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

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

Synopsis

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

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

Keywords: text style; thesis; documentation; sample text

Děkuji, děkuji, děkuji.

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

Obsah

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 6 z nich dokonce ani nepodporuje hru jednoho hráče. Dokonce vznikly sporty v počítačových hrách, tzv. „e-sporty“, ve kterých se utkávají profesionální týmy proti sobě a kompetetivních hrách pro více hráčů. Trochu jiný typ her pro více hráčů, ale o nic míň výdělečný, jsou masivně online hry pro více hráčů. Kompetetivní hry jsou rozdělené do zápasů pro pár hráčů a jsou optimalizované pro minimální odezvu mezi akcemi hráče a reakcí systému. Naopak MMO hry jsou masivní a zvládnou masivní online světy pro tisíce hráčů. Optimalizují pro obrovský datový tok z tisíce klientů na server, který musí rozdělit svou zátěž na vícero dalších serverů.

V práci se zaměřujeme na metody optimalizace datového přenosu v síťových systémech ve hrách více hráčů. To znamená nejen velikost dat, ale i odezvu mezi vstupem hráče a reakcí stavu hry. Tyto optimalizace jsou obecné pro síťové systémy, distribuované nevyjímaje. Hru stavíme na vlastním herním enginu tak, abychom různé algoritmy měli pod kontrolou. Například jsme vytvořili vlastní protokol pro obousměrné posílání spolehlivých i nespolehlivých zpráv zvaný QUICr. Popíšeme, jakým způsobem jsme optimalizace měřili a jaké metriky jsou k tomu potřeba.

2 Distribuované systémy

V této kapitole představíme distribuované systémy. Popíšeme, na čem stojí a jaké jsou cíle, pokud se takový systém rozhodneme implementovat. Začneme od síťových systémů, na kterých jsou distribuované systémy stavěné. Ukážeme softwarové a systémové architektury v distribovaných systémech. Poté podrobněji rozebereme komunikaci v síti.

Počítačový systém 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 systémům, ve kterých jsou procesy rozmístěné přes více počítačů a komunikují spolu posíláním zpráv. Výhod je hned několik. Část systému může být umístěna počítači blíž zákazníkovi a snížit tak odezvu. Když jeden proces selže, může být jiný, který plní stejnou službu a může systém udržet v provozu. Konkrétní skupinou jsou distribuované systémy, které mají mnoho procesů rozmístěných přes více počítačů, které aktivně spolupracují a jeví se jako jeden celek.

Příkladem distribuovaného systému je World Wide Web, zkráceně WWW. Jedná se o informační systém, který umožňuje prohlížet, ukládat a odkazovat 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 snadno přidat svůj server se svými dokumenty, které se tak stanou dostupné v systému. To zároveň rozloží zátěž přes více počítačů a systém tak zvládá miliardy požadavků denně. Zároveň se celý systém jeví jako jeden celek, který z URL adresy vyhledá server a dokument vrátí.

2.1 Architektury

V praktické části jsme navrhli síťový systém. V návrhu jsme využívali známé vzory, které představíme. Návrh systému rozdělíme na dvě části: softwarovou a systémovou architekturu. V softwarové architektuře řešíme komponenty a konektory mezi nimi. Mezi typy softwarových architektur patří například vrstvená, orientovaná na služby nebo publish-subscribe. 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 definuje, že databáze jako služba pro výpočetní vrstvu má roli serveru. Naopak služba pro replikaci ve skupině databázových serverů má roli peer-to-peer. Typy si rozebereme podrobněji v této kapitole.

2.1.1 Softwarová architektura

V první části se zaměříme na softwarovou architekturu. Budeme řešit komponenty a konektory mezi nimi. Představíme tři známé vzory: vrstvená architek-

tura, 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. Občas je možné, aby nižší vrstva volala vyšší, ale mělo by se dít přes rozhraní definované nižší vrstvou, které vyšší vrstva pouze implementuje. 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. Namísto toho lze software organizovat do nezávislých entit, kde každá zapouzdřuje službu. Takovým entitám se říká: služba, objekt nebo mikroslužba. Na komponenty se můžeme dívat jako na objekty a konektory mezi nimi jsou volání metod. Instance objektu mohou být rozmístěny na více počítačích. To, že jiný objekt je na jiném počítači, by mělo být skryté. Když chce klient použít jiný objekt, lokálně si vytvoří jeho instanci, která ale po zavolání metody volání zabalí do zprávy a odešle objektu, který metodu implementuje. Tomuto modelu se říká „remote procedure call“, nebo zkráceně RPC. Klient obdrží zprávu s výsledkem, kterou rozbalí a pokračuje v běhu. Lokální instanci se někdy říká „proxy“ nebo „klient-stub“.

Co občas může vadit je, že jsou služby „referenčně vázané“. To znamená, že služby musejí znát adresu nebo jméno jiné, pokud ji chtějí používat. Od tohoto omezení můžeme upustit například tím, že budou „publikovat“ události do různých „témat“ a dělat na tyto témata dotazy. Pokud například služba publikuje událost do tématu A, jiná služba, která udělá dotaz na téma A, dostane právě tuto událost. Odesílatel tak neví, kdo zprávu zpracuje, kolikrát a kdy. Implementace je přes „brokera“, který slouží jako jednotný bod pro všechny procesy, do kterého publikují události a broker je třídí vhodným jiným procesům. Tento princip lze využít u systémů pro hry více hráčů pro komunikační kanály. Klient hráče chce například napsat do lokálního kanálu pro město, ve kterém se v herním světě nachází. Komponenta pro chatovou službu tuto zprávu zařadí do správného kanálu a rozešle klientům, kteří tento kanál také odebírají.

2.1.2 Systémové architektury

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

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 přijmout fakt ze serveru.

Symetrická architektura, kdy obě strany jsou si rovny, se nazývá „peer-to-peer“. Znamená to, že obě strany mohou posílat požadavky a zprávy na druhou stranu. Využívá se například při replikaci, kdy máme několik replik, které jsou si rovny. Pouze řešíme, která vlastní jaký zdroj tak, aby nedošlo ke konfliktu dvou replik při psaní konkrétního záznamu. Libovolná replika má ale možnost odeslat svůj stav jiné, protože jsou si rovny. Tento model je vhodný například pro horizontální škálování stavových serverů, které si rozdělují část stejné služby. Například vícero serverů může zpracovávat různé části herního světa a předávají si mezi sebou entity, které mají na starost.

2.2 Komunikace

V síťovém systému není sdílená paměť. Místo toho se pro komunikaci mezi procesy používá posíláním primitivních zpráv pomocí systémových volání. To se snažíme v distribuovaných systémech skrýt za vhodnějším rozhraním. V této podkapitole si představíme různé abstrakce a v podkapitole 2.3 ukážeme podrobněji, jak se taková komunikace realizuje.

Pro schování komunikace a platformy, na které proces běží, se využívá „middleware“. Jedná se o komponentu, která 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` přiřadí síťovou IP adresu `77.75.77.222`.

Pro různé účely a situace máme jiné modely komunikace. 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 zpráva je uložena jen po dobu, kdy běží proces odesílatele a příjemce. To znamená, že pokud příjemce není dostupný, zprávu nikdy nepřijme. Protokoly v transportní vrstvě jsou transientní.

Dále může být komunikace „synchronní“ nebo „asynchronní“. Asynchronní znamená, že odesílatel pokračuje v běhu okamžitě po odeslání zprávy. Ta 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 tedy definujeme tři body, kde může nastat synchronizace (proces odesílatele pokračuje v práci). 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.

2.2.1 Modely pro komunikaci

Pro každý účel musíme vhodně zvolit konfiguraci perzistence a synchronizace. Představíme dva modely komunikace, které posílání primitivních zpráv skrývají

a jsou implementovány jako middleware: Message Oriented Middleware (MOM) a Remote Procedure Call (RPC).

Prvním přístupem je Remote Procedure Call, který procesu umožňuje zavolat lokální proceduru s implementací na jiném počítači. Když proces A zavolá proceduru na počítači B, A se pozastaví a začne se vykonávat na B. Jakmile se dokončí, pošle se výsledek zpět na A, které poté pokračuje. Tento model je synchronní a transientní. Cílem je, aby volání vypadalo jako by implementace byla lokální a skrýt tak komunikaci mezi počítači.

Jedná se o intuitivní řešení, ale přináší pár 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 systémového jména, nikoliv fyzické adresy. Tento přístup méně schovává fakt, že procesy jsou rozmístěny na více počítačích a abstrakce z fyzických adres na jména pomáhá. 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.

2.3 Počítačová Síť

Nedílnou součástí distribuovaného systému je síť, přes kterou mohou procesy posílat primitivní zprávy. V této podkapitole se zaměříme na komunikaci mezi dvěma procesy v síti, a to i těch, které nejsou součástí jednoho síťového systému. Počítačová síť je skupina propojených počítačů, které si mezi sebou přenášejí data. Například lokální sítě (LAN) propojují až tisíce počítačů, které jsou geograficky blízko, například v rámci jedné budovy. Rozsáhlé sítě (WAN) propojují miliony různých zařízení po celém světě. Příkladem je síť Internet.

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

Dva počítače, přímo propojené fyzickým médiem, komunikují posíláním n-tic bytů zvané „rámce“. Variantou jsou „přepínané“ sítě, které nemusejí mít mezi všemi dvojicemi počítačů přímé spojení, ale využívají „přepínače“. Proces, kdy doručujeme zprávu na cílový počítač přes vícero přepínačů se nazývá „směrování“. Každý vrchol v takové síti musí mít přiřazenou unikátní „síťovou adresu“ a posílali se formátované rámce zvané „pakety“. Ty se skládají z hlavičky, ve

které najdeme informace ke směrovaní, a tělu obsahujícím samotnou přenášenou zprávu.

2.4 Komunikace v síti

V této části představíme jak je realizována komunikace v síti. Aby si dva počítače navzájem rozuměly, musejí se shodnout na významu jednotlivých bitů rámců, které si mezi sebou budou posílat. K takové definici slouží „komunikační protokol“: soubor pravidel pro výměnu informací mezi počítači. Definuje syntaxi, sémantiku a synchronizaci zpráv.

Každý protokol poskytuje komunikační služby. Tyto služby rozdělujeme na dvě skupiny: ty co před komunikací navážou spojení a ty co ne. 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 telefoní linka. V druhém případě může odeslat zprávu kdykoliv a bez předchozího upozornění druhé strany. Příkladem takové komunikace je posílání emailu.

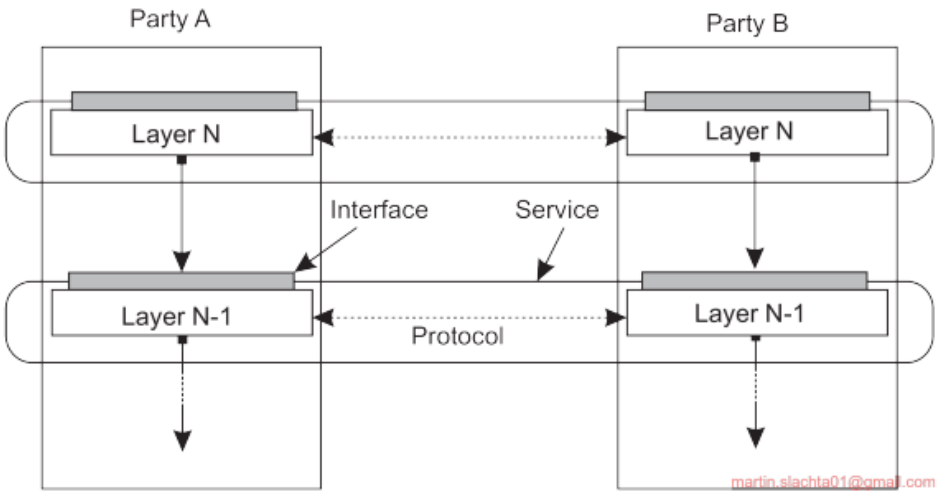
2.4.1 Rodina protokolů TCP/IP

Rodina protokolů pro komunikaci v síti Internet je TCP/IP. Jedná se o více protokolů organizovaných do vrstev, kde každá může používat jen tu pod ní. Vrstvenou architekturu vidíme na obrázku 1. Vrstva N používá rozhraní vrstvy $N-1$. Každá vrstva poskytuje komunikační služby. Počítač A i B tak vidí stejnou službu. Představme si, že rozhraní má dvě funkce: pro čtení a pro zápis. Aplikace do vrstvy N zapíše zprávu, kterou chce, aby si proces využívající stejnou službu, ale na druhém počítači, mohl přečíst. Je potřeba, aby vrstva zapsala zprávu ve formátu, kterému bude rozumnět strana A i strana B. Tento formát je právě protokol, jak vidíme na obrázku 1.

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

Protokol IP má za úkol doručit paket od odesílatele na příjemce pouze podle IP adresy, která je v hlavičce paketu. IP nenavazuje spojení a nezaručuje doručení. Důvodem je princip „end-to-end“, který říká, že body mezi odesílatelem a příjemcem, jako routery a přepínače, by měli být co nejjednodušší. Spolehlivost musejí zaručit až dva koncové body, například číslováním paketů a sledováním stavu doručení. V případě, že paket chybí, ho musí odesílatel odeslat znova.

Router udržuje frontu příchozích paketů, která má fixní velikost. Jakmile objem příchozích paketů překročí kapacitu fronty, router má různé politiky, jak se v takové situaci zachovat. Nejběžnější je politika „tail-drop“, kdy paket zahodí. Proto musejí koncové body hlídat a měřit zahlcení sítě. Měli by hlídat, jak moc paketů se ztrácí a případně snížit frekvenci odesílání. V TCP/IP se o to stará



Obrázek 1: Vrstvená architektura TODO předělat

protokol TCP, který si má frontu paketů, které musí odeslat a udržuje si ve frontě okno paketů, které jsou odeslané ale nepotvrzené neboli „in-flight“. Pokud není potvrzeno až příliš paketů, okno zmenší a tím sníží počet paketů, které mohou být najednou v síti.

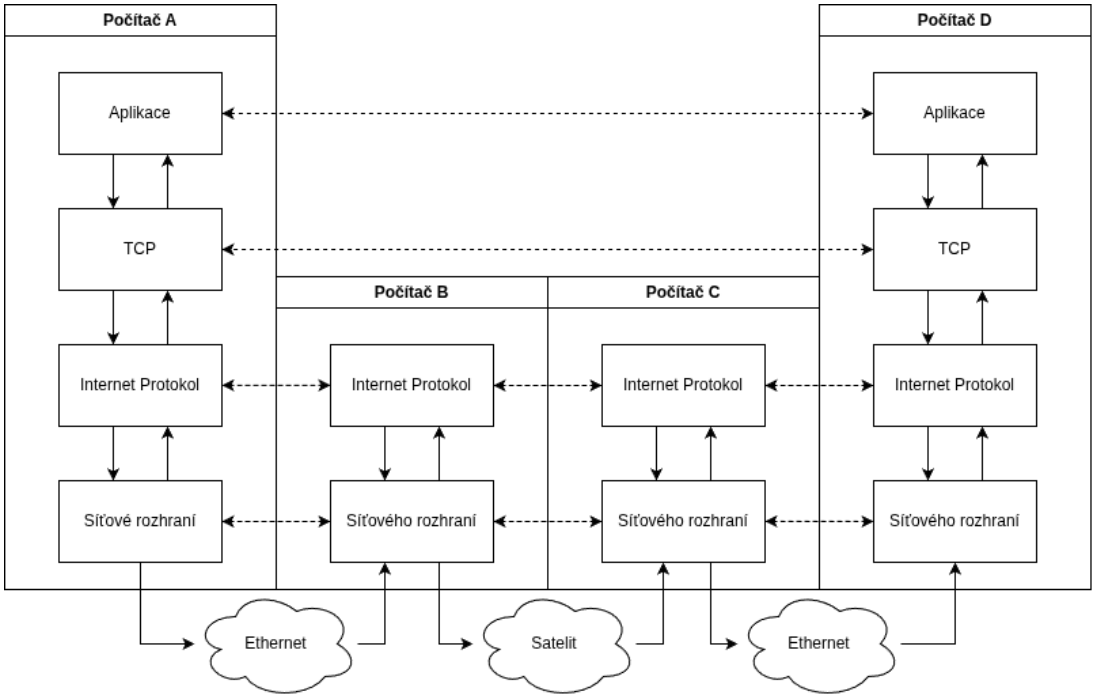
Konkrétně architektura TCP/IP je rozdělena do čtyř vrstev: aplikační, transportní, síťová a síťové rozhraní. Každá vrstva obsahuje množinu protokolů a pro každou situaci lze sestavit ideální čtveřice. Vrstvy podrobněji představíme.

Aplikační
: Nejvyšší je „aplikační“ vrstva, ve které jsou aplikace. Příkladem protokolů jsou FTP, SMTP, HTTP nebo gRPC.

Transportní
: Druhá vrstva je „transportní“. Tato vrstva, stejně jako aplikační, je rozbalena až na koncových bodech, jak vidíme na obrázku 2. Proto se jim někdy říká end-to-end protokoly. Není totiž důležitá pro směrování paketů. Stará se o to, jaké přišli, v jakém pořadí a jestli nejsou poškozené. Příkladem protokolů jsou TCP, UDP nebo QUIC.

Síťová
: Pro adresaci je 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



Obrázek 2: Komunikace vrstev TCP/IP

média bude vyžadovat různé protokoly. Příkladem jsou Ethernet, Token ring, apod.

Na obrázku 2 vidíme, jak mezi sebou jednotlivé vrstvy komunikují. Aplikace na počítači *A* zapíše 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. Šrafovaná čára reprezentuje abstrahovaný datový tok tak, jak ho vidí vrstva. Počítače B a C jsou propojovací body. Všimněme si, že se nepoužívají transportní nebo aplikační vrstvy, pouze si rozbalí IP paket a zjistí, kam mají pakety posílat dál. Vrstva síťového rozhraní pracuje s různými médii, jak také vidíme na obrázku.

2.5 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, gRPC nebo DNS, které jsou pro obecné posílání zpráv.

2.5.1 HTTP

Velmi používaný protokol v systému World Wide Web je HTTP, neboli Hyper Text Transfer Protokol. Především slouží 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 to 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 je krátký a podstatný, může první dotaz blokovat ten druhý. Klienti by se měli snažit nejprve odeslat jednoduché dotazy a ty zdlouhavější až potom.

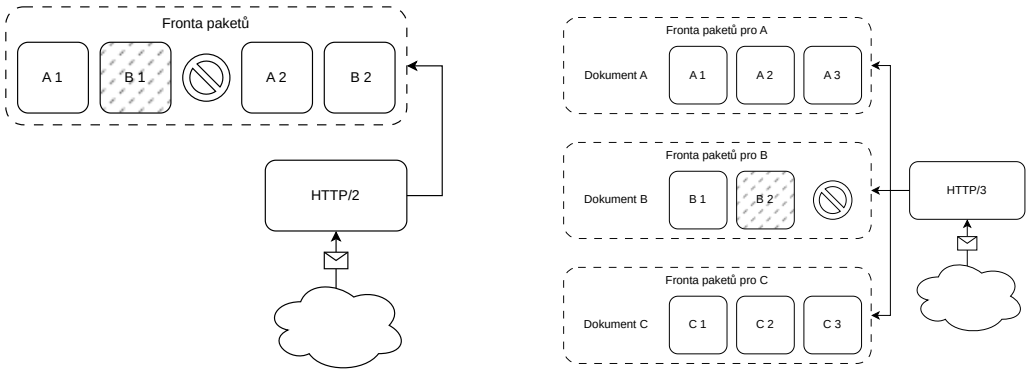
2.5.2 HTTP/3

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 tzv. head-of-line blokování. Na obrázku 3 vidíme porovnání TCP v HTTP/2 a QUIC v HTTP/3. V prvním případě stahujeme dva dokumenty: A z částí $A1$ a $A2$ a B z částí $B1$ a $B2$. Když server odesílá v pořadí $A1$, $B1$, $A2$ a $B2$, a $B1$ se nedoručí, tak i když už se stihli doručit $A2$ a $B2$, tak je nemůžeme začít zpracovávat, protože protokol čeká na B1. Na druhou stranu protokol QUIC může mít pro každý dokument vlastní frontu a tím blokování kompletně eliminovat.

2.6 Protokoly v transportní vrstvě

Protokoly transportní vrstvy poskytují end-to-end komunikační služby pro aplikace. Příkladem takových služeb může být 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. Poslední příklad, který zmíníme, je protokol QUIC od Google, který poskytuje spolehlivou komunikaci přes navázané



(a) HTTP/2 (b) HTTP/3

Obrázek 3: Vizualizace HTTP/2 blokování

spojení a s podporou multiplexování. Byl důležitou inspirací pro náš vlastní protokol z praktické části.

2.6.1 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 si pakety čísluje a příjemce musí za každý odeslat potvrzení přijetí. Pokud odesílatel toto potvrzení nedostane, odešle paket znovu. Pokud nedostává potvrzení pro až moc velkou část okna, může toto okno zmenšit.

Pro navázání spojení a následnou komunikaci musejí oba procesy vytvořit koncový bod, kterému se říká „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 musí jedna strana spojení iniciovat a druhá ho musí přijmout.

Popíšeme si, jak TCP zajišťuje spolehlivost pomocí odesílání potvrzení o přijetí. Pokaždé, co 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, a příjemci přijde nejprve C a B, nebude moct příjemce ze soketu nic přečíst, dokud nepřijde i část A. Většinou je toto vhodná služba, ale jsou situace, například ve hrách, kdy se to

stává problémem. Například posíláním zpráv s pozicí hráče. Pokud strana pošle pozici pro čas t_1, t_2 a pak t_3, tak by 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 začala zpracovávat t_3 hned a docílila tak nižší odezvu mezi serverem a klientem.

Nevýhoda pro nás byla, 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.

2.6.2 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, že to co vrátí jedno systémové volání pro čtení ze soketu je právě jedno volání pro zápis, které udělal odesílatel. 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.

2.6.3 QUIC

QUIC je spolehlivý protokol transportní vrstvy, který poskytuje multiplexní komunikaci založenou na TLS 1.3, od společnosti Google. Jeho cílem je být efektivnější varianta TCP, která využívá protokol UDP. Znamená to například, že lze snížit množství blokování, ke kterému dochází z důvodu ztracených paketů. Mezi dvěma koncovými body vytváří spolehlivé spojení. Na rozdíl od TCP umožňuje otevřít více proudů v jednom spojení a modelovat tak částečné uspořádání. Když se čeká na paket v TCP, tak se pozastaví celý tok dat, ale v případě QUIC se zastaví jen konkrétní proud.

Mezi dvěma koncovými body se posílají pakety. Ty obsahují rámce, které se rozdělují na kontrolní a proudové. Kontrolní slouží pro ovládání koncových bodů: otevírání a zavírání proudů, přesun ve stavovém stroji koncového bodu nebo udržování spojení naživu. Proudové obsahují aplikační data. Stejně jako TCP, i QUIC přenáší proud bytů a oddělení jednotlivých zpráv je nutné implementovat zvlášť.

Pakety

Dvě strany komunikující přes QUIC protokol mezi sebou posílají sekvenční pakety různých typů. Příkladem jsou: `Initial`, `0-RTT`, `Handshake` a `1-RTT`. Liší se v síle šifrování, které používají, každý typ má svůj číselný prostor a liší se i hlavička. Každý paket má své unikátní pořadové číslo,

které se v rozumném čase nesmí použít znova, a to ani při opakovaném poslání stejného paketu z důvodu ztráty. Různé typy paketů mají ale čítač nezávislý. Pakety typu `Initial` a `Handshake` mají dlouhou hlavičku a `0-RTT` a `1-RTT` mají krátkou hlavičku.

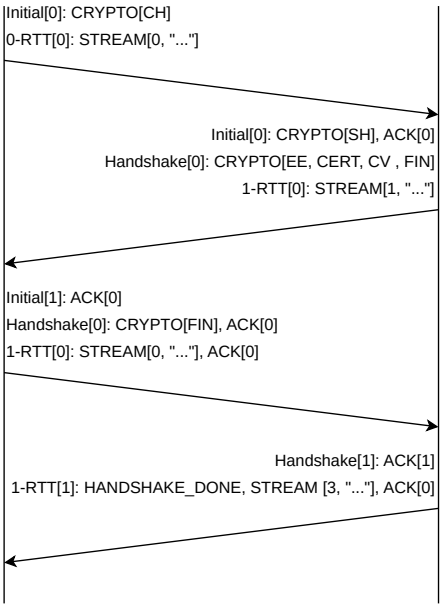
Spolehlivost
Protokol QUIC zajišťuje spolehlivost jednotlivých rámců. Příjemce musí odeslat zpět odesílateli rámec typu `ACK` se zakódovaným číslem paketu, ve kterém rámce získal. Tento rámec může obsahovat i více paketů zároveň a protokol toho využívá tak, že neposílá `ACK` tak často.

Šifrování
QUIC už podporuje šifrování přes TLS 1.3 přímo v transportní vrstvě. To znamená dvě výhody: protokol autentizuje klienta, který nám posílá zprávy a tajný klíč lze domluvit už při navazování spojení. TCP naváže spojení pomocí handshake a následně pokračuje další handshake pro TLS. QUIC je v tomto ohledu schopný navázat spojení rychleji.

Unikátní identifikátory koncových bodů
Protokol UDP směřuje datagramy do portů a identifikovat klienty umožňuje pomocí zdrojové IP adresy. Protokol QUIC přidává další vrstvu směrování pomocí „unikátních identifikátorů koncových bodů“. Před začátkem komunikace se oba koncové body domluví na svých identifikátorech a odesílatel do hlavičky paketu zapíše identifikátor cílového spojení. Toto číslo je náhodně vygenerované a s délkou až 62 bitů. Díky šifrování nemůže potenciální útočník toto číslo podvrhnout a vydávat se za někoho jiného, protože kvůli neznalosti klíče by nebyl schopný správně zašifrovat zprávu. Výhoda je, že klient může změnit IP adresu, např. z důvodu přepnutí z Wi-Fi na mobilní data, není třeba spojení znova navazovat. V případě TCP se často stává, že spojení musí vypršet, a pak znova navázat.

Popíšeme, jak v QUIC funguje handshake, který vidíme na obrázku 4. Nejprve začíná strana, která se chce připojit, posláním paketu typu `Initial` s rámcem CRYPTO s ClientHello. Server odpoví několika rámci: `Initial` s `CRYPTO` rámcem se ServerHello společně s ACK rámcem a paketem typu `Handshake` ve kterém je `CRYPTO` rámec s potřebnými informacemi pro TLS. Aplikační data jsou v paketu typu `1-RTT` a `0-RTT`. Jakmile klient dostane tyto informace, mohou si strany posílat šifrovaná data. Toho QUIC využívá a první odpověď ze serveru už může obsahovat paket `1-RTT` se `STREAM` rámci s šifrovanými daty, jak vidíme na obrázku 4. Dále si všimněme, že každý typ paketu má vlastní prostor čísel. QUIC používá UDP a proto posílá datagramy. Do jednoho datagramu spojuje více paketů. Tři pakety ze serveru, které přijdou jako odpověď, se tak mohou odeslat jako jeden datagram.

Při handshake se dva koncové body mimo jiné domlouvají na svých unikátních číslech. Při Client-Hello se server podívá na Source Connection ID a zapamatuje si ho jako ID připojení druhé strany. Následně vygeneruje své vlastní ID a přiřadí



Obrázek 4: QUIC handshake

mu instanci relace. Pak se podívá na Destination Connection ID a nastaví si, aby pakety, které mají toto číslo jako Destination Connection ID, se připisovaly nově vytvořené relaci. Toto číslo jako takové nebude používat a v odpovědi jako Source Connection ID bude jeho vlastní vygenerované.

Všimněme si speciálního typu paketu `0-RTT`, který slouží pro poslání aplikačních dat ještě před samotným navázáním spojení, ale pouze se základním šifrováním. Pro tento paket se využívá první Destination Connection ID, které ale vygeneroval iniciátor komunikace. Jedno spojení může mít přiřazeno více ID. Spojení, které si založí druhá strana, si přiřadí jak ID vygenerované touto stranou, tak ID, které zvolil iniciátor. Když přijde později paket `0-RTT`, bude správně nasměrován na instanci spojení.

Způsobem, jakým QUIC redukuje nebo dokonce eliminuje head-of-line blocking, je multiplexing. Přes jedno spojení je možné otevřít více paralelních proudů, které se navzájem neovlivňují.

2.7 Sokety

Populární rozhraní pro použití protokolů v transportní vrstvě jsou tzv. „Berkeley sokety“. Soket je komunikační koncový bod, do kterého aplikace zapisuje data, která chce poslat přes síť druhému procesu, a ze kterého zároveň může číst příchozí data. Jedná se o abstrakci nad konkrétním otevřeným portem. Nelze

vytvořit dva sokety se stejným portem.

2.8 Asynchroní vstupně výstupní operace

Čtení a zápis na socket je běžně blokující operace. To znamená, že při systémovém volání pro čtení 10 bytů, bude volání blokovat, dokud na socket nepřijde 10 bytů. V případě serveru pro online hru je to nevhodné, protože pokud nám nic nepřišlo, můžeme mezitím vykonávat jinou logiku hry. Socket lze ale nastavit tak, aby vracel hned a počet bytů neznamenal konkrétní počet, ale pouze maximum. Je ale poté třeba zprávu opět sestavit z postupně přečtených částí v aplikaci.

Podobnému přístupu se říká „asynchronní sockety“. Jinými slovy to znamená oddělení volání a samotného vykonání. V tomto případě sice zavoláme operaci čtení, ale toto volání se vloží do fronty, a jiné samostatné vlákno ho vykoná asynchroně později. Protějšek z reálného světa je hození dopisu do schránky a pokračování v jiné práci. Doručení se stane v nejbližších dnech a to někým jiným, protějšek jiného vlákna. Pro lepší představu si pár existujících implementací představíme.

2.8.1 ZeroMQ

Při použití knihovny ZeroMQ vytváříme objekt kontextu pomocí ctx=zmq_context(num_threa
Ten obsahuje konkrétní počet vláken specifikovaný parametrem num_thread. Když následně zavoláme zmq_send(ctx, x), protějšek klasického send(x), tak nedojde k blokování. Místo toho se zpráva x umístí do fronty, ze které pak vlákna z zmq_context tzv. „kradou“ a odesílají v pozadí.

To samé platí o zmq_receive, které si v pozadí skládá celou zprávu. TCP totiž nemá žádnou strukturu zprávy, a proto když zavoláme operaci pro čtení ze soketu, můžeme dostat jen část. ZeroMQ pracuje v proudu s celými zprávami, které za nás skládá.

2.8.2 Work stealing

Běžnou strategií pro využití dynamického počtu vláken je „work stealing“. Jiné strategie jsou třeba „work sharing“. Implementace obsahuje synchronizovanou frontu, do které vkládáme popisy úkolů, které se mají vykonat. Vlákno, které se snaží z této fronty krást, je „worker“ neboli pracovník. Pokud je fronta správně synchronizovaná, může z ní krást libovolný počet vláken. Jsou různé možnosti, co může být popis úkolu. Například anonymní funkce nebo ID. Zajímavou vlastností tohoto přístupu je, že součást úkolu může být vložení

2.8.3 Async/Await

Populární implementací, kterou vidíme například v Rust nebo C#, je „async/await“. V programu označíme funkci jako asynchronní pomocí klíčového slova async. Zavoláním takové funkce speciálním await voláním způsobí, že se celý aktuální

kontext vloží jako úkol do fronty, ze které pracovníci kradou. Tento přístup vyžaduje, aby měly prvky ve frontě definované závislosti mezi sebou, protože úkol s aktuálním kontextem bude závislý na dokončení právě funkce, kterou jsou zavolali pomocí await. Toto volání tak do fronty vložíme taky. Pro definice závislostí se používají ukazatele na čítače. Pokud úkol A závisí na úkolu B, nastaví si A čítač C_A na hodnotu 1 a B dostane na čítač C_A ukazatel. Po dokončení tento čítač dekrementuje. Pracovník může krást z fronty jen ty úkoly, které mají hodnotu čítače roven nule.

Tento přístup je často používán v kontext asynchronních vstupně výstupních operací. Vlákno, které čeká na blokující operaci může v mezičase pracovat na jiném úkolu v aplikaci namísto čekání. V základu metoda neslouží k paralelismu, k tomu je potřeba další operace zvaná „počkej na všechny“, která vloží do fronty vícero úkolů a aktuální kontext, který závisí právě na všech těchto úkolech. Ty tak mají šanci běžet paralelně.

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

2.8.4 Boost Asio

Velmi podobná ZeroMQ je Boost ASIO. Ta ale nechává vytváření vláken na nás. Pouze vytvoříme objekt kontextu asio_context, který má frontu úkolů a metodu run. Když z vlákna zavoláme tuto metodu, stane se vlákno pracovníkem, který krade práci z fronty, která je v objektu kontextu.

Tento přístup je intuitivnější, protože se jedná přesně o to, jak pracovníci fungují. Jedná se o funkci, která se opakovaně snaží odebrat z fronty úkolů. Pokud fronta není prázdná a pracovník se dostane z ní úkol, začne na něm pracovat. Jakmile práci dokončí, vrací se ke kroku odebírání z fronty.

Důvod, proč nevytvořit thread pro každou odeslanou zprávu je, že vyžaduje systémové volání, které může způsobit zpomalení. Proto se často používají tzv. green thready, fibery, user thready, ... což jsou vlákna kompletně v uživatelském prostoru. Není tak třeba žádné systémové volání.

3 Hra

V této kapitole představíme hru, kterou jsem vyvinuli pro implementaci různých technik a jejich následného 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.

3.1 Engine

Pro hru jsme vyvinuli vlastní herní engine tak, abychom mohli celý systém do hloubky upravovat. V této části popíšeme jeho základní komponenty. 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. Jednotlivé moduly představíme.

Program hry jsme rozdělili na „stav“ a „řídící logiku“. Stav obsahuje množinu entit, které jsou ve hře a jejich vlasnosti a atributy. Entity jsou objekty v herním světě, jako postava hráče, truhly s poklady nebo nepřátele. 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“.

Druhá část je řídící logika, která po iteracích 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.

https://www.riotgames.com/en/news/peeking-valorants-netcode

3.1.1 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.

3.1.2 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.

3.1.3 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.

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

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 na TODO, 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.

```
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

Představíme komponenty, které jsme pro hru definovali a proč:

Transform

Obsahuje transformační matici. Entity, které chceme zobrazit ve světě, tuto

komponentu potřebují. Příkladem entity, která ji mít nebude, je zpráva v chatu.

Mesh

Obsahuje mesh, který má hra využít pro vykreslení entity. Mesh je n-tici bodů, které dohromady dávají 3D povrch objektu. 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 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 5. 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. Ten 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“. Nevýhoda je, že všichni hráči musejí mít historii stejnou, jinak dochází k desynchronizaci.

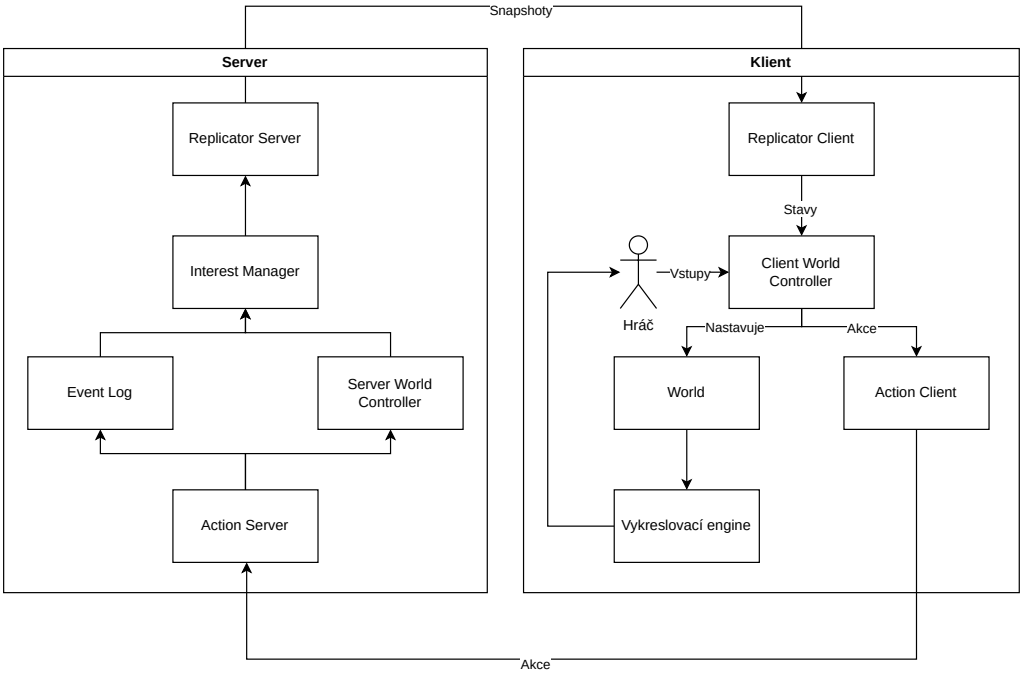
Druhá komplikace je, že výsledek každé aplikace akce být všude stejný. Problém nastává u generování pseudo-náhodných čísel nebo u integrace. Například fyzický engine postupně integruje pozice entit podle derivace, která je reprezentována vektorem zrychlení. Každá integrace musí definovat, o jak velký 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.

Řídící logiku jsme rozdělili na Server World Controller a Client World Controller. Toto rozdělení nás motivovalo dále oddělit integraci světa do vlastní třídy tak, aby logika mohla být sdílená mezi ovladačem pro jednoho hráče a pro více hráčů.

4.1 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“ a „manažer zájmu“. Replikátor 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 se-



Obrázek 5: Softwarová architektura hry

znam 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 z hry pro jednoho hráče a skrývá distribuovanost pomocí vrstvené architektury.

4.1.1 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`.

4.1.2 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.

4.1.3 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, takže je nutné ji synchronizovat, a nedůležitá, která není. 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.

4.2 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.

4.2.1 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.

4.2.2 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. Problém ale najednou byl neplynulost hry. Klient sice vykresloval s frekvencí 60Hz, ale méně časté aktualizace způsobili neplynulý pohyb. Řešení je na klientovi interpolovat spojité proměnné,

jako pozice nebo rotace. Do klienta replikátoru jsme přidali komponentu „interpolátor“. Ta si ukládá historii konkrétních hodnot a jejich časovou známkou do bufferu. Hodnota, kterou nastaví, je pak interpolovaná z bufferu fixní čas zpět.

4.2.3 Rollback

Interpolovat zlepšilo plynulost, ale nepříjemně zvýšilo 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, ale navíc bude nižší než před interpolací.

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.

4.3 Posílání zpráv

Popíšeme náš middleware pro posílání zpráv a jeho protokol založený na TCP. Poskytuje službu oboustranného peer-to-peer posílání zpráv na různé koncové body, které jsou identifikované čtyř bytovým číslem. Strana může pro libovolný identifikátor definovat handler, který se po získání takové zprávy spustí. Protokol zajišťuje spolehlivost na úrovni zpráv. To znamená, že zpráva dorazí buď celá, nebo vůbec. Později v kapitole **??** představíme náš protokol v transportní vrstvě, který umožňuje posílat zprávy spolehlivě i nespolehlivě.

4.3.1 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. 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.

4.3.2 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 lambdu. 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.

4.3.3 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.

4.4 Serializace

Zpráva je v programu reprezentovaná objektem. Proces, 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. Jeho výhoda je možnost vygenerovat třídy v různých programovacích jazycích. Pokud máme architekturu orientovanou na služby, kde každá je implementována v jiném jazyce, 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. Pokud 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.

4.5 Metriky

Představíme metody měření a metriky v systému, které jsme využili pro hledání potenciálních míst pro optimalizaci. Pro sbírání metrik je třeba mít úložiště, kam budeme metriky ukládat a rozhraní, přes které je budeme vizualizovat. Úložiště může být například CSV soubor nebo relační databáze. Existují optimalizované databáze pro vkládání dat s časovou známkou, kterým se říká „timescale“. Jsou ideální právě pro metriky nebo logy, protože narozdíl od klasických databází jsou optimalizované pro rychlé vkládání. Rozhodli jsme se použít relační databázi „PostgreSQL“ s rozšířením „TimescaleDB“. Pro vizualizaci jsme použili službu „Grafana“, která umí z TimescaleDB číst. Všechny tyto služby popíšeme v následujících částech.

Obrázek 6: Příklad Grafana dashboardu

4.5.1 PostgreSQL

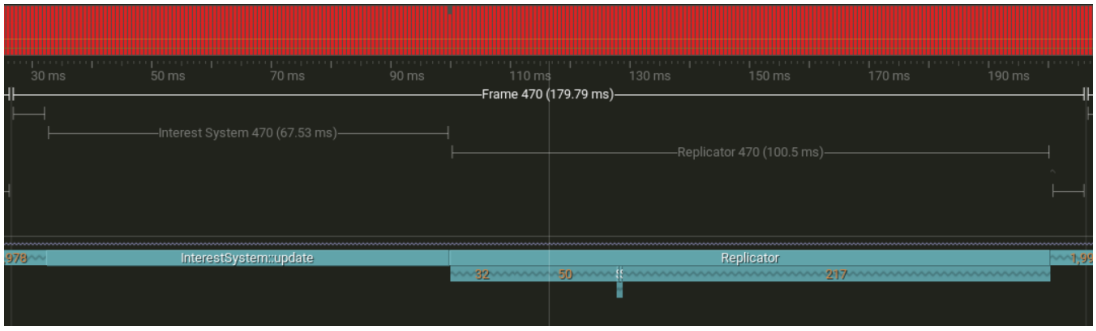
Populární relační databáze je PostgreSQL. Rozhodli jsme se právě pro tu, protože je to otevřený a svobodný software zdarma. Lze si ho snadno nasadit lokálně, ať už daemoném nebo jako Docker kontejner. Skrze rozšíření jako TimescaleDB lze přidat podporu a optimalizace pro různé použití. V našem případě se chceme ukládat informace o hráčích a stav světa pro perzistenci a metriky. Informace o hráčích a stav světa zvládne bez problémů v základní verzi. Pro metriky použijeme zmíněné rozšíření.

4.5.2 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 tzv. hypertables, které jsou optimalizované pro rychlé vkládání. Vytvořili jsme komponentu `NetworkMetricsReporter`, která má na rozhraní různé metody pro získávání metrik. Např. `report_outbound` a `report_inbound`. Tyto údaje posílá do komponenty `TimescaleDbWriter`. To je konkrétní implementace pro TimescaleDB, které má na rozhraní metodu `report (metric_name, value)`. V rámci optimalizace si `NetworkMetricsReporter` metriky ukládá do bufferu, který až pomocí metody `flush` odešle do databáze. To se neděje nutně po každé metrice, ale v našem případě si reportér ukládá hodnoty do bufferu a po intervalech je vkládá do databáze.

4.5.3 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.



Obrázek 7: Jeden snímek v Tracy

4.5.4 Orchestrace

Popíšeme, jak jsme všechny služby zprovoznili a nastavili. Podrobněji se na orchestraci podíváme později.

4.5.5 ImGui

Pro uživatelské rozhraní jsme použili knihovnu ImGui a ImPlot. Tyto knihovny využívají pro vykreslování grafickou kartu a snadno se dali integrovat do našeho vykreslovacího enginu. Knihovna funguje procedurálně, nikoliv objektově. Každý snímek, který vykreslujeme, musíme celé rozhraní definovat znova. To uděláme pomocí volání funkcí. Nejprve zavoláme funkci `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.

Knihovnu jsme využili pro vykreslení metrik přímo uživateli klienta.

4.5.6 Tracy

Pro měření a pokročilejší analýzu výkonu programu klienta nebo serveru používám profiler Tracy. 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 7 vidíme, jak taková analýza vypadá. Vidíme právě jeden snímek. V něm máme označené části replikátoru a manažera zájmu. Také vidíme dole zóny.

4.5.7 Konkrétní metriky

Vyjmenujeme metriky, které budeme měřit. Je dobré zmínit, že místo, ze kterého budeme metriky sbírat, je server.

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

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

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

4.5.8 Sbírání v reálném čase

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

4.6 Protokol QUICr

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

Protokol je obousměrný proud paketů se spolehlivými a nespolehlivými rámci využívající UDP. Jeden poslaný UDP datagram může mít více paketů. Rámce můžeme rozdělit na dva typy: datové a ovládací. Ovládací slouží pro handshake, ukončení spojení nebo nastavení jiných parametrů spojení a budou vždy spolehlivě doručené. Datové obsahují aplikační data a mohou definovat svou spolehlivost. Ty, které budou obsahovat pozice hráčů, tak budeme moct odesílat nespolehlivě, zatímco změny stavu, jako výměna mesh, posíláme spolehlivě. Architektura se skládá ze dvou komponent: „koncový bod“ a „spojení“.

Koncový bod se stará o soket pro UDP: čtení a zápis na něj, a udržuje si registr navázaných spojení. Při přečtení datagramu, který splňuje formát, ho

správně nasměruje podle registru. Nechtěli jsme, aby si implementace QUICr spravovala vlastní asynchronní kontext. Namísto toho jsme definovali metodu pro tik, která posbírá zprávy od jednotlivých navázaných spojení a pošle je. Následně přečte všechna data ze soketu a roztřídí je do navázaných 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 kontrolní rámce. Jeden objekt navázaného spojení může mít přiřazených více ID. Tento objekt si datagram přečte a přeloží na pakety a rámce. Ty postupně prochází. Stavový stroj bude důležitý hlavně v procesu handshake, který definujeme níže.

4.6.1 Rámec

Hlavním prvkem protokolu jsou právě 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 4.6.3.

4.6.2 Handshake

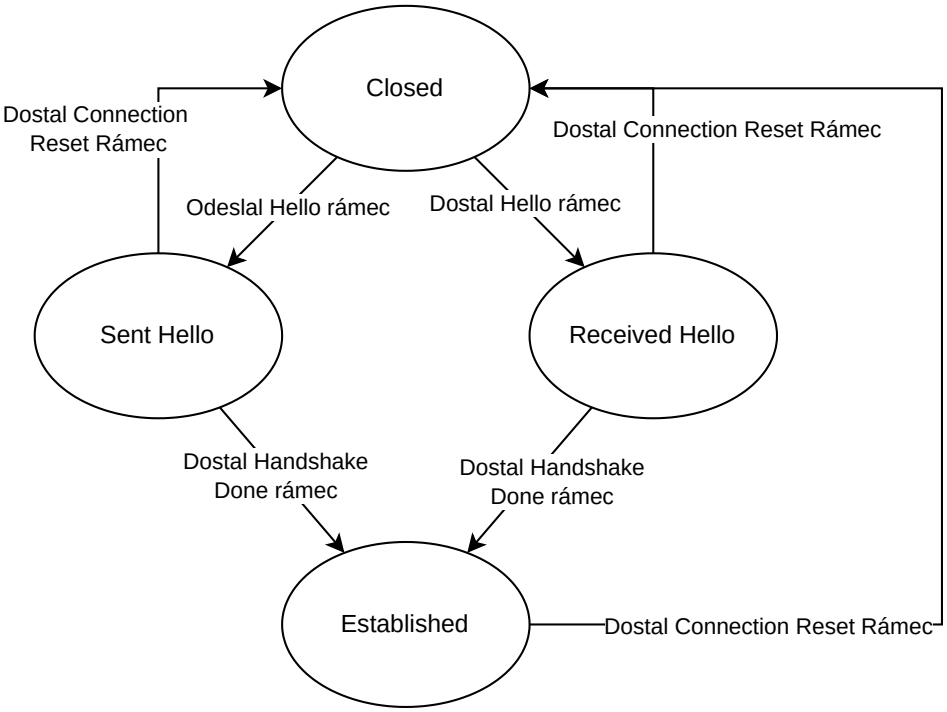
Objekt navázaného spojení obsahuje stavový stroj a postupně mezi stavy přechází. Počáteční stav je `Closed`. Jedna strana musí začít a odeslat paket s `Hello` rámcem. Po obdržení se stavový stroj druhé strany přesune do `ReceivedHello` a spolehlivě odešle `Hello`. Když strana dostane rámec `Hello` tak 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.Celý stavový diagram vidíme na obrázku 8.

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.



Obrázek 8: Stavový stroj QUICr handshake

4.6.3 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 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.

4.6.4 Enkodér

Pro kódování jsme vytvořili komponentu `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. Podobný přístup jsme adaptovali my.

Dále musel umožňovat zakódovat hodnotu zpětně, jako například délku, kterou kvůli omezení bufferu víme až později. Potom, co přes rozhraní zapíšeme vše, co je potřeba, zavoláme metodu `finish`, která hodnoty pro délku přenastaví na správnou hodnotu.

4.6.5 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.

4.6.6 Měření

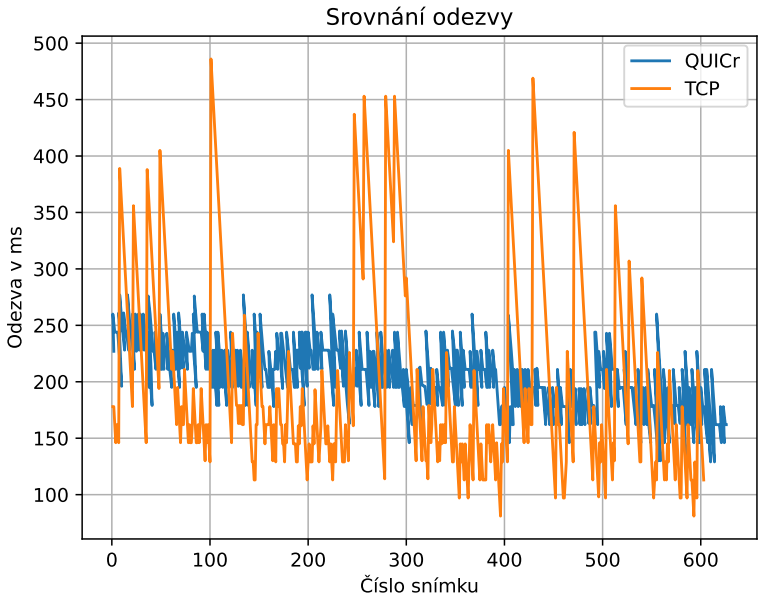
Protokol měl za cíl snížit odezvu na nespolehlivé síti a neblokovat zprávy. Vytvořili jsme test, který simuluje program hry. V testu jsou dvě vlákna, jedno pro klienta a druhé pro server. Oba mají svou vlastní smyčku, ve kterém iterují. Klient odesílá hodnotu derivace typu double. Server si udržuje stav 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“, který obsahuje různá nastavení 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 . Vidíme, že používáme pouze rozhraní `lo`, tedy loopback, které se týká posílání mezi programy na lokální adrese.

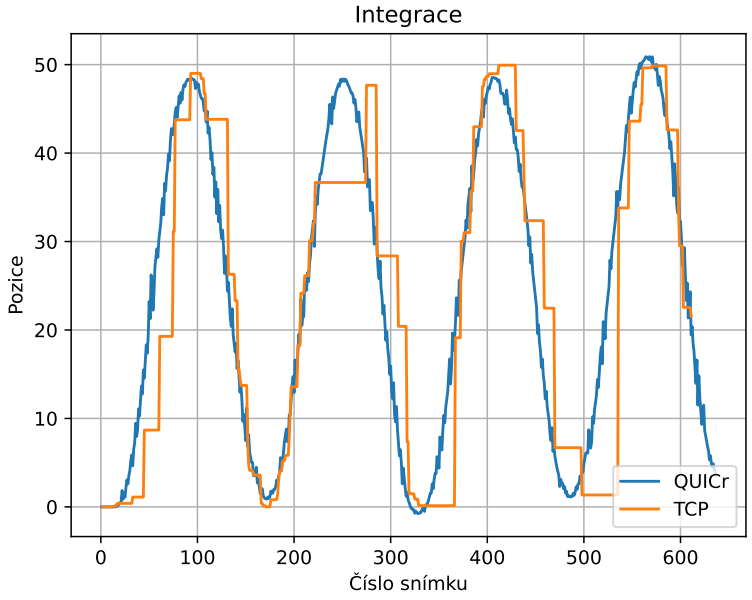
```
1    # tc qdisc add dev lo root netem \
2       delay 200ms 20ms 25% \
3       loss 3% \
4       duplicate 1% \
5       reorder 25% 50%
```

Zdrojový kód 3: Příklad volání TC

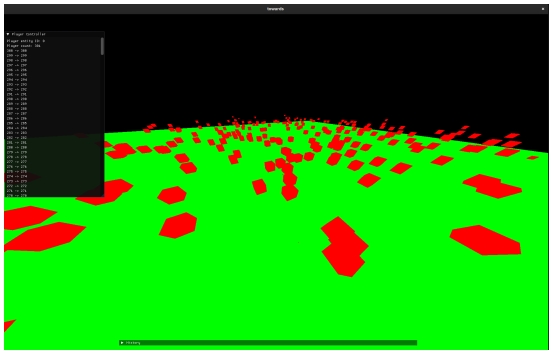
V prvním testu jsme pouze měřili odezvu mezi odesláním vstupu od klienta pro snímek n a získáním jeho integrovaného stavu ze serveru. Na obrázku 9 vidíme, že TCP obsahuje skoky a QUICr je stabilnější.



Obrázek 9: Porovnání odezvy TCP a QUICr



Obrázek 10: Porovnání integrace TCP a QUICr



Obrázek 11: 301 připojených hráčů

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 4.6.6 vidíme, že TCP při ztrátě paketů způsobuje skoky, které nemusí být pro hráče příjemné. Naopak náš protokol má integraci plynulejší.

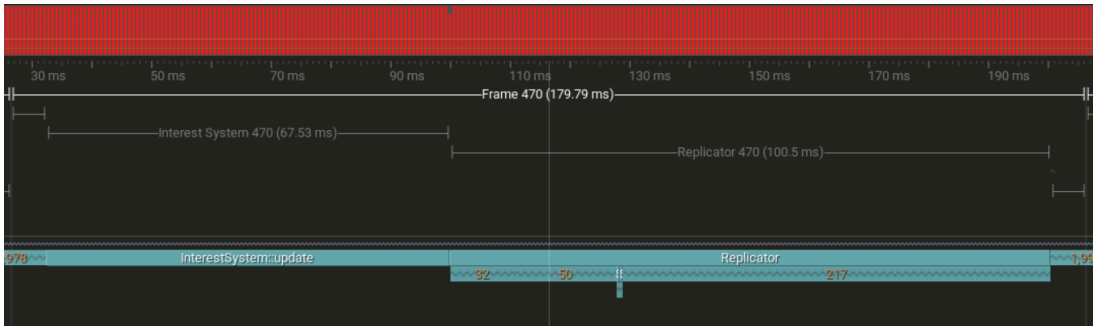
Nakonec jsme integrovali protokol přímo do replikátoru. Vytvořili jsme simulovaného klienta, který na server posílá náhodné vstupy a způsobuje na serveru svůj pohyb. Těchto simulovaných klientů můžeme spouštět libovolný počet. Každý naváže se serverem spojení a komunikuje s ním klasicky přes replikátor a ovladač. Můžeme tak snadno měřit, jak se zatížení projeví v síti, a zároveň můžeme využít nástroje jako tc pro různé podmínky. Na obrázku 4.6.6 vidíme 301 hráčů, kde 300 je simulovaných individuálních připojení a jeden je náš hráč. Server počítal každý snímek dlouho, asi 250ms, ale zátěž zvládl. Dalším krokem bylo zvýšit počet snímku za sekundu pro stovky hráčů.

4.7 Optimalizace replikátoru

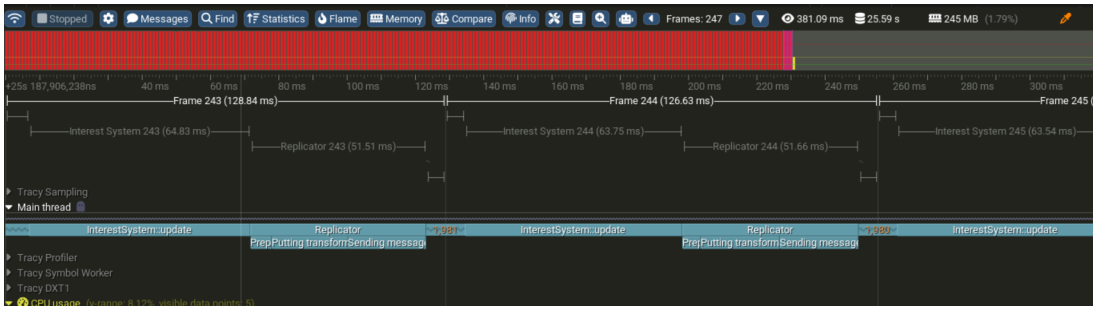
Test s 301 hráči odhalil nedostatečný výkon serveru. Pomocí Tracy jsme začali program analyzovat. Na obrázku 4.7 vidíme, že pouze aktualizace replikátoru trvá každý snímek kolem 100ms. Pro představu, pokud bychom chtěli, aby server integroval 20 krát za sekundu, museli bychom všechno, nejen aktualizaci 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 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



Obrázek 12: Analýza replikátoru

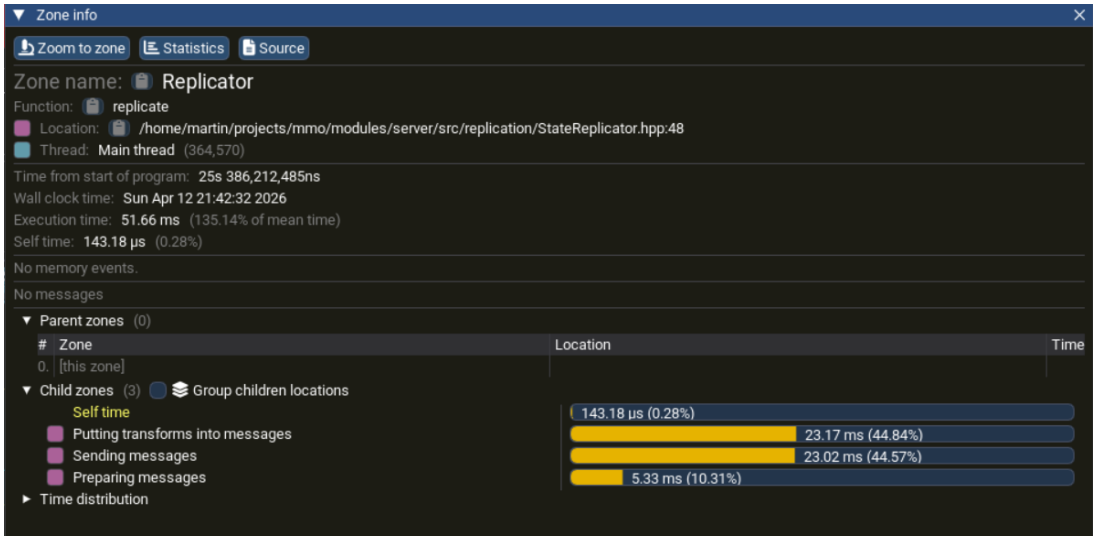


Obrázek 13: Analýza optimalizace replikátoru pro ECS

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 4.7. 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 4.7 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 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. 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 4.7.

Zjistili jsme, že problémem byl ProtoBuf a FlatBuffers by nám nepomohl. Nepřekvapilo nás, že memcpy byl nejrychlejší. Rozhodli jsme se, že pro serializaci



Obrázek 14: Analýza optimalizace replikátoru pro ECS

	ProtoBuf	FlatBuffers	Memcpy
Čas (s)	9.455691762	10.345692894	0.754350866

Obrázek 15: Porovnání serializátorů

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`.

4.8 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

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

Obrázek 16: Porovnání algoritmů

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 o obdobu binárního stromu rozšířenou o rozměr. V tabulce 4.8 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.

4.8.1 Congestion control

4.8.2 Bandwidth control

Entity, které jsou od hráče daleko, není třeba aktualizovat tak často jako ty, které jsou k němu blízko. Již jsme nastínili, že můžeme můžeme spočítat důležitost jednotlivých entit pro klienta a omezit tak počet entit, které musí replikátor synchronizovat. Například můžeme vzdálenější entity synchronizovat méně často. Důležité je, aby se i ty dostali někdy na řadu.

4.9 Sharding

Proces, při kterém hráče rozdělíme do skupin, které se navzájem nevidí, i když jsou na stejném serveru blízko sebe, se říká sharding. Podobný princip funguje i v databázích.

4.10 Horizontální škálování

Další z metod podobná shardování, je „horizontální škálování“, kdy 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 4.10. 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.

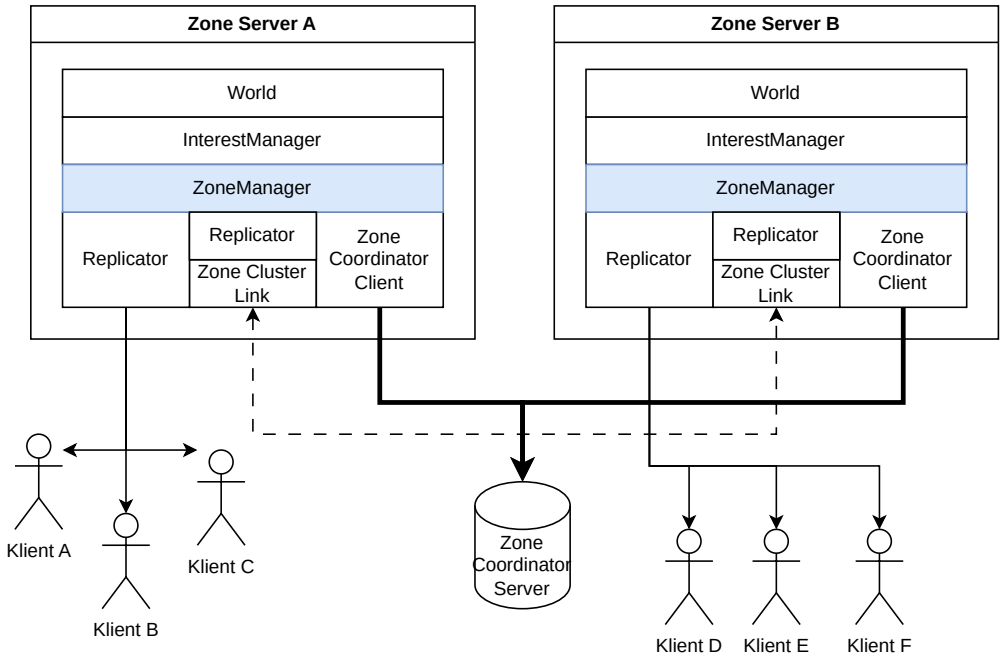
5 Výsledná hra

V této kapitole představíme hru.

6 Měření

6.1 Závěry

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



Obrázek 17: Architektura systému s zone clusterem

```
% Tiskne český závěr práce.
\begin{kiconclusions}
Závěr práce v \uv{českém} jazyce.
\end{kiconclusions}

% Tiskne anglický závěr práce.
\begin{kiconclusions}[english]
Thesis conclusions written in \uv{English}.
\end{kiconclusions}
```

Zdrojový kód 4: Sazba závěrů

Závěr

Závěr práce v „českém“ jazyce.

Conclusions

Thesis conclusions in “English”.

A První příloha

Text první přílohy

B Druhá příloha

Text druhé přílohy

C Obsah elektronických dat

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

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

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

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

Dále mohou data obsahovat například:

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

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