Universiteit Hasselt

Bachelorproef

Latency hiding in NVE's

Auteur:
Pieter Robyns

Promotor:
Prof. dr. Peter Quax

Begeleider:
Prof. dr. Wim Lamotte

Bachelorproef voorgedragen tot het behalen van de graad van bachelor in de informatica/ICT/kennistechnologie

2011 - 2012

Inhoudsopgave

1 Inleiding **1**

2 Doelstelling **2**

3 Networked graphics **3**
3.1 Gebruikte architecturen . . . 3
3.1.1 Peer to peer . . . 3
3.1.2 Peer to peer met master host . . . 3
3.1.3 Peer to peer met rendezvous server . . . 3
3.1.4 Client / server . . . 4
3.1.5 Multicast . . . 4
3.2 Problemen binnen networked graphics . . . 4
3.2.1 Latency . . . 4
3.2.2 Scalability . . . 5
3.2.3 Synchronisation . . . 5
3.2.4 Ownership . . . 5
3.2.5 Persistency . . . 6
3.2.6 Security . . . 6

4 Protocol **8**
4.1 TCP . . . 8
4.2 UDP . . . 8
4.3 Custom protocol: Netfire Protocol (NFP) . . . 9
4.3.1 Reliability . . . 9
4.3.2 Virtual connections . . . 10
4.3.3 Client disconnects . . . 11
4.3.4 Behandelen out-of-order packets . . . 12
4.3.5 Finale pakketstructuur . . . 12

5 Latency hiding **13**
5.1 Interpolation . . . 13
5.2 Extrapolation . . . 13
5.3 Client side prediction . . . 14
5.3.1 Problemen bij client side prediction . . . 14
5.3.2 Input prediction . . . 14
5.3.3 Smooth error correction . . . 16
5.4 Lag compensation . . . 16

6 Implementatie **17**
6.1 Concept . . . 17
6.2 Architectuur . . . 17
6.3 Application protocol . . . 17
6.3.1 Login messages . . . 18
6.3.2 Input messages . . . 18
6.3.3 Position messages . . . 18
6.3.4 Pickup messages . . . 18
6.4 Tools en resources . . . 19
6.5 Server . . . 19

6.5.1 Authoritative environment . . . 19
6.5.2 Asynchronous NFP . . . 20
6.5.3 Update cycle . . . 20
6.5.4 Dead Reckoning . . . 20
6.5.5 Lag compensation . . . 21
6.5.6 Fake lag . . . 22
6.5.7 Ownership and locking . . . 23
6.6 Client . . . 23
6.6.1 Dumb client . . . 23
6.6.2 Interpolation . . . 23
6.6.3 Input prediction . . . 24
6.6.4 Fake lag . . . 25
6.7 Integratie met Unity . . . 25
6.8 Evaluatie . . . 25

7 Conclusie **27**

8 Appendix A: Praktische info **28**
8.1 Setup . . . 28
8.2 Controls . . . 28
8.3 Commands . . . 28
8.3.1 Introduceren van fake lag . . . 28
8.3.2 Interpolation: ratio and time . . . 29
8.3.3 Interpolation and shooting ghosts . . . 29
8.3.4 Input prediction . . . 29
8.3.5 Lag compensation . . . 29
8.4 Testing . . . 29
8.4.1 Testing interpolation . . . 30
8.4.2 Testing dead reckoning . . . 30
8.4.3 Testing input prediction . . . 30
8.4.4 Testing lag compensation . . . 31
8.4.5 Testing ownership . . . 31

9 Appendix B: Screenshots **32**

10 Appendix C: Logboek **33**

1 Inleiding

Deze paper is een uitgebreid verslag van de bachelorproef die vereist is voor het behalen van het bachelordiploma. Als onderwerp van deze bachelorproef werd "Latency hiding in NVE's" gekozen.

In de eerste secties worden de doelstellingen gedefinieerd, gevolgd door algemene informatie binnen het kader "networked graphics". De verschillende architecturen en mogelijke problemen die kunnen optreden worden hier kort besproken.

Vervolgens wordt het custom protocol dat ontwikkeld werd speciaal voor deze bachelorproef beschreven, aangevuld met flowcharts over de precieze werking ervan. Het custom protocol is een uitbreiding op UDP dat moest ontwikkeld worden alvorens aan de effectieve implementatie kon worden begonnen.

De laatste secties gaan over latency hiding en de implementatie ervan. Hierbij worden de gebruikte algoritmes, methodes en voorbeelden verduidelijkt. Ten slotte volgt nog een sectie met nuttige praktische info om de implementatie te testen.

2 Doelstelling

In deze bachelorproef wordt nader bekeken hoe men op een efficiënte manier een netwerklaag kan voorzien voor een interactieve virtuele omgeving, waarbij de gebruiker de indruk krijgt dat gebeurtenissen zich in real time afspelen. Een voorbeeld van een dergelijke omgeving is een multiplayer game. Het doel van deze opdracht is hierbij tweeledig.

Een eerste doel is het verzamelen van zoveel mogelijk informatie over het onderwerp. Meer specifiek zal er binnen networked graphics onderzoek gedaan worden naar de gebruikte netwerkarchitecturen, protocols en de gekende problemen die hierbij kunnen optreden. Als laatste zal ook onderzoek gedaan worden naar de theorie en werking achter latency hiding technieken.

Een tweede doel is om deze informatie te gebruiken in een concrete implementatie. Om tijd te besparen op het grafische aspect van de implementatie werd door het onderwijsteam aangeraden om gebruik te maken van de Unity engine [1]. Hoewel deze engine komt met een werkende netwerkfunctionaliteit, zullen custom functionaliteiten gebruikt worden om de eerder vermeldde technieken te demonstreren.

De implementatie zal de vorm aannemen van een online first person shooter, die gebruik maakt van een zelf ontworpen protocol. Communicatie tussen client en server zal hierbij gebeuren met behulp van sockets. De programmeertaal die werd gekozen om dit te verwezenlijken is C#, omdat deze taal of JavaScript vereist is voor Unity scripts.

Omdat het niet mogelijk is binnen de voorziene tijd *alle* mogelijke technieken omtrent latency hiding in networked graphics te implementeren, wordt in de implementatie echter gefocust op drie deelproblemen uit sectie 3.2, namelijk **latency**, **security** en **ownership**. In sectie 6 wordt beschreven hoe deze problemen werden opgelost.

3 Networked graphics

Deze sectie bevat algemene info over networked graphics. We bespreken kort welke architecturen en datamodellen er bestaan, hoe deze van mekaar verschillen en hoe deze gebruikt worden binnen de context van networked graphics. Tot slot wordt nog besproken welke problemen er kunnen optreden bij de implementatie, voor welke uitdagingen dit zorgt en hoe deze kunnen worden opgelost.

3.1 Gebruikte architecturen

3.1.1 Peer to peer

In de peer to peer network architectuur is elke host zowel een client als een server. De processen waarmee ze communiceren zijn dus gelijk (vandaar ook de naam *peer*). Een populair voorbeeld van een protocol dat deze architectuur gebruikt is "BitTorrent": de hosts die van dit protocol gebruik maken sharen files door ze zowel te hosten (server) als te downloaden (client) [21].

Wanneer P2P wordt gebruikt in genetwerkte virtuele omgevingen is elke participerende host verantwoordelijk voor het sturen van data naar een andere host. De gevolgen voor het data model zijn hierbij logisch: het data model bevat informatie over elke participerende host en bevat tevens ook het IP en de poort van de corresponderende host. In een multiplayer game zou elke host bijvoorbeeld de positie van zichzelf en alle andere hosts (plus IP-adres en poort) in zijn datamodel kunnen bevatten. Het synchroniseren van states tussen de verschillende hosts is hierbij een grote uitdaging [15][7].

3.1.2 Peer to peer met master host

Een van de nadelen bij een gewone peer to peer architectuur is dat alle participerende hosts van elkaars bestaan moeten afweten. Dit bemoeilijkt het toevoegen van een nieuwe host aan de genetwerkte virtuele omgeving.

Dit probleem kan worden opgelost door een peer aan te duiden als *master*. Hosts die willen connecteren met het peer to peer network hebben in dit geval enkel het IP-adres en poortnummer van de master nodig. De master communiceert dan de aanwezigheid van de nieuwe host door aan de andere, reeds geconnecteerde hosts.[15][7]

3.1.3 Peer to peer met rendezvous server

Door gebruik te maken van een master host wordt een nieuw probleem geïntroduceerd: de host die werd aangesteld als master speelt een essentiële rol in het verbinden van nieuwe hosts met het peer to peer netwerk. Wanneer de master host uitvalt of niet meer in staat is om alle network traffic af te handelen kunnen er geen nieuwe hosts meer connecteren.

Een betere manier om dit probleem op te lossen is dus door gebruik te maken van een externe server in plaats van een master host aan te stellen. De server heeft dezelfde rol als een master host, maar doet zelf niet mee in de genetwerkte virtuele omgeving. Het voordeel van deze aanpak is dat de server beter gecontroleerd kan worden en dat er dus een hogere graad van reliability is.[15][7]

3.1.4 Client / server

In de client / server architectuur worden geconnecteerde hosts verdeeld in twee groepen: clients en servers. Hierbij gaan clients requests sturen naar één of meerdere servers. De servers wachten op deze inkomende requests en handelen ze vervolgens af [9]. Er zijn talrijke voorbeelden te vinden van applicaties die gebruik maken van deze architectuur. Een van de meest populaire voorbeelden is het HTTP-protocol, dat veelvuldig gebruikt wordt in het World Wide Web.

In de context van genetwerkte virtuele omgevingen wordt de state van de omgeving en andere clients continu opgevraagd aan de server. Het data model bevat dus ook in deze architectuur informatie over andere hosts. Het verschil is echter dat het niet meer nodig is om het IP-adres en poortnummer van elke participerende host in het netwerk te kennen: enkel het IP-adres en de poort van de server is nodig. Bijgevolg lijkt het data model van deze architectuur sterk op dat van "een peer to peer met rendezvous server"-architectuur (zie sectie 3.1.3) [15][7].

Deze architectuur biedt in vergelijking met peer to peer architecturen een betere controle over security en centralisatie van de netwerkdata. Enkele nadelen zijn dan weer het *single point of failure*, scalability problemen en het feit dat een server meestal "high end" moet zijn en dus veel kost [17].

3.1.5 Multicast

Bij een multicast architectuur wordt het netwerk opgedeeld in verschillende groepen. Hosts die een bepaalde groep joinen zullen elk pakket dat naar deze groep verstuurd wordt ontvangen. Hierbij wordt echter geen gebruik gemaakt van een server: het onderliggende protocol (bijvoorbeeld IGMP) zal hier namelijk voor zorgen [11].

De multicast architectuur lijkt sterk op de peer to peer architectuur, waardoor het data model bijgevolg vrijwel identiek is. Het is echter niet meer nodig om het IP-adres en poortnummer van de andere hosts te weten: het onderliggende netwerk zorgt namelijk voor de distributie van de data [15][7].

3.2 Problemen binnen networked graphics

Bij het ontwikkelen van een genetwerkte virtuele omgeving zijn er een aantal problemen die kunnen optreden omwille van de eigenschappen van network packets en computer networks in het algemeen. In deze sectie wordt elk van deze problemen besproken, samen met een aantal mogelijke oplossingen.

3.2.1 Latency

Wanneer een pakket over het netwerk wordt verstuurd, gebeurt dit niet instantaan. Het pakket zal namelijk verschillende soorten delay ervaren, zoals bijvoorbeeld processing delay, queuing delay, transmission delay en propagation delay [11]. De meting van deze delays noemen we *latency*.

Latency kan ervoor zorgen dat de interactiviteit en "echtheid" van de virtuele omgeving verstoord wordt, omdat updates over de state van de omgeving dan trager aankomen bij een participerende host. Voor de gebruikers in een realtime virtuele omgeving over het netwerk is het dus essentieel dat latency geminimaliseerd wordt.

Een eerste stap om dit te doen begint bij het design van de virtuele omgeving zelf. In geval van een client server architectuur zou men bijvoorbeeld kunnen beslissen om de update rate van de server laag te houden, zodat er geen hoge latency onstaat vanwege bandwidth bottlenecks in het netwerk (zie sectie 6.5.3).

Indien latency dan toch optreedt kan men gebruik maken van verscheidene *latency hiding* technieken om de latency trachten te verbergen. Deze technieken worden uitgebreid behandeld in sectie 5 en werden tevens ook geïmplementeerd in de implementatie (zie sectie 6).

3.2.2 Scalability

Het probleem *scalability* van genetwerkte virtuele omgevingen gaat over de "computational and bandwith bottleneck" van bepaalde architecturen, zoals bijvoorbeeld de client server architectuur (sectie 3.1.4). Bij de client server architectuur ligt deze bottleneck bij de server: elke client wil een reply data over het netwerk ontvangen en neemt tevens ook CPU resources in van de server [4].

Voor deze problemen bestaat er reeds een brede waaier aan oplossingen zoals bijvoorbeeld *load balancing*, *cell-based partitioning* en *area-of-interest management*. De theorie en implementatie achter deze technieken is echter breed genoeg om een onderwerp op zich te zijn, waardoor het oplossen van scalabilityproblemen niet als sub-onderwerp werd opgenomen in deze bachelorproef. De implementatie (zie sectie 6) is daarom bedoeld voor een klein aantal clients, zodat problemen omtrent scalability uitgesloten zijn.

3.2.3 Synchronisation

In sommige genetwerkte virtuele omgevingen is het belangrijk dat de pakketten uitgewisseld tussen verschillende geconnecteerde hosts gesynchroniseerd zijn indien de volgorde, tijd van aankomst en het al dan niet aankomen van pakketten belangrijke implicaties heeft voor de toekomstige state van de omgeving [16][15].

Een voorbeeld van een genetwerkte virtuele omgeving waarbij synchronisatie een probleem is, is een Massively Multiplayer Online Game. Bij deze soort van online games connecteren duizenden clients op een cluster / pool van servers. Elk van deze clients wil uiteraard dezelfde state van de omgeving zien. Hoewel client server architecturen normaalgezien inherent goed presteren qua synchronisatie is dit vanwege de clustering van servers in MMOG's niet het geval en moet men complexe algoritmes zoeken om states tussen de servers zelf te synchroniseren [6].

Een aantal voorbeelden van synchronisatie-algoritmes gebruikt in sommige genetwerkte virtuele omgevingen zijn ten slotte: *lock step synchronisation*, *bucket synchronisation* en *time warp synchronisation*. In de implementatie werd geen van de eerder vermeldde synchronisatie-algoritmes geïmplementeerd, vanwege de doelstellingen van deze bachelorproef (zie sectie 2).

3.2.4 Ownership

Het probleem ownership gaat over het simultaan manipuleren van objecten in een genetwerkte virtuele omgeving. Stel dat twee spelers in een online game bijvoorbeeld tegelijkertijd een deur willen openen. Wat zal er dan gebeuren wanneer er een kleine afwijking is qua latency tussen beide spelers? Zal dit

resulteren in een deur die dichtblijft? Zal de response voor beide clients hetzelfde zijn? Het is duidelijk dat de spelers in een niet-deterministische staat zullen terechtkomen [15].

Een tweede voorbeeld is het oprapen van een powerup: indien beide spelers tegelijk deze willen oprapen. Zou het door latency mogelijk kunnen worden dat ze allebei deze powerup krijgen, terwijl deze normaalgezien maar aan één speler kan worden toegekend?

Deze ownership problemen kunnen worden opgelost door een techniek genaamd *locking*. Deze techniek wordt ook in vergelijkbare situaties gebruikt door applicaties die niets te maken hebben met virtuele omgevingen, zoals SVN repositories (bijvoorbeeld tegelijk committen) of SQL databases (tegelijk een query uitvoeren).

Met behulp van locking wordt ervoor gezorgd dat een resource niet tegelijk door twee entiteiten kan worden gewijzigd. In de implementatie wordt locking geïmplementeerd om te voorkomen dat twee spelers tegelijk een pickup kunnen oprapen (zie sectie 6.5.7).

3.2.5 Persistency

Persistency is een concern die vooral voorkomt in virtuele omgevingen zoals MMORPG's. In deze omgevingen wordt door de gebruikers verwacht dat de acties die ze uitvoeren op de wereld, de items die ze verzamelen, de quests die ze voltooien etc. persistent zijn en dus niet verloren gaan bij een eventuele server crash.

Om dit te bereiken is een goed uitgebouwde database architectuur nodig, samen met de nodige backups in geval van bijvoorbeeld een HDD-crash. Hierbij moet men ermee rekening houden dat de database eventueel een bottleneck kan worden in het systeem [15].

Dit probleem is niet van belang in de implementatie van deze bachelorproef, omdat de game een First Person Shooter is, waarbij geen persistente data moet worden opgeslagen.

3.2.6 Security

Een laatste en meestal ook moeilijk probleem binnen genetwerkte virtuele omgevingen is *security*. Naast het hebben van een werkende applicatie is het namelijk uiteraard ook belangrijk dat een "kwaadaardige" gebruiker de omgeving niet kan manipuleren door network traffic aan te passen. In de context van realtime streaming media zou de gebruiker bijvoorbeeld een video stream kunnen aanpassen door een man-in-the-middle attack [5]. Een ander voorbeeld kan men vinden in de context van realtime games: een gebruiker zou bijvoorbeeld zijn positie kunnen manipuleren door pakketten aan te passen.

Om dit probleem op te lossen kan men gebruik maken van een centrale autoriteit of server. Zelfs bij vele sytemen binnen in het peer to peer model (zie sectie 3.1.1) zoals bijvoorbeeld Pastry, wordt een server gebruikt voor het toekennen van nodeId's bij het joinen van het peer to peer network omwille van veiligheidsredenen. [20]

In een genetwerkte virtuele omgeving is het nodig om de acties van een gebruiker door de centrale autoriteit te laten valideren. Het is hierbij essentieel dat

enkel de centrale autoriteit – en niet de client – controle heeft over belangrijke data zoals positiecoördinaten, levenspunten, snelheid, etc.

4 Protocol

Alvorens we kunnen beginnen met het implementeren van een realtime networking protocol voor een multiplayer game, moeten we best eerst een keuze maken tussen een aantal transport layer standaarden. In deze paper worden enkel de meest bekende en meest gebruikte standaarden (UDP en TCP) met elkaar vergeleken. Na de keuze van een bepaalde standaard kunnen we hierop verderbouwen om tot een custom protocol voor de game in kwestie te komen. Een voorbeeld hiervan wordt gegeven in sectie 6.

4.1 TCP

TCP is een connection-oriented, reliable transport protocol [11] en wordt vaak gebruikt voor applicaties binnen het World Wide Web. Het protocol heeft volgende eigenschappen:

- Point-to-point connections tussen twee hosts.
- Reliable data transfer.
- In order delivery van packets.
- Retransmission mechanismes in geval van lost packets.
- Flow control.
- Segmentatie van pakketten.
- Congestion control met als gevolg fairness tussen connecties.

Hoewel TCP vaak wordt gezien als een ongeschikt protocol voor realtime omgevingen [8], wordt het vanwege bovenstaande eigenschappen toch gebruikt in games waarin een hoge graad van betrouwbaarheid en synchronisatie nodig is [18][13]. Een voorbeeld van een populaire game die TCP gebruikt is de MMOG World Of Warcraft [13].

4.2 UDP

UDP is een lightweight, unreliable en connectionless transport protocol. In tegenstelling tot TCP wordt er hier dus niet met connecties gewerkt en kunnen pakketten in een willekeurige volgorde of zelfs niet aankomen bij de destination host. Enkele belangrijke aspecten van UDP zijn dus [15][11]:

- Afwezigheid van flow control en congestion control zorgt ervoor dat pakketten onmiddellijk verstuurd worden.
- Geen connection setup met als gevolg minder overhead.
- Afwezigheid van retransmissions en reliability in het algemeen.
- Geen packet segmentation.
- Kleine header size.

Uit deze aspecten kunnen we afleiden dat het inderdaad een betere keuze zou zijn om UDP te gebruiken voor een omgeving waarin een hoge graad van interactiviteit vereist is. Dit omdat UDP de latency in deze soort omgevingen zou kunnen verbeteren, bijvoorbeeld door de afwezigheid van retransmissions.

Toch is het meestal niet ideaal om pure UDP te gebruiken. In de meeste fast-paced interactieve omgevingen is de volgorde van pakketten immers wel belangrijk en moet er dus een extensie op UDP geschreven worden. In de volgende sectie wordt een dergelijk protocol uitgewerkt. Dit protocol zal tevens in de implementatie van de interactieve omgeving gebruikt worden.

4.3 Custom protocol: Netfire Protocol (NFP)

Uit de voorgaande secties kunnen we concluderen dat er zowel bij TCP als UDP voordelige en nadelige eigenschappen te vinden zijn in de context van interactieve omgevingen. Vertrekkende van de doelstelling in sectie 2 kunnen dus best eerst een lijst van eigenschappen opstellen die ons eigen protocol zal moeten ondersteunen:

- Reliability.
- Virtual connections.
- Detecteren van een disconnect.
- Behandelen out-of-order packets.

Voortaan zullen we verwijzen naar dit protocol als het "Netfire"-protocol.

4.3.1 Reliability

Voor sommige minder frequent voorkomende pakketten is het nodig om een zekere vorm van reliability te voorzien. Neem bijvoorbeeld pakketten die de eindscores van alle spelers moeten meedelen op het einde van een game of het joinen van een nieuwe speler. Omwille van deze vereiste functionaliteiten zal het NFP-protocol een simpele vorm van guaranteed delivery implementeren.

Pakketstructuur: In elk NFP-pakket zal een field worden voorzien waar men kan aangeven of het pakket essentieel is en dus *moet* aankomen op de bestemming. Verder zal ook een sequence number worden meegegeven aan elk pakket, zodat pakketten uniek geïdentificeerd kunnen worden.

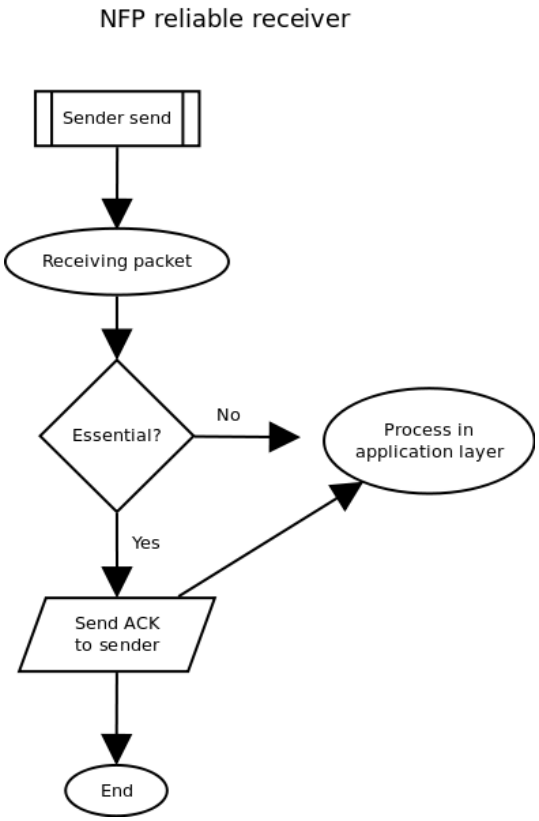
Datastructuur: Zowel op de client als op de server wordt een hashtabel bijgehouden waarbij de key een tuple is van de vorm `(sequenceNumber, endPoint)` en de value een singleton is van de vorm {`DataPacket`}. Sequence numbers moeten ook gemapt worden op de verschillende endPoints en worden daarom ook bijgehouden in een hash table waarbij de key de endPoint is en de value het sequence number.

Werking: Bij het verzenden van het pakket wordt de "essential"-flag gezet en wordt het pakket door de sending host opgeslagen in de hash table. Tegelijk wordt de verzendtijd van het pakket bijgehouden. De receiving host zal bij het ontvangen van het pakket de "essential"-flag zien en zal daarop een `ACK`-pakket

returnen waarbij deze flag niet gezet is. Gaat een van de pakketten (het eerste of de daarop volgende ACK) verloren of klopt de checksum niet, dan zal de timer op de sending host verlopen, wat resulteert in een retransmission van het pakket (zie ter verduidelijking Figuur 2 en Figuur 1).

Verschillen met TCP qua reliability:

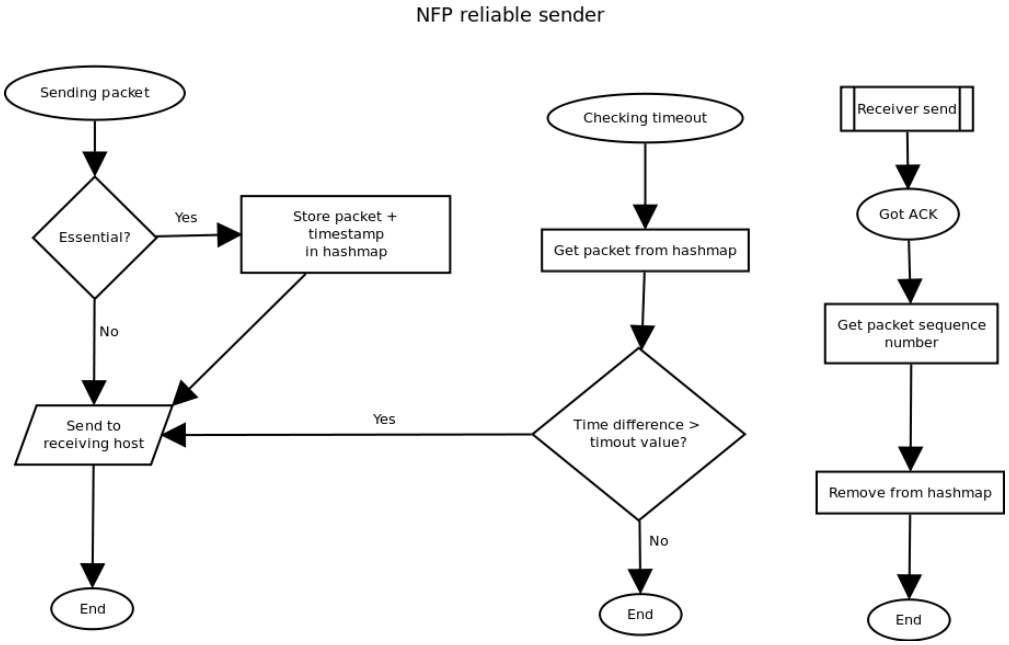
- Per packet ACK: in plaats van cumulative acknowledgement gebruikt NFP acknowledgement per packet. Dit verschil is er omdat TCP eerder stream gebaseerd is, terwijl NFP gebaseerd is op UDP. UDP garandeert dat een pakket in zijn geheel aankomt (sectie 4.2), waardoor het dus niet veel nut heeft om TCP na te bootsen in dit opzicht.
- Statische timeout: in TCP wordt de timeout van een pakket berekend aan de hand van een Smoothed Round Trip Time: $RTO = min[UBOUND, max[LBOUND, (BETA * ((ALPHA * SRTT) + ((1 - ALPHA) * RTT)))]]$ [10]. Voorlopig wordt in NFP een user-configurable timeout gebruikt.



Figuur 1: NFP receiver thread

4.3.2 Virtual connections

In een virtuele omgeving is het belangrijk om te weten wanneer een speler joint of het spel verlaat. De server moet ook weten naar wie een reply moet worden



Figuur 2: NFP sender thread

gestuurd. Omdat UDP connectionless is, zou men ervoor kunnen kiezen om telkens de speler mee te geven aan het pakket om een onderscheid te maken tussen alle binnenkomende pakketten. Deze werkwijze zorgt echter voor veel overbodige data. Een betere manier is het opzetten van een virtuele connectie.

Pakketstructuur: In het `Command`-veld van het pakket staat het commando `LOG`.

Datastructuur: Connections worden gemapt in een hash table op hun respectievelijke endpoints, zodat de server weet welke endpoint staat voor welke connection.

Werking: Om een connectie tot stand te brengen stuurt de client een `LOG`-pakket naar de server. De client zal hierbij de "essential" flag op true zetten waardoor het pakket uiteindelijk zeker zal aankomen. Bij aankomst op de server worden connection en endPoint opgeslagen in de datastructuur. Data over de omgeving kan nu gestuurd worden naar elke geconnecte client.

4.3.3 Client disconnects

Naast het opzetten van een connectie is het voor de server ook essentieel om te weten wanneer een client disconnected geraakt. De server beslist om de client te disconnecten wanneer van deze gedurende een bepaalde periode geen pakketten meer heeft verstuurd.

Pakketstructuur: In het `Command`-veld van het pakket staat het commando `DIS`. Het ARG-veld bevat het IP en de poort van de gedisconnecte client.

Datastructuur: De bestaande tabel van geconnecte clients wordt gebruikt.

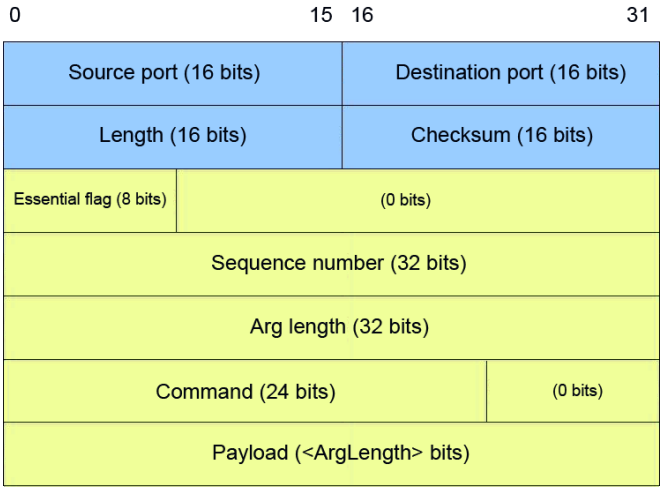
Werking: De server bepaalt dat de client gedisconnect moet worden omdat er gedurende een bepaalde periode geen pakketten meer zijn ontvangen van deze client. De client wordt dan uit de lijst van actieve connecties gehaald en een reliable `DIS`-pakket wordt naar de andere clients gebroadcast. Op deze manier weten de andere clients dat de gedisconnecte client uit de virtuele omgeving mag verdwijnen.

4.3.4 Behandelen out-of-order packets

Omdat de implementatie een online FPS zal worden, werd gekozen om in NFP out-of-order packets te droppen. Dit betekent dus dat enkel het pakket met de hoogste sequence number (dit is het meest recente pakket) gebruikt zal worden en dat pakketten met een lager sequence number die daarna aankomen gedropt zullen worden.

4.3.5 Finale pakketstructuur

Op basis van alle voorgaande eigenschappen kunnen we een structuur bepalen voor onze uiteindelijke packet:



Figuur 3: Headerstructuur voor een NFP packet. Het blauwe gedeelte is UDP. De velden zijn vrij groot qua bits, omdat ze in ASCII leesbaar zijn voor de gebruiker. Dit werd op deze manier geïmplementeerd zodat men in Wireshark makkelijk kan zien wat er over het netwerk gebeurt.

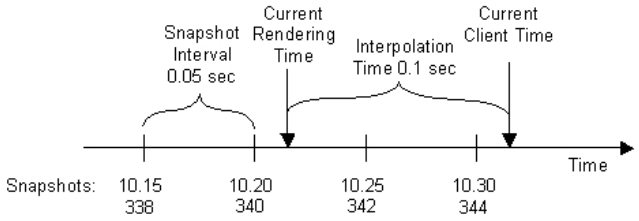
5 Latency hiding

Hoewel we in dit stadium een geschikt protocol hebben ontwikkeld voor onze virtuele omgeving, beketent dit echter niet dat we nooit latency zullen hebben. In virtuele omgevingen varieert de server tickrate van 20 Hz tot 66 Hz [15]. De meeste update-pakketten komen niet gelijktijdig of gewoon niet aan door respectievelijk jitter en packet loss. Omwille van deze reden is het noodzakelijk om latency hiding technieken toe te passen.

5.1 Interpolation

In de context van genetwerkte virtuele omgevingen betekent interpolation: het interpoleren van posities tussen de geconnecteerde entiteiten in deze omgeving. In plaats van rechtstreeks van de server posities aan te nemen, gaat de client dus zelf berekenen waar de andere clients zich bevinden op een bepaald moment (Figuur 4).

Om interpolation te bereiken moet de client twee bekende states van de server hebben ontvangen. Stel bijvoorbeeld dat de server elke 100 ms een update stuurt. In dit geval moet de client dus in principe 100 ms in het verleden renderen, omdat er eerst 2 states moeten zijn om tussen te interpoleren. Wanneer de tweede state update verloren zou gaan, kan gebruik gemaakt worden van extrapolation (sectie 5.2) om een voorspelling te maken van de volgende state.



Figuur 4: Interpolation [19]

5.2 Extrapolation

In sommmige genetwerkte virtuele omgeving wordt in geval van *lag* of hoge latency gebruik gemaakt van extrapolatie. Bij extrapolatie wordt in geval van het ontbreken van een state update gebruik gemaakt van een reeks vorige states om deze te extrapoleren naar een nieuwe state. Extrapolatie kan echter niet gebruikt worden gedurende lange periode's, omdat dan de afwijking met de echte state te groot zou zijn [14].

Deze manier van werken is nuttig in virtuele omgevingen waarbij het gedrag van objecten deterministisch is. Men zou bijvoorbeeld in een racegame gebruik kunnen maken van extrapolatie, omdat de snelheid van een auto bekend is en doorgaans niet snel verandert binnen een korte tijdsspanne.

In virtuele omgevingen zoals first person shooters is het meestal niet nuttig om in grote mate extrapolatie te gebruiken: het gedrag van een speler is

niet deterministisch. Er wordt dus te vaak veranderd van snelheid, rotatie, positie... Toepassen van extrapolatie in dit geval zou dus inconsistentie kunnen veroorzaken tussen verschillende hosts. Hierdoor zal bij het ontvangen van een latere, correcte state de geëxtrapoleerde state te fel afwijken, wat bij correctie waargenomen wordt als een onrealistische, "springende" transitie tussen deze states.

5.3 Client side prediction

In virtuele omgevingen waarin de server autoritair is (zie sectie 3.2.6 moeten de acties van de gebruiker eerst berekend worden door de server alvorens ze op de client kunnen worden toegepast. Dit betekent dat de client een volledige round trip time moet wachten alvorens zijn actie zichtbaar wordt, wat de input van de gebruiker minder responsief en stroever maakt.

Om dit probleem op de lossen maakt men gebruik van client side prediction. Client side prediction gaat ervan uit dat de acties die de gebruiker uitvoert goedgekeurd zullen worden door de server. Men wacht dus niet meer op een effectieve bevestiging [15][19]. In het doorgaans minder voorkomende geval dat de actie niet werd goedgekeurd, kan een rechtstreekse correctie worden gedaan naar een correcte state of kan men interpoleren van de incorrecte naar de correcte state.

5.3.1 Problemen bij client side prediction

Hoewel men gewoonlijk dezelfde movement code gebruikt tussen client server is het toch mogelijk dat er door latency afwijkingen tussen client state en server state onstaan. Hieronder volgen enkele situaties waarbij dit kan voorvallen:

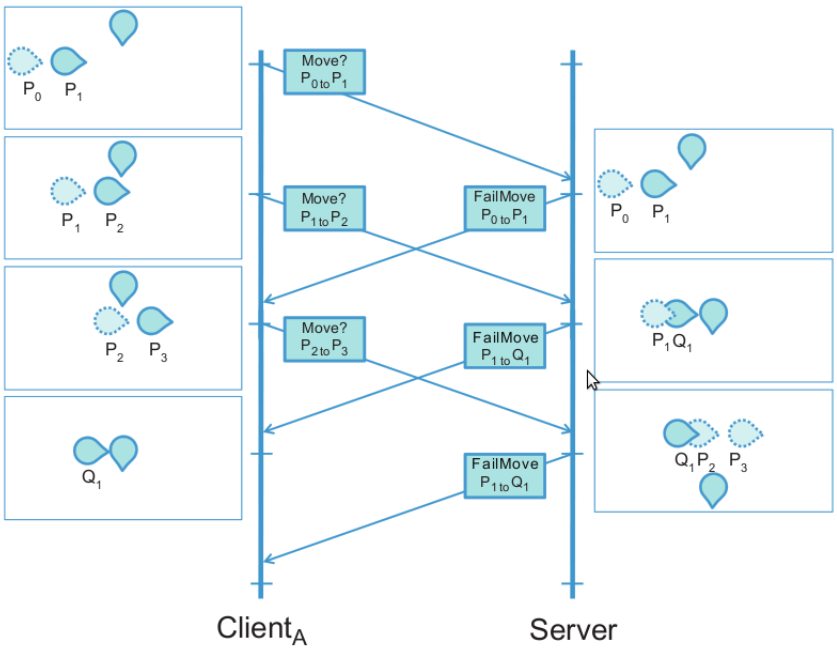
Inconsistente state bij moving object collision: wanneer client 1 te laat een state update over client 2 krijgt van de server, is het mogelijk dat er zich client-side geen collision voordoet, terwijl dat server-side wel het geval is. In dat geval zal er een afwijking ontstaan tussen de state van client 1 en de server en zal er een correctie moeten worden uitgevoerd (Figuur 5).

Inconsistente state bij weapon firing: wanneer client 1 op client 2 kan vuren net voordat client 2 achter een muur verdwijnt, zal het voor client 2 lijken alsof hij geraakt is achter de muur, omdat zijn state voor staat op die van de server door client side prediction (Figuur 6). Hoe client 1 client 2 kan raken is een topic binnen lag compensation en wordt verder besproken in sectie 5.4[1].

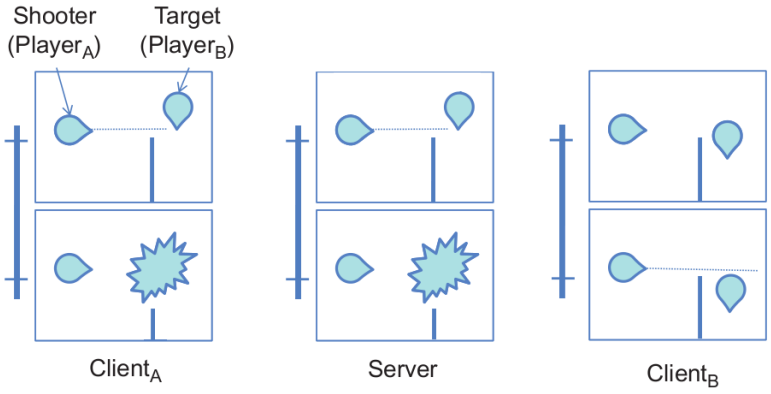
5.3.2 Input prediction

Input prediction is een vorm van client side prediction waarbij de client geen posities doorstuurt naar de server (zie Figuur 5), maar input commands. Hierdoor hoeft de server niet te controleren of een bepaalde positie valid is: de server

[1]De situatie in Figuur 6 werd met betrekking tot client side prediction niet besproken in het boek Networked Graphics [15], maar is wel samengesteld uit grafische elementen van andere figuren uit het boek. Er wordt echter wel een gelijkaardige situatie besproken als het "fire proof player problem" in het hoofdstuk over Lag Compensation.



Figuur 5: Inconsistente state bij moving object collision: client-side lijkt het mogelijk om van P2 naar P3 te gaan. Server-side blijkt er echter een obstakel in de weg te zitten, waardoor een correctie Q1 wordt verstuurd naar de client.[15]



Figuur 6: Inconsistente state bij weapon firing: player A kent perfect de state van de server (we veronderstellen geen lag compensation) en raakt player B. Player B ziet zichzelf local reeds achter de muur door client side prediction, maar deze state werd nog niet door de server aanvaard, waardoor player B toch geraakt zal worden.

voert alle simulations uit aan de hand van de input van de client. Input prediction wordt onder andere gebruikt in de multiplayer games ontwikkeld door Valve [19].

Ook met deze manier van werken is het mogelijk om predictions te doen in plaats van te wachten op een reply van de server. Dit kan door simpelweg de input commands ook lokaal uit te voeren met dezelfde movement code als op de server.

5.3.3 Smooth error correction

Binnen client side prediction is het mogelijk dat de server state afwijkt van de client state. In dit geval moet er dus een correctie worden uitgevoerd op de client, op basis van de response van de server.

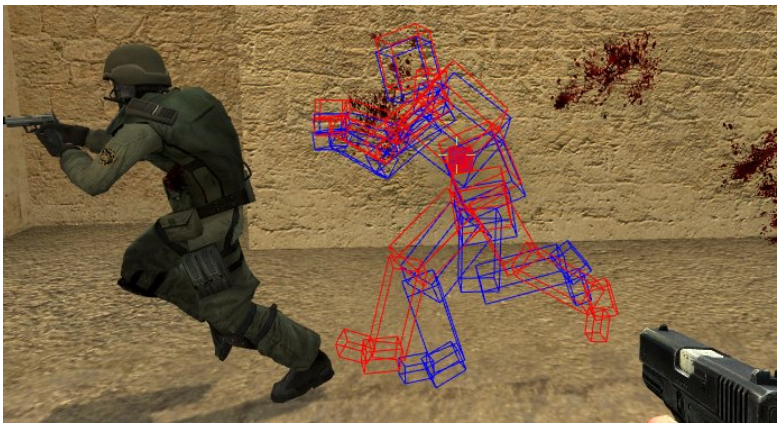
Deze correctie kan ofwel door de state van de client rechtstreeks op de correcte state te zetten (*snapping*), of door *smooth error correction*. Deze laatste is gebruiksvriendelijker omdat het beeld van de client bij deze methode niet gaat "verspringen" bij een correctie.

Over de formele invulling van smooth error correction is niet veel bekend: de meeste informatie over deze algoritmes is te vinden op *wiki*'s en forums zoals *gamedev*. In de implementatie werd daarom empirisch waargenomen welk een goede methode is om fouten smooth te corrigeren (zie sectie 6.6.3).

5.4 Lag compensation

Lag compensation is een techniek die als doel heeft om te vermijden dat spelers met een hoge latency hun acties hieraan moeten aanpassen. Een bekend voorbeeld is het "trailed shooting"-probleem in online FPS games, waarbij spelers vóór een bepaalde speler moeten schieten om hem effectief te raken [3].

Om dit effect te bereiken, wordt server side de *command execution time* berekend. Dit is de tijd waarop een bepaald commando op de client werd uitgevoerd. Vervolgens gaat men alle objecten in de virtuele omgeving laten terug gaan in de tijd naar het moment dat de command werd uitgevoerd. Op deze manier kan men dus eigenlijk de state van de client simuleren op de server en vervolgens het commando uitvoeren [19].



Figuur 7: Lag compensation in Valve's Source engine (Counter Strike: Source) [19]

6 Implementatie

6.1 Concept

Om de implementatie van voorgaande concepten omtrent realtime genetwerkte virtuele omgevingen te verwezenlijken, zal gebruikt gemaakt worden van de Unity engine voor het grafische aspect. Meer bepaald zal een first person shooter game geschreven worden waarbij de gebruiker kan rondlopen in een virtuele omgeving. Andere gebruikers zullen kunnen joinen, waardoor de latency hiding technieken zichtbaar zullen worden.

Zowel op de client als op de server zal het verder mogelijk zijn om een "fake lag" in te stellen en om de latency hiding technieken uit te schakelen / in te schakelen. Hierdoor kan het effect van bepaalde latency hiding technieken ook bekeken worden wanneer er meer of minder lag is.

Ten slotte zal de communicatie tussen client en server ook secure zijn, waardoor men bijvoorbeeld niet de netwerkpakketten kan aanpassen om vals te spelen.

6.2 Architectuur

Qua architectuur werd voor de implementatie gekozen voor een systeem waarbij clients kunnen connecten met een dedicated server. Dit omwille van verschillende redenen:

- P2P games beveiligen is een enorme uitdaging [12]. Men moet namelijk rekening houden met het feit dat de "sensitive data" zoals spelerposities, hit points, ammunition, etc. op een of meerdere clients moet worden bijgehouden. Het mag hierbij niet mogelijk zijn om de data onrechtmatig aan te passen om vals te spelen. Technieken die deze vereiste vervullen vallen ver buiten het bestek van deze bachelorproef en zouden bovendien de complexiteit van het geheel onnodig vergroten.
- De meeste hosts op het internet zitten achter een NAT-router, waardoor men bij P2P applicaties hole punching technieken gaat moeten implementeren om een bidirectionele verbinding te kunnen starten. Ook deze technieken vallen buiten het bestek van deze bachelorproef. De client-server architectuur heeft hier geen last van, omdat men op de server gewoon kan port forwarden.
- Scalability is niet van belang in een standaard first person shooter: de meeste first person shooters ondersteunen tot 16 spelers tegelijk. Het gebruiken van een peer to peer systeem omwille van scalability is in dit geval dus overkill.

Kort samengevat zou een P2P-architectuur dus op bepaalde vlakken onnodige functionaliteit voorzien, wat de keuze voor een client-server systeem verantwoordt.

6.3 Application protocol

In de implementatie wordt gebruik gemaakt van het NFP-protocol (zie sectie 4.3). Hierbij wordt de applicatie-specifieke data of *message* in het `ARGS`-veld

van het NFP `DataPacket` gestoken. Een delegate function `OnReceive` kan ten slotte gedefinieerd worden om applicatie-specifieke acties uit te voeren bij het ontvangen van data in `ARGS`.

6.3.1 Login messages

Structuur: Dezelfde structuur als in het NFP `LOG`-pakket. Het `ARGS`-veld bevat de naam van de speler.

Werking: Wanneer de server een `LOG`-message ontvangt, wordt een instantie aangemaakt van de class `Player`. Deze wordt toegevoegd aan de `PlayerList` dictionary, die paren van `EndPoint`s en `Player`s bevat. Met behulp van een `Instantated` bool in de `Player` instantie, kan Unity controleren of de `GameObject` instantie voor deze speler al bestaat in de scene. Wanneer dit niet het geval is, wordt een nieuw Unity `GameObject` in de scene aangemaakt. In het andere geval wordt de data uit `Player` gebruikt om de positie aan te passen van het bestaande `GameObject`.

6.3.2 Input messages

Structuur: Het `Command`-veld bevat de string "INP". Het `ARGS`-veld bevat 4 bytes voor het IP-adres, 4 bytes voor het poortnummer, 8 bytes voor de tijd waarop het pakket werd verzonden, 4 bytes voor de ingedrukte toetsen op het keyboard, 8 bytes voor de beweegrichting en ten slotte 16 bytes voor de rotatie.

Werking: Input messages worden elke frame vanaf de client verstuurd naar de server en bevatten de toetsen die de gebruiker heeft ingedrukt. Wanneer deze messages aankomen op de server, worden aan de hand van de input de physics daar uitgevoerd. In geval client side prediction is ingeschakeld, worden de physics door middel van dezelfde code ook lokaal uitgevoerd. Ten slotte broadcast de server `PositionMessage`s naar alle clients in de volgende update cycle.

6.3.3 Position messages

Structuur: Het `Command`-veld bevat de string "MOV". Het `ARGS`-veld bevat 4 bytes voor het IP-adres, 4 bytes voor het poortnummer, 8 bytes voor de tijd waarop het pakket werd verzonden en 12 bytes voor de positiecoördinaten.

Werking: Position messages gaan van server naar client. Wanneer deze aankomen op de client worden ze niet onmiddellijk toegepast, omdat dit zou leiden tot "springende" bewegingen. In de plaats wordt de oude positie van de client geïnterpoleerd naar de nieuwe positie. In geval van client side prediction wordt tijdens het voortbewegen smooth error correction toegepast (zie sectie 5.3.3).

6.3.4 Pickup messages

Structuur: Het `Command`-veld bevat de string "PIC". Het `ARGS`-veld bevat 4 bytes voor de `PickupID`, 4 bytes voor de lengte van de pickupnaam en ten slotte de bytes van de pickupnaam.

Werking: Wanneer de gebruiker op `E` drukt, wordt een `PIC` pakket verzonden van de client naar de server. Bij aankomst zal op de server een raycast worden uitgevoerd vanaf de positie van de client die het `PIC` pakket stuurde. Indien zich op de ray een pickup bevindt, wordt een mutex lock gezet zodat andere spelers onmogelijk tegelijk de pickup kunnen oprapen. Ten slotte wordt de pickup in de serveromgeving toegepast en verwijderd en wordt een `PIC` pakket terug gestuurd naar de client om te melden dat de pickup actie gelukt is. Hierop wordt de pickup ook in de clientomgeving verwijderd.

6.4 Tools en resources

Om de vermeldde technieken besproken in sectie 5 makkelijker te kunnen implementeren werd gebruik gemaakt van een aantal tools en resources. Meer bepaald werden deze tools en resources gebruikt om de ontwikkelingstijd van het grafische aspect van de implementatie te verkorten:

- Unity 3D engine: framework voor graphics rendering en physics. Unity ondersteunt ook networking functionaliteiten, maar deze werden uitgeschakeld.
  (http://unity3d.com/)
- De_dust2: originele map ©Valve 2004 uit de game Counter Strike: Source.
  (http://store.steampowered.com/app/240/)
- Models: uit de FPS constructor kit door "Dastardly Banana".
  (http://www.dastardlybanana.com/FPSConstructorWeapons.htm)
- Sounds: verzameld via search engine.
  (http://www.findsounds.com)

6.5 Server

In deze sectie wordt besproken welke rol de server vervult in de implementatie en hoe de doelstellingen vermeld in sectie 2 werden vervuld.

6.5.1 Authoritative environment

De implementatie van de server voorziet een visuele weergave van de omgeving door gebruik te maken van de Unity 3D engine (zie sectie 6.4). Dit zodat de gebruiker kan zien welke state de spelers hebben op de server, of er een client is die last heeft van latency (en dus weinig updates doorstuurt) en wat de eventuele verschillen zijn tussen server state en client state. Concreet betekent dit dat de gebruiker zich tijdens het runnen van de server vrij kan verplaatsen over de map in *spectator mode* door gebruik te maken van het toetsenbord en de muis.

Hetgeen wat de gebruiker op het scherm ziet is de *authoritative state* en wordt dus als enige geldende state beschouwd. De client stuurt enkel input messages (sectie 6.3.2) en de server zal de physics simulation hierop uitvoeren. Op deze manier is het niet mogelijk om client side netwerkpakketten aan te passen en bijvoorbeeld de positie van een speler te veranderen of om door muren te lopen door de map client side aan te passen.

Ten slotte is het ook mogelijk om te kijken naar de server state vanuit het first person oogpunt van één speler in *observer mode* (zie sectie 8). In deze

mode is het duidelijker om onregelmatigheden tussen client state en server state te zien.

6.5.2 Asynchronous NFP

Om de omgeving zonder onderbreking te kunnen laten draaien is het belangrijk dat de server client requests gaat behandelen in een aparte thread. Dit wordt automatisch verzorgd in de NFP-protocol implementatie door gebruik te maken van de `AsyncCallback` delegate in C#.

Een probleem dat zich voordoet bij deze aanpak is dat het in Unity niet mogelijk is om vanuit een secundaire thread calls te doen op `GameObject`s. Wanneer de client een input message stuurt, zal deze dus niet onmiddellijk verwerkt kunnen worden door de positie van de speler in de scene aan te passen. Als oplossing werd gekozen om de input data op te slaan in een dictionary van `Player` instances, die spelers mapt op hun respectievelijke `EndPoint`s of ip / port combinaties. De `Player` dictionary wordt vóór het renderen van elke frame door de main thread van Unity uitgelezen, waardoor de input wel kan worden toegepast.

6.5.3 Update cycle

De update rate van de server is standaard ingesteld op 40 updates per seconde. Dit bekenent concreet dat er elke 25 ms een state update naar elke verbonden client zal worden gestuurd, zodat de nieuwe posities van alle entiteiten kunnen worden afgeleid.

Om een stabiele update rate te behouden, wordt de update cycle in een thread uitgevoerd. Met behulp van een `Stopwatch` wordt verder bijgehouden hoeveel milliseconden de server heeft moeten werken om de state naar alle clients te broadcasten. De thread zal vervolgens een `Thread.Sleep((1000 / updateRate) - millisecondsWorked)` uitvoeren om de update rate exact te behouden.

Ten slotte wordt in de update cycle op regelmatige intervallen ook gecontroleerd of er een client gedisconnect is (bijvoorbeeld wanneer deze voor 10 seconden lang geen input message meer heeft gestuurd).

6.5.4 Dead Reckoning

Werking
Op basis van input messages gestuurd door de client (zie sectie 6.6.1) kan bepaald worden op de server wat de positie, snelheid, versnelling en richting is van deze client. Met behulp van deze gegevens kunnen we dus dead reckoning toepassen.

In de implementatie gebeurt dit door exact dezelfde movement code te gebruiken als in de standard asset `CharacterMotor.js`. De snelheid, richting etc. worden hierbij elke keer aangepast wanneer een nieuw pakket binnenkomt. Een

nieuwe positie wordt dus als volgt berekend: [2]

$$P_1 = P_0 + V_0\Delta T + \frac{1}{2}A_0\Delta T^2 \quad (1)$$

Wanneer de client bijvoorbeeld zou disconnecten tijdens het bewegen, dan zullen andere geconnecteerde clients de gedisconnecte client zien blijven bewegen in dezelfde richting en tegen dezelfde snelheid, totdat de server de disconnect detecteert.

Performance in de implementatie
Het gebruik van deze techniek heeft zowel voordelen als nadelen. Het meest belangrijke voordeel is dat lag van client naar server minder voelbaar wordt. Het lijkt alleen alsof het veranderen van richting of veranderen van acceleratie trager "reageert". Er zullen geen plotse sprongen komen bij de bewegingen van de client. Dit komt omdat de snelheid, richting en acceleratie lineair voorspeld worden via het dead reckoning algoritme op de server.

Een belangrijk nadeel van dead reckoning is dat er, in geval input prediction (zie sectie 6.6.3), snel inconsistenties kunnen ontstaan tussen de client en de server. In geval van 200 ms latency heeft de client bijvoorbeeld 200 ms de tijd om van richting en snelheid te veranderen, terwijl op de server nog steeds aan dead reckoning gedaan wordt op basis van het laatst ontvangen pakket. Dit probleem wordt opgelost door *smooth error correction* (zie sectie 6.6.3).

6.5.5 Lag compensation

Werking
Om de tijd te compenseren tussen input uitgevoerd op de client en input uitgevoerd op de server wordt gebruik gemaakt van lag compensation (zie sectie 5.4). Hiervoor wordt op de server een dictionary bijgehouden van `LCSnapshots`. Deze snapshots bevatten de positie en richting van een bepaalde speler, samen met de tijd waarop deze gegevens gemeten zijn.

De oplagfrequentie van snapshots bepaalt de nauwkeurigheid van de lag compensation. Hoe meer snapshots er worden genomen, hoe accurater men kan teruggaan in de tijd, maar ook hoe meer load er op de server komt de staan. In de implementatie werd gekozen om de opslagfrequentie gelijk te stellen aan de update rate van de server (= 1 snapshot per 25 ms).

Wanneer we de lag van client naar server, de interpolatietijd (er wordt gebruik gemaakt van time based interpolation, zie sectie 6.6.2) en de lag van server naar client optellen bekomen we het totale tijdsverschil tussen de client en de server.[3]

Als de speler nu schiet, komt het pakket later aan op de server, terwijl de client ook nog eens zelf aan het interpoleren is naar een nieuwe positie. Op de server moet dus bij aankomst van het pakket worden teruggerekend naar de world state zoals die was toen de client het schietcommando verstuurd

[2] Deze formule is een benadering van de werkelijke formule die gebruikt wordt om de positie te bepalen in Unity. De code in `CharacterMotor.js` houdt bijvoorbeeld ook rekening met andere factoren uit de fysica zoals de hellingsgraad van bepaalde oppervlakken, wrijving, luchtweerstand, etc.

[3] Wanneer men deze techniek wil testen zonder input prediction, is het belangrijk om de console commands voor input prediction uit te schakelen (sectie 8.3) op zowel de server als de client. Dit om te vermijden dat input prediction incorrect wordt toegepast.

had. Hiervoor roept de server `LagCompensationEngine.Load()` aan met als parameters de beoogde speler en de totale lag zoals hierboven reeds besproken.

Wanneer de correcte state vervolgens geladen is, kan het schietcommando worden uitgevoerd. Dit gebeurt uiteraard analoog als op de client met standaard Unity raycasting methoden. Indien de speler raak schoot zal de client hiervan op de hoogte gebracht worden. Ten slotte wordt de world state hersteld naar de "huidige tijd" zodat het lijkt alsof de lag compensation nooit heeft plaatsgevonden (zie Figuur 9 en Figuur 10).

Implementatiekeuze
Bovenstaande werking werd bepaald aan de hand van de beschrijving van lag compensation in [19] en [3]. Bijna alle bronnen die omtrent lag compensation werden onderzocht tijdens de researchfase van deze bachelorproef wijzen terug naar Valve, wat doet vermoeden dat Valve het eerste bedrijf is dat deze techniek gebruikt heeft. Ondanks de schaarsheid van informatie over de implementatie van lag compensation werd toch een eigen systeem ontwikkeld dat de performance onder delay aanzienlijk verbetert (zie screenshot Figuur 10).[4]

6.5.6 Fake lag

Omdat het niet eenvoudig is om zelf een omgeving met hoge latency op te zetten, werd op de server een "fake" lag mechanisme geïmplementeerd. Op deze manier kan er sneller local gedebugd worden in plaats van telkens op twee systemen te testen. Er werden twee vormen van fake lag geïmplementeerd:

Fake propagation delay: In deze eerste vorm wordt met `Thread.Sleep()` een aantal milliseconden gewacht vóór een pakket verzonden wordt naar alle clients of de server. Omdat het interval waartussen de pakketten aankomen hetzelfde blijft, lijkt deze simulatie dus het meest op propagation delay. Merk op dat er toch nog een aantal verschillen zijn met echte propagation delay. Deze vorm van fake lag is handig om te controleren hoe de host reageert op een trage, maar consistente stream van updates.

Fake processing delay: De fake processing delay gaat gedurende een vooraf bepaald aantal milliseconden pakketten bufferen in plaats van ze te verzenden. Vervolgens worden ze verzonden op een willekeurig moment. Op deze manier kan gecontroleerd worden hoe de host reageert op een inconsistente stream van updates.

Hoe men deze fake lag kan aanzetten of uitzetten wordt verduidelijkt in de appendix (sectie 8.3.1).

[4]Omdat RakNet een van de meest populaire open source network engines is, heb ik een mail gestuurd naar Kevin Jenkins a.k.a. Rakkar om te vragen of lag compensation geïmplementeerd is in RakNet (deze info was niet terug te vinden op de website van RakNet [2]). Hij zei dat dit inderdaad niet het geval was, omdat lag compensation eerder een keuze is qua game code dan qua network code. RakNet laat dus de keuze aan de gebruiker om dit zelf te implementeren of niet.

6.5.7 Ownership and locking

Werking
In de virtuele omgeving is er één pickup item aanwezig. Dit item bevindt zich op een houten doos vlak achter de spawnlocatie. De pickup zal securely ervoor zorgen dat de client een verdubbelde magazinecapaciteit verkrijgt.

Om te garanderen dat er maar één client de pickup krijgt wanneer men bijvoorbeeld met twee clients tegelijk het object probeert op te pakken, wordt server side een mutex lock gelegd rond de procedure die de pickup toekent. Dit werkt omdat elke client een eigen thread heeft waarin requests worden afgehandeld. Die client wiens `PIC`-pakket het eerst de server bereikt, krijgt de pickup toegewezen.

Alternatieve methode
Tijdens de vergadering met het onderwijsteam werd gesuggereerd om de tijd waarop clients het pickupobject hebben opgepakt door te geven aan de server en op basis van deze gegevens te beslissen aan welke client de pickup wordt toegekend in geval ze deze actie tegelijk hadden uitgevoerd. Er zijn echter een aantal redenen waarom gekozen werd om dit niet op deze manier te beslissen:

- Client side de tijd meesturen is spoofable, ook wanneer gebruik wordt gemaakt van encryptie: een client zou zijn netwerkpakket kunnen aanpassen zodat zijn send time altijd kleiner is dan die van een andere speler. Dit is in conflict met de doelstelling *security*.

- Spelers met meer latency worden bevoordeeld: een speler die zelf latency gaat introduceren met behulp van een lag switch of gewoon veel lag heeft zal op deze manier altijd voorrang krijgen op een speler die een goede verbinding heeft. Vanuit het perspectief van de speler met de goede verbinding zouden er bijgevolg vreemde dingen gebeuren: hij zou eerst de pickup toegekend krijgen, maar daarna wordt deze door een antimessage aan een andere speler gegeven.

6.6 Client

6.6.1 Dumb client

In deze implementatie is de clientapplicatie een "dumb client". Dit betekent dat de client enkel input messages stuurt en verder de server laat berekenen wat hiervan het gevolg is.

Het grote voordeel bij deze aanpak is dat de implementatie meer secure is. In eerdere versies van de implementatie werd bijvoorbeeld de positie gewoon in coördinaten vanaf de client naar de server gestuurd. Een kwaadwillige gebruiker zou in dat geval deze coördinaten kunnen aanpassen (zelfs wanneer de packets encrypted zijn!) en dus kunnen vals spelen.

Een nadeel is echter dat er meer load op de server komt te staan, omdat de physics simulaties voor elke client op de server moeten worden uitgevoerd.

6.6.2 Interpolation

Op de client worden twee soorten interpolatie voorzien, waarbij de ene gebaseerd is op het verschil in afstand tussen de positie op de server en de positie

op de client en waarbij de andere gebaseerd is op tijd.

Interpolatie-algoritme 1: Het eerste algoritme is het meest simpele algoritme en wordt gebruikt bij *smooth error correction* (sectie 6.6.3). De functie voor het interpoleren van een waarde x_0 naar een waarde x_1 is hierbij als volgt: $x_0 = x_0 + (x_1 - x_0) * r$ waarbij r de *interpolation ratio* voorstelt en ligt tussen 0 en 1.

Interpolatie-algoritme 2: Dit algoritme werd achteraf bijgevoegd om lag compensation beter te laten werken en is veel complexer. Deze interpolatie garandeert dat de client steeds van positie x_0 naar x_1 gaat in exact m milliseconden:

$$x_0 = x_0 + \frac{x_1 - x_0}{(m - \Delta P)/\Delta T} \tag{2}$$

Hierbij wordt de delta render time voorgesteld als ΔT en de reeds gepasseerde tijd als ΔP. Het algoritme wordt best verduidelijkt aan de hand van een voorbeeld: stel dat x_0 gelijk is aan 0, x_1 gelijk aan 100 en m gelijk aan 100:

1. $\Delta T = 25ms$, $\Delta P = 0$: $0 = 0 + \frac{100-0}{(100-0)/25} \rightarrow x_0 = 25$
2. $\Delta T = 20ms$, $\Delta P = 25$: $25 = 25 + \frac{100-25}{(100-25)/20} \rightarrow x_0 = 45$
3. $\Delta T = 30ms$, $\Delta P = 45$: $45 = 45 + \frac{100-45}{(100-45)/30} \rightarrow x_0 = 75$
4. $\Delta T = 25ms$, $\Delta P = 75$: $75 = 75 + \frac{100-75}{(100-75)/25} \rightarrow x_0 = 100 = x_1$

In dit voorbeeld hebben we inderdaad in 100 ms van 0 naar 100 geïnterpoleerd. Merk op dat het algoritme zich aanpast aan variaties in ΔT en dus ook smoother oogt dan algoritme 1.

Zoals eerder vermeld verbetert dit algoritme ook de performance van lag compensation: we kunnen nu met zekerheid zeggen dat de interpolation delay van de client **altijd** gelijk is aan 100 ms. Bij algoritme 1 is deze tijd onvoorspelbaar. In Figuur 8 wordt interpolatie visueel weergegeven.

6.6.3 Input prediction

De implementatie van input prediction is vrij straight forward: we gaan dezelfde movement code als op de server gebruiken om local een simulatie uit te voeren. Wachten op een `POS`-message van de server hoeft dus niet meer. Dit optimistisch algoritme gaat er vanuit dat er door de gelijkheid in code geen verschil zal ontstaan tussen client state en server state. Eerder (in sectie 6.5.4) werd echter vermeld dat dit in geval van latency echter toch mogelijk is. Daarom werd gebruik gemaakt van smooth error correction.

Smooth error correction
Het smooth error correction algoritme gaat verschillen tussen client state en server state subtiel proberen weg te werken aan de hand van interpolatie-algoritme 1 (sectie 6.6.2) met een *interpolation ratio* gelijk aan $\Delta D/s$, waarbij ΔD gelijk is aan het verschil tussen client state en server state en waarbij s gelijk is aan de *smoothness factor*. Hoe hoger de smoothness factor, hoe minder dat de client gaat merken dat hij latency heeft, maar ook hoe trager de client zijn state gaat

corrigeren. We hebben hier dus te maken met een tradeoff tussen accuracy en smoothness.[5]

6.6.4 Fake lag

Fake lag op de client is identiek aan fake lag op de server. De werking hiervan werd reeds geïllustreerd in sectie 6.5.6.

6.7 Integratie met Unity

Op aanvraag van het onderwijsteam volgt in deze sectie een korte beschrijving over hoe de implementatie werd gekoppeld aan het Unity framework.

Als eerste had ik een klein voorbeeldprogramma gekregen van de heer Jori Liesenborgs waarin kubussen in een scene konden worden verplaatst met behulp van RPC calls. Door dit voorbeeldprogramma werd vrij snel duidelijk hoe in Unity kan worden geïnterageerd met GameObjects.

Vervolgens heb ik dit voorbeeldprogramma multithreaded en socket based gemaakt, zodat ik het `NFP`-protocol kon beginnen ontwikkelen. Uiteindelijk ben ik stap voor stap en feature voor feature gekomen tot het uiteindelijke resultaat.

Dit hele systeem kon gekoppeld worden met Unity door elke frame de functie `UpdatePlayers()` aan te roepen in `ClientScript.cs`. De thread die runt in `CNetwork.cs` gaat deze datastructuur updaten vanaf het moment dat er nieuwe state data binnenkomt. De reden waarom de state data eerst via deze datastructuur moet, werd uitgelegd in sectie 6.5.2: Unity heeft eigenlijk geen support voor multithreading, dus heb ik deze workaround geïmplementeerd.

6.8 Evaluatie

Tot slot volgt nog een korte zelfevaluatie van de implementatie op basis van de features die werden geïmplementeerd.

Sterke punten

- Volledig speelbare online first person shooter.
- Security tegen wallhacks (collisions zijn server side), lag switches, packet tampering en packet replays (voor bijvoorbeeld pickups → gebeurt server side).
- Effectieve latency hiding technieken zoals interpolation, dead reckoning, input prediction en lag compensation (zelfs RakNet heeft niet elk van deze functionaliteiten geïmplementeerd) → resistent tegen matige latency.
- Fake lag systeem om de latency hiding technieken te testen.
- Volledig from scratch geschreven op basis van UDP, dus geen dependencies of andere libraries buiten Unity 3D. Eigen protocol met reliability, virtual connections, ...

[5] In sectie 8.3 wordt vermeld hoe de smoothness parameter kan worden aangepast tijdens het testen van de implementatie.

- Multithreaded en kan dus meerdere clients tegelijk afhandelen → performant.
- Efficiënt genoeg om tot 8 spelers tegelijk in een wereld te spelen. Dit werd getest in de eerdere stages van het project (toen graphics rendering nog niet de bottleneck voor performance was en het dus mogelijk was om 8 windows tegelijk open te zetten).
- Grafische weergave server state: debuggen wordt makkelijk, de verschillen tussen client state en server state kunnen geobserveerd worden, ...
- Spectator mode en observer mode, met als doel om de wereld beter te kunnen bekijken.
- Grafische weergave van interpolation (interpolation ghost) en lag compensation (before correction en after correction).

Zwakke punten

- Synchronisatie niet geïmplementeerd, dus NTP synchronisatie van alle participerende hosts is nodig (tenzij alles local gebeurt).
- Geen security tegen spoofing van het IP-adres: een gebruiker die het IP weet van een andere client kan commando's sturen als deze client. Dit kan echter worden opgelost door een unique ID mee te geven aan een IP en deze te gebruiken om een spoofer te onderscheiden van een echte speler (in dat geval nog steeds vulnerable tegen MitM attack → asynchronous encryption scheme nodig).
- Geen security tegen aimbots. Aimbots zouden kunnen berekenen hoeveel van een bepaalde input moet gegeven worden om een doelwit te raken.
- Clients krijgen ook data van andere clients die niet binnen het gezichtsveld vallen. Indien de client bijvoorbeeld game data aanpast zodat de omgeving 50% transparant is, kan deze alle spelers op de map zien rondlopen.

Mogelijke uitbreidingen

- Alle bovenstaande punten uit "zwakke punten".
- Extra models, animations, grotere hitboxes (ze zijn aan de kleine kant nu), meerdere wapens, ...
- Duidelijkere interface voor het ingeven van commando's.

7 Conclusie

Achter de genetwerkte virtuele omgevingen van vandaag schuilt meer dan men eerst zou verwachten: de latency hiding technieken die men gebruikt zijn transparant en hebben als doel om de gebruikservaring van de speler te verbeteren.

Toch zijn niet alle technieken voor elke soort virtuele omgeving even nuttig: in een online Realtime Strategy Game zou input prediction bijvoorbeeld totaal overbodig zijn, terwijl deze techniek in online First Person Shooters een hogere meerwaarde heeft. Hetzelfde geldt voor technieken zoals dead reckoning en lag compensation: deze technieken zijn meestal niet nuttig bij bijvoorbeeld een virtuele chatroom in de aard van "Second Life".

Deze bachelorproef heeft mij een beter inzicht doen krijgen in de gebruikte latency hiding technieken. Bovendien heb ik deze technieken zelf kunnen implementeren. Een van de moeilijkheden hierbij was dat er voor sommige technieken, waaronder lag compensation, relatief weinig documentatie beschikbaar was. Voor bepaalde problemen heb ik dus zelf een creatieve oplossing moeten zoeken.

De grootste problemen omtrent latency hiding ontstaan doordat men rekening moet houden met *security* – iets wat ik in het begin van deze bachelorproef totaal niet verwacht had. Denk bijvoorbeeld aan het doorsturen van posities: zonder security kan de client gewoon coördinaten doorsturen naar de server om te laten broadcasten. Een techniek als input prediction is in dat geval overbodig.

Een laatste conclusie die ik kan trekken uit deze bachelorproef is dat ik veel nieuwe dingen geleerd heb omtrent networked graphics. Ik had in de zomervakantie 2010 - 2011 mij met dit onderwerp al eens bezig gehouden, maar door gebrek aan research en tijd heb ik het "hobbyproject" toen aan de kant geschoven. Ik ben daarom blij dat ik het onderwerp mocht behandelen als onderdeel van mijn bachelorproef.

8 Appendix A: Praktische info

8.1 Setup

Bij het opstarten van de client wordt automatisch een configuratiefile aangemaakt in dezelfde folder als de executable. Deze file kan worden aangepast naargelang men lokaal of remote de implementatie wil testen.

Lokaal: Om de server en client te testen op localhost moeten de velden `ip` en `myIP` worden ingesteld op `127.0.0.1`. De waarde van `port` kan de gebruiker zelf kiezen.

Remote: Om de implementatie remote te testen dient de server `myIP` op het eigen IP te zetten. De remote client moet in het veld `ip` het IP van de server zetten en in het veld `myIP` het eigen IP (de implementatie voorziet hiervoor geen automatische detectie). **Belangrijke nota:** de implementatie voorziet geen systeem om klokken te synchroniseren. Het is daarom belangrijk om via het `NTP-protocol` de klokken van beide machines *perfect* te synchroniseren. Indien dit niet gedaan wordt, gaat het lag compensation algoritme een foute delay berekenen tussen de twee hosts, wat verkeerde resultaten zal produceren.

8.2 Controls

- Movement: Z, Q, S, D en muis
- Jump: Spacebar
- Shoot: Linkermuisknop
- Reloading: R
- Pick up item: E
- Open command window: TAB
- Enter observer mode: H[6]

8.3 Commands

Commando's kunnen worden ingegeven door in-game op `TAB` te drukken. Hieronder volgt een lijst van mogelijke commando's die essentieel zijn om de latency hiding technieken goed te kunnen testen.

8.3.1 Introduceren van fake lag

De gebruiker kan met behulp van het `fakelag_server x` commando (zie sectie 8) zelf bepalen hoeveel x milleseconden propagation lag er op de server moet zijn. In essentie wordt dan de update rate verlaagd, waardoor alle clients minder snel state updates ontvangen en dus lag wordt gesimuleerd. Bij grote hoeveelheden

[6]Observer mode kan enkel gebruikt worden op de server wanneer er één client reeds is gejoined. In deze mode kan de omgeving op de server bekeken worden vanuit het first person perspectief van de client. Dit is handig om de verschillen tussen server state en client state te zien tijdens het testen van de implementatie.

fake lag is het aangeraden om input prediction in te schakelen, zodat dit effect optimaal kan worden weergegeven (sectie 8.3.4).

Met `fakelag_client x` kan hetzelfde gedaan worden op de client kant.

Met `fakelag_proc x` kan op zowel op de server als op de client fake processing delay worden geactiveerd (zie sectie 6.5.6). Deze vorm van lag is realistischer omdat ze de update rate niet verlaagt, maar pakketten gaat bufferen en random gaat uitsturen.

8.3.2 Interpolation: ratio and time

De interpolation ratio van algoritme 1 kan worden aangepast met het commando `set_ierp x` waarbij x gelijk is aan de nieuwe ratio. De ratio moet tussen 0 en 1 liggen. Een ratio van 1 betekent geen interpolatie. Een ratio van 0 betekent het compleet negeren van network updates en oneindig interpoleren. De interpolatietijd m van algoritme 2 staat vast op 100 ms.

Interpolatie kan volledig worden uitgeschakeld door `enable_ierp 0` in te geven.

8.3.3 Interpolation and shooting ghosts

- Om de interpolatie duidelijk te zien kan op een client `enable_ghost 1` worden ingegeven. Er wordt dan een 'ghost interpolation image' weergegeven van andere clients die daarna de game joinen.
- Correcties van lag compensation worden sowieso weergegeven op de server: de rode ghost is waar de client normaal zou schieten. De blauwe ghost geeft de positie weer na correctie door lag compensation.

8.3.4 Input prediction

Input prediction is default uitgeschakeld en kan als volgt worden ingeschakeld om de effecten van latency te reduceren: `enable_inputprediction 1`. **Belangrijk:** dit commando moet zowel op de client als op de server worden uitgevoerd. Anders zal lag compensation niet correct worden uitgevoerd (bij input prediction is er immers *geen* interpolation delay).

8.3.5 Lag compensation

Lag compensation is default ingeschakeld en kan als volgt worden uitgeschakeld om de effecten van latency te tonen bij het vuren op objecten of spelers: `enable_lagcompensation 0`. Omdat lag compensation een server side techniek is, moet dit commando op de server worden uitgevoerd. Op de client zal dit commando geen effect hebben.

8.4 Testing

In deze sectie wordt beschreven hoe de geïmplementeerde latency hiding technieken getest kunnen worden.

8.4.1 Testing interpolation

1. Start de executable 2 keer op.
2. Laat 1 proces opstarten als server. Laat daarna de andere joinen als client.
3. Druk in het client scherm op `TAB` en geef volgend commando in: `set_ierp 0`.
4. Wanneer men nu rondloopt met de client, zal men zien dat de positie van de client rechtstreeks zal updaten in plaats van smooth te interpoleren. Vermits interpolation default aan staat, kan men dus concluderen dat interpolation naar behoren werkt.

8.4.2 Testing dead reckoning

1. Start de executable 3 keer op.
2. Laat 1 proces opstarten als server. Laat daarna de anderen joinen als client. Zorg hierbij dat de eerste client op een andere positie staat als de tweede client (anders gaan deze 'in mekaar' spawnen vermits de spawn point op coördinaat (0, 0, 0) ligt).
3. Houd de `Z`-knop ingedrukt op de ene client om vooruit te lopen en sluit tegelijkertijd het vester (met de muis of `ALT-F4`).
4. Gedurende 10 seconden kan het dead reckoning effect op de andere client worden waargenomen: de server ontvangt van client 1 geen pakketten meer omdat het venster werd gesloten, maar gaat aan dead reckoning doen op basis van de laatst ontvangen snelheid, positie en richting.

8.4.3 Testing input prediction

1. Start de executable 2 keer op.
2. Laat 1 proces opstarten als server. Laat daarna de andere joinen als client.
3. Druk in het client scherm op `TAB` en geef volgend commando in: `fakelag_client 200`. Van client naar server zal nu 200 ms delay optreden.
4. Druk in het server scherm op `TAB` en geef volgend commando in: `fakelag_server 200`. Van server naar client zal nu ook 200 ms delay optreden.
5. Loop rond in de wereld om de latency vast te stellen.
6. Druk in het client scherm op `TAB` en geef volgend commando in: `enable_inputprediction 1`.
7. Loop opnieuw rond en verifieer dat de client inderdaad niet langer moet wachten op goedkeuring van de server.
8. Nota: indien lag compensation tegelijk wordt getest, is het belangrijk dat ook op de server het commando `enable_inputprediction 1` wordt uitgevoerd! Anders zal de lag compensation incorrect gebeuren.

8.4.4 Testing lag compensation

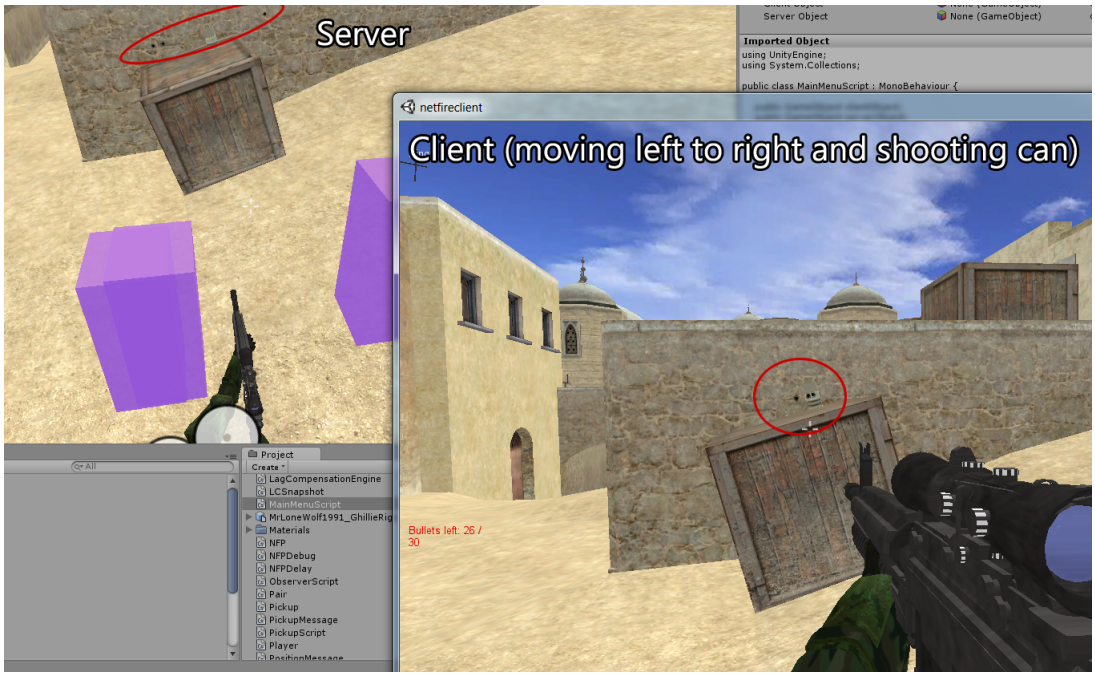
1. Start de executable 2 keer op.
2. Laat 1 proces opstarten als server. Laat daarna de andere joinen als client.
3. Druk in het server scherm op `TAB` en geef volgend commando in: `enable_lagcompensation 0`.
4. Schiet al lopend recht op een bepaald object in de wereld (bijvoorbeeld de pickup).
5. Kijk op het server scherm. De positie van de bullet hole zou sterk moeten verschillen tussen client en server, omdat geen lag compensation werd toegepast.
6. Druk in het server scherm op `TAB` en geef volgend commando in: `enable_lagcompensation 1`.
7. Herhaal stap 4. Deze keer zou de positie van de bullet hole ongeveer exact overeen moeten komen tussen client en server.

8.4.5 Testing ownership

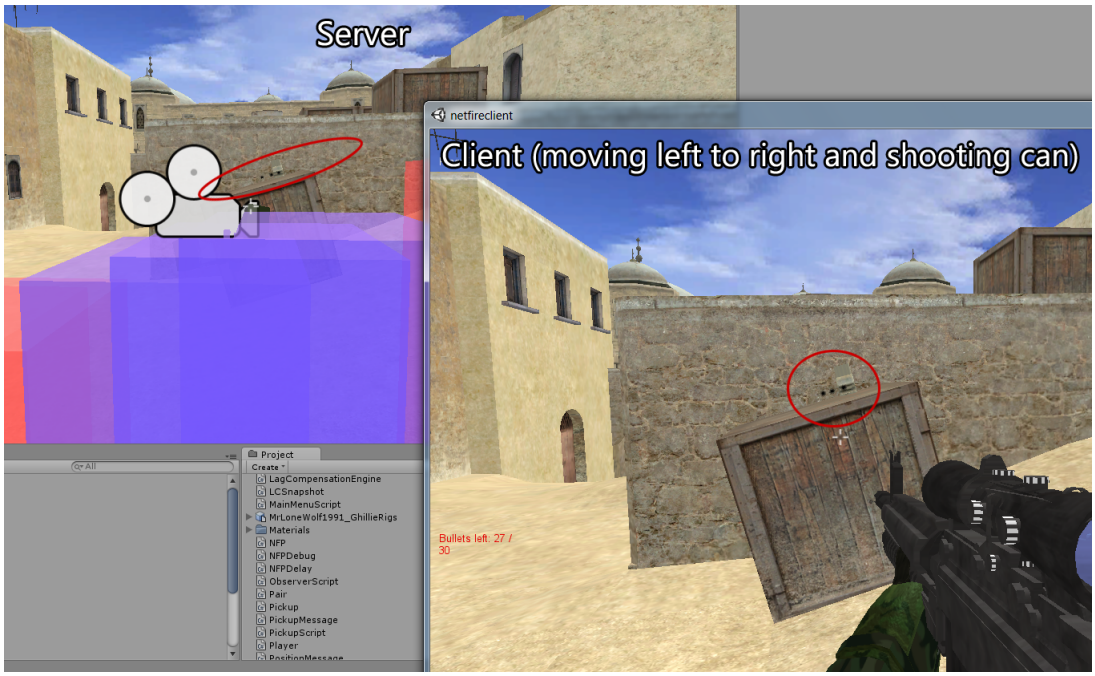
1. Configureer de config file `clientconfig.ini` op 3 PC's.
2. Start op elke PC een executable.
3. Laat PC 1 één proces opstarten als server. Laat daarna de andere PC's joinen als client.
4. Loop met beide clients naar de pickup en mik met de crosshair recht op het object.
5. Spreek met een tweede persoon af om tegelijk op een bepaald moment op de pickup knop (`E`) te drukken.
6. Verifieer dat slechts één speler de pickup krijgt.

9 Appendix B: Screenshots

Figuur 8: Weergave van de interpolation ghost: de rode kubus is de effectieve positie die de client heeft binnen gekregen. De witte kubus interpoleert naar de rode kubus.



Figuur 9: Schieten en bewegen op een object zonder lag compensation: de kogels zijn op de server afgeweken omdat er geen rekening werd gehouden met interpolation delay, server delay en client delay.



Figuur 10: Schieten en bewegen op een object met lag compensation: de kogelinslagen komen overeen op de server en op de client. De rode kubussen zijn plaatsen waar de client oorspronkelijk stond. De blauwe kubussen zijn correcties.

10 Appendix C: Logboek

Lesvrije week: De belangrijkste stukken gelezen uit de boek "Networked Graphics": problemen omtrent het probleem distributed boids en de gerelateerde datamodellen: data push model, data sharing model en data coupled model. Ook de issues uit chapter 5: architectuur, data sharing, schaalproblemen en de rol van client en server binnen networked graphics. Daarna heb ik gelezen over hoe RakNet (open source API voor networked graphics) werkt en over principes zoals ownership van objecten in online games.Ten slotte in part III heb ik geleerd wat latency en jitter juist is, wat voor impact dit heeft en hoe men elk type van latency kan oplossen (dead reckoning, interpolation, extrapolation, ...). Het hoofdstuk over scalability heb ik niet gelezen, omdat ik niet denk dat dit niet de focus is van mijn bachelorproef. Het laatste interessante uit de boek vond ik chapter 13.1 ivm security.

Indien ik een netwerklaag moet schrijven voor de "Little Big Planet"game zoals besproken tijdens onze vergadering, lijkt het mij misschien het beste om een soort transparante netwerklaag te schrijven waarbij de belangrijke data (positie, skills,..) in een object wordt gestoken om over het netwerk verstuurd te worden. Op deze manier moet de andere developer van de game geen speciale dingen doen om de netwerklaag te doen werken. Alhoewel ik nog eens goed moet nadenken over hoe ik dit precies zou moeten doen. Qua architectuur zou

ik ten slotte gaan voor client / server waarbij de client een zogenaamde "dumb clientïs. Dit omdat dumb clients het meest secure lijken vanwege hun "command messages"die ze naar de server sturen.

Week 1: Op het internet nog wat extra research gedaan naar networking in games. Volledige serie "Networking for Game Programmers"gelezen va Glenn Fiedler (http://gafferongames.com/networking-for-game-programmers/). Deze serie beschrijft meer specifiek hoe je de features van networking in games implementeert (virtual connections, reliability indien vereist, flow control, ...). Ik ben ook begonnen met het samenstellen van een paper waarin ik alle kennis opgenomen uit het boek en de artikels op internet compileer.

Week 2: Kennismaking met Unity 3D engine vooral door het volgen van online tutorials. Willekeurige papers gelezen over dead reckoning. Deze week iets minder kunnen doen door naderende examen van het keuzevak "Micro economie".

16/03/12: Gewerkt aan paper (beschrijving features UDP / TCP en vergelijking)

21/03/12: Idee uitgewerkt voor eigen protocol "Netfire". Nagedacht over welke features het protocol moet voorzien en in welke context deze gebruikt moeten worden. Deze ideen verwerkt in paper.

24/03/12: Demo-implementatie gemaakt met RPC calls in Unity. Nog gebruik makende van RakNet

26/03/12: Begonnen aan C# server. Omvormen van RPC calls naar eigen asynchronous socket server. Met netcat connecten om te zien of de server naar behoren werkt

27/03/12: Idem aan maandag, maar vooral gewerkt aan project SE

31/03/12: NFP Protocol verder uitgewerkt. Shared class gemaakt voor client en server zodat reliability kan werken aan beide kanten zonder duplicate code.

2/4/2012: Reliability afgewerkt en connections. Packets kunnen nu op een betrouwbare manier verzonden worden over het netwerk. Dit gebeurt transparant voor de gebruiker. Getest met "TES"packets

3/4/2012: Concurrency problemen opgelost met de asynchrone client en server. Client omgevormd zodat de RPC call voor spawnen vervangen is door een connection in NFP. Clients komen in een hash table op de server en broadcast is supported

4/4/2012: Grotendeels idem aan gisteren. Afgewerkt en begonnen aan het verzenden van MOV pakketten. Probleem: meerdere clients zorgt voor crash op client niveau. Op server niveau crash na steeds reconnecten.

5/4/2012: Server crash was door ICMP message (zie blog). Client deadlockte op schrijven naar een log message (mutex probleem met 2 clients samen). Deze

problemen opgelost resulteren in werkend prototype. Meerdere clients kunnen nu connecten en mekaar smooth zien bewegen op scherm. Echter met hoge bandwith omdat nog geen speciale technieken worden gebruikt. Ten slotte blog aangemaakt met alle details van de afgelopen weken: http://pieterrobyns.wordpress.com/

6/4/2012: Server optimalisaties, basis Entity Interpolation gemplementeerd, detecteren van client disconnects en bijbehorende message,...

8/4/2012: Grafische aspect van de client verbeterd: speler kan nu rondlopen als in een first person shooter. Dit om later lag compensation te kunnen implementeren.

9/4/2012: Stabielere server tickrate door ms worked bij te houden. Op client worden disconnects nu afgehandeld door spelers te verwijderen

12/4/2012: Gewerkt aan paper (interpolation, extrapolation, disconnects,...)

13/4/2012: Animations toegevoegd. Nu ook Z-as in PositionMessage gestoken. Gewerkt aan paper (zie netphyx.be/bachelorproef.pdf).

14/04/2012: Player health, hits, raycasting + shooting, sound en console in UI met mogelijkheid om interpolation uit te zetten etc

17/04/2012: Presentatie gemaakt voor donderdag

21/04/2012: Omvormen Console server naar Unity server

22/04/2012: Omvormen Console server naar Unity server

29/04/2012: Secure movement geimplementeerd

1/5/2012: Tweaks aan secure movement

2/5/2012: Secure movement werkt nu met een vector die wordt doorgestuurd om de kijkrichting te bepalen (zie paper)

3/5/2012: Gewerkt aan paper

7/5/2012: Begonnen aan implementatie van secure shooting

8/5/2012: Afwerking secure shooting

9/5/2012: Begonnen aan input prediction: acties gebruiker moeten onmiddellijk zichtbaar worden

10/5/2012: Input prediction: problemen met synchronisatie server state en client state zorgen voor kleine afwijkingen, waardoor de raycasts voor shooting fel afwijken over grote afstanden. Smooth error correction algoritme gemaakt om dit op te lossen

11/5/2012: Fake lag functionaliteiten toegevoegd. Tweaks aan input prediction, omdat blijkbaar bij hogere latency de afwijkingen groter zijn dan normaal. Is dit normaal bij 220 ms lag? Verder onderzoeken.

12/5/2012: Begonnen aan pickup implementatie. Introductie van PIC-pakket, dat een client laat weten welke pickup hij heeft opgeraapt. Verder is het oprapen van pickups secure (men kan geen packet forgen om het op te rapen).

14/05/2012: Afwerking pickup implementatie. Enkel todo: locking

15/05/2012: Werken aan paper

16/05/2012: Werken aan paper

18/05/2012: Werken aan paper

19/05/2012: Nieuwe blogpost aangemaakt en nog verder gewerkt aan paper.

22/05/2012: Gewerkt aan paper

25/05/2012: Testen van implementatie op remote pc's

26/05/2012: Bugs gefixed bij remote pc test

8/6/2012: Afwerking paper

9/6/2012: Afwerking paper, cleanup van de code

10/6/2012: Afwerking paper

Referenties

[1] Unity 3D. Unity 3d engine. Internet. http://unity3d.com/.

[2] Kevin Jenkins a.k.a. Rakkar. Raknet. Internet. http://www.jenkinssoftware.com/.

[3] Yahn W. Bernier. Latency compensating methods in client/server in-game protocol design and optimization. *Valve*, page 12, /.

[4] Ashwin R. Bharambe. Scalable and secure architectures for online multiplayer games. /, page 1, 2006.

[5] Bunnie. Implementation of mitm attack on hdcp-secured links. Internet. http://events.ccc.de/congress/2011/Fahrplan/events/4686.en.html.

[6] Chia chun Hsu, Jim Ling, Qing Li, and C.-C. Jay Kuo. On the design of multiplayer online video game systems. *University of Southern California*, page 3, 2002.

[7] Clark D. D. and Tennenhouse D. L. Architectural considerations for a new generation of protocols. *Sigcomm Computer Communication Review*, pages 200–208, 1990.

[8] Glenn Fiedler. Udp vs. tcp. Internet. http://gafferongames.com/networking-for-game-programmers/udp-vs-tcp/.

[9] D. Hemmendinger, A. Ralston, and E. D. Reilly. Client/server term definition. *International Thomson Computer Publishing*, page 1, 1998.

[10] Information Sciences Institute. Rfc: 793, transmission control protocol. Internet. http://www.ietf.org/rfc/rfc793.txt.

[11] James F. Kurose and Keith W. Ross. *Computer Networking: A Top-down Approach*. Pearson Addison-Wesley, Boston, MA 02116, 2010.

[12] Christoph Neumann, Nicolas Prigent, Matteo Varvello, and Kyoungwon Suh. Challenges in peer-to-peer gaming. *ACM SIGCOMM Computer Communication Review*, page 1, 2007.

[13] Markus Rupp Philipp Svoboda, Wolfgang Karner. Traffic analysis and modeling for world of warcraft. /, page 1, 2012.

[14] Dan Royer. Network game programming. Internet. http://www.flipcode.com/archives/Network_Game_ProgrammingIssue_07_I_bent_my_Wookie.shtml.

[15] Anthony Steed and Manuel Fradinho Oliveira. *Networked Graphics*. Morgan Kaufmann, 30 Corporate Drive, Suite 400, Burlington, MA 01803, USA, 2010.

[16] Yasui T., Ishibashi Y., and Ikedo T. Influences of network latency and packet loss on consistency in networked racing games. *Sigcomm workshop on network and system support for games*, pages 1–8, 2005.

[17] Enterprise Technology. Client-server vs. peer-to-peer. Internet. http://www.enterprise-technology.net/network2.htm.

[18] TNet. Learn about udp and tcp. Internet. https://sites.google.com/site/tnetsite/tutorials-and-articles/networking-architectures.

[19] Valve. Source multiplayer networking. Internet. https://developer.valvesoftware.com/wiki/Source_Multiplayer_Networking.

[20] Dan S. Wallach. A survey of peer-to-peer security issues. /, page 1, 2012.

[21] Jing Wu and Michel Savoie. Peer-to-peer network architecture. *Communications Research Center Canada*, page 1, /.