GVO : Eindverslag
20 mei 2010

Robin Marx
Hasselt University

Jimmy Cleuren
Hasselt University

Nick Michiels
Hasselt University

ABSTRACT
Dit verslag bespreekt de basisstructuur van ons framework, de reeds gebruikte protocollen voor verschillende informatiestromen, traces voor verschillende situaties en optimalisaties en een kritische bespreking van al deze elementen.

Author Keywords
Architectuur, protocol, trace

CONCEPT
Voor het spel hebben we gekozen voor een hybride Real Time Strategy / First Person Shooter aanpak. Het idee is dat een speler veel avatars in zijn controle heeft en kan verplaatsen zoals in een RTS, waarbij hij ten alle tijde in 1 van deze avatars kan "duiken" en deze kan besturen alsof hij in een FPS zou zitten.

Dit concept laat ons toe om later verschillende optimalistaties door te voeren (het is bijvoorbeeld moeilijk om voor elk object constant exacte posities te sturen omdat het er zoveel zijn) en vergroot de schaal van de NVE ineens gevoelig tov een wereld waarin welke speler slechts 1 object bestuurt. De verschillende onderdelen zijn er dan ook op gericht om een groot aantal objecten per speler te kunnen afhandelen.

ARCHITECTUUR
We hebben gekozen voor een gelaagde architectuur bestaande uit 3 verschillende onderdelen, zie figuur 1. Op deze manier houden we de functionaliteit abstract en afgescheiden van elkaar zodat code efficient kan hergebruikt worden.

Onderaan zit de SocketManager. Deze beheert alle sockets op het laagste niveau, zorgt voor abstractie van send en receive, buffert het verkeer en biedt mogelijkheden voor introductie van delay en packet loss en maakt bandwidth monitoring mogelijk. Naar elke client worden 2 connecties geopend, 1 TCP en 1 UDP, waarbij ook voor de UDP een handshake-algoritme wordt gebruikt om de connectie op te zetten.

Het tweede niveau is de eigenlijke implementatie van de netwerklogica. Afhankelijk van de gebruikte NetworkManager zullen we een andere netwerk-opbouw krijgen en zullen de connecties anders worden aangemaakt. We maken gebruik van een dedicated server en voor alle applicaties zijn er ook pure console-only versies voorzien. De hybrid structuur (waarbij zowel client-server als P2P connecties worden gelegd) is nog niet geïmplementeerd maar zien we als een interessante uitbreiding.

Het laatste niveau zijn de streams. Deze behandelen vooral de gamelogica. Zij maken gebruik van een abstracte view op het netwerk (voorzien door de NetworkManager) om berichten te sturen aan geselecteerde spelers. Voor verschillende berichten zijn er verschillende streams mogelijk, waarbij het type stream bepaalt of een bericht reliable of unreliable wordt verzonden.

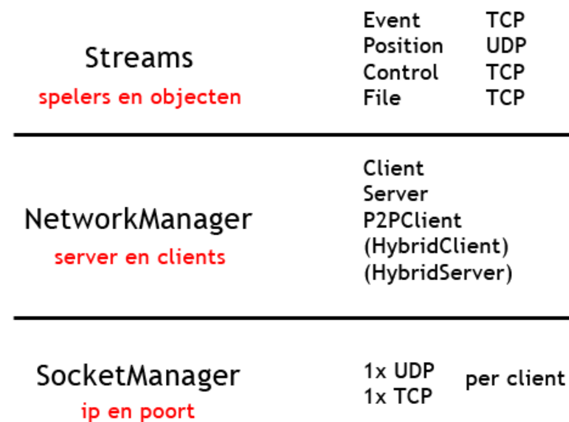

Figure 1. Gelaagde basisstructuur van het framework.

PACKETS
Streams maken intern packets aan met daarin de gegevens die ze willen versturen naar andere spelers. Deze packets geven ze dan aan de NetworkManager en die bepaalt op welke manier deze packets bij de ontvangers aankomen. Al deze packets hebben 1 gemeenschappelijke header van 8-12 bytes groot, bestaande uit volgende gegevens :

```
uint32_t length
uint8_t packetType
uint8_t streamType
uint16_t senderId
```

```
bit timestampIncluded
(uint64_t timestamp)
```

1 bit bepaalt of er een timestamp in het packet zit of niet. Als deze de waarde 1 heeft, zijn de volgende 64 bits de timestamp, anders volgt de rest van de packet inhoud.

Afhankelijk van het packetType zal de rest van het packet worden ingevuld. Hiervoor maken we gebruik van een BitStream class die makkelijk toelaat allerhande variabelen weg te schrijven naar een byte-array voor verzending. Deze bitstream schrijft/leest voor een string bijvoorbeeld automatisch eerst 32 bytes voor de lengte van deze string en dan pas de echte string.

De packet-samenstelling en grootte zal afhangen van het doel waarvoor het wordt gebruikt en verschillende optimalisaties zijn mogelijk per soort packet.

Overzicht

Deze paragraaf bevat een overzicht van de verschillende soorten packets per stream. Bij elke stream is aangegeven of hij gebruik maakt van TCP of UDP en waarom.

ControlStream

Deze stream bevat alle netwerk-gerelateerde controle informatie.

Deze stream maakt gebruik van TCP omdat deze packets zeer belangrijk zijn voor het opzetten van het netwerk, weinig worden gestuurd en niet real-time gevoelig zijn.

- *Hello*: bevat alle informatie over 1 speler en wordt gestuurd bij het joinen van het spel. IP en tcp/udp luisterpoorten worden gestuurd zodat hierop connectie kan worden gemaakt door andere spelers in een P2P scenario.

```
uint16_t playerId
string playerName
string ip
uint16_t tpcListenPort
uint16_t udpListenPort
```

- *PlayerList*: stuurt voor elke gekende speler de inhoud van een HelloPacket. Dit packet is dus eigenlijk een soort aggregatie van verschillende HelloPackets en wordt gebruikt om spelers op de hoogte te brengen van hun medespelers.

PositionStream

Deze stream bevat alle positie-updates. Intern houdt het programma bij of de positie van een object is veranderd. Als dit het geval is, en het is nodig, wordt er een update gestuurd.

Deze stream maakt gebruik van UDP omdat er zeer veel packets gaan worden gestuurd. De overhead van TCP is hierbij ongewenst. Ook is er de real-time requirement en de nutteloosheid van het herverzenden van een gemist packet omdat er vlak erna waarschijnlijk wel een nieuwe update zal ontvangen worden.

Later in dit document bespreken we incrementele updates, waardoor de inhoud van dit packet wordt aangepast zodat het kleiner wordt.

- *Position*: beschrijft de positie, orientatie en huidige animatie voor een bepaald object. Dit packet wordt zeer vaak verstuurd en is het belangrijkste packet om state synchronisatie mee te doen.

```
uint16_t objectId
float32_t posX
float32_t posY
float32_t posZ
float32_t orX
float32_t orY
float32_t orZ
string animationStateId
```

ActionStream

In onze NVE hebben we 2 verschillende soorten bewegingen : wanneer objecten in de RTS modus zitten en wanneer ze in de FPS modus zitten. In de RTS bestuurt de speler de objecten vanuit een zichtpunt dat boven het spelveld hangt. De speler selecteert 1 of meerdere objecten en geeft door ergens in de wereld te klikken de opdracht dat de objecten daarnaartoe moeten bewegen. De objecten bewegen dan automatisch naar het opgegeven punt. Dit is een groot verschil met de FPS modus, waarbij de speler als het ware in 1 van zijn objecten kruipt en het volledig zelf kan besturen.

Deze 2 modussen geven geheel andere soorten bewegingen. In RTS zullen de objecten via een welgekend pad naar het opgegeven punt bewegen, terwijl in FPS de speler de bewegingen van het object op eender welk moment kan aanpassen. We splitsen deze 2 modussen dan ook op in aparte streams, waarbij de PositionStream instaat voor FPS en de ActionStream voor RTS. Waar de PositionStream constant updates verstuurt, zal de ActionStream enkel een update sturen als zijn target-punt verandert. Hierdoor worden er veel minder packets verstuurd door de ActionStream dan voor de PositionStream. Later in dit document worden de resultaten van deze opsplitsing besproken.

- *Action*: beschrijft voor een object waar het zich nu bevindt en waar het zich naartoe aan het bewegen is volgens het huidige actieve pathfinding algoritme in de wereld.

```
uint16_t objectId
float32_t posX
float32_t posY
float32_t posZ
float32_t targetPosX
float32_t targetPosY
float32_t targetPosZ
```

FullStateStream

Deze stream bevat alle full state informatie die wordt uitgewisseld als een nieuwe speler het spel joint. Intern wordt er een persisted database bijgehouden die kan aangesproken worden als een nieuwe speler joint om zijn objecten te kunnen inladen. De informatie uit de database moet dan ook worden uitgewisseld naar de andere spelers.

Deze stream maakt gebruik van TCP omdat het belangrijk is dat alle spelers alle objecten hebben. Ook zijn deze packets zoiezo redelijk omvangrijk en erg laag in frequentie, waardoor de overhead van TCP geen probleem vormt.

- *FullState*: Bevat een lijst van objecten voor 1 speler die moeten worden ingeladen als de speler joint. Per object wordt de inhoud van een PositionPacket gestuurd. Dit packet is dus eigenlijk een aggregatie van verschillende PositiePackets. We gaan ervanuit dat er na het FullStatePacket positieupdates gaan binnenkomen voor de objecten waarvoor het nodig is, maar een initiele positie in dit packet is nodig voor objecten die bijv. niet bewegen. Een extra veld per object is de string-id van elk object. Intern werkt de engine met string-ids, maar voor door te sturen over het netwerk gebruiken we uint16_t ids. De mapping van de ints op de strings wordt gedaan in het fullstatepacket.

EventStream

Naast positieupdates zijn er ook andere events in de wereld die moeten worden gestuurd, bijvoorbeeld het aanmaken van een nieuw object, selecteren van een object of een item oprapen.

Deze stream maakt gebruik van TCP omdat het belangrijk is dat deze events worden ontvangen door de spelers waarvoor ze van belang zijn. Deze events zijn ook van een veel minder hoge frequentie dan positieupdates. Het enige probleem is dat deze packets vaak een zeer kleine payload hebben, waardoor de TCP overhead relatief gezien erg hoog zal zijn. Aggregatie lost volgens ons maar weinig op voor deze packets omdat ze laag in frequentie zijn en er moeilijk kan gewacht worden. Het positieve aan deze lage frequentie is dat de echte hoeveelheid overhead erg laag is.

- *NewGameObject*: Gestuurd als een speler een nieuw object aanmaakt in zijn wereld. Het packet bevat het ID, type en de initiele positie/orientatie van het object.

```
uint16_t objectNumericId
string objectId
string objectType
float32_t posX
float32_t posY
float32_t posZ
float32_t orX
float32_t orY
float32_t orZ
```

- *Selection*: Gestuurd als een speler een object heeft geselecteerd. Dit is niet echt een noodzakelijk onderdeel van het spel maar geeft een interessante extra dataflow die hoger in frequentie is dan de andere TCP flows. Voorlopig stuurt hij een apart packet per object, waardoor aggregatie een zeer goede optimalisatie zou zijn als er meerdere objecten tegelijk worden geselecteerd, wat vaak voorkomt in een RTS.

```
uint16_t objectId
```

- *OwnershipChanged*: Dit packet wordt voornamelijk gebruikt in een context van items die spelers kunnen oprapen. Een item is van geen speler als het eerst in het spel komt. Een speler kan het item dan met een avatar oprapen waardoor het zijn eigendom wordt. Als de avatar die het item heeft opgeraapt sterft komt het item terug in de wereld terecht en kunnen andere spelers het oprapen.

```
uint16_t itemObjectId
uint16_t avatarObjectId
uint16_t newOwnerId
```

- *ZoneChangedPacket*: Gestuurd als een speler verandert van zone. Dit is voor de zoning-optimalisatie, die later besproken wordt.

```
uint16_t playerId
int zoneX
int zoneZ
```

- *ZoningPacket*: Dit packet kan in 2 modussen gebruikt worden : GET en SET. De GET modus is om requests te sturen voor zoning informatie. Een SET bericht is het antwoord op een GET packet.

  Het GET packet definieert over welke zones het informatie opvraagt. Het SET packet bevat informatie over de gevraagde zone en eventueel omliggende zones. Deze informatie bestaat uit de playerIds van alle spelers die zich in deze zone bevinden. Deze informatie is nodig in de zoning-optimalisatie die later verder besproken wordt.

```
GET PACKET :
int zoneX
int zoneZ
```

```
SET PACKET : (meerdere entries)
int zoneX
int zoneZ
uint32_t playerCount
uint16_t playerId
...
uint16_t playerId
```

Bespreking van de packet opbouw

Een grote optimalisatie die aanwezig is in de packet-opbouw zijn de numeric ids voor de verschillende objecten. De engine werkt intern met string ids waarin ook het type van het

object zit verwerkt. Deze zijn typisch van de vorm "type_xxx". De lengte van deze strings hangt dus af van het type object. In onze eerste versie stuurden we de string ids mee in elk PositionPacket. Dit zorgde niet enkel voor grote packets (want de string ids zijn vaak redelijk lang), het was ook onmogelijk te voorspellen hoe groot een PositionPacket precies zou zijn.

Door een mapping te maken van elke string id op een numerieke id van een bepaald aantal bits bekomen we een veel kleinere packetsize die bovendien altijd constant is. Er wordt een klein beetje overhead gecreerd doordat we de mapping tussen numeriek en string moeten doorsturen in andere packets (fullState en newGameObject), maar deze packets worden veel minder frequent gestuurd dan de positionPackets, waardoor de werkelijke overhead zeer beperkt blijft. Geavanceerdere technieken voor GUID-generatie en onderhoud kunnen gebruikt worden om deze manier van object ids toekennen veiliger en zelfs zonder mapping te doen. In onze implementatie kozen we voor een ad-hoc generatie omdat de nadruk van het project hier niet op lag.

Er zijn nog een aantal andere optimalisaties die mogelijk zijn op het packet-niveau:

Zo schrijft de bistream voor elke string eerst de lengte en dan pas de string, wat zorgt voor 32 bits overhead. Als we gebruik maken van een 0-terminated string vraagt dit slechts 8 bits overhead. Dit zou echter welk zorgen voor processing overhead omdat we eerst moeten zoeken naar de 0 op het einde van de string.

De velden rotX en posY zijn mogelijk niet (altijd) nodig als we posities doorsturen omdat deze ofwel niet worden gebruikt, ofwel worden bepaald door de physics-simulatie, die toch bij elke client/peer min of meer hetzelfde zijn. Deze optimalisaties zijn echter packet-afhankelijk en dit zijn al zeer specifieke optimalisaties die veel tests nodig hebben of ze nuttig zijn en wel kunnen doorgevoerd worden.

Grotere optimalisaties zijn mogelijk als we op een hoger niveau nadenken over het netwerkverkeer. Deze worden in een volgend hoofdstuk verder besproken.

PROTOCOLLEN

Deze sectie bevat uitleg over de verschillende protocollen gebruikt in het project. Deze uitleg wordt in volgende hoofdstukken grotendeels herhaald bij de traces die over dat stuk van het protocol gaan. Het is dus mogelijk enkel volgende hoofdstukken te lezen zonder dit hoofdstuk. Andersom kan men dit hoofdstuk lezen om enkel een idee te krijgen van de gebruikte protocollen zonder de bespreking van de traces. De besproken protocollen zijn soms niet helemaal geimplementeerd of slechts voor bepaalde onderdelen. We bespreken hier echter hoe we het conceptueel willen aanpakken voor het framework, ookal zijn sommige dingen niet helemaal afgeraakt.

Network setup

Wanneer een nieuwe speler het spel joint gaat hij eerst een UDP en TCP connectie maken op een gekend IP. Dit is ofwel het IP van de server of van 1 van de peers. Voor UDP maken we gebruik van een eigen handshake-procedure om er zeker van te zijn dat het UDP verkeer mogelijk zal zijn.

Eens deze 2 connecties klaar zijn stuurt de nieuwe speler een HelloPacket naar het gekende IP. Dit bevat zijn spelerinformatie (id, naam, kleur etc.) en IP + listen TCP en UDP port. Op basis van deze gegevens voegt de server of peer de nieuwe speler toe aan de spelerlijst en stuurt deze geupdate spelerlijst naar alle spelers (inclusief de nieuwe speler). Nu is de nieuwe speler "officieel" deel van het spel geworden en iedereen weet dat hij er is.

Vervolgens moet de state van de nieuwe speler worden ingeladen. Aangezien we werken met meerdere objecten per speler houden we deze state bij in een SQLite databank. Bij CS houdt de server al deze state bij, bij P2P houdt elke peer zijn eigen staat en de staat van de gekende "wereldobjecten" bij. Na het versturen van de spelerlijst verstuurt de server een FullStatePacket met de staat van de nieuwe speler naar elke speler zodat iedereen zijn objecten kent. Vervolgens stuurt hij de staat van alle andere spelers naar de nieuwe speler, zodat deze ook alle objecten van de rest kent.

Voor P2P is het iets ingewikkelder. Hier is er immers niet 1 entiteit die alle state bijhoudt. De peerst wisselen elkaars state uit van zodra ze een connectie hebben. De nieuwe speler krijgt de Playerlist van het gekende IP en kan dan met de andere spelers connecties beginnen maken. Telkens zo'n connectie is opgezet wisselen de 2 peers hun FullState uit. Eens de state van alle spelers gekend is, kan de nieuwe speler deelnemen aan het spel.

Een belangrijke impact hierbij is het gebruik van numerieke ID's voor objecten. Intern gebruikt het framework stringids die ook het type van het object bevatten. Deze kunnen echter redelijk lang worden en zijn dus ongeschikt om te versturen in bijv. een frequent gestuurd PositionPacket. Daarom maken we een mapping van een 16-bits integer die in combinatie met de 16-bit ownerID een numerieke ID geeft aan elk object. Deze mapping moet natuurlijk ergens bekend worden gemaakt aan de andere spelers. Dit gebeurt in het FullStatePacket. Dit packet wordt hierdoor wel wat groter, maar uiteindelijk besparen we hiermee heel wat bits in de veel frequenter gestuurde PositionPackets. Voor deze numerieke IDs te berekenen checken we niet expliciet op collisions (wat op zich ook niet nodig is omdat elke speler een unieke ID heeft) waardoor dit ook geen overhead met zich meebrengt.

De zoning optimalisatie heeft enkele interessante invloeden op het joinproces. Zoals het protocol hierboven beschreven staat, is het natuurlijk weinig schaalbaar omdat alle informatie naar alle spelers wordt gestuurd en elke peer naar alle andere peers een connectie moet openhouden. Als we met zoning werken kunnen we enkel de benodigde spelers en state doorsturen bij het joinen van de nieuwe speler en later als hij rondbeweegt dit proces herhalen voor nieuwe zones die de speler ontdekt. Deze optimalisatie is onontbeerlijk bij het ontwerp van een echte NVE.

Tijdens het spel

Tijdens het spel is het bewegen van een avatar de belangrijkste actie die ondernomen kan worden. In ons spel zijn er 2 mogelijkheden om dit te doen. Vooreerst is er de RTS modus waarin meerdere objecten tegelijk kunnen worden aangestuurd. Dit gebeurt door objecten te selecteren en op een punt in de wereld te klikken, waarna ze ernaartoe bewegen in een rechte lijn. Vervolgens is er de FPS modus waarbij de speler meer directe controle heeft over de bewegingen van de avatar waar hij op dat moment ”in zit”.

In de basisversie van het spel was er weinig verschil tussen deze 2 modussen. De posities van de objecten werden gestuurd als ze bewogen en dit zoveel mogelijk (framerate). In de laatste versie maken we een duidelijk onderscheid tussen RTS en FPS. We gaan ervanuit dat in RTS enkel het doelpunt moet worden doorgestuurd via het netwerk, omdat alle objecten toch in een rechte lijn naar dit punt zullen bewegen bij alle spelers. Met een simpele lag-compensation door interpolatie over deze lijn zullen de posities weinig afwijken van de echte posities. Het enige waarmee rekening moet worden gehouden is als de objecten onderweg collisions ondervinden waardoor ze van hun pad kunnen afwijken. Wanneer die collisions optreden kunnen we even overschakelen naar veel frequentere position updates voor de objecten tot de collisions opgelost zijn, waarna ze terug kunnen bewegen naar het doelpunt. Er kan ook geopteerd worden om periodiek toch nog een full position te sturen ookal zijn er geen collisions, om de consistentie zeker te garanderen als het een lang pad is naar het doelpunt.

In de FPS modus worden er wel veel positieupdates gestuurd. Dit is omdat we hiervoor het pad van het object veel minder goed kunnen voorspellen dan in RTS modus omdat de speler het object rechtstreeks bestuurt. Ook hier hebben we echter enkele optimalisaties die gebruikt kunnen worden. Ten eerste is er Dead Reckoning. Afhankelijk van de snelheid en richting van beweging kunnen we voorspellen waar de speler over enkele ogenblikken zal zijn. Zolang deze voorspelling blijft kloppen moeten er geen extra packets worden gestuurd. Dit is bijvoorbeeld zo als de speler in een rechte lijn blijft lopen. Pas als de fout met de voorspelde positie te groot wordt zal er een correctie worden gestuurd. Een tweede optimalisatie zijn incrementele updates. Hierbij sturen we slechts af en toe een volledig positiepacket (met wereldcoordinaten voor de positie) maar meestal worden er slechts offsets gestuurd tov de laatste gekende wereldpositie. Deze optimalisatie werkt het best als de speler in een relatief klein gebied blijft rondlopen.

Het is interessant op te merken dat deze twee optimalisaties elkaar aanvullen. DR is goed als de speler op 1 lijn blijft lopen, maar minder goed als hij veel afwijkt van die lijn. Incrementele Updates hebben weinig probleem met het afwijken van deze lijn maar vereisen zwaardere updates als de speler ver over 1 lijn blijft lopen. Als de speler dus in 1 gebied blijft, maar onregelmatig beweegt, moet DR veel updates sturen. Deze updates kunnen echter klein blijven omdat het referentiepunt voor incrementele updates nog dichtbij is. Wanneer DR wel goed zijn werk doet zorgen de full updates als een extra consistency check, waarbij de tussenliggende offset-updates worden weggelaten.

Voor het afhandelen van de collisions hebben we gekozen voor een authoritive aanpak. Hiermee bedoelen we dat 1 entiteit de beslissing neemt over de uitkomst van de collisions en ze daarna kenbaar maakt aan de andere partijen. Voor CS is dit de server, voor P2P maken we gebruik van een soort ”authoritive peer”. Voor P2P is dit zo nauwgebonden aan zoning. Voor elke zone hebben we immers ook een authoritive peer die verantwoordelijk is voor de zone en die het aanspreekpunt is voor objecten die een zone binnenkomen. Het kiezen van de authoritive peer voor P2P kan op een aantal verschillende manieren, bijv. door elections (op basis van beschikbare bandbreedte, processing power, ...), hardcoded (onze huidige oplossing) of random. Dezelfde peer is dan ook verantwoordelijk voor de collisions binnen deze zone. Hierdoor voorkomen we dat er long-term inconsistencies optreden (hoewel short term nog wel mogelijk is onder invloed van delay etc.). Deze aanpak zorgt er voor dat er geen extra informatie moet uitgewisseld worden voor de collisions. Elke speler stuurt gewoon de updates van zijn objecten door. Op de authoritive entiteit draait dan een physics-simulatie die het resultaat berekent en eventuele correcties van de posities en uitkomsten kan verspreiden. In het geval van wereldobjecten (geen speler is er eigenaar van) mag enkel de authoritive entiteit hiervoor updates verzenden, die dan toegepast worden bij alle spelers waarvoor ze van belang zijn. Eenzelfde aanpak kan genomen worden voor objecten die de spelers kunnen opnemen in de wereld, hoewel we dit niet helemaal geimplementeerd hebben (een andere mogelijkheid voor het oprapen van items is te werken met een locking mechanisme vooraleer de actie mag doorgaan. Dit maakt de interactie op een meer expliciete manier duidelijk (en vraagt ook meer packets) terwijl de authoritive aanpak het meer op een impliciete manier doet).

Voor zoning zijn er nog wat extra voorzieningen nodig. Daar moeten de updates immers enkel gestuurd worden naar de huidige zone (en eventueel de omliggende), maar er is natuurlijk wat meer verkeer nodig als objecten van zone veranderen. Voor CS kan dit impliciet gebeuren, waarbij enkel de server weet heeft van de zones en de clients niet. Voor P2P is het moeilijker. Dit lossen we zoals reeds vermeld op met een authoritive peer per zone toe te kennen die als aanspreekpunt voor nieuwe speler dient. Als een speler verandert van zone vraagt hij de authoritive peer van deze zone op (bijv. hardcoded) en vraagt hij deze peer voor de state informatie van de nieuwe zone. De authoritive peer laat dan ook aan de andere spelers in zijn zone weten dat er een nieuw object is bijgekomen. Dit vraagt zoals eerder gezegd wat meer verkeer als men van zone verandert, maar nog altijd veel minder dan wanneer deze veranderingen naar alle spelers in de wereld zouden moeten gestuurd worden.

Merk op dat deze authoritive peer niet echt als een server fungeert. De peers sturen nog altijd naar alle peers hun updates, niet enkel naar de authoritive peer. Deze kan echter extra updates uitsturen als dit nodig is, zoals bij collisions. Het is dus niet zo dat de peers in 1 zone enkel communiceren

met de authoritive peer, die dan alles zou doorsturen naar de anderen (waardoor we meer een soort van zone-servers zouden krijgen waarop telkens 1 speler speelt, maar dit is hier niet de bedoeling, hoewel het ook een interessante hybride architectuur zou kunnen zijn). Als laatste moeten we opmerken dat onze aanpak natuurlijk iets gevoeliger is voor cheating, maar dit is hoe dan ook een groot probleem in P2P systemen.

ANALYSE BASISVERKEER

Deze traces zijn uitgevoerd met onze eigen traffic-calculator in het framework. Bijgevolg zijn er bij deze traces geen transport protocol headers opgenomen in de resulaten omdat we deze niet konden meten. In de volgende sectie hebben we een afzonderlijke vergelijking gedaan van onze traces met enkele wireshark captures die aangeven dat onze eigen calculator goed werkt en een goed beeld geeft van de werkelijke trafiek.

De calculator berekent de bandwidth in ticks van 100ms. De x-as van de grafieken is in seconden, de y-as in Bytes.

Voor deze traces maken we onderscheid tussen CS en P2P enerzijds, en netwerk setup/join vs. spel spelen anderzijds. Geen enkele van de later besproken optimalisaties zijn actief in deze simulaties.

Network setup

Deze simulatie start automatisch enkele processen en meet de informatie die ze uitwisselen om goed en wel het spel te joinen.

Wanneer een speler wil joinen maakt hij eerst een connectie naar de server of een andere peer en stuurt een HelloPacket. Als antwoord krijgt hij eerst een PlayerListPacket binnen met daarin alle gekende spelers, en daarna 1 of meerdere FullStatePacket's.

Voor P2P krijgt de nieuwe speler deze FullState van de andere spelers als hij er een connectie naar legt, in CS krijgt hij alle FullState van de server.

CS met 2 clients

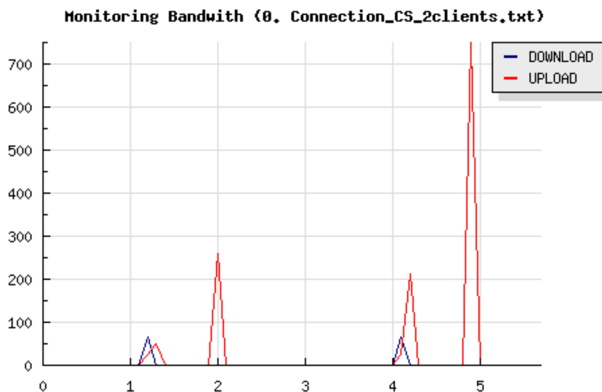

Figure 2. Server

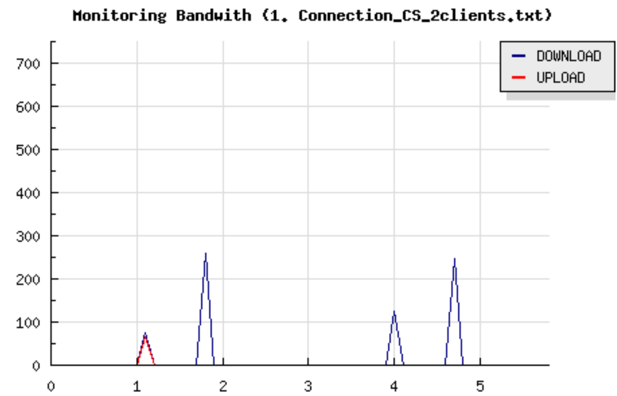

Figure 3. Client 1

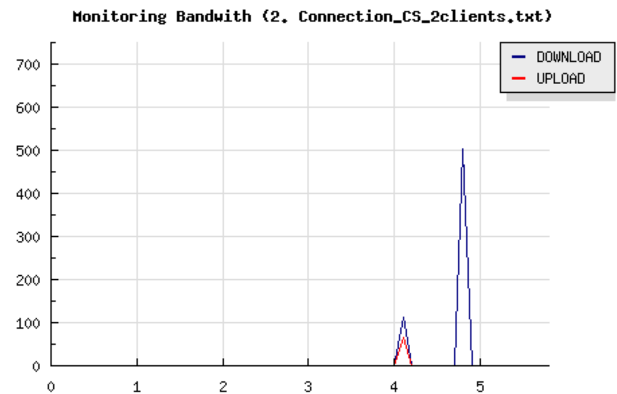

Figure 4. Client 2

De grafieken laten het protocol mooi zien. Client1 joint eerst en krijgt een PlayerListPacket van de server na zijn HelloPacket, en daarna een FullStatePacket met zijn objecten. Wanneer Client2 even later joint krijgt hij een PlayerList van de server die ook een kopie stuurt naar Client1. Daarna stuurt de server FullStatePackets voor de objecten van Client2 naar beide spelers, en een FullStatePacket met de objecten van Client1 naar Client2.

De traffiek op de server is precies de som van de trafiek op de 2 Clients, wat de bedoeling is.

P2P met 2 peers

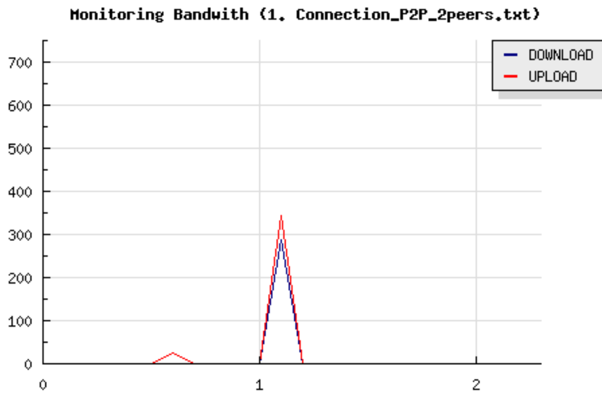

Figure 5. Peer 1

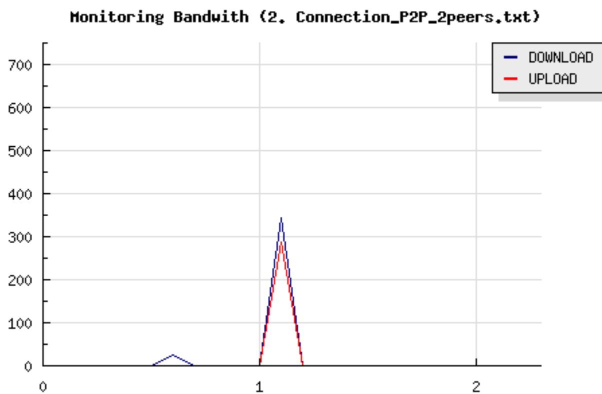

Figure 6. Peer 2

Ook voor P2P is het protocol duidelijk zichtbaar. Eerst stuurt Peer1 een PlayerListPacket naar Peer2, waarna ze FullState uitwisselen. Peer1 heeft 1 object meer waardoor zijn upload iets hoger is dan zijn download, en andersom bij Peer2.

CS met 4 clients

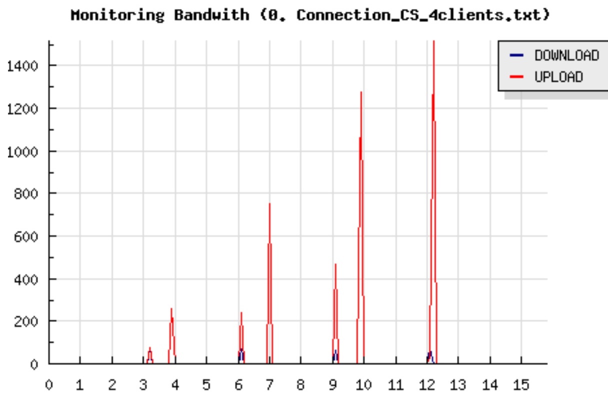

Figure 7. Server

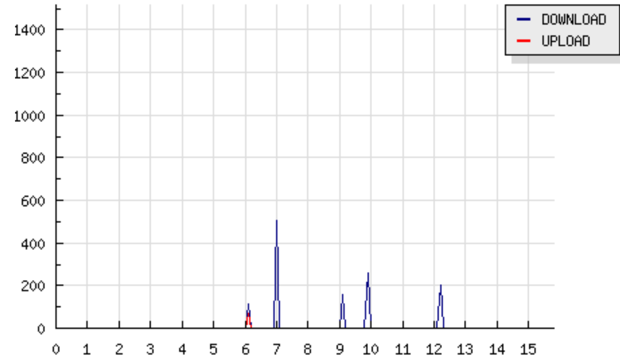

Figure 8. Client 2

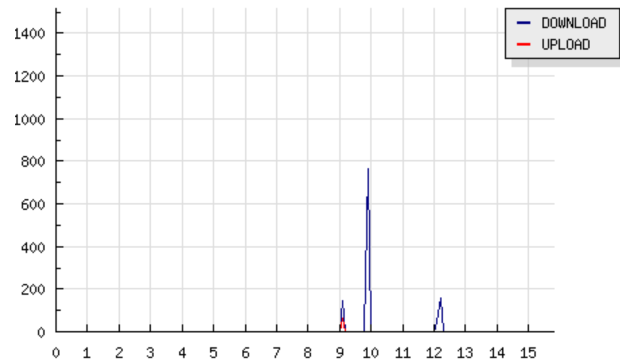

Figure 9. Client 3

Een setup met 4 clients toont dat ook hier alles verloopt als verwacht. Het verkeer dat de server moet uitsturen stijgt lineair met het aantal spelers en de download van de spelers stijgt gelijk mee. Zo moet client 2 telkens maar FullState van 1 andere speler tegelijk binnenhalen, terwijl Peer3 er bij zijn join direct 2 te verwerken krijgt (van speler 1 en speler 2) en zijn download van de fullstate is dan ook groot. De upload van de spelers wordt niet beïnvloed door het aantal spelers.

Opmerking : Speler 4 heeft geen opgeslagen objecten dus er moet ook geen FullState gestuurd worden. De upload van de Server stijgt omdat hij meer PlayerListPackets moet sturen naar de spelers, niet omdat hij nog eens FullState moet sturen (de stijging is dan ook veel minder als bij join van client 2 of 3).

P2P met 4 clients

In deze setup wordt Peer 1 altlijd gebruikt als peer waarop de anderen joinen. Dit betekent dat hij altijd naar iedereen PlayerLists gaat moeten sturen, wat zijn lineaire stijging van upload verklaart. Voor Peer 3 is de download en upload 2x zo hoog omdat hij FullState moet sturen en krijgen van zowel Peer1 als Peer2.

Ook hier heeft speler 4 geen objecten en moet dus geen full-

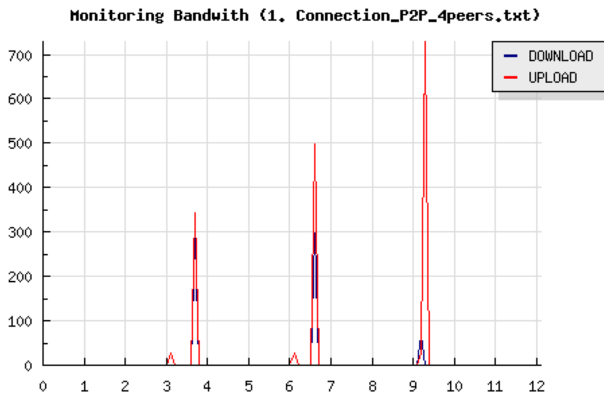

Figure 10. Peer 1

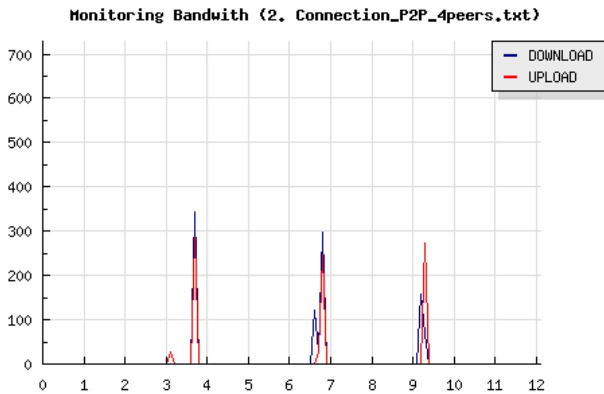

Figure 11. Peer 2

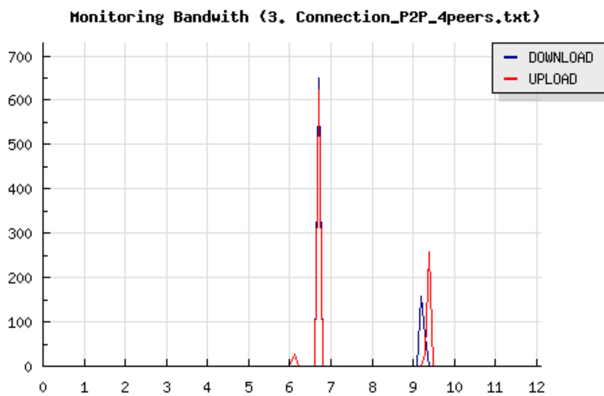

Figure 12. Peer 3

state sturen, enkel ontvangen van 3 peers. Hierdoor ligt de download bij de 3 anderen laag als Peer 4 joint, ze krijgen enkel de nieuwe playerlist binnen.

Moving objects

Bij deze simulaties werden er telkens enkele objecten bewogen bij 1 of meerdere spelers tegelijk. In ons framework hebben we de mogelijkheid ingebouwd om simulaties automatisch te kunnen starten en stoppen en zodoende telkens opnieuw exact dezelfde omstandigheden voor de traces te genereren.

3 bewegende objecten in CS

In deze simulatie beweegt speler 2 een object, waarna speler 1 ook begint te bewegen. Op het einde begint speler 2 te bewegen met een extra object.

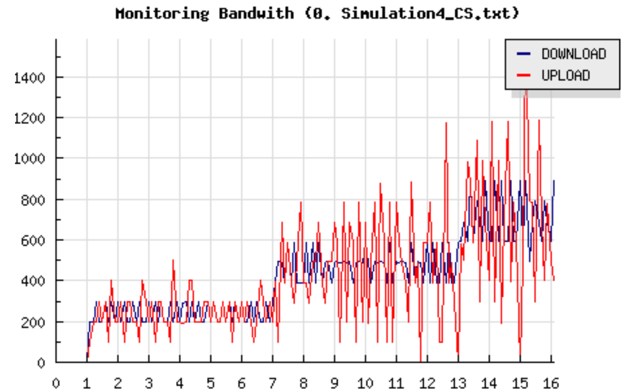

Figure 13. Server

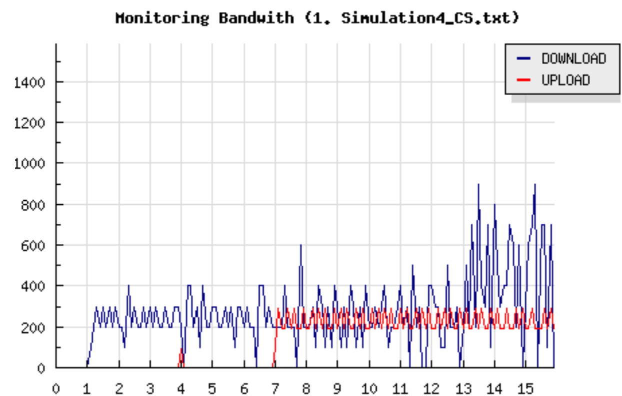

Figure 14. Client 1

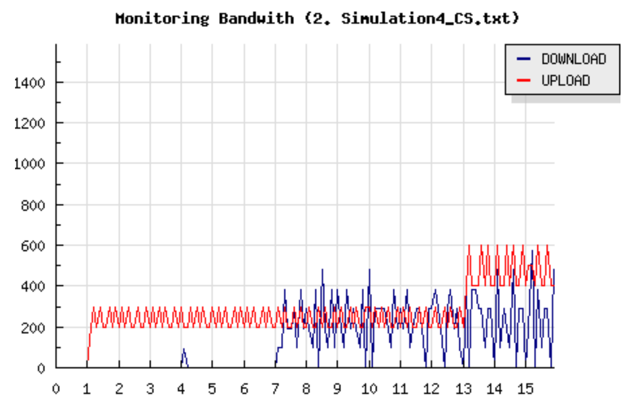

Figure 15. Client 2

In deze traces zien we zeer duidelijk dat we geen vast of constant bandwidthverbruik hebben. De grafieken schommelen

vaak tussen redelijk uiteenliggende waarden. Dit komt omdat we geen vaste framerate aanhouden binnen het framework en het netwerkverkeer gegenereerd wordt op basis van frameevents. Wanneer de framerate hoog ligt zal er met andere woorden meer worden verstuurd dan wanneer de framerate laag is. Hierbij komt dat we slechts om de 100ms een opname nemen van het verkeer.

Een mooi voorbeeld hier is het begin van de grafiek van client 1. Deze upload schommelt tussen 200 en 300 bytes per 100ms. Als we weten dat 1 PositionPacket ongeveer 100 bytes groot is, wil dit zeggen dat we soms 2 en soms 3 dergelijke packets verstuurd hebben tijdens de laatste 100ms. Dit komt omdat de framerate schommelt tussen 20 en 30 fps en als we dan om de 100ms een capture nemen, kan het zijn dat we juist wel of juist niet een packet meenemen in de berekeningen. Hierdoor komt het dat deze grafiek schommelt tussen 2 "grenzen".

Om aan te tonen dat deze schommelingen wel degelijk komen door een wisselende framerate volgen hier nog 2 grafieken, de eerste opgenomen door ons eigen framework, de tweede door wireshark. Ook wireshark toont hetzelfde schommelende gedrag. Bij deze simulatie beweegt er slechts 1 object op client 2. Opmerking : op wireshark worden de UDP/IP/Ethernet headers ook meegeteld dus de schaal is iets anders.

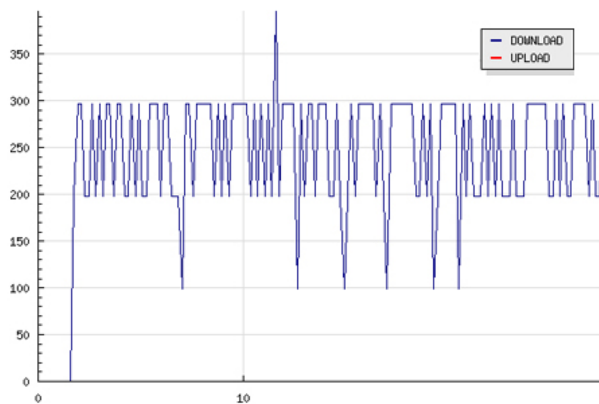

Figure 16. Client 1, eigen capture

De traces voor 3 bewegende objecten geven mooi het verwachtte gedrag. Zo zien we het serververkeer gradueel stijgen naarmate er meer bewegende objecten bijkomen. De download bij de verschillende clients weerspiegelt het uploadgedrag van de andere client. Wanneer client 2 een 2de object laat bewegen zien we zijn upload verdubbelen. Hierdoor verdubbelt ook de download bij client1.

3 bewegende objecten in P2P

Bij deze simulatie met 2 peers zien we hetzelfde gedrag als bij de clients in de CS opstelling. Download en upload zijn min of meer tegenovergestelde van elkaar op de 2 verschillende peers, wat eens te meer volledig binnen de verwachting valt.

De hevige schommelingen kunnen hier verklaard worden

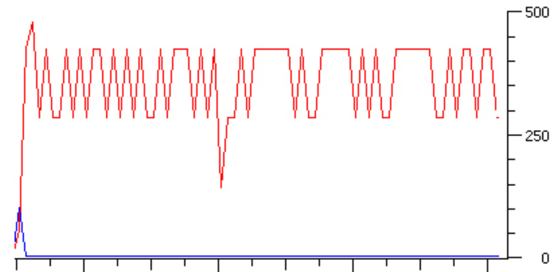

Figure 17. Client 2, wireshark capture

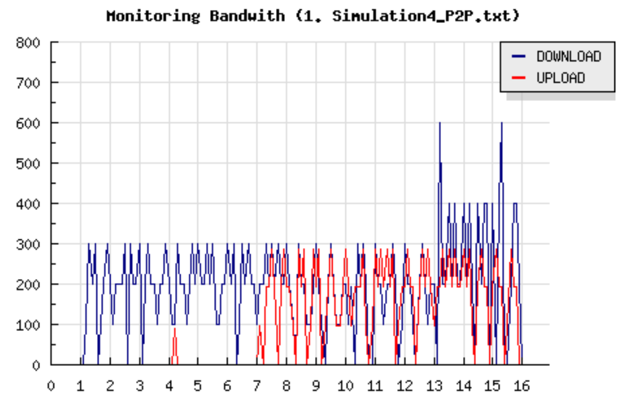

Figure 18. Peer 1

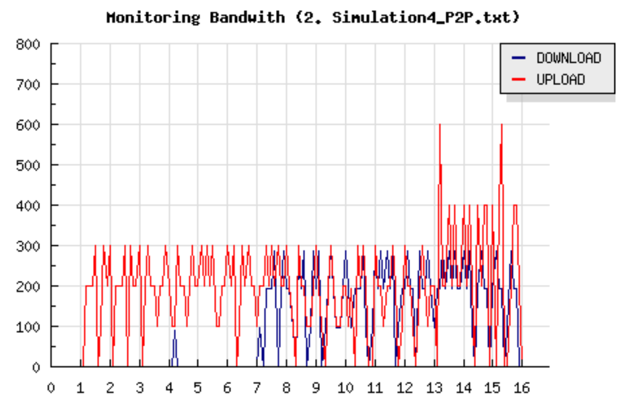

Figure 19. Peer 2

door grotere verschillen in de framerate tijdens het uitvoeren van de simulatie.

Nieuwe speler in CS

Bij deze simulatie kijken we wat er gebeurt als er een nieuwe speler bijkomt (client3) en later verdwijnt terwijl een andere speler (client2) aan het bewegen is.

Zoals we zien heeft vooral de server "last" van het joinen van

een nieuwe speler. Client2 moet zijn upload niet verhogen om te blijven rondlopen, maar de server moet deze data wel naar een extra speler doorsturen, waardoor zijn upload verhoogt. De 2 bestaande clients merken weinig van het joinen van client3, buiten dan de hoge download-piek bij het joinen van de playerlist en fullstate van de nieuwe speler.

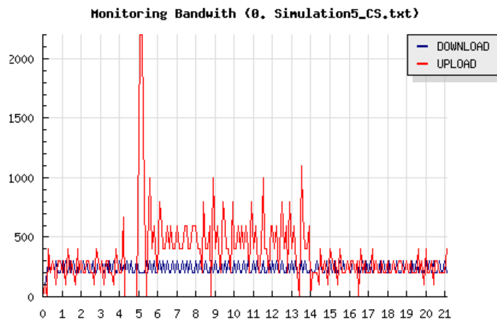

Figure 20. Server

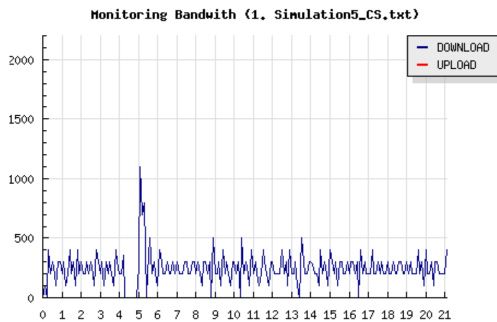

Figure 21. Client 1

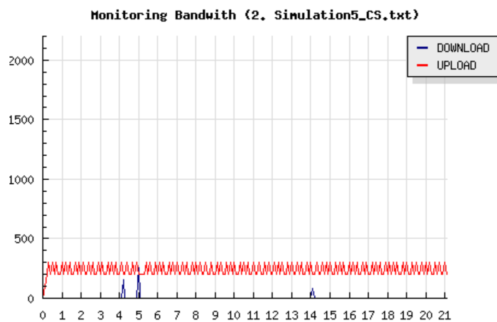

Figure 22. Client 2

Nieuwe speler in P2P

We merken hier meteen een groot verschil met de CS setup. Waar daar client2 geen verhoging van zijn upload zag gebeuren wanneer client3 erbij kwam, is dit hier heel duidelijk wel het geval. Peer2 moet immers zelf zijn updates ook versturen aan peer3, waardoor de upload effectief verdubbeld. Wanneer peer3 het spel verlaat, daalt de upload van peer2 terug.

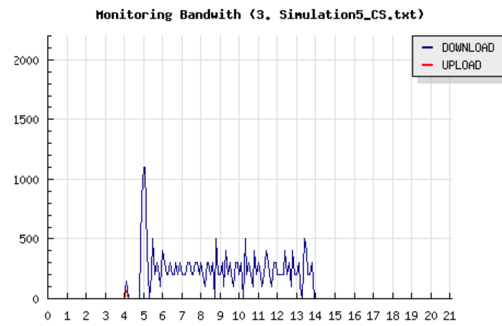

Figure 23. Client 3

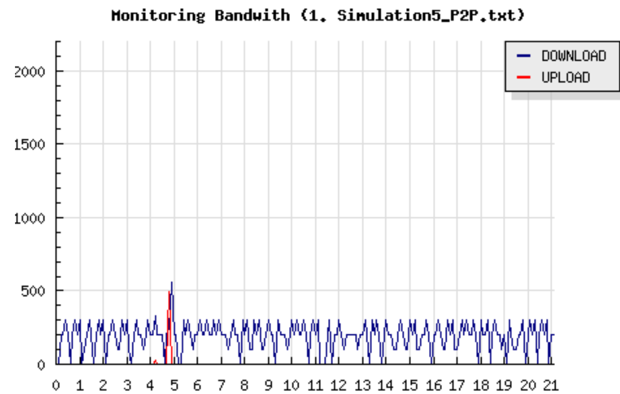

Figure 24. Peer 1

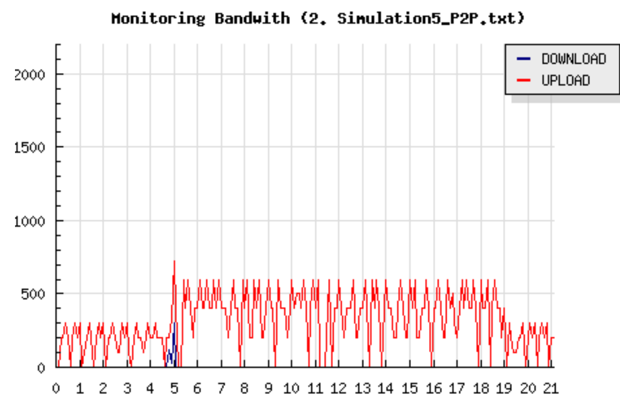

Figure 25. Peer 2

CONCLUSIE VERGELIJKING CS/P2P

Uit de verschillende traces blijkt vooreerst dat het framework zich gedraagt zoals verwacht wordt in de gestelde simulaties.

Voorts kunnen we de observatie maken dat de server het meeste werk op zich neemt in een CS-architectuur. Naarmate het aantal spelers stijgt, stijgt ook het verkeer van en naar de server. De andere clients zien hun verkeer veel minder schommelen onder invloed van andere spelers.

Bij P2P is het werk meer verspreid over de verschillende

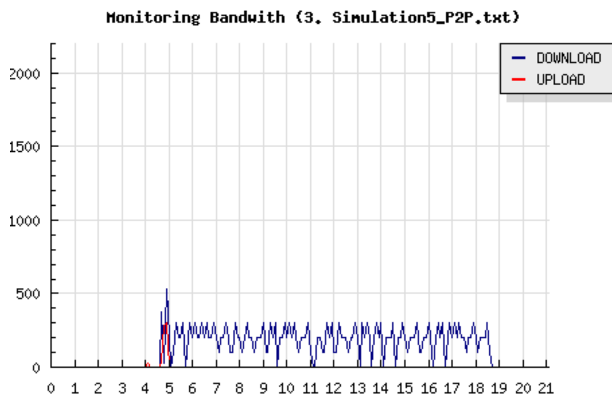

Figure 26. Peer 3

peers. Afhankelijk van wat een peer nu precies aan het doen is of welke rol hij op zich neemt zal ook zijn verkeer stijgen. Het verkeer per peer neemt nooit zulke proporties aan als dat van een server, maar blijft ook zelden zo laag als dat van een client. De resources worden dus beter verdeeld in de P2P opstelling.

PROTOCOL OVERHEAD

Eerst een opmerking over de traces tot hiertoe: deze zijn allemaal genomen mbv ons eigen framework en bevatten bijgevolg geen informatie over de headeroverhead van oa. de transportprotocollen. We hebben echter ook enkele traces gedaan met wireshark, waarbij we hebben gekeken wat de werkelijke grootte was van de packets.

TCP

De meeste streams sturen hun packets over TCP. Deze packets worden echter infrequent gestuurd, waardoor de absolute overhead van de TCP header beperkt blijft. Enkele voorbeelden:

```
Hello : 41 bytes payload, 95 totaal
PlayerList : 85 - 139
FullStatePacket : 246 - 300
NewGameObject : 93 - 147
```

Het verschil tussen de echte payload en de totale grootte is telkens 54 bytes. Dit zijn 40 bytes voor TCP + IP headers en 14 voor de Ethernet headers. De grootte van de payloads is ook volledig binnen verwachting gezien de opstelling van elk packet zoals gezien eerder in dit verslag.

UDP

PositionStream stuurt zijn data via UDP en doet dit zeer frequent (bijna elke frameupdate). Voor een PositionPacket van ongeveer 100 bytes zorgt dit tegen een framerate van 30fps voor ongeveer 300 bytes verkeer per 100ms. Deze berekening wordt ondersteund door de traces.

```
Position : 95 bytes payload, 137 totaal
```

Hier is het verschil 42 bytes, waarvan er 28 voor UDP + IP headers zijn en 14 voor Ethernet. Dit zorgt voor een goede 70% aan payload in het totaalpakket. Dit aantal kan zeker nog verhoogd worden door aggregatie van verschillende PositiePackets in 1 groot packet, aangezien we nog ver onder de MTU size zitten.

Deze overhead is goed zichtbaar als we figuur 16 (eigen capture) vergelijken met figuur 17 (wireshark capture). Terwijl het verkeer zonder headers schommelt rond de 250 bytes per 100 ms, schommelt het in wireshark rond de 370 bytes per 100 ms. Het verschil van 120 bytes/100ms kan verklaard worden doordat er ongeveer 3 PositionPackets worden verstuurd per 100ms. Per PositionPacket is er 42 bytes overhead, dus 3 x 42 is ongeveer 120 bytes overhead.

OPTIMALISATIES

Wegens tijdgebrek voor de deadline voor dit verslag bespreek ik niet alle optimalisaties in-depth. Ik ga ervanuit dat de lezer het basisprincipe snapt en bespreek vooral hoe wij het in ons framework hebben gebruikt en hoe we het nut ervan aantonen met traces.

Dead Reckoning

Hoewel deze optimalisatie in principe vooral handig is voor het verbergen van latency en jitter voor bewegingen van objecten, kan het ook gebruikt worden als een zeer handige manier om bandwidth omlaag te halen. Door lokaal dezelfde predictions te doen als de remote medespeler en enkel updates te sturen als het lokale pad te fel afwijkt van het voorspelde pad, kunnen we heel wat minder netwerkverkeer bekomen als de speler in een min of meer rechte lijn loopt of weinig afwijkt van zijn pad.

In onze implementatie doen de peers aan locale predictie voor hun eigen objecten (en sturen dus enkel updates als dit nodig is) en aan remote predictie voor objecten van anderen (waarmee we bedoelen dat ze de posities van het object voorspellen en ook echt tonen op het scherm).

Voor Client/Server werkt het iets anders. Hier doen de Clients zelf geen lokale predictie. In de praktijk wil dit zeggen dat ze zoveel mogelijk positieupdates naar de server sturen, typisch 1 per frameupdate. De server gaat dan wel lokale predictie toepassen en de medespelers pas iets laten weten als het pad teveel is afgeweken. De clients passen wel remote predictie toe voor objecten die ze niet zelf controleren. Hierdoor wordt vooral het uploadverkeer van de server dramatisch verminderd.

Het is ook mogelijk om de clients zelf ook lokale predictie te laten toepassen en zo de download van de server te verminderen. We willen echter een zo goed mogelijke consistentie behouden en daarvoor is het belangrijk dat de server zoveel mogelijk informatie heeft voor de objecten. Hij kan dan zelf beslissen welke dingen met dead reckoning kunnen worden afgehandeld en welke niet, terwijl hij anders aangewezen is op de berekeningen van de spelers, die onder andere door factoren als delay fouten kunnen vertonen.

Om de consistency zo hoog mogelijk te houden moeten we rekening houden met delay en jitter voor de dead reckoning. Wanneer er een nieuw positiepacket binnenkomt is het originele object mogelijk al een heel stuk van deze positie bewogen. Hiervoor gebruiken we absolute timestamps die gesynchroniseerd zijn bij alle spelers. Bij elke positieupdate sturen we de tijd dat deze gezonden wordt mee in het packet. Wanneer het packet dan aankomt weten we precies hoe lang het onderweg geweest is door de receivetime op te vragen, en kunnen we het verschil compenseren. De timestamps zijn ook belangrijk om de snelheden van de objecten te berekenen. We sturen immers enkel positieupdates door naar medespelers, waaruit zij zelf de juiste snelheid van het object kunnen berekenen door de timestamps.

Voor het testen van DeadReckoning namen we een simulatie waarbij speler 2 rondloopt met een object. De grootteordes van het verkeer zijn niet altijd hetzelfde omdat de simulaties niet helemaal hetzelfde werden uitgevoerd. Het is echter op alle grafieken duidelijk zichtbaar dat de optimalisatie een serieuze vermindering van het verkeer teweegbrengt.

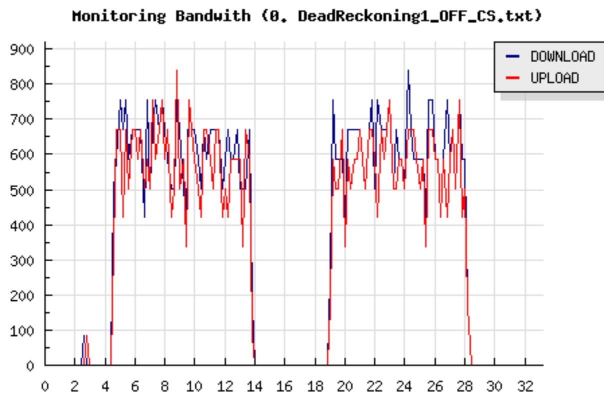

Figure 27. Server zonder DR

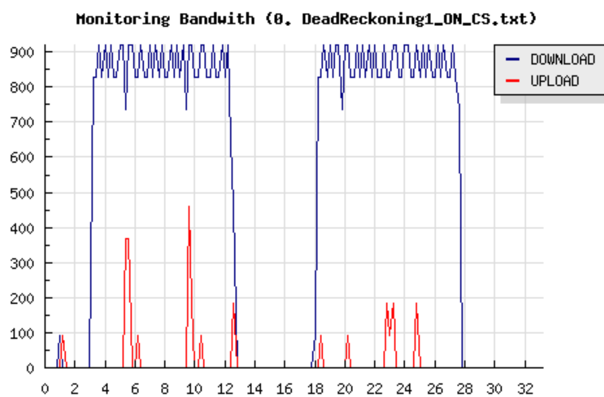

Figure 28. Server met DR

Herinner u dat enkel de server DR toepast. De download is dus even hoog als normaal, maar de upload naar de andere speler van de posities van de rondlopende speler is gevoelig verschillend eens DR opstaat en enkel updates stuurt als de error te hoog wordt.

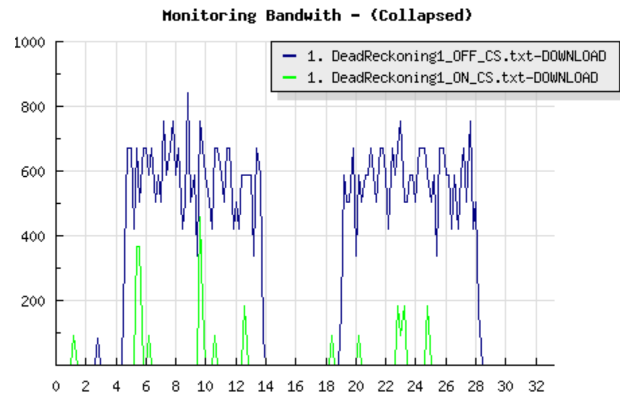

Figure 29. Client1 collapsed

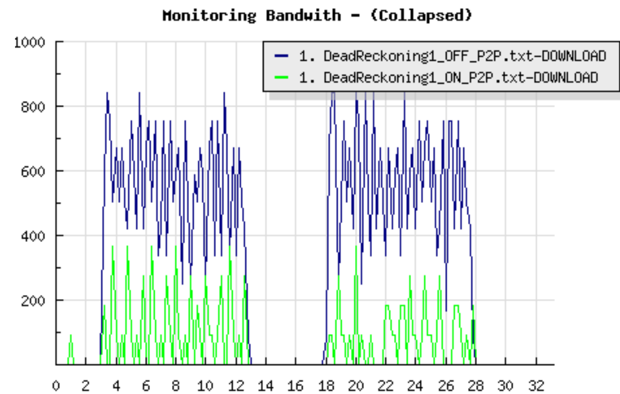

Figure 30. Peer1 collapsed

Deze 2 collapsed grafieken geven nog mooier de verschillen weer. Hier wordt het verkeer met DR en zonder DR in 1 grafiek getoond, met een andere kleur. Het is in beide gevallen duidelijk dat de groene grafiek (DR staat op) veel lager is dan de blauwe grafiek (DR staat af).

Actionbased events

Zoals reeds besproken bij de werking van de ActionStream kunnen we voor de RTS bewegingen gewoon de doelpositie sturen. Aangezien alle spelers hetzelfde pathfinding algoritme draaien, zullen de objecten overal op de juiste manier naar het punt bewegen. Hier moet dan enkel rekening worden gehouden met wat er gebeurt als er objecten met elkaar botsen tijdens het bewegen naar het punt.

Onze oplossing voor dit laatste probleem is om, van zodra er een botsing plaatsvind, volledige positieupdates te sturen voor de objecten in de botsing, alsof we ze in FPS modus aan het bewegen waren. Hierdoor zal de consistentie bij botsingen bewaard blijven en kunnen we genieten van de enorme optimalisatie van de het action-based doorsturen van de RTS doelposities.

Ook voor de actionbased events moeten we rekening houden met delay. Wanneer het nieuwe targetpunt aankomt bij een medespeler is het originele object immers al een heel stuk

ernaartoe bewogen. Hier kunnen we een gelijkaardig convergence algoritme gebruiken als bij Dead Reckoning dat de snelheid even wat hoger neemt tot het object terug meekan met het remote object.

Voor het testen van de actionbased events hebben we een simulatie waarbij een object 4x de opdracht krijgt naar een punt in de wereld te bewegen. Bij dat punt aangekomen stopt het object heel even alvorens naar het nieuwe punt te bewegen.

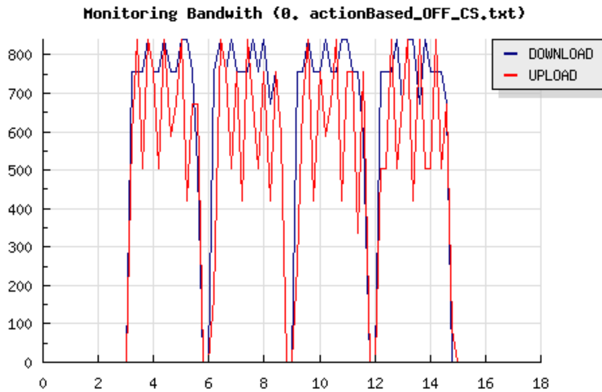

Figure 31. Server zonder Action Based

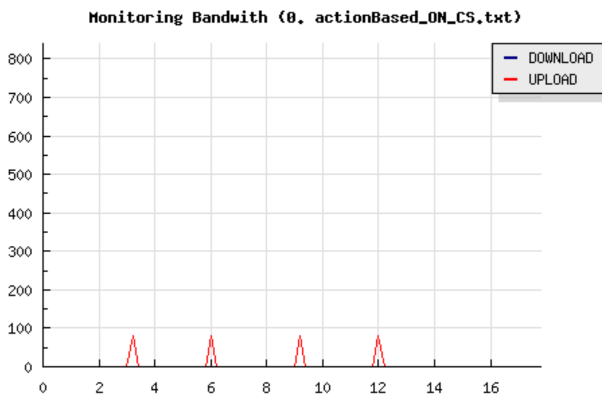

Figure 32. Server met Action Based

Het verschil is hier opmerkelijk. Zonder Action Based gaan de posities worden verstuurd als het object beweegt. We zien kleine rustpunten wanneer het object naar een nieuw punt overgaat. Als Action Based opstaat zijn deze rustpunten echter de enige moment waarop verkeer plaatsvind. Daar wordt er immers gestuurd dat het object een nieuw doelwit heeft gekregen om naartoe te bewegen. Deze traces zijn voor CS maar P2P geeft volledig analoge resultaten.

Incrementele updates

Iets wat opvalt bij de samenstelling van het PositionPacket is dat we voor elke coordinaat en orientatie een 32-bit float gebruiken. Deze 32 bits zijn zeker nodig als het spel zich afspeelt in een grote wereld en men zou zelfs kunnen opperen voor 64-bits preciesie voor objecten in gigantische werelden.

Het zou echter dom zijn om telkens de posities in wereldcoordinaten door te sturen als deze zo'n enorme bitsize innemen. Een slimmere manier van aanpakken is posities te sturen ten opzichte van een andere, gekende positie. Deze offsets zijn vaak heel klein en kunnen opgeslaan worden in bijv. 4 of 8 bit. Als we dan weten dat positiepackets heel vaak gestuurd worden is 4 bit vs 32 bit een hele winst.

De vraag is dan welke offsetpositie we kiezen. 1 mogelijkheid is om te kijken in welke zone (zie ook later) een object zich bevindt en de posities relatief ten opzichte van een hoekpunt van deze zone te versturen. Als alle medespelers dan dezelfde voorstelling van de zones hebben ontstaan er geen inconsistenties. Een andere mogelijkheid is om af en toe een volledige positieupdate te sturen met volledige precisie, en alle daarna komende updates relatief tov deze volledige positie te sturen. Deze keuze is veel onafhankelijker van bijv. zoning en heeft niet als eis dat alle spelers dezelfde wereldvoorstelling hebben (wat bijvoorbeeld handig is voor ons want de clients weten niks van de zones af).

We kozen voor de offset tov een volledige positieupdate. Deze volledige updates sturen we via TCP zodat ze zeker door iedereen ontvangen worden, waarna we de incrementele updates versturen via UDP. Als de verschillen te groot worden, maw ze passen niet meer in 4 of 8 bits, wordt er terug een volledige update gestuurd met volledige precisie, waarna de incrementele terug als kleine waardes kunnen beginnen.

Dit soort optimalisatie werkt goed als spelers in 1 gebied blijven, wat vaak het geval is in spellen. Ook wanneer ze rondlopen en er relatief veel volledige positieupdates moeten gestuurd worden, is dit nog altijd minder dan wanneer we altijd volledige precisie zouden moeten sturen. De TCP overhead neemt gedeeltelijk toe, maar de frequentie van de volledige updates is volgens ons laag genoeg om dit te kunnen rechtvaardigen.

Om deze optimalisatie te testen laten we 1 object rondlopen in de wereld. In de grafieken is duidelijk te zien dat de incrementele updates zorgen voor een pak minder verkeer. Merk op dat we wel vertelden dat we periodiek een fullstatepacket sturen, maar dat deze niet duidelijk te zien zijn in de traces. Dit komt omdat elk punt in de grafiek een optelling is van alle verkeer gedurende 200ms. In die tijd zijn er al heel wat packets verstuurd. Zonder incrementele updates hebben we op 200ms bijv. 20 packets van elk 100bytes. Met incrementele updates hebben we bijv. 1 packet van 100 bytes en 19 van 30 bytes. Door deze optelling is het dus perfect mogelijk dat de uitzonderlijke volledige updates niet duidelijk zichtbaar zijn in de grafieken.

Om deze toch duidelijk te maken deden we 2 captures met wireshark waarbij we het TCP verkeer (rood) en UDP verkeer (groen) apart tonen. In deze grafieken zien we duidelijk de uitschieters in TCP van de volledige updates en het constante UDP verkeer van de incrementele updates. We zien ook duidelijk dat wanneer een object in 1 gebied blijft rondlopen, er een tijd lang geen volledige updates en dus ook geen TCP verkeer nodig is.

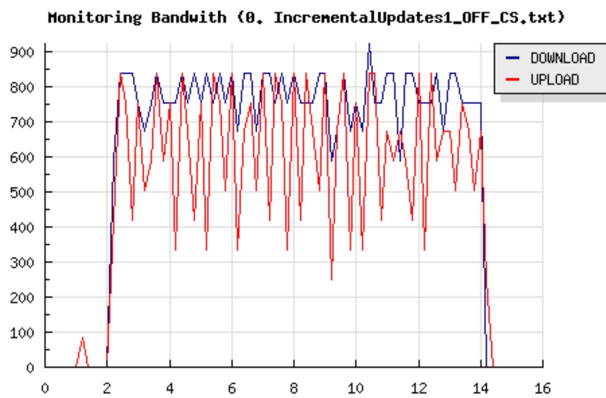

Figure 33. Server zonder Incremental Updates

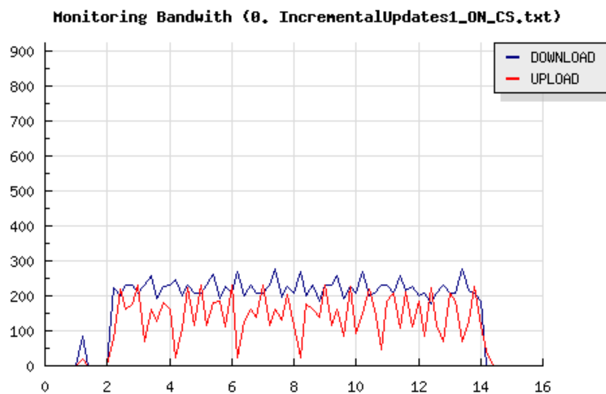

Figure 34. Server met Incremental Updates

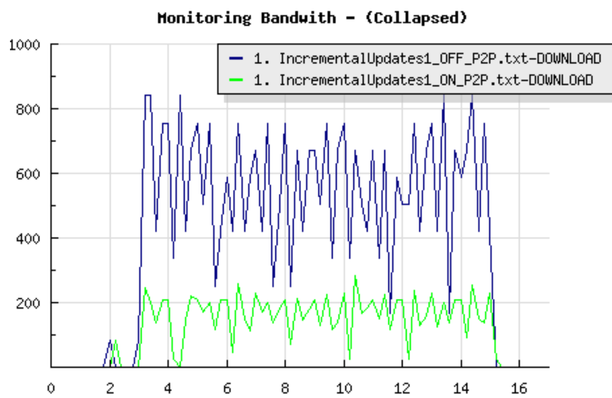

Figure 35. Peer1 incremental updates collapsed

Compressie

Hoewel we het vervangen van string-ids door numerieke ids en het invoeren van incrementele updates ook kunnen beschouwen als vormen van compressie, bestaan er ook meer algemeen inzetbare compressiemethodes. Wij kozen Run-length-encoding en huffman-encoding om onze packets nog eens extra mee te encoderen. We kwamen hier echter voor een raar resultaat te staan: de compressie geeft lang niet altijd een kleinere packetsize!

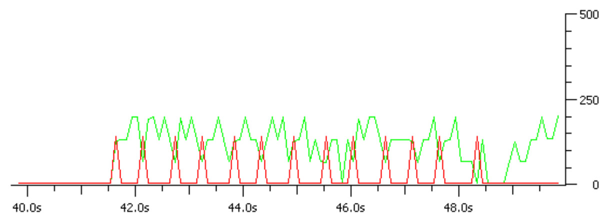

Figure 36. Wireshark capture : object loopt op rechte lijn

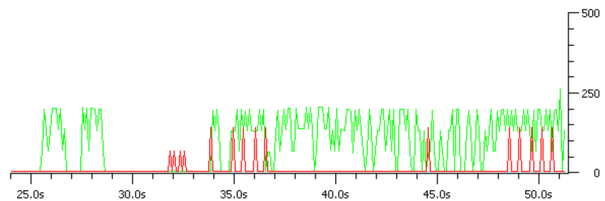

Figure 37. Wireshark capture : object blijft een tijdje in hetzelfde gebied

```
RLE:

identify: 13 -> 11
hello: 41 -> 53
playerlist: 85 -> 99
fullstate: 252 -> 219
selection: 11 -> 14
position: 84 -> 66
action: 79 -> 86
zonechange: 21 -> 20


RLE 10 * aggregatie:

identify: 130 -> 95
hello: 410 -> 188
playerlist: 850 -> 263
fullstate: 2520 -> 557
selection: 110 -> 123
position: 840 -> 254
action: 790 -> 254
zonechanged: 210 -> 138

RLE 10 verschillende: 750 -> 516


huffman:

identify: 13 -> 26
hello: 41 -> 99
playerlist: 85 -> 149
fullstate: 252 -> 263
selection: 11 -> 33
position: 84 -> 115
action: 79 -> 92
zonechanged: 21 -> 35
```

```
huffman 10 * aggregatie:

identify: 130 -> 58
hello: 410 -> 189
playerlist: 850 -> 301
fullstate: 2520 -> 657
selection: 110 -> 75
position: 840 -> 284
action: 790 -> 246
zonechanged: 210 -> 87

huffman 10 verschillende: 750 -> 795
```

Zoals we kunnen zien zijn de packetsizes soms veel groter dan de originele sizes, zeker bij huffman encoding. Dit kan verklaard worden doordat de gebruikte encoderingen vooral goed werken op bepaalde types data. Huffman zal bijvoorbeeld goed werken op data waar veel herhaling in zit, terwijl RLE goed werkt als er veel dezelfde getallen achter elkaar zitten.

Wanneer we echter kijken naar groepen van meerdere packets samen, zien we dat de compressiealgoritmes wel degelijk grote winsten kunnen boeken. Zoals verwacht doet huffman het een pak beter als we 10 identieke packets tesamen comprimeren. Dit geef aan dat de compressiealgoritmes wel nuttig zouden zijn als er aggregatie van packets wordt toegepast, wat in ons framework niet het geval is. Er moet wel nog altijd opgelet worden met de keuze van algoritme: op 10 verschillende packets gaf RLE een goed resultaat, maar huffman liet ons eens te meer in de steek en produceerde meer data dan er origineel aanwezig was. Uitgebreid testen in real-life omstandigheden is de boodschap voor dit soort generieke compressiealgoritmen.

Zoning

De wereld wordt opgedeeld worden in een aantal vierkante zones. De positie van elk object in de wereld bepaalt bij welke zone het object hoort. Elk object kan een eigenaar hebben en zo zal elke zone ook weten welke spelers actief zijn in die zone. Statusveranderingen van objecten worden nu enkel nog doorgestuurd naar andere spelers die ook actief zijn in dezelfde zone. Dit zorgt dat er zeer veel overbodige netwerkinformatie wordt weggesnoeid. In theorie kan deze filtering op eender welk packet worden toegepast. Voorlopig kiezen we er bewust voor om dit enkel op position packets te doen, omdat zoning op deze de meeste invoed heeft. In een uitgebreidere implementatie zouden alle events hiervan kunnen profiteren. Het beheer van de zoning informatie gebeurt verschillend voor een P2P- of CS-architectuur.

Zoning bij Client/Server

Binnen een C/S structuur zal enkel de server weet hebben van zoning. Alle clients sturen enkel de statusinformatie van al hun eigen objecten door naar de server. De server bepaalt vervolgens bij welke zone het object hoort en gaat reageren op eventuele zone-overgangen. De server bepaalt op basis van deze zoninginformatie welke actieve spelers aanwezig zijn in de desbetreffende zone en gaat enkel naar deze spelers de status van het object doorsturen. Inconsistenties kunnen zich voordoen aan de rand van twee zones en bij zone-overgangen. Als een speler naar een andere zone gaat moet hij de status te weten komen van die nieuwe zone. Er kunnen namelijk objecten zijn verplaatst waarvan de speler nog geen weet heeft. Doordat de server de staat weet van alle objecten binnen de nieuwe zone kan deze ook gemakkelijk de staat doorsturen naar de nieuwe speler. Omdat enkel nog maar positions worden verwerkt met zoning en niet de volledige state tranfer gebeurt, zal de nieuwe client niet weten of andere objecten zijn verdwenen uit de nieuwe zone. In onze implementatie lost de server deze inconsistentie op door bij een zone-overgang ook de andere clients te laten weten dat het object is veranderd van zone. Deze clients zullen het object in de nieuwe zone plaatsen waardoor hij niet op een andere plaats foutief blijft staan. Een verbetering van dit systeem is om een volledige state transfer te laten gebeuren wanneer een object naar een andere zone gaat zodat het weet welke objecten op die moment in de zone zitten. Deze full state transfer kan zelfs in stappen gebeuren waarbij de dichtsbijzijnde objecten eerst worden gestuurd en later pas de verderafgelegen objecten. De pseudocode voor zoning op CS vindt u in figuur 38.

```
Server
onZoneChanged(player, zone)
        foreach  ( object in zone )
                sendAll ('POSITION', object->position);
        end
end

onPositionChanged(object)
        zone = getZoneFromPosition(object->position);
        foreach ( player in zone->activePlayers )
                send('POSITION', player, object->position);
        end
end

Client
onPositionChanged(object)
        send(server, 'POSITION', object->position);
end
```

Figure 38. Pseudocode Zoning CS.

Zoning bij P2P

In tegenstelling tot de C/S structuur, waar de zoning centraal wordt beheerd, zal bij P2P elke zone een eigenaar toegekend krijgen. Deze eigenaar beheert al de objecten die in de zone zitten. Uiteindelijk is het de bedoeling dat de eigenaars met behulp van elections worden gekozen, maar om het makkelijk te houden, krijgt bij het opstarten elke zone hardcoded een eigenaar toegewezen. Lokaal heeft elke peer ook weet van de verschillende zones. Het is de bedoeling dat elke speler weet heeft van alle andere actieve spelers binnen de zones waar hij zelf actief is. Als hij zelf de eigenaar van de zone is, heeft hij direct de goede staat. Maar als de zone een andere eigenaar heeft, zal hij de state van de zone moeten opvragen. Ook hier is deze state voorlopig beperkt tot de positions van de objecten. Het doorsturen van de informatie gebeurt in een ZoningPacket. Dit packet zal alle actieve spelers bevatten alsook eventueel actieve spelers van de aangrenzende zones (instelbaar met een threshold). Net zoals bij CS kan deze implementatie inconsistenties bevatten. Om dit op te lossen zal de eigenaar van de zone bij een zone-overgang een zoneChangedEvent uitsturen naar alle peers in de NVE. Dit heeft als effect dat alle peers hun status van de objecten die hij zelf actief heeft in de zone nogmaals doorstuurt naar de anderen. Ook dit is weer een zeer onperformante oplossing omdat het niet schaalbaar is met meerdere peers. Dit kan

eens te meer makkelijk worden opgelost door een full state transfer wanneer er een zone-overgang zich voordoet. Hier kan de verantwoordelijke peer de nieuwe peer laten weten welke de aanwezige spelers zijn, waarna deze on request laten weten welke objecten ze in die zone hebben. Alle andere objecten die de nieuwe peer eventueel nog in de zone had staan die niet in deze lijsten zitten kan hij dan verwijderen, waardoor hij enkel de objecten die er echt in zitten overhoudt. De pseudo code voor zoning met P2P vindt u in figuur 39.

```
Peer
onZoneChanged(object, zone)
        requestZoningInformation(zone, zone->owner);
end

onZoneInformationReceived(zone, remoteZone)
        zone->setActivePlayesr(remoteZone->activePlayers);
end

onZoneChanged(player, zone)
        foreach  ( object in zone )
                sendAll (object->position);
        end
end
onPositionChanged(object)
        zone = getZoneFromPosition(object->position);
        foreach ( player in zone->activePlayers )
                send(player, object->position);
        end
end

Peer (owner side)
onRequestZoneInformation(zone,  player)
        send(player, zone->activePlayers);
        sendAll('ZONE_CHANGED', player, zone);
        onZoneChanged(player, zone);
end
```

Figure 39. Pseudocode Zoning P2P.

Analyse CS

In de opstelling zullen er twee actieve spelers zijn met elk drie objecten. Elke speler bevindt zich met zijn objecten in een eigen zone. Beide spelers zullen doorheen de simulatie een object bewegen waarbij ze op bepaalde momenten samen in eenzelfde zone actief zijn. Dit is geïllustreerd in figuur 40.

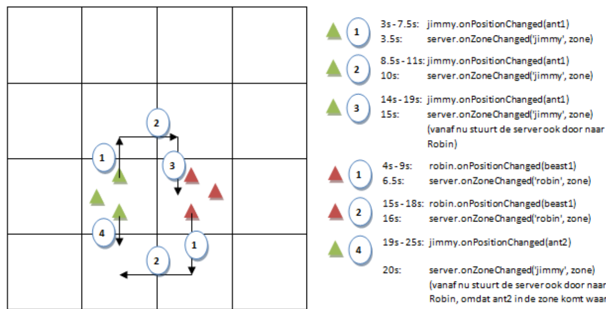

Figure 40. Opstelling van de simulatie met de verschillende events die belangrijk zijn voor zoning in CS (Robin is client 1 en Jimmy is client 2). Rechts van de figuur vindt u de verschillende eventen die plaatsvinden.

In figuren 41, 42 en 43 vindt u de traces van de simulatie met zoning. Dezelfde simulatie is ook getraced zonder zoning (figuren 44, 45 en 46).

Van 3s tot 7s beweegt client 2 (Jimmy) met een object naar boven (groen event 1 op figuur 40). Deze position updates worden door de server niet doorgestuurd naar Robin omdat Robin niet actief is in de zone. De server merkt vrijwel direct een zoneverandering op en laat de andere clients weten dat het object is veranderd van zone. Deze kleine uploadpiek van de server kan je waarnemen rond de 3s. Hetzelfde

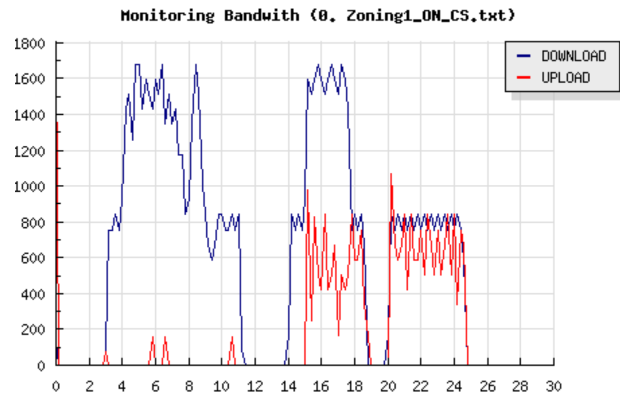

Figure 41. Server met zoning

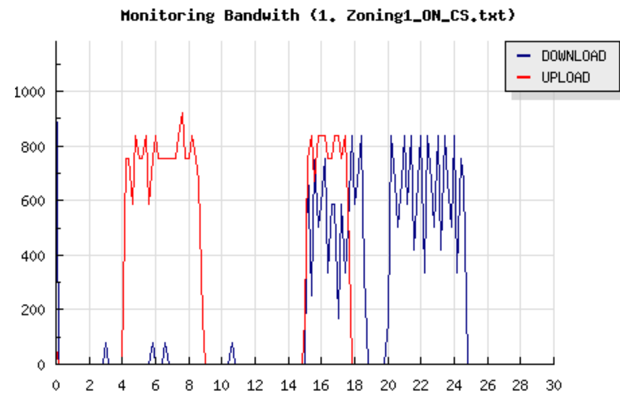

Figure 42. Client 1 met zoning

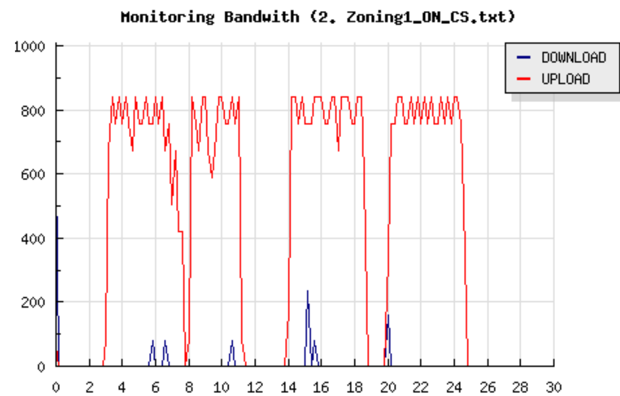

Figure 43. Client 2 met zoning

gebeurt ook met een object van Robin dat beweegt van 4s tot 9s (rood event 1). Een beetje later gebeurt een soortgelijke overgang voor hetzelfde object van Robin en Jimmy. Op 19s gaat Robin zijn object geplaatst hebben in de zones onder de objecten van Jimmy. Van 19s tot 25s gaat groen event 4 plaatsvinden. Deze gaat een object van Jimmy in een zone brengen waar Robin wel actief is. De server gaat namelijk op 20s de zone-overgang waarnemen. Op dit moment gaan alle positie veranderingen van het object van Jimmy ook

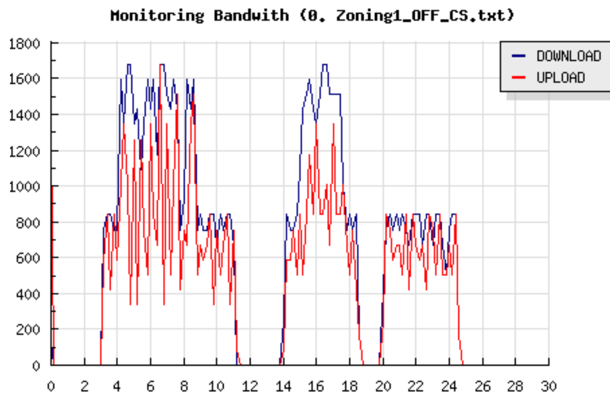

Figure 44. Server zonder zoning

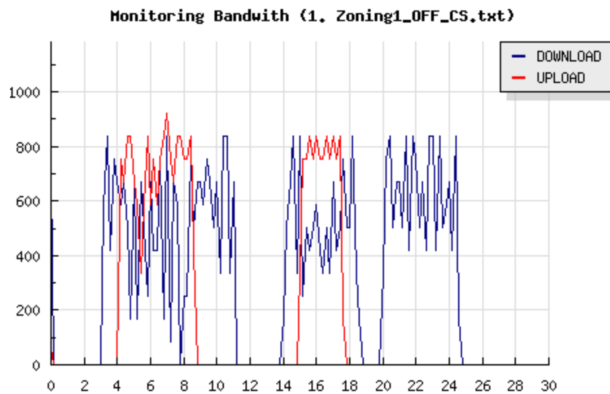

Figure 45. Client 1 zonder zoning

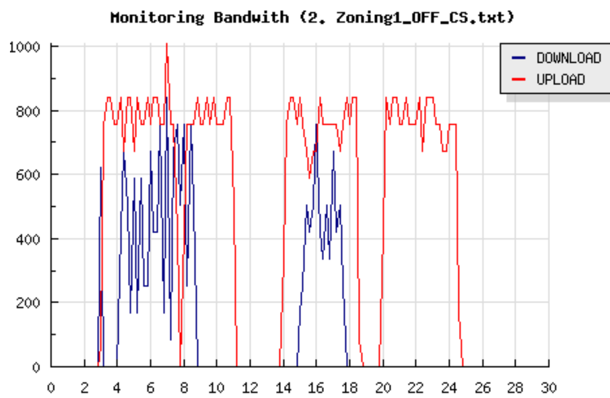

Figure 46. Client 2 zonder zoning

aankomen bij Robin. Als we de traces met en zonder zoning naast elkaar leggen, kunnen we concluderen dat de upload van alle clients geen verschil maakt en de download van de server ook hetzelfde blijft. Wel zal de server niet meer naar iedereen doorsturen. Doordat hij enkel naar de actieve spelers binnen de zone doorstuurt krijgen we een enorme optimalisatie van het netwerkverkeer.

Analyse P2P

Voor P2P hebben we dezelfde simulatie gebruikt. Het enige verschil is dat nu elke zone een eigenaar heeft toegewezen. Dit is geïllustreerd in figuur 47. De groene zones vallen onder het beheer van peer 2 (Jimmy). De rode zones vallen onder het beheer van peer 1 (Robin).

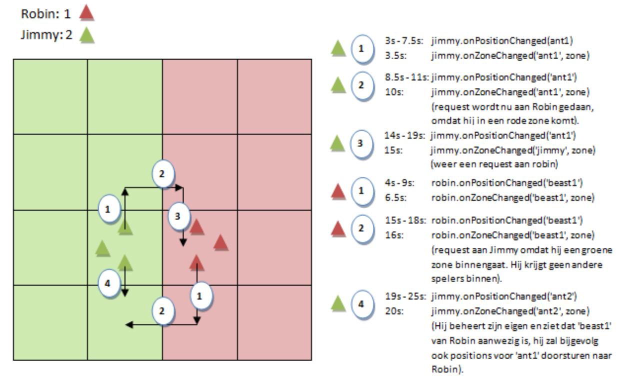

Figure 47. Opstelling van de simulatie met de verschillende events die belangrijk zijn voor zoning in P2P. Rechts van de figuur vindt u de verschillende eventen die plaatsvinden.

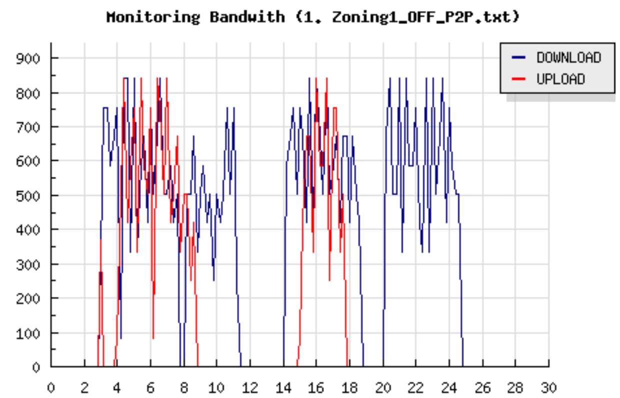

Figure 48. Peer 1 zonder zoning

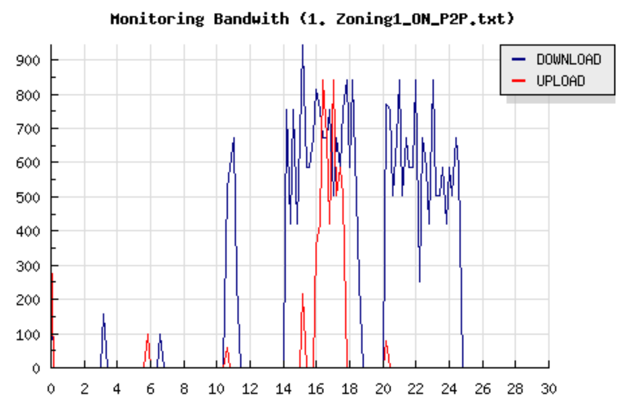

Figure 49. Peer 1 met zoning

Het belangrijkste verschil bij P2P is dat bij zone-overgangen de staat van de nieuwe zone moet worden opgevraagd. Bij event 1 op het groene object en event 1 op het rode object

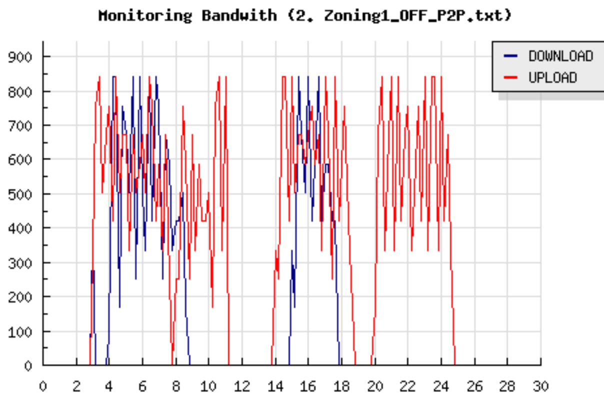

Figure 50. Peer 2 zonder zoning

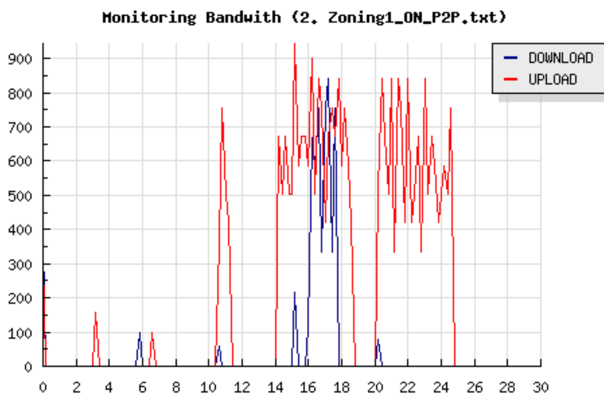

Figure 51. Peer 2 met zoning

geeft dit niet veel problemen. Ze gaan naar een zone die beheerd wordt door de eigenaar van het object zelf. Het enige dat zal gebeuren bij zo'n overgang, is het uitsturen van een event zodat de andere peers weten dat het object is verplaatst van zone. De posities van de twee objecten moeten nog altijd niet doorgestuurd worden naar andere peers omdat er geen andere actieve peers aanwezig zijn. Doordat in de nieuwe zones geen andere objecten zitten zullen er ook geen reacties komen op het zonechanged event. Pas vanaf event 2 op beide objecten wordt het interessanter. Het groene object van jimmy (peer 2) gaat zich verplaatsen in een rode zone van Robin. Dit zal er voor zorgen dat Jimmy een request doet voor informatie van die zone. Er zitten geen andere objecten in de andere zone (enkel de eigenaar is er actief), dus zal de reply enkel de eigenaar van de zone bevatten. Hierdoor zal peer 1 wel alle updates van het object ontvangen. Net hetzelfde voor het object van Robin dat in de zone van Jimmy gaat. Het groene object gaat nogmaals overgaan naar een andere rode zone (event 3). Nu zijn er wel andere objecten actief in de nieuwe zone, dit zijn allemaal objecten van de eigenaar zelf en heeft dus geen invloed op het zenden van updates. Een laatste event dat er gebeurt is event 4. Er gaat een groen object van een groene zone naar een andere groene zone, maar nu heeft de groene zone al een object van Robin. Robin gaat een zonechanged event aankrijgen van Jimmy. Deze ziet dat hij zelf actief is in de zone en gaat voor al zijn objecten de posities doorsturen. Jimmy krijgt nu direct de goede positie aan van het rode object. Verder zal Jimmy voor het groene object zijn positie updates ook doorsturen naar Robin.

Voor P2P is er heel wat meer state transfer nodig voor de zones. Maar indien er niet te veel zone overgangen plaatsvinden is dit redelijk schaalbaar. Als er weinig actieve peers binnen een zone zijn, gaat er heel veel netwerkverkeer wegvallen.

AOI

Als uitbreiding op zoning hebben we een soort van AOI gemaakt. Deze werkt niet helemaal op de traditionele manier van een AOI (het aura-nimbus idee) maar is eigenlijk vooral een manier om zoning nog beter te laten werken. Een probleem bij zoning is dat we rekening moeten houden met de randen van zones en overgang tussen verschillende zones. Als we deze overgangen te hard maken gaan er rare artefacten optreden omdat er opeens objecten bijkomen of verdwijnen uit de zones. Het beste is dus dat we niet enkel updates sturen naar de zone waar we momenteel in zitten, maar ook al naar de zones daarrond (maar dan eventueel niet zoveel of aangepaste updates) zodat de overgangen beter zijn.

Een naïve manier om dit te doen is de 8 zones rondom de huidige zone te nemen en zodoende de updates naar 9 zones in totaal te sturen. Dit heeft echter overhead want er is een grote kans dat we naar 1 specifieke richting gaan bewegen en we dus slechts 2 of 3 van de aangrenzende zones uiteindelijk zullen bezoeken, waardoor de overige 5 of 6 dus nutteloze updates krijgen. Een andere manier om een zachtere overgang te bekomen is om te kijken of we dichtbij een andere zone zijn (dichtbij de rand dus) en als dit zo is ook reeds updates sturen naar de aangrenzende zone. Hierdoor sturen we enkel updates naar de zones waar we echt dichtbij zijn en waar de kans dat we gezien worden groot is.

Dit effect bereiken wij door een soort van cirkelvormige Area of Interest te definiëren rondom alle objecten. Wanneer een object beweegt kijken we welke zones een overlap hebben met deze cirkel. We sturen onze updates dan naar elke zone die de cirkel snijdt. Op deze manier moeten we naar maximaal 4 zones tegelijk updates sturen en dit enkel als ze dichtbij genoeg liggen (aangegeven door de straal van de cirkel). Dit zorgt meteen voor zachte overgangen tussen zones omdat we al voor de echte overgang begint updates beginnen te sturen.

In de implementatie is echter bewust niet opgenomen dat de updates van de zones waar het object niet helemaal inzit ook gestuurd worden het object op de rand. Het object op de rand is dus zichtbaar in de aangrenzende zones, maar de eigenaar van het object op de rand ziet geen objecten uit de andere zones. Een uitzondering hierbij is natuurlijk objecten in aangrenzende zones die ook op de rand zitten, aangezien deze automatisch hun updates ook sturen omdat ze bij de rand zitten. Het concrete gevolg van deze werkwijze is dat een speler die oversteekt redelijk wat verderafgelegen objecten

zal zien verschijnen (artefacten) maar de spelers die al in de zone zaten zullen weinig merken van het binnenkomen van de nieuwe speler in termen van anomaliën. De verhoogde artefacten van de speler die oversteekt kunnen we verdedigen door de grote besparing op netwerkverkeer dan wanneer we echt alle objecten van aangrenzende zones zouden zien op de rand. Het is immers niet altijd zo dat een speler op de rand ook effectief naar de andere zone zal overgaan en als de AOI's niet te klein zijn zullen ze toch al redelijk wat objecten van de aangrenzende zones te zien krijgen die ook dichtbij de rand zijn in hun zone.

COLLISION CONSISTENCY

Wanneer er collisions plaatsvinden tussen objecten in de wereld kunnen er onder invloed van delays op het netwerk inconsistencies optreden voor de uitkomsten van deze collisions. Speler 1 gaat immers een iets andere positie weten voor de 2 objecten die botsen dan speler 2, waardoor ook de uitkomst van de collision iets anders gaat zijn. Deze verschillen in positie zijn groter naarmate er meer delay zit op de verbinding (waardoor de posities lang onderweg zijn). Dit kan er zelfs toe leiden dat speler 1 een collision ziet en speler 2 niet.

Een mogelijke oplossing hiervoor is om de spelers het voorkomen van de collision en de uitkomsten ervan met elkaar te laten communiceren. Op deze manier kunnen ze onderling tot een overeenkomst komen over "wat er nu echt is gebeurt". Afhankelijk van het gebruikte algoritme kan dit echter heel wat extra netwerkverkeer ten gevolg hebben. Een tweede probleem is dat ook dit verkeer delay zal ondervinden waardoor het een hele tijd kan duren eer de partijen het eens zijn over wat er gebeurt is, wat kan leiden tot verwarrende artefacten bij de spelers.

Een tweede mogelijkheid is om te werken met een locking mechanisme wanneer we als speler een object dat niet van ons is willen verplaatsen of manipuleren. Deze methode kan goed gebruikt worden voor "wereldobjecten" die niet bestuurd worden door een andere speler (zoals items), maar is minder bruikbaar voor collisions met objecten die bestuurd worden door andere spelers en het is ook moeilijker om simultane manipulatie van een object door meerdere spelers toe te laten (omdat slechts 1 speler tegelijk een lock kan hebben en de locks elkaar dan zeer snel zouden moeten opvolgen). Een laatste nadeel is dat ook deze methode afhankelijk is van de netwerkdelay. Als er twee spelers ongeveer tegelijk een lock aanvragen op hetzelfde object, zal degene met de snelste internetverbinding altijd winnen, wat een oneerlijk voordeel kan geven.

De oplossing die wij gekozen hebben is om 1 entiteit verantwoordelijk te laten zijn voor de collisions en het berekenen van hun uitkomsten. Hierbij kan er wel op een gegeven moment een beetje lokale inconsistency optreden bij enkele spelers (door delay), maar de uiteindelijke consistency zal hetzelfde zijn bij alle spelers omdat er maar 1 entiteit is die de uitkomst bepaalt en communiceert. In onze implementatie heeft dit zelfs geen extra netwerkverkeer tot gevolg. De spelers sturen gewoon hun posities van hun objecten. De authoriteit detecteert eventuele collisions en stuurt updates voor de posities van de objecten die collision ondervinden. De spelers kunnen lokaal ook een physics-simulatie draaien om in het geval van hoge delay toch een direct resultaat te zien, ookal kan dat later aangepast worden door updates van de server. In onze implementatie hebben we ervoor gekozen dit niet te doen en de spelers niet zelf een physics-simulatie te laten draaien, dit geeft immers minder onlogische artefacten bij hoge delay. Een voordeel van deze aanpak is dat ze werkt voor zowel collisions tussen speler-objecten als tussen speler- en wereldobjecten. Hierbij stuurt enkel de autoriteit updates van de wereldobjecten en de individuele spelers enkel updates van hun eigen objecten. Dit concept van 1 authoriteit die alles beslist wordt vaak toegepast in andere games (bijv. FPS) waarbij geavanceerde technieken als lag compensation en interpolatie worden gebruikt om de effecten van de delays zo goed mogelijk te verbergen en het spel eerlijk te houden. Een bijkomend voordeel van deze methode is dat het moeilijker is om te cheaten als 1 entiteit de beslissingen neemt.

Voor de Client/Server architectuur is het logisch dat we de Server nemen als autoriteit. Deze heeft toch al alle informatie, krijgt alle updates van de spelers binnen en is dan ook uitermate geschikt om alles te reguleren. Dit is dan ook de methode die vaak gebruikt wordt in bestaande games en architecturen. Voor P2P is het iets ingewikkelder. Hier is er immers niet zomaar 1 aanspreekpunt omdat alle peers een gelijkaardige rol vervullen in de wereld. Voor de optimalisatie zoning hadden we echter ook al gebruik gemaakt van 1 verantwoordelijke peer per zone. Voor zoning had deze enkel de verantwoordelijkheid om de spelers per zone bij te houden, maar we kunnen deze peer ook wat meer verantwoordelijkheid geven omdat hij toch gekend is door de andere peers (of toch tenminste kan opgevraagd worden). Hierdoor krijgen we in feite 1 peer per zone die verantwoordelijk is voor alle collisions in deze zone, alsook voor het bijhouden van de spelerinformatie in deze zone. Hierbij is het echter niet zo dat alle verkeer enkel via deze ene peer gaat: de peers sturen nog steeds alle updates gewoon naar elkaar door (en dus ook naar de verantwoordelijke peer). De verantwoordelijke peer kan dan wel extra updates sturen naar de peers na een collision om de consistency te bewaren, maar hij neemt enkel in dit opzicht de rol van server op zich en niet in de andere opzichten. Als er bij het aanduiden van een verantwoordelijke peer rekening wordt gehouden met zijn waarschijnlijkheid tot cheaten (bijvoorbeeld vertrouwde spelers of zelfs peers gecontroleerd door de eigenaar van de NVE) kan ook dit probleem binnen de perken worden gehouden (het zou immers erg slecht zijn als de verantwoordelijke peer slechte code zou draaien, waardoor heel wat spelers negatieve gevolgen zouden ondervinden).

Omdat er door onze implementatie geen extra netwerkverkeer wordt veroorzaakt, leek het ons niet nuttig nog uitgebreide traces van collisions op te nemen in het verslag. Er is wel een simulatie voorzien om te draaien tijdens de presentatie zodat het behoud van consistentie over de verschillende spelers duidelijk wordt.

VERGELIJKING MET ANDERE GROEPEN

Een eerste groot verschil van onze implementatie met de meeste andere groepen is het gebruik van een PeerServer. Bij ons kan een nieuwe speler bij eender welke andere peer terecht om te beginnen deelnemen aan het spel, terwijl bij de andere groepen dit via een gekende server gaat. De precieze invulling van deze server verschilt echter sterk van groep tot groep. Sommige groepen gebruiken deze gewoon om de lijst van actieve spelers op te vragen terwijl anderen veel meer door de server laten doen, zoals het afhandelen van zoning. Voor een echte P2P NVE is een server voor het opvragen van andere peers volgens ons zeker een goede oplossing. Bij ons kende elke peer nog elke andere peer (omdat niet alles in zoning zit) maar in een echte NVE zal dit niet het geval zijn. Een server die voor een nieuwe speler kan zeggen bij welke peers hij in de buurt zal spawnen is daar zeker interessant. Een oplossing zonder server kan bijvoorbeeld zijn dat er een bepaalde structuur zit aan de toekenningen van zones aan verantwoordelijke peers (zoals bijv. een hashtable zoals gezien voor PASTRY) waardoor nieuwe spelers via het netwerk snel het juiste aanspreekpunt kunnen vinden. Aparte servers voor zoning en andere dingen gebruiken in een P2P omgeving lijkt ons wat te veel afstappen van het P2P ideaal aangezien de servers dan al heel wat meer load te verwerken krijgen en we daar weer met scalability issues komen te zitten.

Een tweede groot verschil zit hem in het toekennen van authoriteit aan de clients in een C/S structuur. Bij heel wat groepen hebben de clients bijv. zelf ook kennis van zoning en sturen de clients zelf packets wanneer ze bijv. van zone veranderen. Ook voor de state van de wereld is het vaak zo dat de server eerder een forwarding rol aanneemt in plaats van zelf een actieve entiteit te zijn. In onze implementatie neemt de server erg impliciet controle over heel wat aspecten, zoals bijv. zoning, waar bij ons de clients geen enkele weet van hebben. De clients bekommeren zich enkel om hun eigen objecten en luisteren voor de rest naar de server die zelf nieuwe berichten genereert uit de game-state in plaats van gewoon berichten die de clients sturen verder te forwarden. Onze aanpak legt wat meer processing requirements bij de server maar wij vinden dat 1 van de belangrijkste voordelen van C/S is dat we de controle over de NVE en vooral de consistency in de handen van 1 entiteit kunnen leggen. Hierdoor verlaagt de kans op fouten gevoelig en wordt ook de mogelijkheid tot valsspelen vermindert.

Een derde groot verschil is in de implementatie van een lock-systeem voor het opnemen van objecten. In onze implementatie worden objecten niet zozeer expliciet opgepakt en meegenomen door een avatar om later weer expliciet te worden neergezet, maar neemt een avatar het object impliciet op als hij erover loopt (zoals bijv. een health kit). Dit in tegenstelling tot andere groepen waarbij objecten echt kunnen worden vastgepakt, verplaatst en terug neergezet door een speciale actie via het keyboard. Hiervoor gebruiken ze een uitgebreid lock-systeem waarbij locks worden aangevraagd op de objecten wanneer ze worden opgepakt en vrijgegeven als ze worden neergezet. We denken dat dit mechanisme vooral nut heeft in P2P systemen waar er niet 1 entiteit verantwoordelijkheid heeft over de objecten en er beter even kan overlegd worden vooraleer er iets mee gedaan wordt (wat in onze implementatie dus ontbreekt). Voor CS waarbij 1 server alle beslissingen kan nemen zijn extra berichten volgens ons vaak overbodig omdat de server kan beslissen of een actie op een object mag uitgevoerd worden of niet, zonder dat de gebruiker eerst een lock daarvoor moet aanvragen. Zo nemen bij ons avatars een object op als ze erover lopen (een collision dus) waarbij de server perfect zelf kan beslissen welke van 2 spelers eventueel eerst bij het item was (zeker als we lag compensation ed. zouden gebruiken). In onze P2P omgeving wordt het probleem ook deels opgelost omdat er ook maar 1 entiteit beslist over de collisions, waarbij het opnemen van items wel alle nadelen meeneemt van de authoritive peer aanpak. Voor het terug neerleggen van een object zorgt de authoriteit ervoor dat de positie van het object wordt gestuurd naar de andere spelers, waarna er weer kan geluisterd worden op physics-collisions van avatars.

Er zijn echter ook veel gelijkenissen terug te vinden tussen de groepen. Zo gebruikt bijna elke groep UDP enkel voor de positieupdates en voor al de rest gebruiken ze TCP. Dit is op zich niet verwonderlijk aangezien het meeste ander verkeer zeker niet verloren mag gaan en dus wel betrouwbaar moet worden verstuurd. Er zijn ook enkele andere groepen die het idee van incrementele updates gebruiken, al doen ze dat niet altijd 100% hetzelfde als wij. Hiervoor gebruiken ze ook TCP voor de full positions te sturen en UDP voor de offset berichten, wat eens te meer een zeer logische keuze is. 1 groep maakt ook gebruik van Area Of Interest, maar ze gaan hierin verder dan wij. Waar wij AOI vooral gebruiken om zoning wat beter te maken, gaat de andere groep ook expliciet verkeer filteren afhankelijk of een object binnen de AOI is of niet, terwijl dit bij ons enkel gebaseerd is op de zoning. 1 ding wat al de andere groepen ook hebben, maar wij niet, zijn de keepalive messages. Wij hebben deze niet geïmplementeerd omdat dit reeds gezien was in het vak GS en het detecteren van een timeout van een speler vaak direct mogelijk was via het uitvallen van een TCP connectie. Interessant vinden we wel dat de meeste groepen niet vermelden of ze de keepalives altijd sturen of enkel als er al een tijdje geen ander verkeer meer is gestuurd. De tweede optie is natuurlijk het meest performante qua verkeer, terwijl de eerste optie eigenlijk overbodig is omdat er tegelijk ander verkeer plaatsvind waaruit af te leiden valt dat de speler nog aanwezig is en enkel zorgt voor heel wat extra verkeer.

Wat de precieze structuur van de pakketten betreft zijn er natuurlijk wat project-specifieke verschillen te vinden. We denken hierbij aan het gebruik van timestamps, id toekenningen van objecten en spelers, etc. Uiteindelijk denken we dat de meeste groepen wel min of meer dezelfde inhoud doorsturen, zij het vaak in iets andere vorm. Dit is op zich niet verwonderlijk omdat het allemaal basisgames zijn met min of meer dezelfde features. We vinden het wel opvallend dat geen groep rechtstreeks de inputkeys communiceert maar enkel de posities en rotatie van de avatars op de lokale client. Dit terwijl veel online games eerder sturen welke keys ingedrukt zijn op elk moment. Deze manier van werken is volgens ons echter vooral geschikt voor CS en aangezien de

groepen zoveel mogelijk code proberen te delen tussen CS en P2P is het te begrijpen dat er geen enkele groep voor deze manier heeft gekozen.