Katedra informatiky
Přírodovědecká fakulta
Univerzita Palackého v Olomouci

DIPLOMOVÁ PRÁCE

Metody optimalizace přenosu dat v distribuovaném systému

2026

Martin Šlachta

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

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:	51
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:	51
Supplements:	electronic data in the storage of department of computer science
Thesis language:	Czech

Anotace

V diplomové práci se zabýváme optimalizacemi datového přenosu v distribuovaných systémech. Konkrétně se zaměřujeme na systémy pro hry více hráčů. Analyzujeme a popisujeme jejich problematiku a specifika. Následně představujeme optimalizace, které jsme implementovali. Mezi hlavní části patří náš vlastní protokol transportní vrstvy a vývoj našeho vlastního herního enginu pro implementaci hry pro více hráčů.

Synopsis

In this theses we focus on optimization of the data transfer in distributed systems. We focus on systems for multiplayer games. We analyse and describe their issues and specifics. Then we show our implementation of those optimization. Main parts are our own protocol in the transport layer and development of our own game engine to create multiplayer game for showcase.

Klíčová slova: distribuované systémy, C++, QUIC, optimalizace, videohra, herní engine

Keywords: distributed systems, C++, QUIC, optimalizations, video game, game engine

Děkuji vedoucímu této práce Mgr. Tomáši Urbancovi, Ph.D. za spolupráci, ochotu a čas, který mi věnoval při konzultacích. Dále bych rád poděkoval rodině a přátelům za podporu a motivaci.

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

Seznam tabulek

1 Úvod

Hry pro více hráčů jsou stále populárnější. Například na internetovém tržišti her Steam 9 z 10 nejhranějších her podporuje hru více hráčů a 6 z nich dokonce ani nepodporuje hru pro jednoho hráče. Kvůli rostoucí popularitě online her vznikly sporty v počítačových hrách, tzv. „e-sporty“, ve kterých se utkávají profesionální týmy proti sobě v kompetetivních hrách pro více hráčů. Kompetetivní hry jsou často rozděleny do zápasů a na jejich konci se provede evaluace hráčova skóre.

Dalším typem jsou masivní multiplayerové online hry, tzv. MMO, které zvládnou online světy pro tisíce hráčů. Používají poněkud odlišné principy a optimalizace, aby systém fungoval optimálně. Kompetetivní hry se soustředí na minimální odezvu a masivní online hry na zvládání co nejvíce hráčů současně pro co nejživější svět.

V práci se zaměřujeme právě na metody optimalizace datového přenosu v síťových systémech pro hry více hráčů. Charakteristiky zanalyzujeme a implementujeme různá řešení. V závěru práce představíme naše testovací scénáře pro měření optimality těchto technik. Zároveň popíšeme nástroje k měření a jejich způsob použití.

Nejprve v následující kapitole popíšeme distribuované systémy obecně: jak probíhá komunikace mezi dvěma procesy na dvou různých počítačích. Tyto procesy budeme skládat do distribuovaného systému. Uvedeme různé modely a atributy, které může komunikace nebo distribuovaný systém mít.

V navazujících kapitolách popisujeme samotnou hru. Představíme náš vlastní herní engine, který jsme vytvořili. Engine je navržen jako modulární monolit. Zároveň s ním popíšeme architekturu samotné hry. Začneme hrou pro jednoho hráče a návrh rozšíříme o hru více hráčů tak, aby byl přehledný a snadno se s kódem pracovalo.

Pro demonstraci různých technik jsme vytvořili hru. Jedná se o jednoduchou hru ve 3D prostoru, kde každý hráč má svou postavu, se kterou může pohybovat. Tu stavíme na vlastním herním enginu tak, abychom měli pod kontrolou různé algoritmy. Například jsme vytvořili vlastní protokol pro obousměrné posílání spolehlivých i nespolehlivých zpráv zvaný QUICr. Celý engine jsme navrhovali jako modulární monolit a snažíme se organizovat jednotlivé řešení problémů do modulů. Díky tomu je návrh intuitivní a snadno rozšiřitelný.

V poslední kapitole představíme měření, které jsme provedli a jejich výsledky. Ukážeme důležité metriky a uvidíme, že u vysokých frekvencí zpráv bylo třeba optimalizovat nejen datový přenos, ale i samotné kódování zpráv.

2 Distribuované systémy

V této kapitole představíme distribuované systémy. Nejprve si zadefinujeme, co distribuovaný systém je. Podíváme se, čím jsou specifické a jaké vlastnosti mohou mít. Následně hlouběji vysvětlíme jak takové systémy fungují. Distribuované systémy jsou často složité, proto představíme různé architektonické vzory, které i později při implementaci vlastního distribuovaného systému využijeme. Ty pomáhají s udržitelností systému.

Začneme pojmem počítačový systém. Ten se skládá ze služeb, každá implementovaná jako kolekce procesů, které dohromady plní společný úkol. Moderní systémy jsou ale čím dál větší a úkoly, které musí plnit, jsou složitější. To vedlo ke vzniku síťových systémů, ve kterých jsou procesy rozmístěné přes více počítačů a komunikují spolu posíláním zpráv. Výhod je hned několik. Část systému může být umístěna na počítači blíž uživateli pro snížení odezvy. Pokud jeden proces selže, může existovat jiný, který plní stejnou službu a může systém udržet v provozu. Konkrétní skupinou jsou distribuované systémy, které se skládají z mnoha procesů rozmístěných na více počítačích. Tyto procesy spolu aktivně spolupracují a navenek se jeví jako jeden celek.

Příkladem distribuovaného systému je World Wide Web, zkráceně WWW. Jedná se o informační systém, který umožňuje prohlížet, ukládat a odkazovat na dokumenty umístěné na internetu. Dokumenty mohou být například webové stránky, obrázky nebo videa a jsou uloženy na webových serverech. Odkazy na ně jsou ve formátu URL. Distribuovanost systému umožňuje snadné rozšíření, protože každý může jednoduše přidat vlastní server s dokumenty, které se tak stanou dostupné v systému. To zároveň rozloží zátěž přes více počítačů a systém tak zvládá miliardy požadavků denně. Současně se celý systém jeví jako jeden celek, který z URL adresy vyhledá server a vrátí dokument.

2.1 Počítačová síť

Distribuované systémy stojí na počítačových sítích. Ty umožňují komunikaci mezi dvěma procesy na dvou různých počítačích. Počítačová síť je skupina propojených počítačů, které si mezi sebou přenášejí data. Definujeme několik typů sítí, které se liší velikostí a provedením. Například lokální sítě (LAN) propojují až tisíce počítačů, které jsou geograficky blízko, typicky v rámci jedné budovy. Rozsáhlé sítě (WAN) propojují miliony různých zařízení po celém světě. Příkladem je rozsáhlá síť Internet.

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“. Díky propojovacím prvkům je možné poslat zprávu přes více vrcholů na cílový počítač. Příkladem takových prvků jsou přepínače, rozbočovače a opakovače. Tyto prvky mají za úkol třídit poslané zprávy a doručit je správnému koncovému bodu. Fungují tak podobně jako třídící centra pošty. Sítím, které využívají tyto prvky, se říká „přepínané sítě“. Stejně jako u pošty, musejí mít koncové body přiřazenou unikátní „síťovou adresu“. Koncové body

jsou například stolní počítače, mobilní telefony a jiná zařízení, na kterých běží komunikující procesy.

Dva počítače, přímo propojené fyzickým médiem, komunikují posíláním n-tic bytů zvané „rámce“. V přepínaných sítích mají rámce konkrétní formát, který pomáhá při hledání cesty v grafu, neboli „směrování“. Takový formátovaný rámec se nazývá „paket“. Skládá se z hlavičky, kde jsou informace ke směrování, jako je například síťová adresa, a těla, kde se nachází samotná zpráva.

2.1.1 Komunikace

V této části popíšeme, jak samotná komunikace posíláním zpráv funguje. Představíme, jaké nástroje a vzory používáme. Konkrétně nás bude zajímat, jak můžeme komunikaci co nejvíce skrýt. Dále zmíníme vlastnosti komunikace.

Ke skrytí komunikace v distribuovaných systémech používáme komponenty, kterým se říká „middleware“. Ty leží mezi aplikací a operačním systémem a poskytuje komunikační služby. Není závislá na žádné konkrétní aplikaci. Příkladem je distribuovaná služba DNS, která podle doménového jména vyhledá síťovou adresu, jako například adrese `www.seznam.cz` přiřadila dne 3. sprna 2026 síťovou IP adresu `77.75.77.222`.

Komunikace může mít různé vlasnosti. Například email je typický příklad „persistentní“ komunikace. Po odeslání si persistentní middleware zprávu uloží do té doby, dokud ji příjemce nepřijme. To znamená, že proces příjemce nemusí běžět v době, kdy odesílatel odesílá. Na druhou stranu máme „transientní“ komunikaci, kdy je zpráva uložena jen po dobu, kdy běží proces odesílatele a příjemce. To znamená, že pokud příjemce není dostupný, zprávu nikdy nepřijme.

Asynchronní komunikace znamená, že odesílatel po odeslání zprávy nemusí čekat na odpověď příjemce. Zpráva se dočasně uloží v middleware do doby, než se odešle. Na druhou stranu při synchronní komunikaci volající proces zastaví, dokud neobdrží odpověď z cílového procesu. Obecně definujeme tři různé momenty, ve kterých může proces odesílatele pokračovat v běhu. Zaprvé hned poté, co middleware převzal zprávu a zodpovědnost za její doručení. Zadruhé v moment, kdy byla zpráva doručena druhému procesu. Zatřetí až v moment, kdy druhá strana zpracovala požadavek a dostali jsme odpověď.

2.1.2 Modely pro komunikaci

Představíme dva konkrétní pohledy na komunikaci. Nejedná se o konkrétní implementace, ale pouze modely, které můžeme při vývoji middleware použít. Zároveň si řekneme, jaké vlastnosti výsledná komunikace může mít.

Prvním přístupem je Remote Procedure Call, který procesu umožňuje zavolat lokální proceduru s implementací na jiném počítači. Tím kompletně skrývá, že objekt, který jsme zavolali, je na jiném počítači. Pokud proces A zavolá proceduru na počítači B, proces A se pozastaví a začne se vykonávat nový proces na počítači B. Ten spouští zavolanou metodu. Jakmile metoda vrátí výsledek, pošle ho proces zpět na počítač s procesem A, který poté i s výsledkem pokračuje. Tento model je

synchronní a transientní. Cílem je, aby volání vypadalo jako lokální implementace a skryla tak komunikaci mezi počítači.

Jedná se o intuitivní řešení, ale přináší několik problémů. Dva počítače mají různý adresní prostor, proto je třeba vyřešit, jak bude procedura využívat ukazatele do svého adresního prostoru, nebo jestli tuto vlastnost zakáže.

Druhým přístupem je posílání zpráv. V tomto přístupu posílají procesy zprávy na logické cíle pomocí jména v systému, nikoliv fyzické adresy. Tento přístup méně schovává fakt, že procesy jsou rozmístěny na více počítačích. Abstrakce z fyzických adres na jména pomáhá jednoduchosti. Využívá se v architektuře publish-subscribe nebo občas v architektuře orientované na služby.

Opět je potřeba, aby se dva koncové body shodli na významu bitů jednotlivých zpráv. Většinou jsou v programu zprávy reprezentovány objektem, který je pro síť převeden na n-tici bitů, procesem zvaným „serializace“. V praktické části představíme serializaci podrobněji.

2.2 Architektury systémů

V praktické části představujeme náš ukázkový síťový systém, který umožňuje hru pro více hráčů. V návrhu využijeme známé vzory, které podrobněji představíme. Návrh systému rozdělíme na dvě části: softwarovou a systémovou architekturu. V softwarové architektuře se zabýváme komponentami a konektory mezi nimi. Druhá část architektury, která řeší role jednotlivých služeb, se nazývá „systémová architektura“. Mezi příklady patří peer-to-peer nebo klient-server.

Pro ilustraci rozdílu představíme známý příklad třívrstvé softwarové architektury: databázová, výpočetní a frontendová vrstva. Systémová architektura dále definuje databázi jako službu, která vůči výpočetní vrstvě plní roli serveru. Naopak služba pro replikaci ve skupině databázových serverů využívající model peer-to-peer má roli jak serveru, tak klienta. Typy si rozebereme podrobněji v této podkapitole.

2.2.1 Softwarová architektura

Nejprve se zaměříme na softwarovou architekturu. Zde řešíme komponenty a konektory mezi nimi. Představíme tři známé vzory: vrstvená architektura, architektura orientovaná na služby a publish-subscribe architektura.

Pokud komponenty organizujeme do vrstev, kde komponenta ve vrstvě N může volat rozhraní vrstvy $N - 1$, říkáme tomu „vrstvená architektura“. Příkladem je vrstva operačního systému, nad kterým je vrstva uživatelského prostoru. Vrstva $N - 1$ nemá možnost volat rozhraní vrstvy N. Díky tomu lze na sebe vrstvy snadno skládat.

Občas je možné, aby nižší vrstva volala vyšší. Takové volání by však mělo probíhat prostřednictvím rozhraní definovaného nižší vrstvou, které vyšší vrstva pouze implementuje. Tomuto principu se říká „obrácení závislostí“. Udržíme tak závislost N na $N - 1$ a $N - 1$ zůstane nezávislá. Příkladem je operační systém, který oznamuje událost aplikaci. Aplikace proto registruje funkci, která se

v případě události zavolá. Rozhraní funkce ale určuje vrstva pod ní: operační systém.

Nevýhodou vrstvené architektury je silná provázanost mezi vrstvami. Možnost je software organizovat do nezávislých entit, kde každá zapouzdřuje službu. Ty pak mezi sebou mohou volně komunikovat. Takovým entitám se říká: služba, objekt nebo mikroslužba. Na komponenty se můžeme dívat jako na objekty a konektory mezi nimi jsou volání metod neboli posílání zpráv. Tento přístup je relevantní pro distribuované systémy, protože instance objektů mohou být rozmístěny na více počítačích. K tomu lze použít například RPC, které jsme zmínili.

V případě služeb musí služba znát adresu nebo jméno jiné služby, kterou chce využívat. Někdy říkáme, že služby jsou „referenčně vázané“. Tuto závislost lze odstranit pomocí publish-subscribe architektury. V ní každá služba publikuje „události“ do „témat“. Služby mohou témata odebírat, a to znamená, že budou dostávat všechny události, které jsou publikované do daného tématu. Tuto službu distribuce události a správy témat zajišťuje „broker“, který má všem známou adresu. Odesílatel události neví, kdo na jeho událost zareaguje a jak.

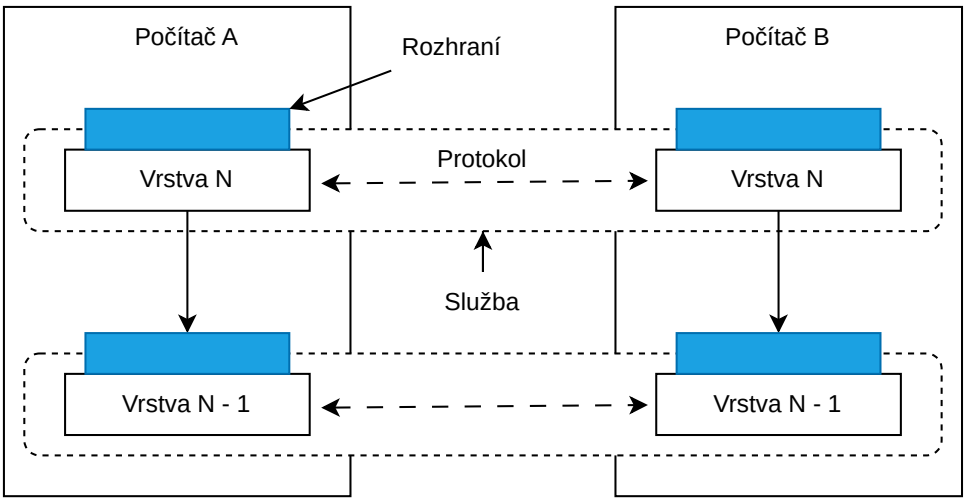
Tento princip lze využít u systémů pro hry více hráčů pro komunikační kanály. Klient hráče chce například napsat do lokálního kanálu města, ve kterém se v herním světě nachází. Komponenta pro chatovou službu tuto zprávu zařadí do správného kanálu a rozešle ji klientům, kteří tento kanál také odebírají.

2.2.2 Systémové architektury

Pro systémové architektury představíme a později využijeme dva modely: asymetrickou klient-server a symetrickou peer-to-peer architekturu.

Asymetrickou architekturou, kde je komponenta buď server nebo klient, se nazývá „klient-server“. Pouze klienti mohou serverům posílat dotazy a dostávat od nich odpovědi. Vztah je tedy asymetrický. Příkladem jsou webové servery a webové prohlížeče, které fungují jako klienti. Tento model se hodí například pro autoritativní server, který určuje stav hry a pouze jej replikuje klientům. Usnadňuje synchronizaci jednotlivých klientů a zvyšuje efektivitu, protože se účastníci nemusí shodovat, ale pouze přijmou pravdu ze serveru.

Symetrická architektura, kdy jsou obě strany rovny, se nazývá „peer-to-peer“. Znamená to, že obě strany mohou posílat požadavky a zprávy na druhou stranu. Tento přístup je užitečný, pokud dva procesy mají stejnou roli a jedná se tedy pouze o repliku té stejné služby. To je potřeba, pokud je zátěž na systém příliš velká a jeden počítač ji nezvládne. Například můžeme mít více počítačů, na kterých běží herní server stejné instance hry. Herní svět můžeme rozdělit na zóny a každý proces dostane svou zónu. Žádná zona není nadřazená jiné, proto spolu mohou procesy komunikovat modelem peer-to-peer a předávat si tak události, které v zónách nastaly. Například sousední zóny může zajímat, že se hráč blíží k okraji a za chvíli přejde do jiné zóny. Následně může původní zóna hráče „předat“ druhému procesu.



Obrázek 1: Vrstvená architektura

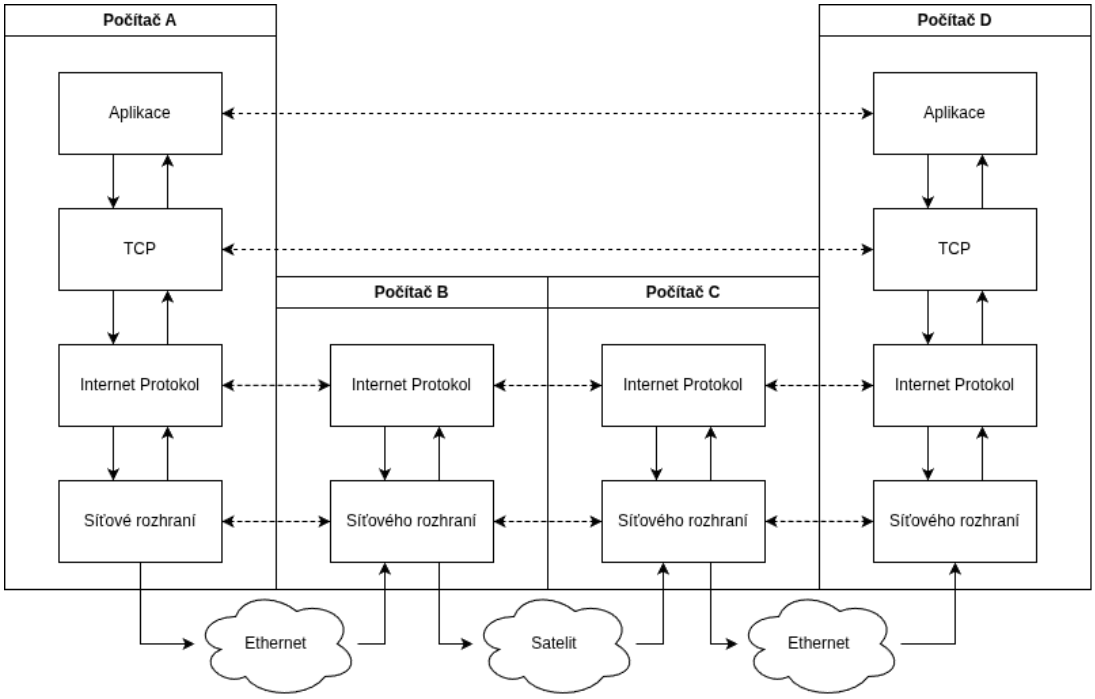
3 Protokoly

V této kapitole podrobně popíšeme, jak se komunikace realizuje. Aby si dva počítače rozumněly, musejí se shodnout na významu jednotlivých bytů rámců, které si posílají. To definuje „komunikační protokol“ - soubor pravidel pro výměnu informací mezi počítači. Představíme několik už existujících protokolů a jejich vlasnosti. V praktické části jsme vypracovali vlastní protokol optimalizovaný pro distribuované systémy her více hráčů.

Každý protokol poskytuje komunikační služby. Tyto služby rozdělujeme na dvě skupiny: služby, které před zahájením komunikace navazují spojení, a služby, které spojení nenavazují. V prvním případě musejí obě strany přijmout a navázat spojení a potencionálně se domluvit na jeho dalších parametrech. Jakmile jejich komunikace skončí, spojení se ukončí. Příkladem takové služby je telefonní linka. V druhém případě může odeslat zprávu kdykoliv a bez předchozího upozornění druhé strany. Příkladem takové komunikace je posílání emailu.

3.1 Rodina protokolů TCP/IP

Rodina protokolů pro komunikaci v síti Internet se nazývá TCP/IP. Jedná se o více komunikačních protokolů, organizovaných do vrstev. Tuto architekturu vidíme na obrázku 1. Každá vrstva má svůj konkrétní význam a jinou zodpovědnost. Tuto zodpovědnost pak plní služba v dané vrstvě. V jedné službě vidíme dva koncové body na dvou počítačích. Oba poskytují rozhraní pro svou instanci. To, že jsou instance fyzicky oddělené, je skryté právě za komunikační službou. Vrstva N zapisuje a čte z vrsvy $N-1$ a neřeší, jak si dva koncové body ve vrstvě $N-1$ zapsané informace předají, aby je druhá strana mohla číst. Způsob, jakým



Obrázek 2: Komunikace vrstev TCP/IP

si koncové body ve vrstvě předávají informace, je právě protokol.

Rodina protokolů TCP/IP se řídí principem „end-to-end“, který říká, že body mezi odesílatelem a příjemcem, jako směrovače a přepínače, by měly být co nejjednodušší. Spolehlivost a navázání spojení musejí implementovat až dva koncové body. Případné ztráty paketu musí odesílatel zjistit a ztracené pakety odeslat znovu.

Ztráta paketu nastává, když je některý z přepínačů na cestě mezi odesílatelem a příjemcem, zahlcený. Přepínač si přijaté pakety ukládá do fixně velké fronty, ze které poté postupně odebírá a snaží se najít další vhodný uzel, kam daný paket poslat. Pro případ přehlcení fronty má přepínač definovanou politiku[1]. Běžně se používá politika „tail-drop“, při níž je nově příchozí paket, který se nevejde do fronty, zahozen. V tento moment se paket ztrácí.

Název rodiny protokolů se skládá ze dvou důležitých protokolů: IP (Internet Protocol) a TCP (Transmission Control Protocol). IP umožňuje komunikaci libovolné dvojice uzlů počítačů v propojených sítích. Definuje formát adresy koncových bodů a směrování. Protokol nenavazuje spojení ani nezaručuje doručení. TCP zajišťuje spolehlivý obousměrný přenos dat mezi procesy na dvou počítačích (ne nutně různých).

Popíšeme vrstvy, ze kterých se rodina TCP/IP skládá. Konkrétně jsou čtyři: aplikační, transportní, síťová a síťové rozhraní. Každá vrstva obsahuje množinu protokolů a pro různé situace můžeme protokoly ve vrstvách kombinovat. Na obrázku 2 vidíme, jak mezi sebou jednotlivé vrstvy komunikují. Aplikace na počítači

A zapíše zprávu do transportní vrstvy, kterou počítač *D* ze stejné vrsvy přečte. Vrstva zná pouze vrstvu pod sebou. Plnou čarou vidíme datový tok zprávy, jak putuje přes jednotlivé vrstvy. Přerušovaná čára reprezentuje protokol. 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.

Aplikační
: První nejvyšší je „aplikační“ vrstva, ve které se nachází protokoly přímo pro aplikace. Příkladem protokolů jsou FTP pro přenos souborů, SMTP pro emailovou komunikaci, HTTP pro přenos hypertextových dokumentů nebo gRPC pro implementaci RPC. Tato vrstva je rozbalena až na koncovém bodu, protože neobsahuje žádné podstatné informace pro směrování. To je vidět i na obrázku 2.

Transportní
: Druhá vrstva je „transportní“. Tato vrstva, stejně jako aplikační, je rozbalena až na koncových bodech. Stará se o to, jaké pakety přišly, v jakém pořadí a jestli nejsou poškozené. Příkladem protokolů jsou TCP, UDP nebo QUIC.

Síťová
: Pro adresaci slouží „síťová“ vrstva. 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 komunikovat přes internet.

Síťové rozhraní
: Nejnižší vrstva, která se stará o přenos přes fyzické médium. Pro různé fyzické média existují různé protokoly. Příkladem jsou Ethernet, Token ring a další.

3.2 Protokoly v aplikační vrstvě

Nejvyšší v rodině protokolů TCP/IP je aplikační vrstva s protokoly jako HTTP, gRPC nebo SMTP. Aplikačním protokolům, které nejsou specifické pro konkrétní aplikaci, se říká middleware. Mezi takové řadíme právě HTTP a gRPC, které jsou pro obecné posílání zpráv a poskytují jednoduché rozhraní pro aplikace.

3.2.1 HTTP

Velmi používaný protokol v systému World Wide Web je HTTP, neboli Hyper Text Transfer Protokol. Slouží především pro přenos hypertextových dokumentů, jako například HTML. Původně byl navržen pro komunikaci mezi prohlížečem, neboli klientským agentem, a webovým serverem. Dnes se využívá i pro API

dotazy. Protokol je klient-server, to znamená, že jedna strana je klient a druhá server. Pouze klient může na server posílat dotazy a server posílá zpátky odpověď. Server nemůže poslat dotaz na klienta. I když už jsou způsoby, jak může server posílat alespoň události.

Protokol je textový a snadno čitelný člověkem. Navíc je rozšiřitelný pomocí hlaviček. Protokol je bezestavový. To znamená, že server mezi dvěma dotazy nemá žádnou vazbu. Podporuje ale relace, kdy v dotazu můžeme serveru sdělit identifikátor kontextu, který chceme použít. Server si musí tyto kontexty ukládat. Je request-reply. V první verzi, HTTP/1, bylo třeba otevřít nové TCP připojení pro všechny. Ve verzi HTTP/2 je podpora pro multiplexní dotazování. To znamená, že jedno navázané spojení lze využít na více dotazů. To může ušetřit čas, protože se tak vyhneme navazování spojení. Funguje tak, že přes jedno spojení lze odeslat více dotazů a odpovědi musejí chodit ve stejném pořadí. Už zde je vidět možný problém, který se nazývá „head-of-line“ blokování. Znamená to, že pokud první dotaz trvá dlouho a přitom je nepodstatný, a druhý dotaz krátký a podstatný, může první dotaz blokovat ten druhý. Klienti by se měli snažit nejprve odeslat jednoduché dotazy a teprve poté ty zdlouhavější.

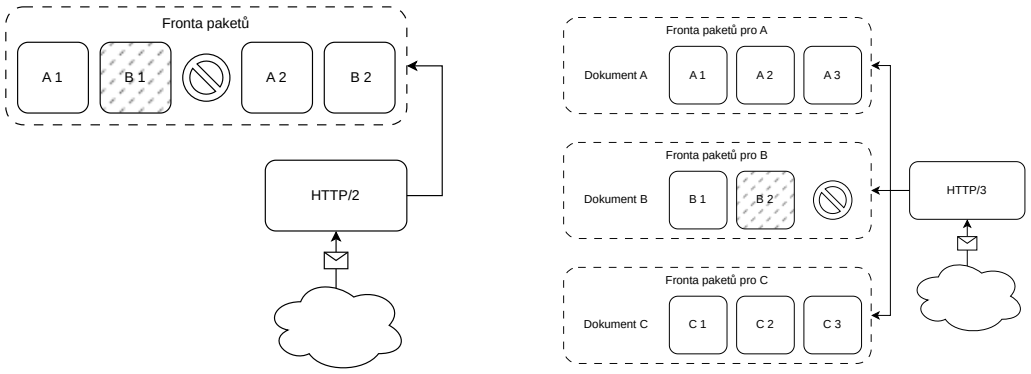
3.2.2 HTTP/3

Zajímavou novinkou je HTTP/3, která je založena na protokolu QUIC, který je založen na UDP namísto TCP. Podporuje více „multiplexových“ proudů už na transportní vrstvě. To znamená, že agent klienta může v jednom HTTP/3 stahovat najednou vícero dokumentů nezávisle. Pro jednu stránku je třeba stáhnout CSS, JS i HTML dokumenty. V HTTP/2 se všechny stahují jedním proudem střídavě, což způsobuje „head-of-line“ blokování. Na obrázku 3 vidíme porovnání TCP v HTTP/2 a QUIC v HTTP/3. V prvním případě stahujeme dva dokumenty: dokument A složen z částí $A1$ a $A2$ a dokument B složen z částí $B1$ a $B2$. Pokud server odesílá části v pořadí $A1$, $B1$, $A2$ a $B2$, a část $B1$ se nedoručí, nelze začít zpracovávat ani části $A2$ a $B2$, přestože byly doručeny. Protokol totiž čeká na doručení části $B1$. Nezná závislosti mezi částmi a předpokládá tedy úplné seřazení tak, jak byly odeslány. Na druhou stranu protokol QUIC může mít pro každý dokument vlastní frontu a tím blokování kompletně eliminovat.

3.3 Protokoly v transportní vrstvě

Protokoly transportní vrstvy poskytují end-to-end komunikační služby pro aplikace. Služba může mít různé vlastnosti, jako komunikace s navázáním spojení, spolehlivost doručení, zamezení zahlcení sítě nebo multiplexing.

Nejčastějším protokolem je TCP, který naváže spojení a zaručuje spolehlivé doručení dat jako proud bytů. Alternativou je UDP, které nezaručuje doručení, ale je vhodnější pro streamovací služby, které to neomezuje. Poslední příklad, který zmíníme, je protokol QUIC od Google, který poskytuje spolehlivou komunikaci přes navázané spojení a s podporou multiplexování. Byl důležitou inspirací pro náš vlastní protokol z praktické části.



(a) HTTP/2

(b) HTTP/3

Obrázek 3: Vizualizace HTTP/2 blokování

3.3.1 TCP

Nejběžnější protokol je TCP. Při použití mezi sebou dva koncové body vytvoří spojení s obousměrným proudem bytů. Protokol garantuje spolehlivé doručení dat v pořadí, ve kterém byly odeslány. Zároveň řeší zahlcení sítě udržováním okna paketů, které jsou v oběhu a redukuje jeho velikost v případě, že je síť přehlcená. To pozná tak, že pakety čísluje a příjemce musí za každý paket odeslat potvrzení, že paket přijal. Pokud odesílatel toto potvrzení do časového limitu od odeslání nedostane, odešle paket znovu. Pokud nedostává potvrzení pro příliš velkou část okna, může toto okno zmenšit. Tím sníží i zatížení sítě.

Pro navázání spojení a následnou komunikaci musejí oba procesy vytvořit koncový bod. Velmi často používané rozhraní je přes „soket“. Jedná se o objekt, který se chová jako soubor. Odesílatel do něj zapisuje data, která chce odeslat a příjemce je z něj může přečíst. Na jednom počítači může být otevřeno více soketů a jsou identifikované „portem“. Ten slouží pro směrování paketů správnému soketu, ze kterého příjemce čte. Pro navázání spojení musí jedna strana spojení iniciovat a druhá jej musí přijmout.

Popíšeme si, jak TCP zajišťuje spolehlivost pomocí odesílání potvrzení o přijetí. Pokaždé, kdy příjemce dostane paket, odešle druhému koncovému bodu zprávu označenou jako „ACK“ s číslem paketu. V případě, že odesílatel potvrzení o přijetí nedostane, odešle paket znova. Příjemce si u sebe postupně skládá seřazený proud bytů. Jakmile je na socketu seřazená posloupnost bytů, umožní ji ze soketu přečíst.

V protokolu na transportní vrstvě dochází k head-of-line blokování. Pokud jedna strana odešle proud bytů po částech A, B a C, ale příjemci dorazí nejprve části C a B, nebude moci příjemce ze soketu nic přečíst, dokud nepřijde i část A. Většinou je toto chování žádoucí, existují však situace, například v počítačových hrách, kdy představuje problém. Uvažujme například posílání zpráv o pozici hráče. Pokud jsou odeslány pozice pro časy t_1, t_2 a následně pro čas t_3, tak

by případně ztracená zpráva t_1 blokovala příjemce, který by ji v zápětí přepsal zprávou pro čas t_3. Lepší by bylo, aby seřazení definovala až aplikace, která by ihned začala zpracovávat pozici pro čas t_3 hned a docílila tak nižší odezvy mezi serverem a klientem.

Nevýhodou pro nás je, že přenáší proud dat. Jinými slovy, v protokolu není proud rozdělen na jednotlivé bloky se zprávou. Tuto logiku si musíme implementovat sami. Proto jsme v praktické části vytvořili vlastní protokol, který umí v proudu zprávy správně oddělit.

Pro šifrovanou komunikaci je potřeba použít další protokol, například TLS. Ten opět vyžaduje „handshake“, protože se strany musejí dohodnout na tajném klíči.

3.3.2 UDP

Méně spolehlivá alternativa je UDP, neboli User Datagram Protocol. Ten komunikuje posíláním „datagramů“, na rozdíl od proudu bytů, jako je tomu v případě TCP. Znamená to, jednomu systémovému volání pro čtení ze soketu odpovídá právě jedno volání pro zápis provedené odesílatelem. Protokol nenavazuje spojení a nezaručuje doručení datagramů.

Je vhodný jako základ pro vlastní transportní protokol. Například protokol QUIC, který podrobně popíšeme níže, ho využívá pro vlastní implementaci spolehlivosti a multiplexing. My jsme na něm založili vlastní protokol, inspirovaný QUIC, který umožňuje aplikaci odeslat zprávu i nespolehlivě, například pokud se jedná o pozici hráče.

3.3.3 QUIC

QUIC[2] je spolehlivý protokol transportní vrstvy od společnosti Google, který poskytuje multiplexní šifrovanou komunikaci založenou na TLS 1.3. Multiplexní komunikace znamená, že v jednom navázaném spojení můžeme vytvořit více proudů bytů, ať už jednosměrných nebo obousměrných. To značně redukuje problém ahead-of-line blokování. Naopak protokol TCP má vždy právě jeden obousměrný proud.Protokol QUIC je pro nás důležitý, protože popisuje implementaci spolehlivosti, handshake a další vlastnosti nad protokolem UDP. V pozdější kapitole představíme náš protokol, který je protokolem QUIC inspirovaný a dále ho upravuje. Mezi naše úpravy patří možnost odeslání zpráv nespolehlivě a s různým seřazením. V této části podrobněji popíšeme části protokolu QUIC, které jsou pro náš protokol důležité.

Koncové body komunikují posíláním paketů. Pakety obsahují rámce, které dělíme na ovládací a proudové. Ovládací rámce slouží pro ovládání koncových bodů a vlastností navázaného spojení: otevírání a zavírání proudů, změna stavu nebo udržování spojení naživu. Proudové rámce obsahují aplikační data. Stejně jako TCP, i QUIC přenáší data jako proud bytů a oddělení jednotlivých zpráv je nutné implementovat zvlášť.

Popíšeme charakteristiky navázaného spojení v QUIC. Obě strany mají unikátní ID, které je specifické pro navázané spojení. Posílané pakety obsahují ID příjemce i ID odesílatele. Tyto unikátní ID používají koncové body k autentizaci. V protokolu TCP se ke stejnému účelu využívá IP adresa.

Problém nastává, pokud se IP adresa jednoho koncového bodu změní. To například nastává jakmile přepneme z mobilních dat na Wi-Fi. V ten moment by u TCP nastal proces vypršení relace, kdy obě strany zjišťují, že už se nevidí a spojení by ukončily. Následně by se původní iniciátor komunikace, nyní s novou IP adresou, pokusil spojení opět navázat. To by znamenalo zopakovat proces handshake.

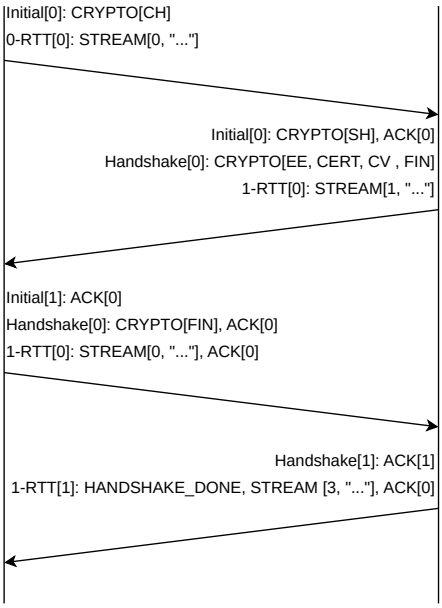
V případě QUIC je možné se opakovanému navazování spojení vyhnout. Jakmile příjemci dorazí paket na původní ID, ale z nové IP adresy, provede authentizaci a pokud je úspěšná, přenastaví IP adresu druhé strany.

Aby bylo možné v protokolu provést autentizaci, využívá QUIC kryptografii. Konkrétně už v základu implementuje TLS 1.3 pro šifrovanou komunikaci. V případě TCP by TLS nebo jiný šifrovací protokol stál ještě nad TCP protokolem. Po provedení handshake v TCP je potřeba udělat druhý handshake a shodnout se na tajném klíči. V případě QUIC představuje navázání spojení a domluva na tajném klíči jednu výměnu. Další zajímavostí je, že ve verzi TLS 1.3 je handshake zkrácen, protože není třeba se domlouvat na použitém šifrovacím algoritmu, jako tomu bylo v předešlích verzích. Nová verze má definovaných pouze pár možných algoritmů a iniciátor rovnou posílá svou část veřejného klíče pro každý možný algoritmus. Druhá strana si pak jeden algoritmus vybere.

Paket v QUIC se skládá z hlavičky a těla. Pakety mají různé typy, které popíšeme později. Prozatím předpokládejme, že v hlavičce je ID odesílatele a ID příjemce. Taktéž je každý paket očíslovaný. Pokud dva pakety v jedné instanci spojení od stejného odesílatele mají stejné číslo, tak jsou považovány za identické. To znamená, že každý další příchozí s již zpracovaným číslem může příjemce zahodit. Každý paket obsahuje rámce. Ty mají také typ. Nejčastěji bude v proudu rámec typu STREAM, který obsahuje data odeslaná aplikací. Příjemce si ze STREAM rámců lokálně skládá proud bytů. Každý takový rámec totiž obsahuje informace, o který úsek v odeslaném proudu, se jedná. Příjemce tak může snadno zjistit, která část proudu mu ještě chybí.

Popíšeme, jak celý proces handshake vypadá. Diagram můžeme vidět na obrázku 4. Vidíme dva koncové body vyjádřené svislými čarami. Ty se mezi sebou domlouvají na svých ID a tajném klíči.

Nejprve popíšeme domluvu na unikátních ID. Iniciátor si vygeneruje svoje ID a posílá rámec, kde v hlavičce je jeho ID a ID příjemce zvolí dočasně náhodně. Jakmile rámec dorazí a příjemce zjistí, že tohle ID odesílatele nemá v registru, vytvoří novou instanci spojení ve stavu ReceivedHello. Přiřadí mu dvě ID - jako první to, co vybral iniciátor a druhé si vybere sám. Od té doby všechny pakety, které ve spojení odešle, budou mít ID odesílatele právě jeho zvolené ID. I přesto si dočasně ponechává ID, které zvolila druhá strana. Je to řešení případu, kdy druhá strana odeslala více paketů ještě před tím, než se dozvěděla zvolené ID.



Obrázek 4: QUIC handshake

Všimněme si teď různých typů paketů: Initial, Handshake, 0-RTT a 1-RTT. Liší se především v síle šifrovaní. Pakety typu Initial a 0-RTT se používají v případě, kdy ještě není domluvený tajný klíč a jsou snadněji dešifrovatelné. Handshake a 1-RTT používají plné šifrování. Význam Initial je zřejmý a 0-RTT se používá pro rychlé sdělení informací druhé straně ještě před navázáním spojení. Není ale vhodné ale do takového paketu ukládat tajné informace, které nechceme, aby četla třetí strana. Zároveň QUIC umožňuje poslat více paketů v jednom datagramu. Proto v diagramu vidíme, že iniciátor posílá paket Initial i 0-RTT najednou.

4 Hra

V této kapitole představíme hru, kterou jsem vyvinuli pro demonstraci různých technik a jejich měření. Ve hře má každý hráč svou postavu, se kterou může volně pohybovat ve 3D prostoru. Hra simuluje na postavách hráčů realistickou fyziku. Pokud se do hry připojí více hráčů, tak se navzájem vidí. Každý hráč spustí program klienta, který zobrazuje stav hry a postavy hráčů v 3D prostoru. V další kapitole porovnáme plynulost pohybu a rychlost odezvy, které jsou pro hratelnost důležité.

V první části popíšeme architekturu hry a našeho enginu. Následně rozšíříme a jinak upravíme tento model pro hru více hráčů. Vytvoříme tak distribuovaný systém.

Naše první implementace využívá model klient-server, kde máme dva různé programy: server a klient. Podrobně popíšeme, které komponenty jsme přidali, které pouze přesunuli do programu serveru a které naopak ponechali v programu klienta.

Mezi hlavní prvky patří náš vlastní komunikační protokol transportní vrstvy: „QUICr“. Je to obousměrný protokol pro posílání zpráv jako n-tic bytů, který se inspiruje protokolem QUIC, ale upravuje ho pro lepší využití pro hry. Náš protokol lépe zachycuje závislosti mezi zprávami a umožňuje i posílat zprávy nespolehlivě. Díky tomu se úplně zbavuje ahead-of-line blokování.

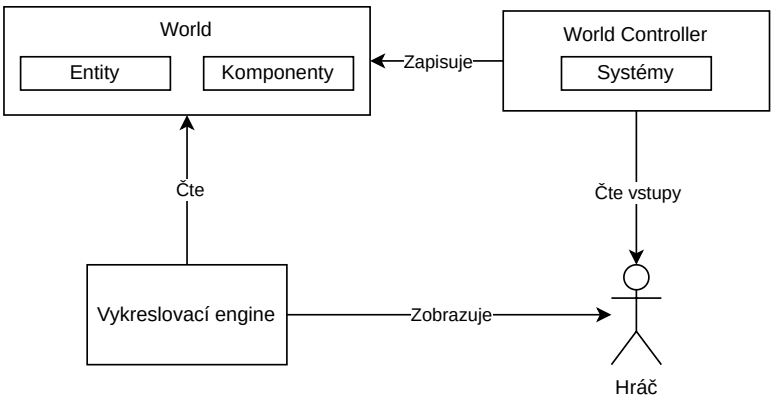
Zjistili jsme, že důležitým vylepšením je vlastní serializace primitivních zpráv. Pokud například obsahuje pouze seznam pozic, může být vlastní serializace rozdíl mezi hratelnou a nehratelnou odezvou. Popíšeme, jak jsme rychlost měřili a jaké metody jsme pro vlastní serializaci použili.

Nakonec představíme i architekturu peer-to-peer, kdy umožníme, aby v systému bylo více spolupracujících serverů, které si mezi sebou budou rozdělovat práci. Zároveň porovnáme i případ, kdy není žádný autoritativní server a hráči se mezi sebou synchronizují samovolně.

4.1 Engine

Pro hru jsme vyvinuli vlastní herní engine tak, abychom mohli celý systém do hloubky upravovat. Implementace je v jazyce C++, protože je v tomto jazyce napsáno spousty nástrojů a knihoven právě pro vývoj her. Díky tomu se vývoj značně akceleroval. Zároveň je to vhodný jazyk pro práci na nižší úrovni, a to je důležité pro některé metody optimalizace. Engine je implementovaný jako modulární monolit. Dosavadní architekturu vidíme na obrázku 5. Jednotlivé komponenty si popíšeme.

Mezi hlavní komponenty patří „stav“ hry a „řídící logika“. Stav obsahuje množinu entit, které jsou ve hře a jejich vlasnosti a atributy. Entity jsou objekty v herním světě, jako postava hráče, nepřátele nebo interaktivní prvky jako truhla s pokladem. Entita má přiřazené vlastnosti, které ji dávají stav. Entita nepřítele může mít vlastnost "množství životů"a truhla vlastnost "poklad", která definuje, co je v truhle obsaženo. Stavu hry se někdy říká pouze „svět“.



Obrázek 5: Architektura hry pro jednoho hráče

Druhá část je řídící logika, která iterativně mění stav hry. Tato změna může být pouze na základě stavu světa, například posune objekt, který má nenulový vektor zrychlení. Mimo to může řídící logika reagovat na údalosti. Například stisk tlačítka W je událost, který vyvolá změnu vektoru zrychlení hráče, který tlačítko stiskl. V důsledku toho se postava hráče pohne.

Každá iterace řídící logiky trvá přibližně stejně dlouho, většinou 1/60 sekundy nebo 1/24 sekundy. Vyšší frekvence znamená nižší odezvu a lepší plynulost. Nižší frekvence znamená menší výpočetní náročnost. Pro pomalejší hry bez rychlých akčních pasáží se využívá frekvence 24Hz a u kompetentních her i 120Hz [1].

4.1.1 Fyzický engine

Hra je 3D a chtěli jsme simulovat otevřený svět, ve kterém platí zákony fyziky. Pro simulaci jsme využili existující fyzický engine „Jolt“ a zabalili ho do modulu. Jolt je populární projekt, který je využíván například v Horizon: Forbidden West nebo Death Stranding 2.

Pro použití jsme vytvořili instanci fyzického světa, do kterého naskládáme objekty a jejich vlastnosti, jako hmotnost a tvar. Zavoláním metody pro aktualizaci engine posune stav entit (zrychlení a pozice). Vše jsme obalili do `JoltPhysicsWorld` .

Jolt umí využít více vláken procesoru a podporuje „rollback“, který umožňuje vracet stav v historii simulace dozadu. Je to důležitá věc pro hry více hráčů a obecně distribuovaných systémů, pro řešení desynchronizace.

[1] Článek od Riot Games ukazuje, jak důležitá je odezva u kompetetivních her. U profesionálních hráčů je poznat i rozdíl mezi 120Hz a 240Hz. https://www.riotgames.com/en/news/peeking-valorants-netcode

4.1.2 Vykreslování

K vykreslování jsme použili náš vlastní engine. Pro vykreslení 3D objektu je potřeba popsat jeho povrch jako množinu polygonů. Této množině se říká „mesh“. Vykreslení jednoho snímku lze v enginu popsat jako graf operací. Běžně bude obsahovat operaci, která seznam mesh vykreslí do obrázku. Tento celý proces lze shrnout jednou operací, protože o celý algoritmus a logiku se stará grafická karta. Vývojář jen popisuje mesh a například jakou má mít barvu apod.

Mezi operacemi v grafu definujeme závislosti. Díky tomu můžeme definovat operaci, která vykreslí 3D mesh a druhou operaci, která je na ni závislá, a která přes obrázek vykreslí uživatelské rozhraní.

V našem případě máme právě operaci pro vykreslení statických objektů, jako povrchu, po kterém se hráči pohybují apod. Druhá operace vykreslí samotné hráče a jiné dynamické objekty. Poslední operaci vykreslí uživatelské rozhraní obsahující různé nástroje pro různé ladění programu.

4.1.3 Entity-Component-System

V programu využíváme architekturu entity-component-system, zkráceně ECS, která je pro hry běžná. Entity už jsme popsali výše. Ke konkrétním entitám vážeme instance komponent, které entitě dají stav. Většinou je komponenta přiřazena právě jedné entitě. Poslední částí jsou systémy, které reprezentují řídící logiku. Systém je definován funkcí, která prochází entity, které splňují definovanou podmínku a mění stav její komponent. Tato podmínka většinou pouze omezuje na entity, které mají přiřazené komponenty pro určité typy. Systém může například simulovat fyziku, a to přičítáním zrychlení k pozici. K tomu definuje podmínku, která procházenou množinu omezí na entity, které mají potřebné komponenty. Těmi jsou komponenta transformace a komponenta zrychlení.

Motivací této architektury je jednoduché skládání různých entit, které jsou pro rozmanité herní světy typické. Pomocí stromu dědičnosti bychom tyto různé scénáře hůře skládali. Další výhoda je zlepšení výkonu. Protože si engine může poskládat entity a jejich komponenty do paměti jak chce, může umístění optimalizovat pro definované systémy tak, aby iterování bylo co nejrychlejší. Tomuto se říká problém lokality.

Pro ECS využíváme knihovnu „Entt“, která usnadňuje definici entit, přiřazování komponent a definici systémů. Entita je v této knihovně pouze unikátně identifikační číslo a komponenta je libovolný typ. V registru pak můžeme vytvářet „pohledy“ na n-tici typů komponent, které jsou seznam právě všech entit v registru, které všechny tyto komponenty mají. Ten můžeme procházet a libovolně měnit atributy komponent. Příklad použití vidíme v 1, kde nejprve vytvoříme registr do kterého přidáme novou entity a přiřadíme ji komponentu typu Transform. Nakonec definujeme pohled na všechny entity, které mají komponentu obou typů: Transform i CharacterBody. Před tento pohled iterujeme a v každé iteraci kopírujeme. Představíme komponenty, které jsme pro hru definovali a proč:

Transform

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

Zdrojový kód 1: Příklad použití knihovny Entt

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. Komponenta neobsahuje body, ale pouze odkaz na buffer, ve kterém je najdeme a kde. 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 zrychlení, které na entitu budeme aplikovat. Systém pro tuto komponentu bude ve fyzickém enginu.

4.1.4 ImGui

Některé metriky jsme chtěli zobrazit přímo v klientovi pro hru. Jsou to například příchozí a odchozí množství dat klienta. Tyto data neukládáme nikam do databáze a musíme si je spravovat lokálně. Proto jsme v klientovi potřebovali zobrazit základní uživatelské rozhraní. Zvolili jsme knihovnu ImGui a ImPlot, které se integrují přímo do vykreslovacího enginu. Pro knihovnu jsme přidali jednu novou úlohu do vykreslovacího grafu.

S knihovnou se pracuje procedurálně, nikoliv objektově. Každý snímek je třeba definovat celé rozhraní znova. Pro definici voláme různé funkce, kde každá definuje v rozhraní některý z možných prvků. Například funkce `InputText` zobrazí vstup pro text. Důležitá je funkce `Begin("Název okna")`, která zobrazí okno, do kterého můžeme 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.

5 Hra více hráčů

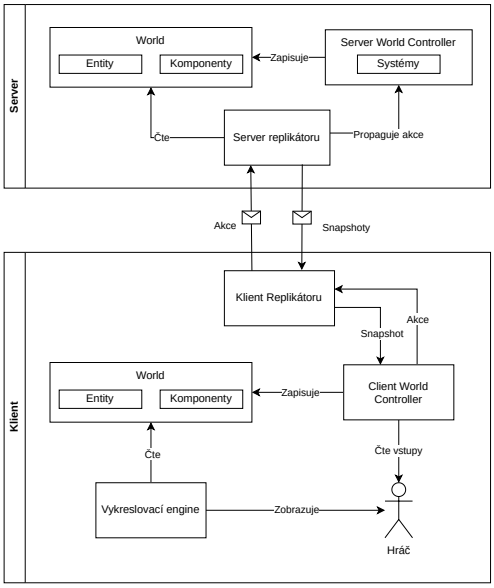
V předchozí části jsme představili základní implementaci hry pro jednoho hráče. V této části ukážeme, jak jsme hru rozšířili na síťový systém a umožnili hru více hráčů. To znamená, že více hráčů se může připojit do stejné instance hry a navzájem spolu interagovat.

Jako první jsme implementovali model klient-server. Diagram systému vidíme na obrázku 6. Stav hry řídí autoritativní server, který ho replikuje klientům posíláním snapshotů. Klient zobrazuje stav hry hráči, stejně jako v případě hry jednoho hráče. Autorita serveru zjednodušuje konzistenci a detekci podvádění. Klienti na server neposílají svůj stav, ale pouze akce, které chtějí provést. Příkladem může být akce pohybu dopředu, kterou vyvolal hráč. Server má možnost akce odmítnout a libovolně interpretovat, protože konečné slovo má právě server. Server rozesílá klientům zprávy zvané snapshoty, které obsahují aktuální stav hry (např. pozice entit) a události pro jednorázové akce (např. událost o konci hry).

Druhá varianta je model peer-to-peer. Ta je složitější z hlediska konzistence. Účastníci si mezi sebou posílají akce a každý si udržuje svůj stav, který ale nikomu nereplikuje. Každý se tak chová jako server v modelu klient-server. Je potřeba, aby každý účastník měl kompletní historii akcí všech ostatních v systému ve správném pořadí. Tomuto se říká „event-sourcing“. V případě, že se historie liší, dochází k nekonzistenci.

Druhá komplikace peer-to-peer je, že aplikace stejné akce na stejný stav musí mít vždy stejný výsledek. Pokud by dva procesy ve stejném stavu po aplikaci stejné akce na stav dospěli do jiného stavu, dochází k desynchronizaci. Problémové oblasti jsou generování pseudo-náhodných čísel a integrace. Výsledek integrace je často závislý na frekvenci. Například fyzický engine postupně integruje pozice entit podle derivace, která je reprezentována vektorem zrychlení. Každá integrace musí definovat, o jak velký časový krok se jedná (tzv. delta-time) a všichni účastníci se na něm musejí shodnout. Různé délky by rychle způsobily desynchronizaci. Zároveň všechny pseudonáhodné generátory musejí mít stejný počáteční seed. Je třeba si dát pozor na aritmetriku s plovoucí desetinou čárkou, která na různých platformách může dopadnout trochu jinak. I malé rozdíly se mohou rychle projevit.

Rozšířenou architekturu vidíme na obrázku 6. Hned si všimneme, že se svět rozdvojil na instanci na serveru a instanci na klientovi. Jsou to právě tyto dvě instance, které se snažíme posíláním zpráv synchronizovat. Řídící logiku jsme rozdělili na Server World Controller a Client World Controller. Všimněme si, že ovladač na klientovi ne nutně obsahuje systémy, které by měnili stav hry, ale pouze kopíruje to, co mu server poslal ve zprávě pro snapshot. Na druhou stranu posílá akce, které přečetl od hráče. Oba ovladače ale zapisují do své instance světa.



Obrázek 6: Architektura hry pro více hráčů

5.1 Server

Představíme, jak vypadá architektura pro server a komponenty, ze kterých se skládá. Stejně jako hra jednoho hráče si ukládá stav hry, tedy instanci `World`. Navíc pro hru více hráčů jsou komponenty „replikátor“. Ten posílá snapshoty a replikuje tak stav na serveru. Může posílat i jednorázové události.

Replikátor pracuje pouze s entitami a komponentami. Proto můžeme snapshot zjednoduššit na seznam entit s komponentami. Dále bude obsahovat seznam entit, které jsou nové a které naopak už mají zmizet. Stará se jen o to, jak tento stav synchronizovat pomocí posílání zpráv.

Různé entity na serveru jsou pro některé hráče jinak důležité. Například ty, které jsou daleko, nemusíme synchronizovat, protože po vykreslení by nebyly vidět. Komponenta, která se o to stará je „manažer zájmu“.

Obě tyto komponenty, společně se stavem a řídící logikou, jsou ve třídě `WorldServerController`. Architektura tak využívá komponenty ze hry pro jednoho hráče a skrývá distribuovanost pomocí vrstvené architektury.

5.1.1 Registr klientů

Komponenta, která spravuje a udržuje spojení s klienty hráčů, se nazývá „registr klientů“. Slouží jako sifon pro všechny zprávy, které klienti odesílají. Zároveň z ní lze zjistit kdo se právě připojil a kdo odpojil. K tomu slouží metody: `popDisconnectedPlayers` a `popConnectedPlayers`. Udržuje seznam nových připojení od posledního zavolání `popConnectedPlayers`.

Třída si u každého klienta hlídá počet po sobě jdoucích selhání. Jakmile počet překročí určitou hranici, např. 5 chyb, relaci s klientem ukončí. Tato komponenty skrývá samotné připojení s klientem a zbytek systému tak odpojení a připojení nemusí řešit. Pro získání seznamu existuje metoda `getClients`.

5.1.2 Server Replikátoru

Komponenta, která se snaží synchronizovat stav klientů s lokálním stavem na serveru, se nazývá „replikátor“. Pro každého klienta si drží seznam entit, které musí danému klientovi synchronizovat, aby svůj úkol splnil. Tento seznam entit lze z venku nastavovat, většinou komponentou pro správu zájmů, kterou představíme v další podkapitole. Základní implementace by posílala každý snímek kompletní stav entity se všemi komponentami. Lepší varianta je posílat pouze změněnou část stavu. Pokud se například pozice entity X nezmění, není třeba posílat stejnou pozici pro entitu X jako minule. Na druhé straně u klienta máme klienta replikátoru. Ten se stará o dekódování zpráv a správné aktualizaci stavu.

5.1.3 Správa zájmů

Komponenta, která řeší důležitost entit pro jednotlivé hráče, je „manažer zájmu“. Důležitost definujeme pro každou dvojici klienta a entity. Manažer zájmu podle definované logiky, například na základě vzdálenosti, určí důležitost entity pro klienta. V základní verzi jsme měli pouze dvě hodnoty: důležitá a nedůležitá. Synchronizovali jsme pouze důležité. Replikátor se dotazuje manažera zájmu v moment, kdy klientovi chce stav synchronizovat.

Dotazu na všechny body, které jsou dostatečně blízko od konkrétního bodu, se říká ‘range-query‘. Datové struktury které tuto operaci akcelerují jsou například fixní mřížka, dynamická mřížka nebo quad-tree.

5.2 Klient

Program klienta se skládá z třídy `World`, která reprezentuje stav a ClientWorldController, která obsahuje jednodušší řídící logiku. Řídící logika posbírá každou iteraci vstupy, převede je na akce a ty odešle na server.

5.2.1 Klient Replikátoru

Abychom oddělili to, jak replikátor funguje, vytvořili jsme pro něj na straně klienta ovladač. To nám později umožnilo měnit protokol mezi klientem a server podle potřeby. Komponenta pouze přijme snapshot od serveru, provede autentizaci a pak podle něj změní stav na klientovi.

5.2.2 Interpolace

Když jsme začali systém měřit, zjistili jsme, že vysoká frekvence replikace až příliš zatěžuje síť. Graf s průměry pro počet hráčů vidíme na grafu. Snížili jsme

proto frekvenci na 20Hz. To snížilo zátěž na třetinu. Hra potom nebyla plynulá. Klient vykresloval s frekvencí 60Hz, ale aktualizace pozice s frekvencí 20Hz způsobily neplynulý pohyb. Řešení je na klientovi interpolovat spojité proměnné, jako pozice nebo rotace.

Vytvořili jsme novou komponentu „interpolátor“. Ta umožňuje zapsat pozici entity v konkrétní čas. Komponenta si udržuje časovou osu stavu pozice. Zároveň umožňuje číst pozici entity pro konkrétní čas. Pokud je vybraný čas na ose mezi dvěma body, tak vrátí mezi nimi interpolovanou pozici. Pokud je vybraný čas až po posledním známém bodě na ose, tak vrací poslední známou hodnotu.

Změnili jsme objekt, do kterého klient replikátoru zapisuje pozice entit na právě objekt interpolátoru. Do něj zapisuje původní sníženou frekvencí 20Hz. Řídící logika ale může z interpolátoru číst s vyšší frekvencí, například 60Hz. Aby čtecí funkce interpolátoru měla mezi čím interpolovat, není vhodné číst hned nejnovější hodnoty. Lepší je počkat, a interpolovat pozici určitý čas zpět. To má za následek zvýšení odezvy mezi vstupem od hráče a fyzickým posunem postavy hráče. Klient nejprve čeká na odpověď serveru a poté čeká ještě o jednu odpověď navíc. To není problém, pokud takto interpolujeme postavu, kterou neovládá přímo hráč. Rozdíl odezvy pár set milisekund bude zanedbatelný.

5.2.3 Rollback

Interpolace zlepšila plynulost, ale nepříjemně zvýšila odezvu mezi vstupem od hráče a vyvolanou změnou stavu. Možným řešením je simulovat řídící logiku i na klientovi, ale pouze pro konkrétní entitu, kterou hráč ovládá. Odezva bude v podstatě nulová.

Vytvořili jsme buffer, do kterého ukládáme konkrétní moment stavu, číslo iterace stavu a seznam akcí, která se v danou iteraci mají aplikovat. Pokud dostaneme snapshot pro snímek x a z bufferu přečteme, jaký byl v iteraci stav. Pokud se neshodují, celý původní stav načteme a opravíme podle snapshotu. Následně znova aplikujeme všechny akce a přepíšeme tak náš buffer, až se dostaneme do aktuální iterace. Poté normálně pokračujeme s opraveným stavem.

5.3 Posílání zpráv

V aplikační vrstvě jsme definovali middleware protokol, který umožňuje obousměrné posílání typovaných zpráv. Typ zprávy je definován čtyř bytovým kladným číslem. Protokol jsme umístili do modulu `message_protocol`. Protokol využívá TCP, ale později implementaci změníme tak, aby používal náš protokol. Koncové body v protokolu se chovají jako fronta příchozích a fronta odchozích zpráv. Pro koncový bod můžeme definovat tzv. „dispečer“, definovaný typem zprávy a anonymní funkci, která každou příchozí zprávu tohoto typu zpracuje.

5.3.1 Kódování zpráv

Middleware využívá TCP, které na rozhraní při čtení a zápisu pracuje s proudem bytů, který není logicky rozdělený. Jediné pevné body jsou začátek a konec proudu. UDP na druhou stranu zaručuje, že celý buffer dat odeslaný přes `write` operaci bude přečtený vždy právě jedním voláním `read` operace. Náš protokol definuje oddělení zpráv tak, aby šli v proudu najít.

První čtyři byty každé zprávy jsou tzv. MAGIC číslo. Konstanta, která nemá jiný význam než záchytný bod při čtení. Dekodér v počátečním stavu hledá v proudu tuto hodnotu. Pokud ji do určitého počtu bytů nenajde, ukončí spojení z důvodu narušení protokolu. Když konstantu najde, přečte další čtyři byty, které reprezentují délku těla a další čtyři byty reprezentující ID koncového bodu.

Vlastnost, kterou jsme v průběhu testování potřebovali, byla možnost request-reply modelu. Zároveň jsme ale chtěli mít možnost posílat zprávy stylem fire-and-forget. To znamená, že zprávu odešleme a nečekáme odpověď. Nemohli jsme tedy použít způsob, jaký používá HTTP/2. Místo toho lze při odeslání zprávy definovat, že čeká na odpověď a její identifikační číslo. Druhá strana naopak může poslat odpověď, když zprávu definuje jako odpověď pomocí příznaku a identifikačního čísla zprávy, na kterou odpovídá.

5.3.2 Implementace

Vytvořili jsme třídu `MessagingSession`, která má na rozhraní metody `set_handler (endpoint_id)`, `send` a `request`. První metoda nastaví pro koncový bod `endpoint_id` handler, který je definovaný anonymní funkcí. Druhá pošle zprávu stylem fire-and-forget. To znamená, že se nestaráme o odpověď a synchronizace je v moment, kdy si zprávu převezme middleware. Poslední metoda umožňuje poslat požadavek a nastavit handler pro odpověď.

5.3.3 Detekce a náprava chyb

Občas mohou nastat v proudu chyby, kdy zprávu nelze dekódovat, nebo nastane chyba při deserializaci. Proto jsme do kódování implementovali nápravu chyb. Už jsme definovali maximální délku pro hledání magické konstanty. Dále si dekodér pro spojení počítá chyby. Ten se po každé úspěšně dekódované zprávě resetuje na 0. Pokud počet překročí definovanou toleranci 3 chyb, spojení bude ukončeno. Příkladem takové chyby je selhání deserializace nebo že TCP spojení bylo přerušeno.

5.4 Serializace

Většinou nechceme používat middleware protokol, který na rozhraní používá n-tice bytů. Lepší je postavit obal, který umožní poslat zprávy reprezentované strukturou. Procesu, který ji převede na n-tici bytů se říká „serializace“. Existují různé knihovny, které umí objekty serializovat. Některé umí serializovat přímo

třídy definované v konkrétním programovacím jazyce. Jazyk C++ ale nemá runtime reflexi jako C# nebo silná makra jako Rust, a tento způsob není možný. Druhá varianta je definovat tyto třídy ve speciálním jazyce pro definici schémat. Před sestavením programu z této definice vygenerujeme třídy v jazyce, který potřebujeme. Do tříd se vygenerují i metody pro serializaci. Tento přístup umožňuje například ProtoBuf, Cap'n'Proto a další. Velká motivace pro použití tohoto přístupu byla možnost reflexe, kterou tyto knihovny umožňují. Další výhoda je pro architekturu orientovanou na služby, kde každá služba je implementována v jiném jazyce, tak nemusíme definice lokalizovat.

Hned na začátku jsme se rozhodli nepoužít JSON nebo XML, protože jsou to textové formáty a jejich forma je často větší než binárních formátů. 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 téměř dvojnásobný. Každý atribut objektu se v JSONu reprezentuje řetězcem, namísto identifikačním číslem o 2 bytech, které by bylo menší. Výhody formátu jsou především liberalní zpracování, kdy se schémata dvou koncových bodů mohou i trochu lišit, ale pokud má zpráva všechny potřebné atributy, tak se strany domluví.

Binární formáty tuto problematiku také řeší, například číslováním atributů, ale stále není řešení tak volné, jako u textových formátů. Obecně hrozí, že v binárním formátu je atribut navíc, nemusí se druhé straně podařit zprávu deserializovat. Proto se někdy JSON používá jako záložní formát, kdy dvě strany provedou handshake a domluví se na formátu. Pokud zjistí, že jedna strana používá starší definici schématu, mohou použít záložní JSON.

Rozhodli jsme se začít velmi rozšířeným ProtoBuf formátem. Začneme definicí zpráv mezi serverem a klientem. Vytvořili jsme WorldServerMessages.proto soubor, do kterého budeme zprávy definovat. Trochu formát `.proto` představíme. V kódu **??** vidíme definici zprávy, která říká, že si příjemce má do svého stavu přidat novou entitu. Obsahuje tři atributy: ID pro navázání budoucích zpráv o entitě, příznak, jestli se jedná o hráče a jméno.

```
1  message EntitySpawnMessage {
2    uint32 entity_id = 1;
3    bool is_player = 2;
4    string name = 3;
5  }
6  \label{code:entity_spawn_message}
```

Zdrojový kód 2: cpp

Jazyk vypadá podobně jako C. Nejprve použijeme klíčové slovo `message` a název typu. Do těla píšeme atributy oddělené středníkem. Všimněme si, že pro každý atribut musíme definovat jeho unikátní číslo. To slouží při identifikaci a pomáhá při úpravách definice, abychom měli kontrolu nad tím, pod jakým klíčem se hodnota serializuje.

FlatBuffers je opět protokol od Googlu, který se vyvýjel 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.

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.

5.5 Protokol QUICr

Naše hra používala ke komunikaci mezi vrcholy protokol TCP. Zmínili jsme ale, že ten pro hry nemusí být vhodný. Vytvořili jsme vlastní protokol inspirovaný QUIC, který také využívá UDP, ale uvolnili jsme podmínky pro spolehlivost, konkrétně jistotu doručení a uspořádání zpráv. Spolehlivost a uspořádání lze definovat pro každou zprávu zvlášť. V další kapitole TCP a QUICr porovnáme a ukážeme, jak se výhody QUICr projeví.

Upustili jsme od QUIC paketů a definovali jsme právě jeden typ, který má v hlavičce ID příjemce a ID odesílatele. Každý paket obsahuje rámce. Rámce jsme také rozdělili na dva typy: datové a ovládací. Ovládací slouží pro handshake, ukončení spojení nebo nastavení jiných parametrů spojení a budou vždy doručené spolehlivě. Datové rámce obsahují aplikační data a mohou definovat svou spolehlivost. Ty, které budou obsahovat pozice hráčů, tak budeme moct odesílat nespolehlivě, zatímco změny stavu, jako změna barvy postavy, posíláme spolehlivě. Implementace se skládá ze dvou tříd: „koncový bod“ a „spojení“.

Objekt navázaného spojení je stavový stroj a fronta příchozích a fronta odchozích zpráv, podobně jako TCP spojení. Stavy se mění v reakci na příchozí ovládací rámce. Každý objekt spojení má přiřazeno alespoň jedno ID reprezentované osmi bytovým číslem. Tento objekt přímo nepracuje se socketem, ale poskytuje rozhraní, kterým dostává získané zprávy, které odeslala druhá strana. Právě tento objekt překládá n-tici bytů na rámce a postupně je zpracovává. Stavový stroj je důležitý hlavně v procesu handshake, který definujeme níže.

Objekt pro koncový bod se stará o soket pro UDP: čtení a zápis na něj, a udržuje si registr objektů spojení. Při získání datagramu ze socketu ověří, že má správný formát. Podle protokolu by každý datagram měl obsahovat hlavičku s magickým číslem a ID spojení. Koncový bod přečte ID spojení a podle svého registru vyhledá objekt spojení, kterému tělo zprávy předá. Pokud v registru klíč pro získané ID není, koncový bod si objekt spojení vytvoří v počátečním stavu a začíná proces zvaný handshake. Tento proces popíšeme v podkapitole 5.5.2.

Nechtěli jsme, aby si implementace QUICr spravovala vlastní asynchronní kontext. Namísto toho jsme definovali metodu pro aktualizaci, která posbírá zprávy od jednotlivých navázaných spojení a pošle je. Následně přečte všechna data ze soketu a roztřídí je do navázaných spojení.

5.5.1 Rámce

Hlavním prvkem datového přenosu v protokolu jsou rámce. Jak už bylo nastíněno, dělíme je na kontrolní a datové. Pro využití nespolehlivosti jsme u každého datového rámce umožňili definovat, jestli je spolehlivý nebo nespolehlivý. O spolehlivost obecně se stará komponenta `ReliabilityUnit`, kterou podpobněji představíme v kapitole 5.5.3.

Rámec je n-tice bytů, která se skládá z hlavičky a těla. V hlavičce máme ID cílového spojení a ID počátečního spojení.

5.5.2 Handshake

Objekt navázaného spojení obsahuje stavový stroj a postupně mezi stavy přechází. Počáteční stav je `Closed`. Jedna strana musí začít a odeslat paket s `Hello` rámcem. Po obdržení se stavový stroj druhé strany přesune do `ReceivedHello` a spolehlivě odešle `Hello`. Když strana dostane rámec `Hello` tak ví, že v hlavičce paketu je cílové i lokální ID. Proto už touto výměnou se strany shodli na ID navázaného spojení, které budou používat. Poté už si vymění rámec `HandshakeDone` a spojení je navázané a připravené k posílání.C elý stavový diagram vidíme na obrázku 7.

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

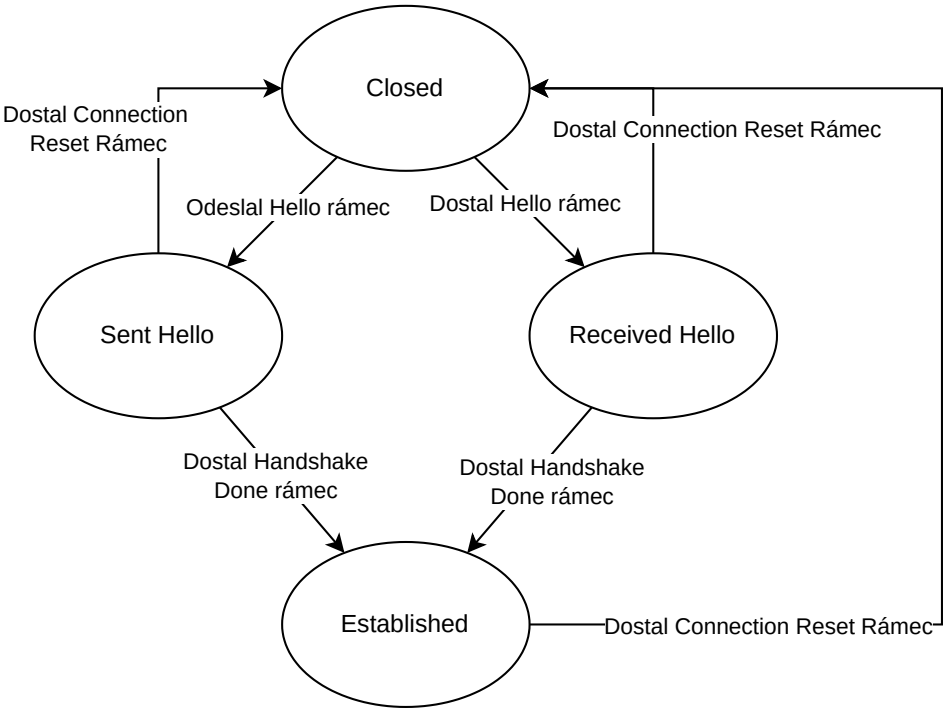
SentHello
: Po odeslání `Hello` rámce se spojení dostane do stavu `SentHello`. V tomto stavu čeká na `Hello` rámec od druhé strany.

ReceivedHello
: Jakmile strana dostane rámec `Hello`, přejde do stavu `ReceivedHello`. V ten moment čeká na potvrzení, že spojení bylo navázáno, tedy rámec `HandshakeDone`.

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

5.5.3 Spolehlivost

O spolehlivost doručení se protokol stará posíláním `Ack` rámců s číslem paketu, který získal. Navíc ale rozlišujeme spolehlivost na základě jednotlivých rámců. Pokud paket neobsahuje žádný spolehlivý rámec, nemusí pro něj příjemce posílat `Ack` rámec. Protokol v tento moment neřeší spolehlivost pořadí paketů. V našem případě posíláme například akce z klienta na server a u každé definujeme číslo



Obrázek 7: Stavový stroj QUICr handshake

snímku. Server si udržuje nejvyšší číslo snímku, které dostal, a všechny zprávy z nižších snímků zahazuje. Stejně to dělá klient se snapshoty. Proto neřešíme pořadí na transportní vrstvě.

Spolehlivost řeší komponenta `ReliabilityUnit`, která si udržuje frontu rámců, které je potřeba odeslat, čas, kdy je potřeba rámec odeslat a jejich spolehlivost. Čas slouží pouze pro prioritu a málokdy se stane, že se rámec odešle právě v tento čas. Objekt `QuicrEndpoint` se při tiku dotazuje `QuicrConnection` na další datagram, který chce odeslat. Ten pomocí `ReliabilityUnit` začne datagram skládat z paketů. Když přijde rámec na řadu a je spolehlivý, hned se umístí na konec fronty a nastaví se mu čas odeslání. Většinou aktuální čas s přičteným konstantním intervalem. Pokud je nespolehlivý, do fronty se nevrátí. Krátce představíme rozhraní `ReliabilityUnit` objektu.

`push_reliable_frame_to_send(deadline, frame)`
: Uloží si zakódovaný rámec a nastaví si u něj čas odeslání. Jakmile bude po deadline, jednotka se bude snažit dostat rámec do dalšího datagramu. Rámec se ukládá v zakódované podobě tak, aby byla předpovídatelná jeho velikost a snadno se odhadovalo, jestli se do dalšího datagramu vleze.

`peek/pop_reliable_frames_to_send(packet_number)`
: Vrátí rámce, které je potřeba v tomto okamžiku odeslat. Číslo paketu `packet_number` si jednotka přiřadí jako verifikátor doručení rámce, který metoda vrátí. To znamená, že když toto číslo přijde v `Ack` rámci, bude jednotka vědět, který rámec může odebrat. Zároveň může mít vícero vazeb čísla paketu na jeden rámec. Libovolné číslo paketu smaže provázaný rámec. Důvod je, aby mohlo být v oběhu více paketů se stejným rámcem.

`push_ack(packet_number)`
: Vloží do seznamu čísel paketů. Celý tento seznam se odešle v `Ack` rámci v dalším datagramu.

`peek/pop_acks_to_send()`
: Vrátí seznam čísel paketů, které musí druhé straně oznámit jako doručené.

5.5.4 Enkodér

Pro kódování jsme vytvořili třídu `QuicrEncoder`, která přes rozhraní umožňuje kódovat pakety a rámce do libovolné alokované paměti. 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.

Další problém, který musel enkodér řešit, bylo zpětně zapsat hodnoty do hlavičky. Na začátku zápisu, kdy zapisuje do hlavičky, tak ještě není známá velikost těla, která je ale potřebný atribut v hlavičce. Proto se zapisuje zpětně.

5.5.5 Testování

Při implementaci jsme zjistili, že je dobré začít testy, které definují jednotlivé vlastnosti protokolu, 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. Původně jsme začali implementací, ale bylo těžké domyslet, jak by se měl protokol implementovat.

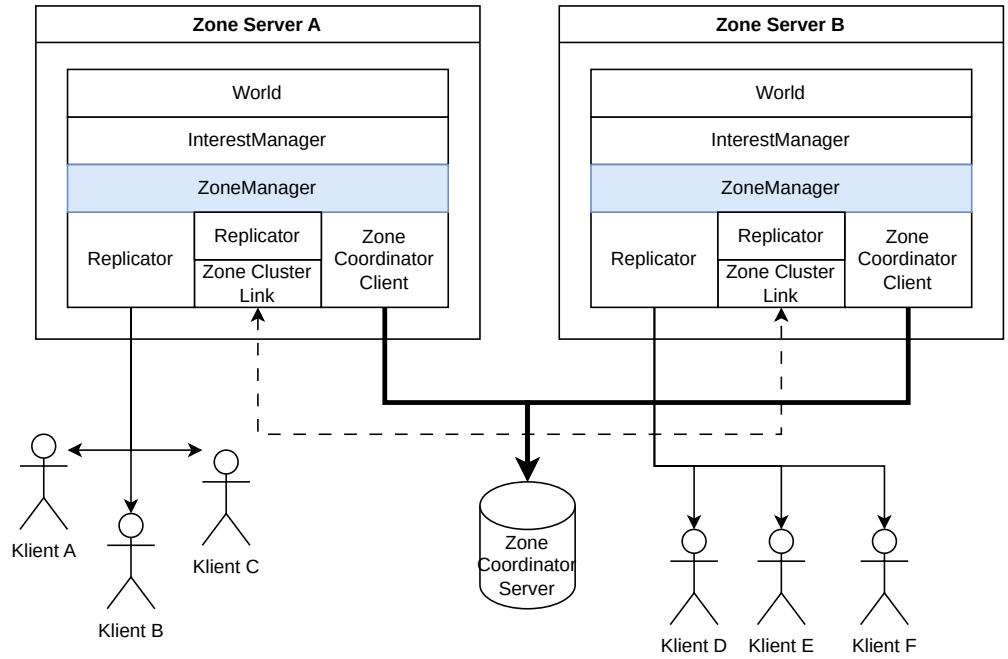
5.6 Horizontální škálování

Při nárustu počtu hráčů už nemusí stačit jeden výpočetní server. Poté je nutné využít výkon více počítačů technikou zvanou „horizontální škálování“. To znamená, že do systému přidáváme servery, které si tak rovnoměrně rozdělí práci. V případě naší hry jsme rozdělili herní svět na zóny a o každou se stará jiný server. Takový server je „zone server“ a dohromady tvoří „zone cluster“. Do systému jsme dále přidali „cluster koordinátora“, který říká, který zone server má jakou zónu. Model mezi koordinátorem a zone serverem je klient-server. Na druhou stranu zone servery mezi sebou komunikují přímo a využívají model peer-to-peer.

Představíme, jak spolu zone servery komunikují. Využijeme k tomu už existující komponenty: replikátor a manažera zájmu. Nejprve jsme stav hry, manažera zájmu a fyzický svět do třídy `ZoneManager`. Tuto třídu obalíme `ZoneServer` třídou. Ta bude obsahovat replikátor a `ZoneCoordinatorClient`. Manažer zón si bude udržovat registr sousedních zón a bude se k ním chovat jako klasickým klientům. Rozdíl bude ten, že klient má jako zájem definovaný pouze bod a manažer zájmu klientovi přiřadí do zájmu entity, které jsou od bodu v dostatečné vzdálenosti. Sousedi budou mít jako zájem definovanou celou jejich plochu. Manažer zájmu má nyní na rozhraní dvě metody: `register_client_interest(interest_id, entity)` a `register_zone_interest(interest_id, area)`. Každý zájem je definovaný unikátním ID. O to, jaké entity pak konkrétní zájem zajímají získáme voláním `get_interest(interest_id)`.

Replikátor poté replikuje sousedům ty entity, které jsou od plochy souseda v dostatečné blízkosti. Pokud klient překročí do jiné zóny, měl by manažer zóny předat klienta druhé zóně. Toto modelujeme přes `ZoneProxy`, která reprezentuje souseda pro manažera konkrétní zóny. Implementace obsahuje `ZoneClusterLink`, která obsahuje QUICr endpoint, na který ostatní zóny mohou posílat cluster data.

Systémovou architekturu vidíme na obrázku 5.6. Vidíme, že `ZoneClusterLink` slouží pro komunikaci mezi servery v clusteru a zároveň pro replikaci entit, které jsou blízko hranic se sousední zónou. Když se nový server spustí, nejprve se registruje koordinátorovi. Ten mu přiřadí ID, plochu a seznam jeho sousedů včetně `ZoneClusterLink` adres. S každým sousedem přes QUICr naváže spojení a představí se pomocí zprávy `ZoneHello`. Tím si ho druhý server přidá do registru a může na něj začít přenášet klienty.



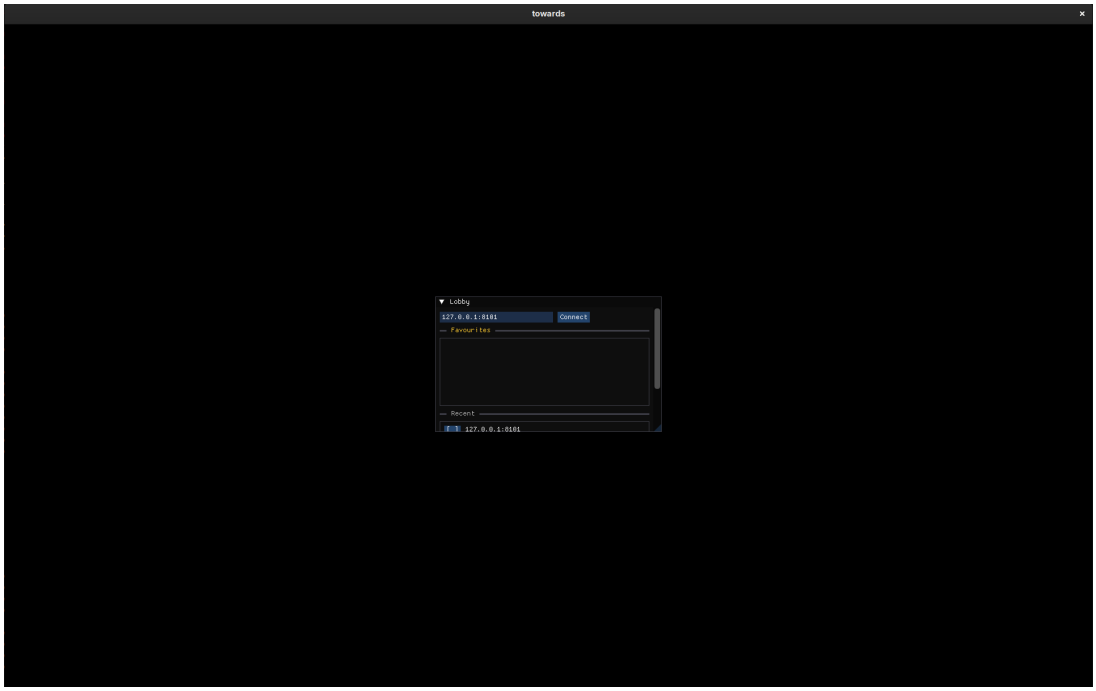
Obrázek 8: Architektura systému s zone clusterem

5.7 Výsledná hra

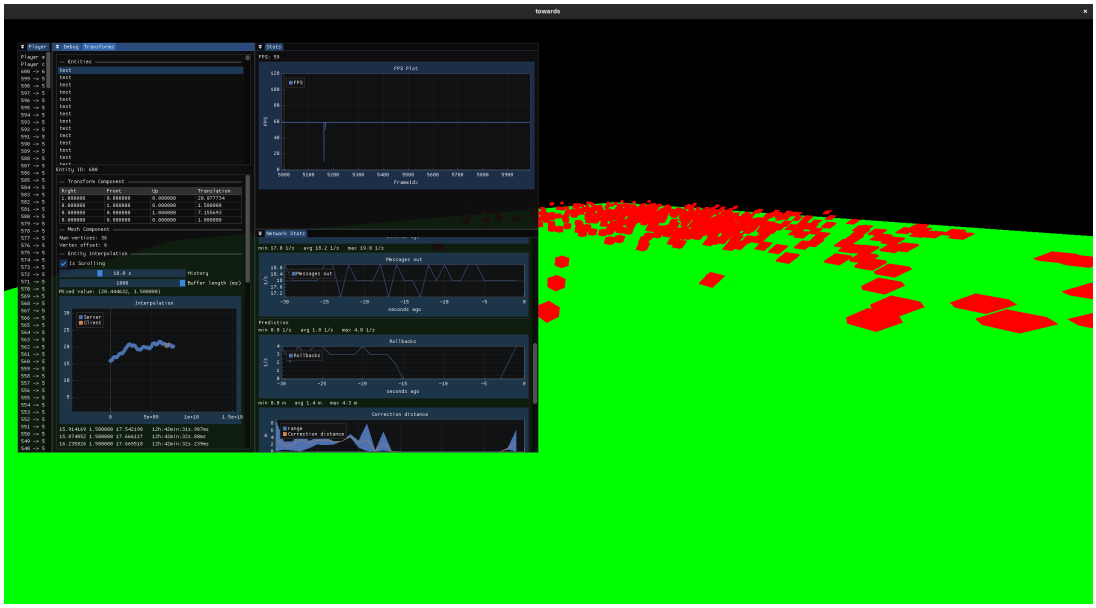
V této části výslednou hru popíšeme. Po spuštění klienta dostaneme možnost zadat IP adresu serveru. V tento moment je aplikace ve stavu `lobby`. Hlavní třída, která řídí chod včetně nekonečné smyčky, se nazývá `Runtime`. Ta se chová jako stavový stroj a umožňuje stavům vracet zprávy, podle kterých mezi stavy přechází. Po kliknutí na „connect“ se klient pokusí připojit na server.

V případě že uspěje, přejde `Runtime` do stavu `game`. V něm už vidíme stav hry a uživatelské rozhraní, které se skládá z oken. Okna je možné skládat do sebe. TODO: Otevírání různých oken pro přehlednost.

Po připojení je možné se volně pohybovat pomocí WASD. Do hry se může připojit více hráčů. Například je možné použít program `tw_mock_client`.



Obrázek 9: Uživatelské rozhraní v lobby



Obrázek 10: Uživatelské rozhraní v klientovi

6 Měření

V této kapitole popíšeme, jaké testy jsme provedli a jejich výsledky. Nejprve představíme jak samotné metriky můžeme sbírat. Implementovali jsme jednoduchý systém pro sběr metrik, který je schopný je různě agregovat do oken dlouhých například jednu sekundu. Data jsme různě vizualizovali ve webovém rozhraní, tak i přímo v aplikaci klienta.

6.1 Metriky

Představíme metody měření a metriky v systému, které jsme využili pro hledání potenciálních míst pro optimalizaci. Problém jde rozdělit na tří úlohy: jak metriky sbírat, jak je ukládat a jak je zobrazit. Pro sběr jsme definovali třídy `NetworkMetricsReporter`, který obsahuje metody pro přidávání nových vzorků a různé možnosti čtení daných metrik. Každá lze přečíst jako celá série s určitou historií. Příkladem metod jsou `report_outbound` a `get_outbound`.

Implementace si data ukládá pouze do paměti RAM a po ukončení programu jsou nenávratně ztracena. Druhý problém šel ale řešit i perzistentně. Existují databáze, které jsou pro rychlé vkládání vzorků s časovou známkou vhodné. Mezi využití takových databází patří logy nebo právě metriky. Náš výběr představíme.

Nakonec zbývalo metriky zobrazit. Rozhodli jsme se použít webové rozhraní pro rychlou inspekci stavu. Umožní rychle se podívat na množství hráčů nebo velikost datového toku. Metriky, které sbírá klient, zobrazujeme přímo v klientovi. Obě řešení ukážeme.

6.1.1 PostgreSQL

Pro perzistentní úložiště jsme vybrali populární relační databázový server PostgreSQL. Rozhodli jsme se právě pro ten, protože je to otevřený a svobodný software, který je zdarma. Databázi si lze snadno nasadit lokálně, ať už daemoném nebo jako Docker kontejner. Skrze rozšíření jako TimescaleDB lze přidat optimalizace pro rychlé vkládání dat o metrikách.

6.1.2 TimescaleDB

Pro optimalizaci PostgreSQL pro časová data jsme zvolili TimescaleDB. Díky tomu, že se jedná o rozšíření, nemusíme spravovat jinou službu, než PostgreSQL. TimescaleDB umožňuje vytvářet „hypertables“, které jsou rychlé vkládání optimalizované. Vytvořili jsme komponentu `TimescaleDbMetricsWriter`, která čte z `NetworkMetricsReporter` a postupně zapisuje metriky do databáze. Jedná se o konkrétní implementace pro TimescaleDB. Komponenty jsou tak oddělené a snadno lze přidat rozšíření pro další typy úložišť, jako CSV nebo jiný databázový server.

Obrázek 11: Příklad Grafana dashboardu

6.1.3 Grafana

Grafana je služba, která skrze webovou stránku poskytuje rozhraní pro vizualizaci metrik. Lze nastavit svůj dashboard a v něm různé grafy. Pro naše účely jsme vytvořili grafy pro odchozí a příchozí množství bytů. Služba slouží spíše pro sledování stavu a neumožňuje pokročilejší analýzu. Na obrázku 11 vidíme tři grafy. Horní graf ukazuje více metrik najednou, konkrétně množství odchozích a příchozích dat a počet hráčů. V tomto konkrétním případě jsme připojili 300 hráčů. Vidíme, že množství příchozích a odchozích dat se zvýšil. Množství odchozích dat se zvýšil očekávaně daleko více.

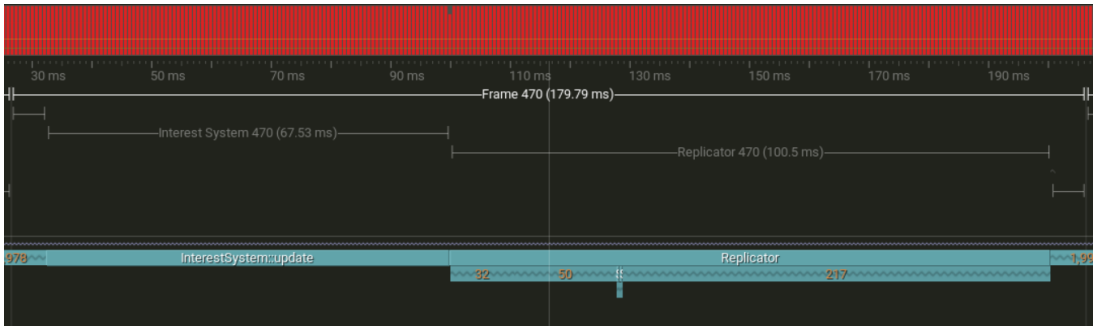
6.1.4 Tracy

Pro měření a pokročilejší analýzu výkonu programu klienta nebo serveru používáme profiler Tracy. Program je zdarma, open-source, napsaný v C++ a pro vykreslování UI využívá ImGui. Poskytuje přesnost na nanosekundy, což je užitečné u serverů, na kterém může být miliony různých entity. Navíc umožňuje připojit se k programu, který měří, vzdáleně. Případně sbírat metriky do souboru, ten si ze serveru stáhnout a analyzovat později.

V kódu umožňuje definovat jednotlivé iterace řídící logiky zvané snímky. Následně umožňuje definovat tzv. „zóny“, které následně měří. Zóna může být průběh funkce nebo její konkrétní část. Na obrázku 12 vidíme, jak taková analýza vypadá. Vidíme právě jeden snímek. V něm máme označené části replikátoru a manažera zájmu. Také vidíme dole zóny.

6.1.5 Klientská aplikace

Přímo v hráčově klientské aplikaci jsme chtěli mít možnost vidět metriky v reálném čase. K implementaci grafického uživatelského rozhraní jsme použili knihovnu ImGui představenou dříve. Komponenty pro zobrazení různých metrik opět čtou z `NetworkMetricsReporter`.



Obrázek 12: Jeden snímek v Tracy

6.2 QUICr

Protokol měl za cíl snížit odezvu na nespolehlivé síti a neblokovat zprávy. 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. Klient odesílá hodnotu derivace typu double. Server si udržuje stav hodnoty, nazvěme ji „pozice“. Při obdržení derivace ji přičte k aktuálnímu stavu. Každý snímek server odesílá na klienta stav pozice, kterou si klient aplikuje.

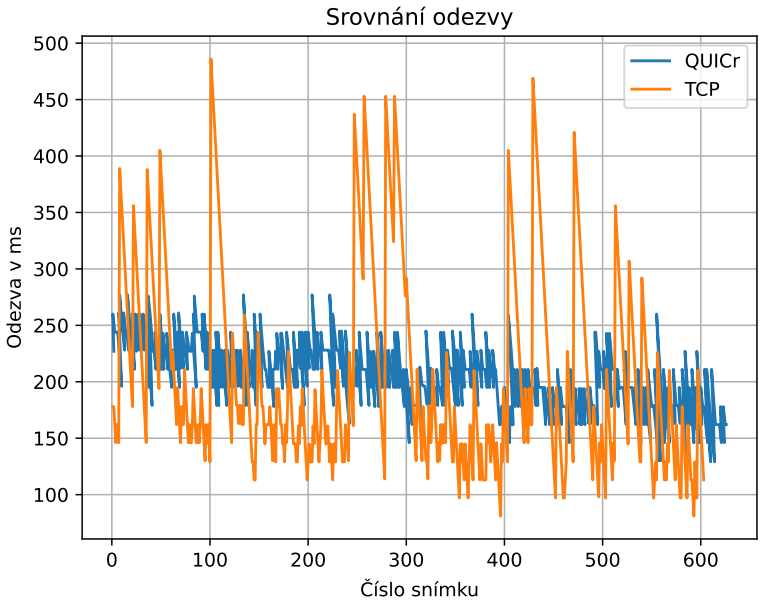
Pro nasimulování nespolehlivosti sítě jsme využili Linuxový nástroj „tc“. Ten obsahuje „network emulator“ s různými nastaveními spolehlivosti sítě. Například kolik procent paketů má ztratit, jaká má být odezva nebo jak moc náhodné bude pořadí paketů. Budeme simulovat odezvu 200 milisekund s rozptylem 20ms, ztrátu 3% paketů a 1% paketů bude duplikovaných. Celý příkaz pro nastavení můžeme vidět v příkladu 3. Vidíme, že používáme pouze rozhraní `lo`, tedy loopback, které se týká posílání mezi programy na lokální adrese.

```
1    # tc qdisc add dev lo root netem \
2       delay 200ms 20ms 25% \
3       loss 3% \
4       duplicate 1% \
5       reorder 25% 50%
```

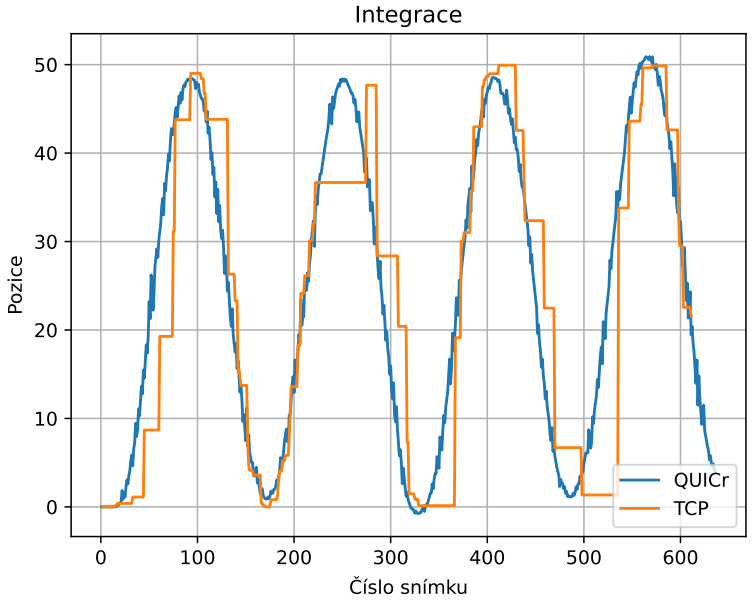
Zdrojový kód 3: Příklad volání TC

V prvním testu jsme pouze měřili odezvu mezi odesláním vstupu od klienta pro snímek n a získáním jeho integrovaného stavu ze serveru. Na obrázku 13 vidíme, že TCP obsahuje skoky a QUICr je stabilnější.

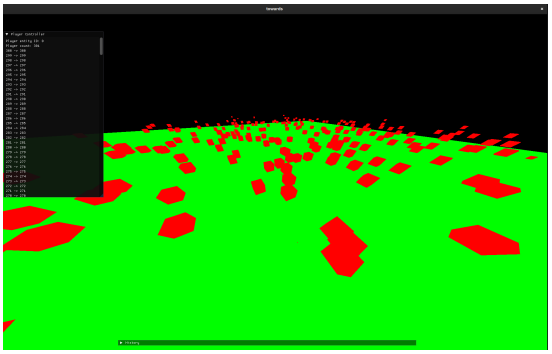
Následně jsme zkusili, jak se nespolehlivost sítě a skoky odezvy z prvního měření projeví v integraci proměnné. Pro derivaci jsme použili sinusoidu, abychom ji mohli zreplikovat snadno v TCP i QUICr. Na obrázku 14 vidíme, že TCP při ztrátě paketů způsobuje skoky, které nemusí být pro hráče příjemné. Naopak náš protokol má integraci plynulejší.



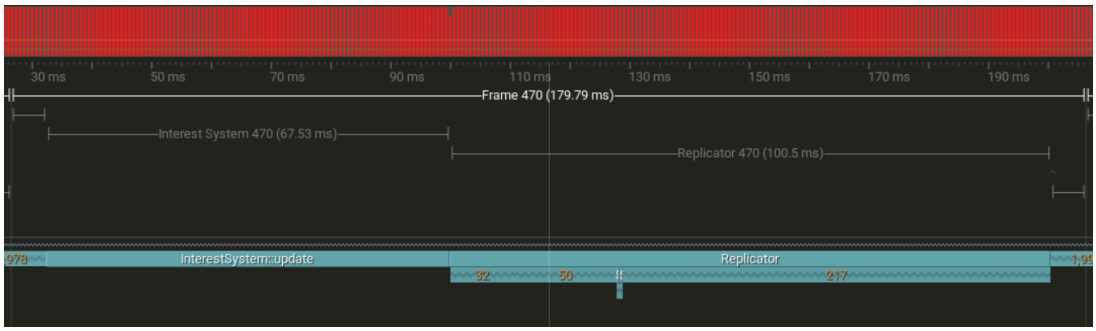
Obrázek 13: Porovnání odezvy TCP a QUICr



Obrázek 14: Porovnání integrace TCP a QUICr



Obrázek 15: 301 připojených hráčů



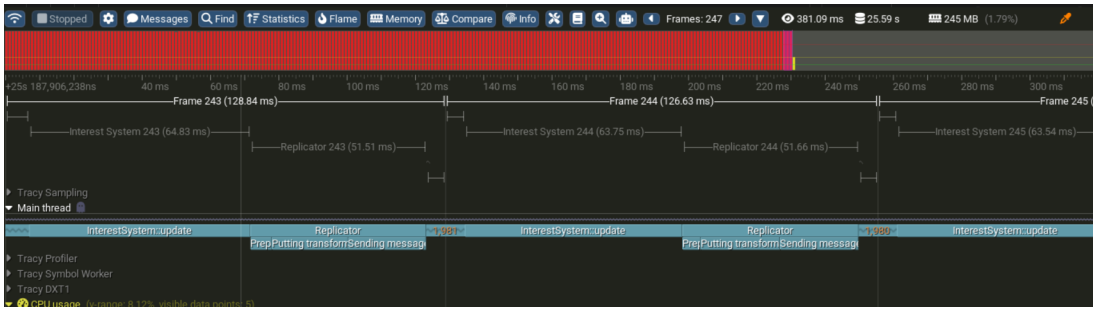
Obrázek 16: Analýza replikátoru

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 15 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.

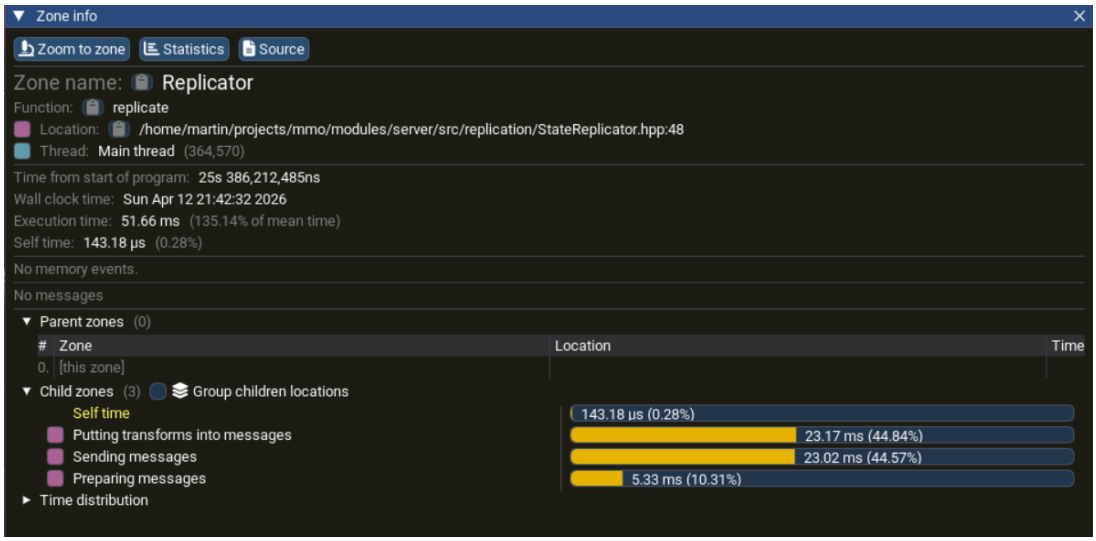
6.3 Optimalizace replikátoru

Test s 301 hráči odhalil nedostatečný výkon serveru. Pomocí Tracy jsme začali program analyzovat. Na obrázku 6.3 vidíme, že pouze aktualizace replikátoru trvá každý snímek kolem 100ms. Pro představu, pokud bychom chtěli, aby server integroval 20 krát za sekundu, museli bychom všechno, nejen aktualizaci replikátoru, stihnout do 50 milisekund. Proto je aktuální stav nepřijatelně 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 při skládání zpráv. Tedy kódování



Obrázek 17: Analýza optimalizace replikátoru pro ECS



Obrázek 18: Analýza optimalizace replikátoru pro ECS

pozic entit, o které se klient zajímá, do objektu. Je to právě tento objekt, který se serializuje přes ProtoBuf a odesílá přes soket klientům.

Replikátor pro každého klienta zjistil jeho zájem a pro každou entitu v něm vyhledával jeho komponenty. Vyhledávání je v případě Entt implementováno hashovací tabulkou a časová složitost je v $O(1)$. Začali jsme měřit i jiná řešení. Primárně nás zajímali metody, které jsou vhodné pro ECS architekturu. První řešení, které jsme zkusili, bylo vytvořit buffer instancí zpráv pro každého klienta najednou. Přes komponentu, kterou jsme chtěli replikovat, např. Transform, jsme vytvořili Entt pohled a iterovali. Pro každý prvek jsem komponentu zapsali do zprávy pro každého klienta, kterého zajímala. Výslednou analýzu vidíme na obrázku 6.3. Tato optimalizace, která jen změnila pořadí, v jakém nahlížíme na data, snížila dobu, kterou trvá aktualizace replikátoru, na polovinu.

Replikátor jsme začali analyzovat do větších detailů. Na obrázku 6.3 vidíme, které jeho části zabrali jakou dobu. Zjistili jsme, že problém byl samotný ProtoBuf. Formát je podobně jako JSON dělaný pro složité struktury objektů. V

	ProtoBuf	FlatBuffers	Memcpy
Čas (s)	9.455691762	10.345692894	0.754350866

Tabulka 1: Porovnání serializátorů

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 dopředu alokovat všechnu potřebnou paměť, kterou můžeme mezi snímky využívat. Dopředu si vytvoříme vektor pozic, které v testu budeme serializovat. Tímto zajistíme, že nebudeme měřit čas včetně alokace paměti. Stejně jako v replikátoru používáme architekturu orientovanou na ECS a procházíme všechny pozice, ty pak umisťujeme 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 1.

Zjistili jsme, že problémem byl ProtoBuf a FlatBuffers by nám nepomohl. Nepřekvapilo nás, že memcpy byl nejrychlejší. Rozhodli jsme se, že pro serializaci a deserializaci snapshotů světa budeme používat vlastní serializaci.

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. Pro čtení slouží `WorldStateReader`.

6.4 Optimalizace manažera zájmů

Všimli jsme si, že replikátor trval každou iteraci 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 přepočet, ale pouze vložit do předem alokovaného bufferu, nebo z něj naopak odebrat.

Testovali jsme 4 implementace, z toho 3 různé datové struktury a 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. První je fixní mřížka, která 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

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%)

Tabulka 2: Porovnání algoritmů

o obdobu stromu, kde vrcholy mají právě 4 potomky nebo žádného potomka. V tabulce 2 vidíme výsledky testu. Celkový čas je milisekundách. Ve třetím a čvrtém sloupci je změřená část vkládání a dotazování. Díky tomu vidíme, že Quad tree má rychlejší dotazování, ale delší vkládání. Nejrychlejší je fixní mřížka. Má ale vysokou paměťovou náročnost a při použití je tak nutné zvážit, jestli není hashovací mřížka vhodnější. Quad tree implementace se ukázala jako nevhodná. S přibývající hloubkou je navíc pomalejší. Ideální se ukázali varianty s fixní a hashovací mřížkou.

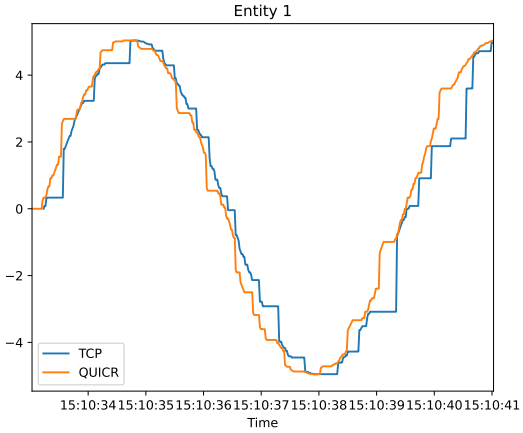
6.5 Peer-to-peer

Chtěli jsme změřit i hru s modelem peer-to-peer. Zároveň jsme chtěli dokázat, že náš protokol QUICr pomáhá i v tomto modelu. Vytvořili jsme proto program, který simuluje jednoho účastníka peer-to-peer systému, tzv. „peer“. Na začátku program čeká, až se připojí a autentizují ostatní peer. Proto jsme definovali číslo hráčů, v základu rovno 2.

Hry peer-to-peer fungují podobně jako v modelu klient-server. Každý účastník si udržuje svůj svět a chová se jako server. Tedy pro něj je zdroj pravdy právě jeho stav. Následně je hra rozdělena na iterace. V každé iteraci všichni odešlou všem svou akci, kterou učinili. Jakmile peer v iteraci i získá akce od všech ostatních pro iteraci i, tak si je lokálně aplikuje na svůj stav. Poté se posune do iterace $i+1$ a proces se opakuje. Tento způsob má nevýhodu v tom, že hra je stejně rychlá jako nejpomalejší peer. Pokud někomu bude dlouho trvat, než odešle svou akci, všichni ostatní jsou nuceni na něj čekat. Je totiž třeba žádnou akci nevynechat, jinak dojde k desynchronizaci.

Optimalizaci, kterou jsme implementovali, byl rollback. Ten umožňil, aby nebylo nutné čekat na všechny akce. Pokud peer A v iteraci i chybí akce pro peer B, tak peer A zkusí udělat pro peer B predikci. Pokud eventuélně akce od B do A pro iteraci i dorazí a liší se od predikce, tak se peer A vrátí v historii, akci změní a aplikuje znova.

Zde je nutné, aby peer posílal ne pouze poslední akci, ale celou historii svých akcí. Představme si, že akce pro iteraci i chybí, takže peer iteraci predikuje a



Obrázek 19: Porovnání TCP a QUICr v peer-to-peer

přejde do $i+1$. Akce pro iteraci i se ztratila a už nedorazí. Jakmile ale dorazí akce pro $i+1$, tak obsahuje i akci pro i. Peer si tak může historii vrátit, nasimulovat správně a tím se synchronizovat. Díky tomu je synchronizace daleko plynulejší.

Účastníci nemusejí vždy posílat celou svou historii, ale pouze tu část, kterou druhá strana ještě nemá. To lze zjistit tak, že příjemce posílá zprávu o tom, které iterace už obdržel. Tento přístup sice stále iteruje rychlostí nejpomalejšího účastníka, ale alespoň řeší lehké záseky při ztrátě paketu. Používá se často v bojových hrách, kde je odezva hráčových vstupů kritická.

V testu jsou tedy dva hráči, každý má svou postavu s pozicí ve 3D prostoru. Výše zmíněným způsobem provedou X iterací. Postupně v čase jsme pozice obou postav exportovali do CSV pro oba hráče. Udělali jsme dvě měření, jedno pro TCP a druhé pro QUICr. Na obrázku 19 vidíme naměřené hodnoty pozice postavy hráče 1 jak ji viděl hráč 2. V případě TCP hráč silně skákal, právě kvůli ahead-of-line blokování. To snižuje hratelnost. Protokol QUICr je plynulejší a tedy i hratelnější.

Závěr

V práci jsme představili problematiku distribuovaných systémů a soustředili jsme se na konkrétní využití pro hry více hráčů. Popsali jsme různé modely a architektury nejen systému, ale i programů v něm. Jednotlivé atributy, které systémy nebo komunikace mohou mít, jsme identifikovali a přenesli na náš konkrétní případ.

Vytvořili jsme si vlastní herní engine, který umožňuje vytvářet hry pro více hráčů. Programy jsme skládali jako modulární monolity a popisovali jejich architekturu a implementaci. Ukázali jsme, že náš přístup umožnil snadno herní engine rozšiřovat. Pro distribuovaný systém jsme vytvořili hned dva protokoly: jeden v transportní a druhý v aplikační vrstvě. První protokol zrychluje tempo, jakým se zprávy dostanou ke zpracování, oproti TCP. Jinými slovy eliminuje ahead-of-line blokování. Implementovali jsme různé vlastnosti jako handshake a spolehlivost. Druhý protokol umožňuje zprávy rozdělovat podle typu. Na koncových bodech je tak možné pro typ definovat funkci, která každou příchozí zprávu tohoto typu zpracuje. Pro snadnější práci se zprávami jsme přidali serializaci objektů. To znamená, že ve vrstvě, kde implementujeme herní logiku, nepracujeme se zprávami jako n-ticemi bytů, ale objekty.

Na konci jsme definovali, jaké metriky chceme měřit a jaké chceme optimalizovat. Představili jsme nástroje a možnosti měření, které jsme využili. Naše metriky se netýkali jen komunikace v síti, ale i výkonu jednolivých programů. Například jsme naměřili, že v případě stovek hráčů začala být problém serializace a obecně sestavení smysluplných zpráv, nikoliv samotné odesílání. Implementovali a popsali jsme naše řešení. Zároveň jsme dokázali, že implementované optimalizace jsou vhodné nejen pro model klient-server ale i peer-to-peer.

Do budoucna by bylo dobré dokončit další vlastnosti QUICr protokolu, jako například sekvenční číslo zprávy. Bylo by nutné se zamyslet, do jaké hloubky by pořadí zpráv měl řešit QUICr. Aplikace sama nejlépe ví, jestli musí čekat na předchozí zprávu nebo ne, jako v našem případě snapshoty ze serveru. Možnost je definovat předchůdce pro kompletní odstranění ahead-of-line blokování. Protokol také neimplementuje dynamické změny datového toku. Ten je nutné snížit, když je síť zatížená.[2]

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 (softwarového) 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.*`.

Literatura

[1] Peterson Larry L., Davie Bruce S. *Computer networks: a system approach* [online]. 6. vyd. Cambridge: Elsevier, 2022 [cit. 2026-8-3]. ISBN 978-0128182000.

[2] QUIC Working Group. *RFC 9000: QUIC: A UDP-Based Multiplexed and Secure Transport* [online]. 2021 [cit. 2026-8-3]. Dostupný z: ⟨https://www.rfc-editor.org/rfc/rfc9000.html⟩.

[3] Fiedler, Glenn. *Snapshot Interpolation* [online]. 2015 [cit. 2026-8-3]. Dostupný z: ⟨https://www.gafferongames.com/post/snapshot_interpolation/⟩.

[4] Steen, M. van; Tanenbaum, A.S. *Distributed Systems* [online]. 4. vyd. Praha: distributed-systems.net, 2023 [cit. 2026-8-3]. ISBN 978-1543057386.

[5] KABELOVÁ Alena a DOSTÁLEK, Libor. *Velký průvodce protokoly TCP/IP a systémem DNS* [online]. 5. vyd. Brno: Computer Press, a.s., 2008 [cit. 2026-8-3]. ISBN 978-80-251-2236-5.