořený položek bude zahrnut seznam obrázků.
tables	true	Je-li zapnuto, pak v seznamech položek bude zahrnut seznam tabulek.
theorems	false	Je-li zapnuto, pak v seznamech bude zahrnut seznam teorémů.
sourcecodes	false	Je-li zapnuto, pak v seznamech bude zahrnut seznam zdrojových kódů.
joinlists	true	Je-li zapnuto, pak seznamy obrázků, tabulek, vět a zdrojových kódů sázené za obsahem nebudou rozděleny na samostatné stránky.
glossaries	false	Je-li zapnuto, pak na konci dokumentu bude vysázen seznam zkratk.
index	false	Zapíná podporu sazby rejstříku.
encoding	utf8	Kódování souboru dokumentu, doporučuje se ponechat výchozí hodnotu.
bibencoding	utf8	Kódování souboru bibliografie. Tato volba má smysl pouze, pokud je použita bibliografie skrze balíček BIBL _A T _E X.

program	infpvs	Specifikuje studijní program/obor
	ainfvs	(specializaci):
	infoi	Informatika (Obecná informatika) – bakalářský i navazující magisterský,
	infpvs	Informatika (Programování a vývoj software) – bakalářský,
	itp	Informační technologie – bakalářský, prezenční forma,
	itk	Informační technologie – bakalářský, kombinovaná forma,
	infui	Informatika (Umělá inteligence) – navazující magisterský,
	ainfvs	Aplikovaná informatika (Vývoj software) – navazující magisterský,
	ainfpst	Aplikovaná informatika (Počítačové systémy a technologie) – navazující magisterský,
	infv	Informatika pro vzdělávání – bakalářský,
	uinf	Učitelství informatiky pro střední školy – navazující magisterský,
	binf	Bioinformatika – bakalářský i navazující magisterský,
	inf	Informatika (bez specializací) – bakalářský i navazující magisterský,
	ainfp	Aplikovaná informatika (bez specializací) – bakalářský, prezenční forma,
	ainfk	Aplikovaná informatika (bez specializací) – bakalářský, kombinovaná forma,
	ainf	Aplikovaná informatika (bez specializací) – navazující magisterský.

4.3 Geometrie stránky

Tento styl používá list velikosti A4. Pro sazbu prací je třeba použít jednostrannou sazbu. Levý okraj je rozšířen s ohledem na vazbu výsledné knižní podoby práce.

5 Sazba částí dokumentu

5.1 Sazba úvodní strany či obsahu

Vysázení všech podstatných částí úvodu práce obstará makro `\maketitle`. Pro správné vysázení všech částí a meta-informací je potřeba použít makra `\title`, `\author` a další. Jejich přehled lze najít ve zdrojovém souboru tohoto dokumentu. V případě použití **pdf** výstupu se generuje i dodatečná hlavička souboru s meta-informacemi jako je autor dokumentu, název práce či dalšími.

5.2 Závěry

Závěr práce by se měl poskytnout jak v původním jazyce práce, tak v jazyce anglickém. Pro sazbu závěru jsou k dispozici příslušná makra. Berte na vědomí, že v anglickém závěru se aktivuje plně anglická sazba se všemi konvencemi. Tedy je třeba používat anglické uvozovky a další správné typografické prvky.

```
1 % Tiskne český závěr práce.
2 \begin{kiconclusions}
3 Závěr práce v \uv{českém} jazyce.
4 \end{kiconclusions}
5
6 % Tiskne anglický závěr práce.
7 \begin{kiconclusions}[english]
8 Thesis conclusions written in \uv{English}.
9 \end{kiconclusions}
```

Zdrojový kód 5: Sazba závěrů

5.3 Matematika

Pro sazbu matematiky je k dispozici sada standardních maker.

$$\langle f \rangle, [g], [h], \lceil i \rceil$$

$$\left\{ \frac{x^2}{y^3} \right\}$$

$$A_{m,n} = \begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m,1} & a_{m,2} & \cdots & a_{m,n} \end{pmatrix}$$

$$M = \begin{bmatrix} \frac{5}{6} & \frac{1}{6} & 0 \\ \frac{5}{6} & 0 & \frac{1}{6} \\ 0 & \frac{5}{6} & \frac{1}{6} \end{bmatrix}$$

5.4 Sazba literatury

Pro sazbu literatury má uživatel dvě možnosti. Může použít služeb balíků `BIBLATEX`, který je pro `kistyles` zapnutý, či lze použít manuální sazbu bibliografie.

5.4.1 Sazba bibliografie přes BibL^AT_EX

Při použití tohoto balíku se data o použité literatuře ukládají do dedikovaného textového souboru, ukázkou najdete i v tomto stylu pod jménem `bibliografie.bib`.

Formát daného souboru je nad rámec této dokumentace a je na každém uživateli, aby si jej nastudoval. Bibliografie se tiskne makrem `\printbibliography`. Taktéž v preambuli dokumentu je třeba definovat, který soubor data bibliografie obsahuje, tedy například `\bibliography{bibliografie.bib}`.

Dokument, který využívá `BIBLATEX` je následně nutné přeložit jak pomocí překladače zvoleného ovladače, tak pomocí aplikace `biber`. Více informací poskytne soubor `Makefile` z distribuce tohoto stylu.

Výhodou tohoto přístupu je, že bibliografie se vysází automaticky a (obvykle) není třeba manuální úprava formátování.

5.4.2 Manuální sazba bibliografie

Manuální sazba obnáší vysazení prostředí `thebibliography` ručně. To je nad rámec tohoto dokumentu. Ukázkou tohoto přístupu lze samozřejmě nalézt ve zdrojovém souboru tohoto dokumentu nebo také [zde](#).

Pro aktivaci manuální sazby bibliografie je třeba volat třídu `kidiplom` s parametrem `biblatex=false`. Mějte, prosím, na paměti, že v tomto módu jsou makra `\bibliography` a `\printbibliography` nedostupná.

5.5 Drobná makra

Základní styl definuje hned několik maker pro usnadnění práce. Například makro `\buno` vysází řetězec „bez újmy na obecnosti“. Je k dispozici i verze s prvním velkým písmenem, `\Buno`.

Tabulka 3: Odstavce v tabulkách

Donec et nisl id sapien blandit mattis. Aenean dictum odio sit amet risus. Morbi purus. Nulla a est sit amet purus venenatis iaculis. Vivamus viverra purus vel magna. Donec in justo sed odio malesuada dapibus. Nunc ultrices aliquam nunc. Vivamus facilisis pellentesque velit. Nulla nunc velit, vulputate dapibus, vulputate id, mattis ac, justo. Nam mattis elit dapibus purus. Quisque enim risus, congue non, elementum ut, mattis quis, sem. Quisque elit.	Etiam suscipit aliquam arcu. Aliquam sit amet est ac purus bibendum congue. Sed in eros. Morbi non orci. Pellentesque mattis lacinia elit. Fusce molestie velit in ligula. Nullam et orci vitae nibh vulputate auctor. Aliquam eget purus. Nulla auctor wisi sed ipsum. Morbi porttitor tellus ac enim. Fusce ornare. Proin ipsum enim, tincidunt in, ornare venenatis, molestie a, augue. Donec vel pede in lacus sagittis porta. Sed hendrerit ipsum quis nisl. Suspendisse quis massa ac nibh pretium cursus. Sed sodales. Nam eu neque quis pede dignissim ornare. Maecenas eu purus ac urna tincidunt congue.	Etiam pede massa, dapibus vitae, rhoncus in, placerat posuere, odio. Vestibulum luctus commodo lacus. Morbi lacus dui, tempor sed, euismod eget, condimentum at, tortor. Phasellus aliquet odio ac lacus tempor faucibus. Praesent sed sem. Praesent iaculis. Cras rhoncus tellus sed justo ullamcorper sagittis. Donec quis orci. Sed ut tortor quis tellus euismod tincidunt. Suspendisse congue nisl eu elit. Aliquam tortor diam, tempus id, tristique eget, sodales vel, nulla. Praesent tellus mi, condimentum sed, viverra at, consectetur quis, lectus. In auctor vehicula orci. Sed pede sapien, euismod in, suscipit in, pharetra placerat, metus. Vivamus commodo dui non odio. Donec et felis.
---	--	--

Důkaz (Název důkazu)

Abcd. Abcd. Abcd. Abcd. Abcd. Abcd. Abcd. Abcd. Abcd. Abcd. Abcd. Abcd.
Abcd. Abcd. Abcd. Abcd. Abcd. Abcd. Abcd. Abcd. Abcd. Abcd. Abcd. Abcd.
Abcd. Abcd. Abcd. Abcd. Abcd. Abcd. Abcd. Abcd. □

POZNÁMKA 2 (PUMPOVACÍ VĚTA)

Abcd. Abcd. Abcd. Abcd. Abcd. Abcd. Abcd. Abcd. Abcd. Abcd. Abcd. Abcd.
Abcd. Abcd. Abcd. Abcd. Abcd. Abcd. Abcd. Abcd. Abcd. Abcd. Abcd. Abcd.
Abcd. Abcd. Abcd. Abcd. Abcd. Abcd. Abcd. Abcd.

PŘÍKLAD 3 (PUMPOVACÍ VĚTA)

Abcd. Abcd. Abcd. Abcd. Abcd. Abcd. Abcd. Abcd. Abcd. Abcd. Abcd. Abcd.
Abcd. Abcd. Abcd. Abcd. Abcd. Abcd. Abcd. Abcd. Abcd. Abcd. Abcd. Abcd.
Abcd. Abcd. Abcd. Abcd. Abcd. Abcd. Abcd. Abcd.

Lemma 4 (Název definice)

Abcd. Abcd. Abcd. Abcd. Abcd. Abcd. Abcd. Abcd. Abcd. Abcd. Abcd. Abcd.
Abcd. Abcd. Abcd. Abcd. Abcd. Abcd. Abcd. Abcd. Abcd. Abcd. Abcd. Abcd.
Abcd. Abcd. Abcd. Abcd. Abcd.

Důsledek 5 (Název důkazu)

Abcd. Abcd. Abcd. Abcd. Abcd. Abcd. Abcd. Abcd. Abcd. Abcd. Abcd. Abcd.
Abcd. Abcd. Abcd. Abcd. Abcd. Abcd. Abcd. Abcd. Abcd. Abcd. Abcd. Abcd.
Abcd. Abcd. Abcd. Abcd.

Věta 6 (Pumpovací věta)

Abcd. Abcd. Abcd. Abcd. Abcd. Abcd. Abcd. Abcd. Abcd. Abcd. Abcd. Abcd.
Abcd. Abcd. Abcd. Abcd. Abcd. Abcd. Abcd. Abcd. Abcd. Abcd. Abcd. Abcd.
Abcd. Abcd. Abcd. Abcd. Abcd.

```
1 int main("cs acsa") // komentár
2 int main("cs acsa") // komentár
3 int main("cs acsa") // komentár
4 int main("cs acsa") // komentár
5 int main("cs acsa") // komentár
```

Zdrojový kód 6: C++

```
1 new object() // komentár
```

Zdrojový kód 7: JS

```
1 public static int main("cs acsa") // komentar
```

Zdrojový kód 8: C#

```
1 SELECT * FROM table_1; /* komentar */
```

Zdrojový kód 9: SQL

```
1 table_1 AND table_2;
```

Zdrojový kód 10: TutorialD

Závěr

Závěr práce v „českém“ jazyce.

Conclusions

Thesis conclusions in “English”.

A První příloha

Text první přílohy

B Druhá příloha

Text druhé přílohy

C Obsah elektronických dat

Na samotném konci textu práce je uveden stručný popis obsahu elektronických dat odevzdaných v systému katedry informatiky spolu s textem. Tato data jsou nedílnou součástí práce a tvoří (datovou) přílohu textu práce. Povinné položky struktury dat jsou:

text/

Adresář s textem práce ve formátu PDF, vytvořený s použitím závazného stylu KI PřF UP v Olomouci pro závěrečné práce, včetně všech (textových) příloh, a všechny soubory potřebné pro bezproblémové vytvoření PDF dokumentu textu (případně v ZIP archivu), tj. zdrojový text textu a příloh, vložené obrázky, apod.

README.*

Textový soubor (s příponou např. `.txt`) s informacemi o opakovatelném způsobu použití ostatních dat práce – typicky plně reprodukovatelný co nejúplnější funkční postup zprovoznění software vytvořeného v rámci práce, tzn. jeho případné instalace/nasazení a spuštění, včetně uvedení všech požadavků pro bezproblémový provoz; za zprovoznění software se nepovažuje zpřístupnění (např. po Internetu) již někde zprovozněného software.

Adresáře a soubory s veškerými ostatními autorskými daty práce (případně v ZIP archivu) – typicky spustitelné a další soubory software vytvořeného v rámci práce potřebné pro bezproblémový provoz software, případně jeho instalační program, a kompletní zdrojové texty software a další data nutná pro plně reprodukovatelné korektní vytvoření spustitelných souborů.

Dále mohou data obsahovat například:

- ukázková a testovací data použitá v práci nebo pro potřeby posouzení práce v rámci její obhajoby,
- položky bibliografie v elektronické podobě, příp. jiná relevantní literatura a dokumentace vztahující se k práci,

- cizí data (software) potřebná pro bezproblémové použití autorských dat práce (software), která nejsou standardní součástí předpokládaného (softwareového) vybavení uživatele.

U veškerých cizích obsažených materiálů jejich zahrnutí dovolují podmínky pro jejich veřejné šíření nebo přiložený souhlas držitele práv k užití. Pro všechny použité (a citované) materiály, u kterých toto není splněno a nejsou tak obsaženy, je uveden jejich zdroj, např. webová adresa, v bibliografii nebo textu práce nebo souboru README.*.

Rejstřík

výraz, [45](#)