DYNAMIC BANDWIDTH SCALABILITY IN LARGE SCALE NETWORKED VIRTUAL ENVIRONMENTS

Eindwerk voorgedragen tot het behalen van de graad van master in de informatica/ICT

Robin Marx

Promotor: Dr. Peter Quax
Copromotor: Prof. dr. Wim Lamotte

Academiejaar 2010–2011

Foreword

I would like to thank everyone that has helped me in creating this thesis. My promotor, dr. Peter Quax, for his support and valuable tips and suggestions. Wouter Vanmontfort for reading my drafts and helping me with the codebase. dr. Maarten Wijnants for being very helpful and thorough in his feedback. Prof. dr. Wim Lamotte for his never-ending enthusiasm in his classes and passion for his work and students.
Many thanks to my parents without whom this thesis and my studies would not have been possible. To my girlfriend for her support and encouragement. To my friends for their good humor and help in many aspects.

Ever since four years ago I've been drawn to network programming because for me, it is one of the easiest and one of the most difficult subjects in IT at the same time. The algorithms and approaches used are often very logical and simple in nature (compared to for example computer graphics techniques), but the distributed nature and the properties of the underlying network make that these simple ideas are often challenging to implement and test correctly. Furthermore, networking is becoming increasingly important in all aspects of society and even more so in online games and environments, illustrated by the huge amounts of users for products like World of Warcraft or Second Life. A thesis that focusses on NVEs in particular can hopefully bring me enough knowledge to one day be able to architect and implement these systems myself, which I would see as a huge accomplishment.

Abstract

Ever since network technologies and applications have been developed, bandwidth has been considered a scarce resource. Although an increase in absolute numbers can clearly be observed (comparable but not exactly equal in trend to Moore's law for processing speed), the demands put on the network infrastructure by applications have also surged in recent years. Several factors contribute to this fact, including the increased use of multimedia data (most importantly audio and video) and highly interactive application types. The latter are the subject of the thesis at hand, more precisely the class of applications known as 'Networked Virtual Environments'. Fluctuations in the amount of users and the data flows associated with each individual entity present in these virtual worlds necessitate a means for limiting the amount of data to be either sent or received by each participant. To be able to adjust these requirements and resources at run-time would clearly be very beneficial for the scalability of the entire system.

This thesis provides an overview of the network flows associated with and the bandwidth requirements of various types of Networked Virtual Environments. Ever since NVEs have been studied as a topic in (academic) research, ways in which to reduce the data flow have been proposed. They can be classified into four high-level categories: compression, aggregation, space partitioning and prediction. Each of these is discussed in this thesis, supported by examples, details and a summary of possible drawbacks. By combining these elements and comparison, the latter two techniques were found to be the most promising for further study.

NVE technology has widespread applications, but the most well-known examples can be found in so-called massive multiplayer games. At their core, these games use engines for network functionality. A number of these (commercial) software libraries and architectures are discussed and investigated, revealing a wide variety of techniques and (sometimes ad-hoc) solutions to known issues. Each of these engines has specific optimizations that are linked to the genre of game they are developed for, indicating the fact that several solutions will have to be combined in order to obtain a generic solution that works well under most conditions.

The implementation part of this thesis encompasses a generic and extensible Area-of-Interest

system (the AOI-API), which includes the ability to discriminate several Levels-of-Detail in information streams. By generalizing the software implementation, the range of applications that can be supported grows substantially and extends even beyond typical applications genres (i.e. MMORPG, MMOFPS, MMORTS,...). Furthermore and as a showcase example for its flexibility, the AOI-API is integrated into an existing architecture which is intended to support large-scale networked virtual environments - ALVIC-NG. By combining ideas and elements from the NIProxy system, which is a platform for dynamic bandwidth partitioning over various network streams, a very powerful system is obtained that can be applied under many circumstances. This fact is supported by extensive tests, classified into seven main experiments, each highlighting either the versatility, bandwidth shaping capabilities or large scale features of the proposed solution. Although additional work is needed to investigate the applicability in more extensive scenarios and to quantify the net gains in terms of bandwidth for more large-scale setups, it is clear from the test results that the functionality of the AOI-API, coupled with the NIProxy principles and the integration into the ALVIC-NG architecture is a promising step towards a truly scalable solution for client/server based networked virtual environments.

Abstract (Nederlands)

Al sinds de begindagen van computernetwerken en het internet wordt bandbreedte beschouwd als zijnde slechts beperkt voorhanden. Hoewel er duidelijk een stijging in absolute cijfers is opgetekend doorheen de jaren (gelijkaardig maar niet gelijk aan Moore's wet voor processor snelheid), geldt dit ook voor de eisen die applicaties stellen aan de netwerkinfrastructuur. Verscheidene factoren dragen hiertoe bij, onder andere het verhoogde gebruik van multimediale data (voornamelijk audio en video) en applicaties met een hoge interactiegraad. Deze laatsten zijn het onderwerp van deze thesis, meer specifiek de verzameling applicaties gekend als 'Genetwerkte Virtuele Omgevingen'. Fluctuaties in het aantal gebruikers en de datastromen geassocieerd met elk individueel object in deze virtuele werelden zorgen voor de nood om een techniek toe te passen voor het beperken van de data die wordt gestuurd door of naar elke gebruiker. Het dynamisch kunnen aanpassen van deze datastromen zou duidelijk heel heilzaam kunnen zijn voor het hele systeem.

Deze thesis geeft een overzicht van de verschillende netwerkstromen die geassocieerd zijn met verschillende types van Genetwerkte Virtuele Omgevingen (GVOs) en hun bandbreedte behoeften. Vanaf het moment dat GVOs bestudeerd zijn in (academisch) onderzoek, zijn er manieren voorgesteld om de datastromen te verkleinen. Deze technieken kunnen geclassificeerd worden in vier grote categorieën: compressie, aggregatie, ruimtelijke opdeling en voorspelling. Elk van deze wordt behandeld in deze thesis, ondersteund door voorbeelden, details en een overzicht van mogelijke nadelen. Door deze verschillende elementen te vergelijken, werden de laatste twee technieken het meest beloftevol bevonden voor verder onderzoek en bespreking.

GVO technologie heeft veel mogelijke toepassingsgebieden, maar de meest bekende voorbeelden kunnen gevonden worden in de zogenaamde massive multiplayer games. Deze games gebruiken specifieke softwarepakketten en structuren voor hun netwerk functionaliteit. Enkele van deze (commerciële) software bibliotheken en architecturen worden onderzocht en besproken, waarbij we ontdekken dat er een grote verscheidenheid aan oplossingen voor bekende problemen wordt toegepast. Elk van deze softwarepakketten heeft specifieke optimalisaties die gericht zijn op het genre game waar ze voor gemaakt zijn, wat aangeeft dat verschillende

technieken moeten worden gecombineerd om een meer algemene oplossing te bekomen die werkt onder de meeste omstandigheden.

Het implementatie-deel van deze thesis omvat een algemeen en uitbreidbaar Area-of-Interest systeem (de AOI-API), dewelke de mogelijkheid bevat om verschillende Levels-of-Detail te introduceren in datastromen. Door de software implementatie te generaliseren groeit de verscheidenheid aan applicaties die we kunnen ondersteunen gevoelig en zelfs tot buiten traditionele GVO genres zoals MMORPG, MMOFPS, MMORTS, ... Daarnaast, en als een voorbeeld van zijn flexibiliteit, wordt de AOI-API geïntegreerd in een bestaande architectuur voor grootschalige GVOs - ALVIC-NG. Door het gebruik van ideeën en elementen van het NIProxy systeem (een platform om dynamisch bandbreedte te beheren over verschillende netwerkstromen heen), verkrijgen we een erg krachtig systeem dat onder veel omstandigheden kan ingezet worden. Dit feit wordt onderbouwd met uitgebreide tests die zijn onderverdeeld in zeven experimenten die elk ofwel de flexibiliteit, ofwel de mogelijkheden voor bandbreedte schaalbaarheid ofwel de grootschalige eigenschappen van het voorgestelde systeem aantonen. Hoewel er opvolgend onderzoek nodig is om de toepasbaarheid te bepalen in uitgebreidere scenarios en applicaties op grotere schaal, maken de testresultaten duidelijk dat de functionaliteit van de AOI-API, gekoppeld met de NIProxy concepten en de integratie in de ALVIC-NG architectuur een goede stap is naar een schaalbare oplossing voor client/server-gebaseerde genetwerkte virtuele omgevingen.

Contents

1 Introduction **1**

1.1 What is a NVE? . . . 1

1.2 What are the problems with practical NVE design? . . . 2

1.2.1 Effects of network drawbacks . . . 2

1.2.2 Consistency and scalability . . . 3

1.3 Focus on bandwidth scalability . . . 4

2 Current state of affairs **8**

2.1 Network Architectures . . . 8

2.1.1 Peer to Peer . . . 8

2.1.2 Client Server . . . 12

2.1.3 Comparison . . . 13

2.2 Network Characteristics . . . 14

2.3 Game Types, Architectures and Protocols . . . 16

2.3.1 First Person Shooter game (FPS) . . . 16

2.3.2 Real Time Strategy game (RTS) . . . 18

2.3.3 MMORPG . . . 19

2.3.4 Physics simulations . . . 20

2.3.5 Collaboration . . . 21

2.3.6 Conclusions . . . 21

2.4 Network Streams . . . 21

2.4.1 User actions . . . 22

2.4.2 World state . . . 22

2.4.3 Audio/Video . . . 23

2.4.4 Low-bandwidth streams . . . 24

2.4.5 Case-study: Managing stream bandwidth with the NIProxy . . . 24

2.4.6 Conclusion . . . 27

2.5 Programming Idioms . . . 27

2.5.1 RPC, Replicas and attribute synchronization . . . 27

2.5.2 Message passing . . . 28

2.5.3 Command synchronization 28

3 Protocol Optimization **30**
3.1 Compression 31
3.1.1 Logical compression 31
3.1.2 Generic compression 35
3.1.3 Delta compression 38
3.1.4 Case-study: DIS PDUs 43
3.1.5 Case-study: Video compression 45
3.1.6 Conclusion 47
3.2 Aggregation 48
3.2.1 Packet headers on the internet 48
3.2.2 Quorum versus Timeout based 50
3.2.3 Where to perform aggregation? 51
3.2.4 Conclusions 52

4 Traffic Filtering **53**
4.1 Spatial subdivision 53
4.1.1 Sharding 54
4.1.2 Zoning 55
4.1.3 Instancing 59
4.1.4 Case-studies: Second Life, Eve Online and World of Warcraft 60
4.1.5 Case-study: ALVIC-NG 63
4.2 Area Of Interest 64
4.2.1 Traditional aura/nimbus model 64
4.2.2 Practical usage 66
4.2.3 Network architectures 69
4.3 Dead Reckoning 71
4.3.1 Dead reckoning models 72
4.3.2 Convergence and consistency 73
4.3.3 Dynamic bandwidth adjustment 77
4.3.4 Network architectures 79

5 Real-life examples and implementation recommendations **82**
5.1 Engines and middleware 82
5.1.1 RakNet 83
5.1.2 ReplicaNet 84
5.1.3 Quazal Eterna 85
5.1.4 XNA and XBOX Live 87
5.1.5 Unity 89

5.1.6 DoIT . . . 90
5.1.7 NetDog . . . 91
5.1.8 Ryzom engine . . . 92
5.1.9 BigWorld Technology . . . 93
5.1.10 HeroEngine . . . 94
5.1.11 Other systems . . . 95
5.2 Games and Virtual Worlds . . . 98
5.2.1 Second Life, There and World of Warcraft . . . 98
5.2.2 MMOFPS and MMORTS examples . . . 99
5.2.3 Browser-based games . . . 102
5.3 Choosing a technique for implementation . . . 102
5.3.1 Which techniques to use? . . . 103
5.3.2 Choice for implementation . . . 105

6 Implementation **108**
6.1 AOI-API . . . 108
6.1.1 Dynamic shape definitions . . . 110
6.1.2 LOD assignment . . . 111
6.1.3 Interest flags . . . 111
6.1.4 Multiple sets of AOIs per user . . . 112
6.1.5 API-centric design . . . 112
6.1.6 Extendability . . . 113
6.2 NIProxy . . . 113
6.2.1 NIProxy limitations and issues . . . 114
6.2.2 Practical use of the NIProxy . . . 117
6.3 Integration into ALVIC-NG . . . 121
6.3.1 Where to perform the filtering? . . . 121
6.3.2 Separation via plugin-like system . . . 125
6.4 Simulating client behaviour . . . 126
6.4.1 Behaviours . . . 126
6.4.2 Many clients in one process . . . 128
6.5 Graphical user interface . . . 129
6.6 Deployment and result analysis . . . 132
6.6.1 Deployment to a server cluster . . . 132
6.6.2 Result data gathering . . . 134
6.6.3 Result visualization and analysis . . . 136

7 Results **139**
7.1 Experiment 1 : First Person Shooter . . . 141
7.1.1 Description and setup . . . 142
7.1.2 Results . . . 144
7.1.3 Discussion . . . 147
7.2 Experiment 2 : Collaborative Environment . . . 148
7.2.1 Description and setup . . . 149
7.2.2 Results . . . 149
7.2.3 Discussion . . . 151
7.3 Experiment 3 : Real Time Strategy . . . 152
7.3.1 Description and setup . . . 152
7.3.2 Discussion . . . 154
7.4 Experiment 4 : Sniper . . . 155
7.4.1 Description and setup . . . 155
7.4.2 Results . . . 158
7.4.3 Discussion . . . 162
7.5 Experiment 5 : Collaborative Environment version 2 . . . 164
7.5.1 Description and setup . . . 164
7.5.2 Results . . . 167
7.5.3 Discussion . . . 168
7.6 Experiment 6 : Medium Scale . . . 169
7.6.1 Description and setup . . . 169
7.6.2 Results . . . 170
7.6.3 Discussion . . . 174
7.7 Experiment 7 : Large Scale . . . 175
7.7.1 Description and setup . . . 175
7.7.2 Results . . . 175
7.7.3 Discussion . . . 181
7.8 Conclusions . . . 182

8 Conclusion **185**

9 Future work **187**

List of Figures

1.1 Elements of consistency and scalability 5

2.1 Multicast vs. Unicast 11
2.2 NIProxy tree example 26

3.1 Graphical comparison of techniques for bandwidth reduction 30
3.2 Object Views 33
3.3 Huffman encoding example 37
3.4 Delta compression for positions 40
3.5 DIS Entity State PDU 44
3.6 JPEG generic compression 46

4.1 Types of spatial subdivision 54
4.2 Zoning types 57
4.3 Eve Online network architecture 61
4.4 Aura nimbus AOI model 65
4.5 Level of detail with AOI 66
4.6 Dead reckoning lag errors 73
4.7 Dead reckoning consistency problems 76
4.8 Dead reckoning dynamic error treshold 78

6.1 AOI-API structure 109
6.2 Server-level bandwidth shaping tree 120
6.3 Possible locations of filtering in the ALVIC-NG network 122
6.4 GUI comparison 130
6.5 GUI 131
6.6 GUI extra functionality 132
6.7 Deployment process 133
6.8 Visualization tool 138

7.1 FPS scenario 142

7.2 FPS AOI definitions 143
7.3 FPS player 1 145
7.4 FPS player 3 and 4 146
7.5 FPS comparison 147
7.6 Collaboration scenario 150
7.7 Collaboration player 1 151
7.8 Sniper scenario 156
7.9 Sniper bandwidth shaping trees 157
7.10 Sniper configurations and heuristics 158
7.11 Sniper best case scenario 159
7.12 Sniper best case scenario 160
7.13 Collaboration bandwidth shaping trees 165
7.14 Collaborative bandwidth usage player 1 167
7.15 Circle vs Flocking 171
7.16 Circle vs Flocking bandwidth limits 172
7.17 RTT Filtering on vs off 173
7.18 Circle large scale 10 updates 176
7.19 Flocking large scale 10 updates 177
7.20 Circle large scale 2 updates 178
7.21 Flocking large scale 2 updates 179
7.22 No filter large scale 2 updates 180

Chapter 1

Introduction

1.1 What is a NVE?

The title of this thesis contains the concept of a "networked virtual environment" (NVE). How can we define such a NVE? Taking a definition that will suit the coming chapters, we quote Singhal & Zyda [86]:

> A Networked Virtual Environment (NVE) is a software system in which multiple users interact with each other in real-time, even though those users may be located around the world.

There are a number of very important items in this definition. First of all, a NVE consists of multiple users. A typical NVE will not stop at 100, 200 or even 1000 users, but will have hundreds of thousands or even millions of (concurrent) users in the environment [46]. These users will usually have a graphical representation of themselves in the environment, often called an avatar. These avatars are often human-like and perform human actions like walking, running, jumping, shooting, etc. and are the user's main way of interacting with the environment.
Secondly, there is the real-time aspect. Users are going to interact with each other in a plethora of ways, depending on the setting of the NVE. These interactions are preferably executed without delay and with immediate feedback to the user. Users do not like to wait for their actions to take effect and in some types of NVE, even the slightest of delay can influence the experience [84]. Ideally, the NVE should appear to be running on the user's local system and all users should have the same consistent view on the environment at all times.
Lastly, the NVE should be accessible to users across the world and they should be able to interact with each other. Practically this means the NVE should work over the modern internet, as this is the main technology available to users today to connect to other computers and users worldwide. As we will see in more detail later, this is not such a trivial requirement

as it might seem at first.

In recent years there have been a number of (commercial) NVEs that are very popular and have large numbers of active users. Most of these NVEs are multiplayer games, more specifically MMORPGs (Massive Multiplayer Online Role Playing Games), such as World of Warcraft [53], Everquest [15] and Guildwars [21]. These games allow players to team up with their friends and undertake epic quests in a fantasy world.
Another type of NVE which is popular, is a Virtual Community. These environments are not so much games as recreations of real or fictive worlds in which people engage in different social interactions that are often mirrors of real-life activities. Examples include Second Life [46] and the now discontinued There [48].
These kinds of worlds in which users can interact in so many different ways with a large amount of other people are still increasing in popularity and are gaining a lot of attention from game developers and community creators alike. It is the belief of some that large scale NVEs will be the future of online interactions and gaming, making them an interesting subject for study in disciplines ranging from sociology to computer sciences [80, 67].

While the previously seen definition touches on three important characteristics of a NVE, it barely gives an idea of the huge numbers of technical difficulties that come with actually implementing and running a NVE with these characteristics. These problems can be divided into a number of smaller ones, such as security, cheating prevention, persistence, virtual economy, user generated content, ... , and two big ones : consistency and scalability.
This thesis will focus primarily on scalability and more precisely on bandwidth scalability. What this encompasses exactly is discussed later, as we first describe the abstract problems of consistency and scalability in more detail.

1.2 What are the problems with practical NVE design?

1.2.1 Effects of network drawbacks

Whilst the problems of consistency and scalability are quite different in concept, they have a lot to do with two basic properties of the network the NVE uses to transmit messages. These are especially present if we look at the internet as our network for deploying a NVE.
To understand this we must first understand the basic inner workings of a NVE and the network. When we perform an action in a NVE, other users have to see this action in order to be able to react to it. Using a network, we can send messages to the other users, notifying them of our actions. The other user can then update his local view of the environment upon receipt of the message and see the result of the action the same way we saw the result of the action when we did it. This is sometimes called synchronizing world state.

This way, NVEs are actually a kind of advanced distributed systems, a term which indicates any application that uses a network to communicate data between different entities in network packets. The problems we discuss here are thus inherent to any type of distributed system, but as we will see, they are enlarged and even more difficult to overcome in NVEs because of their huge scales and requirements.

The first problem is the network latency [65]. This means that any message you send will be delayed by the network and it will take a while for it to arrive at the other users. To complicate matters further, this network delay is not constant and can vary, not only between users depending on their physical distance from each other, but also on one link between users, called network jitter. This problem makes achieving real-time synchronization between users a lot more difficult.
The second problem is the limited network bandwidth [65]. Computer systems work in bits and bytes and every message a user wants to send to other users has a certain size in bytes. The problem is that network links can only process a limited amount of bytes every second, called the bandwidth. So if your messages are 100 bytes and your network link has a bandwidth of 1000 bytes/second, then you will only be able to send 10 messages per second (or even less, as most network protocols induce a certain amount of bandwidth overhead per packet). If you would try to send more, the network would become overloaded, causing lost messages and bigger delays between messages. This problem makes it difficult to support a large number of players because every player requires more packets to be sent as every player also introduces new interactions to the world.

A lot of academic and military NVEs are only used on local and proprietary LAN or specifically designed WLAN networks (see chapter 5) and as such do not have to deal with these problems the same way game and entertainment developers have to. This thesis will explicitly try to deal with the problems the real internet brings and as such dismiss some of the academic theories for practical usage and focus on viable techniques for usage in entertainment-based NVEs over the internet.

1.2.2 Consistency and scalability

Whilst consistency and scalability are similar to the real-time and multi-user requirements of a NVE they are broader in concept and problem:

Consistency means that all users have to have the same view of the environment at the same time. As discussed before, networking delay and jitter makes this real-time aspect a lot more difficult to achieve.
Consistency is not just limited by the network delay and jitter; the available bandwidth

will also have a big influence on achievable consistency. As discussed previously, the users communicate with each other through network messages. To achieve good consistency, all users will have to send a very large number of messages every second, especially in a fast-paced simulation where a lot of actions can happen in just one second. If we need to send all these actions to all other users when they appear, there simply will not be enough bandwidth to support this kind of traffic. Simply put: the higher the consistency requirements, the higher the theoretical bandwidth usage will be. This concept was called the consistency-throughput trade-off by Singhal & Zyda [86]. They state you can either have a highly dynamic world which is not very consistent, or a very consistent world which is not so dynamic, but you cannot have both high consistency and a dynamic world at the same time.
In conclusion, consistency is not only limited by the networking delay but also by the available bandwidth. This is a very important aspect which forms one of the motivations for this research.

Scalability means that the NVE should be able to handle large numbers of users without large problems. New users should be able to join and participate in the experience without other users having any kind of negative effect on performance because of the other user's arrival and participation. As discussed before, with every new user the bandwidth usage will go up. This is because the new user is now also transmitting his actions through messages over the network which have to be received by other users. So there is a limited number of users the NVE can support, dependent on the available network bandwidth.
Scalability is not just limited by bandwidth constraints; the hardware of the end-systems and possibly intermediate systems also has a big influence on the number of users that can be supported. The more users, the more interactions that will have to be processed, the more models will have to be rendered, the more calculations that need to be done. Every message that is sent through the network also has to be interpreted at the intermediate and end-stations and with large numbers of players this is often not trivial, even on very modern hardware. So even if we would have ample bandwidth in the network, the hardware and performance could still be a bottleneck.
In conclusion, scalability is not just about dealing with limited bandwidth, it is also about the performance of the hardware of the end-stations in the network.

1.3 Focus on bandwidth scalability

We can now shortly summarize the previous paragraphs.
For the users, the main requirement of a NVE is consistency. Without a sufficiently consistent world, there can be little meaningful real-time interaction between users. This consistency is dependent on network delay/jitter and network bandwidth and thus these factors should be taken into account when designing a NVE.

For the developers, a big problem is scalability. Users would not appreciate their experience being hindered for something simple as other users joining the environment. Developers have to keep everything up and running, even in the presence of large numbers of concurrent users. Therefore they need to optimize the performance of their software and also the bandwidth usage.
We can put these conclusions into a simple table:

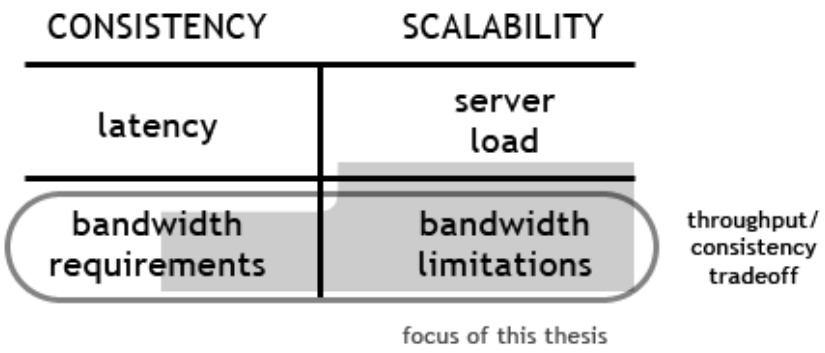

Figure 1.1: Elements of consistency and scalability

As we can clearly see, bandwidth usage plays a large role in both consistency and scalability of a NVE. Furthermore, they are intertwined in such a way that one will be affected if we change something to the other. This can be used as a positive effect as well: if there is an excess of bandwidth available, we can increase the measure of consistency. On the other hand, we can use some logical concepts about consistency as observed by the user to control bandwidth usage of the NVE.

Bandwidth is a very important resource for a NVE and an effective implementation for (dynamic) bandwidth scalability is important for a number of reasons. First of all, bandwidth is costly [7], which can make it expensive to maintain a large NVE. With a bandwidth management system, the amount of bandwidth can be limited to lower the costs. For instance, in some countries bandwidth costs less during nighttime than it does during daytime. The bandwidth limit can be kept lower during daytime to ensure lower expenses.
Another argument is that there is a lot of heterogeneity in contemporary internet connections. Users with high-speed, broadband connections should be able to interact with mobile players using a slow, smallband mobile network. The NVE should be able to adjust its network usage to the type of connection a user uses to connect to it.
A third aspect is that bandwidth limitations can fluctuate considerably during the usage of the NVE. This is caused by other people using the same network and other outside factors, such as a router going down in the internal network for example [62, 74]. The NVE should be

able to compensate for these fluctuations and dynamically compensate it's bandwidth usage to stay within the changing limits.
A final reason is that there are a number of game and application concepts that have not yet been very successful in a NVE setting. One of the reasons for this is that it is difficult to sufficiently limit their bandwidth usage to make them possible on the modern internet. One of the goals of this thesis is to see how we can bring these small-scale concepts to a NVE setting and how this would affect bandwidth usage. This is especially true for at least three kinds of applications that we will focus on in this thesis. Each of these examples will be discussed in more detail later in this text:

- Applications with a high real-time consistency requirement (ex. First Person Shooter)
  Most modern NVEs use an interaction model for which little and infrequent traffic is needed to transmit the user's actions. There are game types however that have a much higher bandwidth requirement to enable full consistency. This is true for real-time action games such as First Person Shooters, where players control their avatar directly and actions are very fast paced. In order to support the same number of concurrent players as is common for the other slower paced interaction models, bandwidth scalability is needed.

- Applications with a huge number of objects (ex. Real Time Strategy, physics)
  Large, strategic simulations will traditionally not be played by a lot of players at the same time, but these players will be controlling a large number of objects each. Historically, other methods have been used to deal with this problem, but as we will discuss, these methods are not easily transferrable to a large scale NVE. Another example are applications in which physics calculations are used to determine how objects realistically react in the world. These physics calculations have to be done on a large number of objects to appear realistic and all these objects have to be kept consistent for a large amount of players. Bandwidth scalability can help bring these concepts to a NVE setting.

- Applications which support user generated content and complex communications (ex. Collaborative environments)
  When we look at how people can best interact and collaborate across a network to perform more serious or realistic tasks, these applications often have special types of communication. Example are for instance audio and video communication between different users, special input devices such as multitouch tables, realtime manipulation of 3D objects by multiple users, user generated content, etc. These types of communication often require large amounts of bandwidth and thus bandwidth scalability is needed to bring them to a NVE setting.

In conclusion, we can say that bandwidth scalability is needed for a number of different rea-

sons and will help enable larger NVEs which are built around new concepts . This scalability can be achieved mainly by manipulating some consistency parameters, as there is a large overlap between the two concepts.

This thesis will research techniques that can be used to ensure bandwidth scalability for a NVE while keeping consistency as good as possible. It will also be discussed how these techniques can be used for dynamic scalability, i.e. how they can be used to adjust bandwidth usage of the NVE at runtime to deal with changing bandwidth limitations on the network connections. Some of these techniques will be chosen to be implemented in a larger NVE framework and simulations will be run to see what effect the suggested solutions really have on the bandwidth usage. In addition, the versatility and applicability of the system will be shown by simulating a few situations in NVEs that are difficult to keep scalable without the proposed techniques.

The hypothesis is that by lowering some consistency requirements, we can achieve good dynamic scalability without it having a dramatic impact on the user's subjective experience. It is expected that special situations will arise in certain interaction types that need special consistency management. The objective is to enable the framework to handle these special requirements, enabling a wide range of NVEs to be deployed using the suggested approach.

In chapter 2, the current state of affairs for network infrastructures and typical network characteristics, game architectures, interaction models, network streams and network programming idioms will be discussed. This will provide a solid basis to discuss methods for bandwidth reduction and to place them in the correct context. Chapter 3 will deal with protocol optimization. The two main concepts of compression and aggregation will be discussed. Chapter 4 discusses traffic filtering in the forms of zoning, Area-of-Interest specifications and prediction techniques such as dead reckoning. Chapter 5 will then look at how the discussed techniques are employed in academic and commercial engines, networking middleware solutions, games and environments. Chapter 6 described our versatile Area-of-Interest implementation in ALVIC-NG [85] using the NIProxy [94]. Chapter 7 describes the performed experiments and analyses the generated results. Finally, chapter 8 and 9 conclude this text and discuss future work.

Chapter 2

Current state of affairs

Here we look at how common real-life networks and games are structured and implemented and what influence this can have on bandwidth usage. We do not limit ourselves to NVEs or large scale ideas because one of the goals of this thesis is to see how existing game concepts and applications like FPS and RTS that have not yet known a large breakthrough on a larger scale can be deployed in an NVE setting. Therefore we also look at how existing small-scale multiplayers games and applications work and see if these implementations are still usable on a larger scale.

2.1 Network Architectures

To make an NVE possible, users have to be connected through a network, and this can be done in various ways. Generally there are two important possibilities: directly (Peer to Peer, P2P) or through an intermediary (Client-Server, CS). Both have advantages and disadvantages. Here we discuss the most common architectures for both P2P and CS. This should not be considered an exhaustive list of all possibilities, which are possibly endless, but a good introduction that touches on all aspects that are important for later use in this text.

2.1.1 Peer to Peer

This is the simplest concept to understand: players connect directly to each other. The network expands with every new player and so the performance scalability is relatively easy: every user adds new processing power to the network.
P2P is also very cheap as no extra infrastructure is needed to maintain the network.
The biggest issues for using P2P as network model are those of security and cheating. When there is no central controlling entity, any peer can cheat or send hazardous information to other peers and it is very difficult to protect against this. This can be very negative for the user's experience.

Another big issue is that of persistence. It is difficult to decide where to store the world state. We cannot choose just any peer as there is a very real possibility that he will not be online all the time. When the user responsible for a certain part of the world would disconnect, this part would become unavailable. Thus, P2P systems have to take extra care as to how and where they store the information about the world that has to be maintained throughout users sessions.
The two main issues mentioned in the introduction, namely performance and bandwidth scalability, are discussed in more detail for P2P systems later.

Nearest Neighbour Problem

One of the main problems in P2P is how to know which users one has to connect to and how these users can be reached. Logically, it is not possible to have every user in the world connect to every other user if the user count is high.
P2P is used for a number of different purposes, mostly filesharing. Several technologies are used there to obtain the information about other users. One is to use a single tracker entity, known to every user, which lists all files and which user has them available. BitTorrent is an example of a popular filesharing system that uses this approach [6]. Another system is that of the Distributed Hash Table (DHT) [9]. With a DHT, there is no central place were all information is kept but this information is stored in several places across the network. Using so-called hash-functions and possibly a special routing overlay network, users are able to find the wanted files. This approach is very scalable and fault-tolerant. An example of a DHT system is PASTRY, which stores resources on nodes with similar hashes to the resources themselves [36].

For filesharing, we are searching those users that have the files we are interested in. For gaming or virtual worlds, we are mostly interested in the users who are geographically closest to us in the virtual world. This concept is mostly referred to as Area of Interest or Zoning and will be explained in further detail in chapter 4.
A first approach to do this is to use a DHT, as is done for filesharing. SimMud [73] is an example of a system that uses this approach through a PASTRY implementation. In SimMud, the world is divided into adjacent regions (see chapter 4 for more details) and every region is assigned a hash. Each peer also has a unique hash. The peer with the hash that is most similar to the region hash, becomes the coordinator/master of this specific region. This assignment is not related to where the peer is in the world, and so it is unlikely that the peer is coordinator of the region he himself is in at this time. This reduces the security risks as the incentive for cheating will probably be less. Another advantage is that the coordinator of a zone does not need to change if the peer transfers to a new zone. By replicating all data of a coordinator on another peer as well, the system is made fault-tolerant.
A second approach is to use voronoi diagrams. Given a number of 2D points (sites), a voronoi

diagram subdivides the other points in regions so that every point in a region is closest to the site of that region. VAST [52] is a system that uses voronoi diagrams to determine interesting neighbours. It sends updates to these neighbours and if a peer detects an important change in the voronoi diagram (for instance, a user has moved closer to another user), the peer will communicate this to his neighbours. This way, the set of closest neighbours will always be up to date for all peers. However, this may lead to problems if there are many users together in a small place, as many connections would still have to be made. For this, VAST uses a dynamic AOI (see chapter 4 for more details) to determine how far a peer can be removed for it to still be an interesting peer. If there are many peers nearby, this distance will be smaller than when there are less peers in the direct vicinity.
Even other approaches are possible but their basic outcome stays the same: find the n nearest neighbours for the current user to connect to and update this set of neighbours as the users move through the world.

Practical connection problems

Once we know the nearest neighbours, we need to actually connect to them. On the internet, this is not always trivial.
Many routers use a technique called Network Address Translation (NAT) [33]. This means that the IP address of a client as seen from the outside world is not the real IP address, but that of the user's router. Consequently, we cannot connect directly to the user. A number of techniques can be used to overcome this [69], like Universal Plug and Play (UPnP), UDP hole punching or worst case: relaying. The latter approach uses an entity which is not behind a NAT so the peers can connect to this entity and exchange data through the relay instead of directly to each other. This technique is for instance used in the P2P VOIP application Skype.
However in a true P2P system, it can be difficult to find a peer that is not behind a NAT and that has sufficient bandwidth and processing power to act as a relay. Furthermore, even if a direct connection from a peer to another peer would work, it is possible that the firewall will block any incoming connections. This is a typical setting as firewalls usually only allow outgoing connections.
In conclusion we can say that connecting users in a P2P fashion on the contemporary internet is not free of practical issues, which mostly have to be solved using relay entities, which are not always optimally available.

IP Multicast

IP Multicast is a technology that allows a peer to send one message to a multicast group, upon which every member of that multicast group will receive that message. The network itself is responsible for replicating the message for the other users and this is only done at

nodes in the so called multicast-tree. This is the inverse of making the connections ourselves and sending the packet to the other peers ourselves, which is also called unicast, see figure 2.1.
Using IP Multicast we can assign one multicast address to every region of the world. Every user sends his updates to the multicast group of the region and every other user in this region receives all the updates. It is simple, elegant and very logical. Another technique would be to use a multicast group per entity/avatar. This way, users can choose which avatars they are interested in by subscribing to their multicast groups. A plethora of multicast-based schemes have been described in academic literature, as multicast is a very interesting technology in research. A downside is that it can be challenging to provide enough fine-grained levels of detail to enable users to determine exactly what information they want to receive [88]. Later sections will explain this problem in further detail.

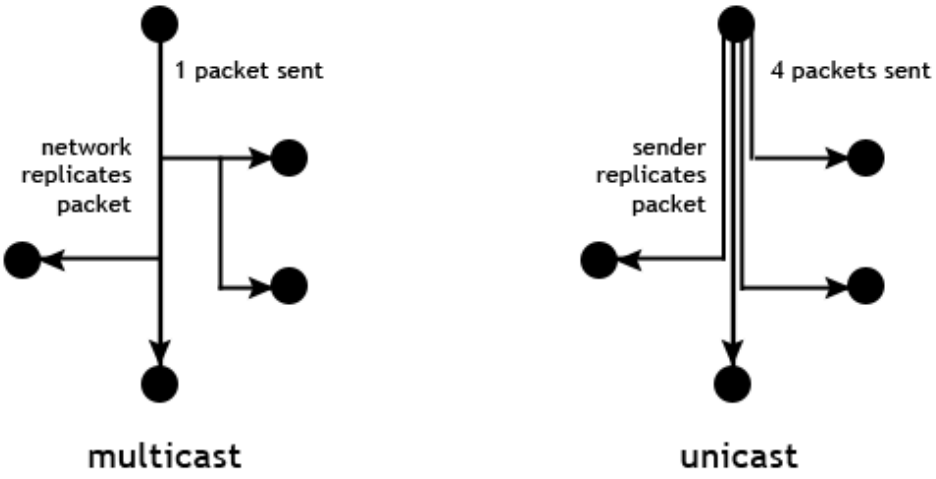

Figure 2.1: Multicast versus unicast.

Sadly, multicast is not very widespread or supported in every router on the internet, even though the internet backbone has good multicast support. As such, we cannot assume that every potential user of the NVE will have access to a fully multicast-enabled network when using the internet. This renders basic multicast all but unusable for any practical application on the contemporary internet. A possible solution is to simulate multicast using application-layer protocols [95], but this requires extra work and it is not guaranteed to provide the versatility of the basic multicast idea. So even though it is a very good technology for NVE systems, it is not practical and only usable on LANs or controlled simulation setups. There are hopes for improved multicast support with IPV6 [27], but the adoption of this new protocol has been very slow.
It should be noted that even if IP Multicast would become available, there would still be

a problem with reliable delivery of the messages. IP Multicast is UDP-only and it can be shown that implementing a reliable protocol requires a lot of communication overhead (more so than is the case with unicast connections) [65].

2.1.2 Client Server

As explained before, when we allow users to connect directly to each other, there is no way to control what data they exchange and so it is possible for a malevolent user to cheat in a game or pose a security risk in a system. This is one of the main reasons why the Client-Server architecture is very popular for online gaming. We introduce an intermediate entity in the network, the server, through which all messages have to pass. This server can check for cheating, violations against gameplay logic or unauthorized operations. In addition, this allows the server to be in control of consistency and decision making. For example, deciding whether one player has successfully shot and killed the other player is a lot more difficult in a P2P than it is in a Client-Server setup, where we only have one authority.
The Client-Server approach needs extra hardware however, often in the form of dedicated server machines. Especially in the case of an NVE, a very large number of servers is needed to be able to support a large number of users. Additionally, servers can induce extra latency for the messages being sent, because they first need to pass through the server and they often also incur some processing overhead at the server, creating even more delay.

The dedicated or host server

For games or applications with a small number of users (typically up to 32 users for a FPS game), one server often suffices. All players connect directly to the server, which calculates the world state at set intervals and sends the changes to every player.
This setup allows one of the players to act as server. A player chooses to "host" the game, after which all other players connect to the server. The host can either play on his server or run a so called dedicated server, a separate process on his computer. This eliminates the need for extra hardware as any player can start a new game on their local computer. In practice however, players often rent a server from a company to act as a 24/7 game server for their favourite game. This is because local players can also suffer from the same practical problems as a P2P setup when they host a server themselves, such as the need for NAT traversal techniques.

Server clusters

As the number of users grows, one server will no longer suffice to process all the data. The approach here is to add extra servers and have all the servers interconnect so they can exchange information about the world and users they have connected to them.

Typically, each server in a cluster is responsible for one part of the world (a region or zone) and processes information of the users currently present in that part. This technique is called zoning and will be discussed in chapter 4.
A good example of a cluster-based NVE is Second Life [46].

Advanced systems

When the virtual world becomes too large, the cluster approach will not be sufficient. The servers will have to maintain too many connections to the users (which is very resource intensive) and process too much data.
A possible solution is to add an extra layer of proxy-servers on the outside of the network [71]. These proxy servers have the task of managing the user connections so the normal (logic) servers do not have to worry about the users, they just have to deal with the gameplay data. One proxy server typically connects a large number of users to the internal network of logic servers. These proxies can filter certain traffic to the logic servers or take certain decisions themselves, reducing the load on the logic servers. This is an approach taken in for example Eve-Online [13] and ALVIC-NG [85], who will both be discussed in more detail later.
Other more advanced setups will try and create a hybrid between P2P and Client-Server. Non-critical data, like audio and video streams, can be exchanged between users directly without having to pass through the server, removing a large processing load. One could also use P2P as the basic networking architecture (possibly using multicast) but incorporate servers for certain tasks, such as neighbour discovery, critical gameplay decisions or traffic aggregation [86].
This kind of P2P-CS hybrid could be interesting for gaming consoles like the XBOX 360 or PlayStation 3. These are mostly closed platforms with a lot less possibilities for users to cheat (compared to online PC gaming), which makes P2P technologies a lot more interesting. Combined with servers for persistence, it would be possible to make a very scalable and affordable NVE.

2.1.3 Comparison

As the focus for this text is on scalability, let us compare P2P and CS setups in that respect. As stated before, the difference in terms of processing scalability is quite straightforward. For P2P, each user in the system automatically adds new processing power and some users with a large number of resources can play a more important role in the network. This way, the system automatically scales as the number of users changes. For client-server, there is often a need for extra and expensive hardware to maintain a good processing scalability, especially in large-scale setups.

The difference in bandwidth scalability is a bit less straightforward. When using P2P with

unicast, there is a lot of traffic on every client as they have to send every packet n times, where n is the number of currently connected peers.
P2P with multicast enables the user to only send one packet and the network will replicate it where needed. As users can choose themselves to which multicast groups they subscribe, they can also regulate their incoming traffic to a certain extent. Mostly however, multicast does not allow a very fine-grained control of the traffic. The user often has to choose to receive all or no traffic from a certain region or entity, with little or no possibility to receive a certain Level of Detail.
The reader should also note that there can still be a large amount of traffic in the internal network. Even though multicast will try to optimize the amount of packets that have to be replicated, there is no guarantee or measurement about how effective this will be. As a consequence, multicast is certainly better for bandwidth reduction when sending packets, but not necessarily for bandwidth usage in the NVE itself.

From the user perspective, the Client-server approach has all the advantages of the multicast approach and more. The user only needs to send his data to the server, and what is more, it only receives data it needs from the server. The server can perform very fine-grained control of what data is being sent to the user and this is a very important concept for the rest of this text.
However, this optimal situation for the client comes at a huge cost for the server. The server needs to process all incoming data from sometimes a very large number of clients, causing the incoming traffic to be directly tied to the number of users connected to the server. The server can perform a lot of operations on the data so that users only receive the data they need, and thus severely reduce its own outgoing traffic. These operations require a lot of processing power however, putting a higher load on the server.

As can be seen, both approaches have benefits and drawbacks for both aspects of scalability. Client-server appears to be the less interesting technology, but practical issues with P2P as well as the problems with cheating and persistence management cause that Client-server is still the most dominant architecture in real-life applications over the internet.
As P2P is still a very active area in academic research and because support for multicast will increase with IPV6 in the future, P2P will be kept in mind when describing techniques for bandwidth scalability later, but the primary focus will be on Client-Server architectures.

2.2 Network Characteristics

There are many different kinds of networks with different characteristics. For this thesis, we are especially interested in the internet, which uses a specific set of protocols and has a certain hierarchical architecture. We are especially interested in two parameters: bandwidth

and packet loss.
The maximum bandwidth of a connection depends on a large amount of factors, such as the physical capability of the connection wires, the buffer sizes of the internal routers, and so on. We will not discuss all these aspects here, only those that are most comprehensive in our explanation of bandwidth usage and scalability in NVEs.

One aspect that is very important, is that the available bandwidth at the core of the internet is typically very high, and so is the bandwidth in proprietary networks connecting most commercial NVE servers. However, the available bandwidth from the internet core to the user, the so-called "last-mile" can be significantly smaller. This means this last connection to the user acts as a bottleneck which will determine how much bandwidth the NVE can use.
The reasons why this last mile has less bandwidth can depend on a couple of reasons. For instance, in a cable-based network, the last mile is often shared between several users and if other users are creating a lot of traffic, the available bandwidth for the NVE user will go down. Another reason can be that the user is connected through a wireless network, which is already slower because of the wireless transmissions.
In a recent survey for an existing game [55], creators of a popular game measured the common bandwidths of the players of the game. It was found that most users have around 176 kb/s download bandwidth and 226 kb/s upload bandwidth. This is relatively low if you know a typical CounterStrike server for 22 players will send around 886 kb/s [88].

Maybe this would not be a very big problem if this bandwidth was fixed in time. Sadly, this is not the case on the internet. Many factors can influence the available bandwidth [62], most notably how many users are using a particular network link and how much traffic they generate. As we have said, if these users would generate more traffic than the available bandwidth can handle, the intermediate routers in the network will start to drop packets and the latency will go up as they cannot process all this traffic anymore. This situation is called network congestion and is very dangerous as it can completely close off a big part of a network if not dealt with.
Either way it happens, the available bandwidth will fluctuate (just as there is jitter on the latency of a connection) and the NVE will have to take care not to exceed this bandwidth to prevent congestion and even worse network problems for the user.

A final issue is that of packet loss. Once again, many factors can cause a packet to be dropped in the network. This can be a problem for consistency of the NVE as some users will not receive the dropped packets and will lose some (important) information. However, for reliable streams, it can also be a problem for bandwidth usage. This is because reliable protocols are reliable because they retransmit packets that are not confirmed as received by the receiver. This means that when a packet is dropped, it will be sent over and over again

until it reaches it's destination and a confirmation packet successfully finds it's way back to the original sender. This can be a problem in a congestion situation when packets are being dropped by routers because they cannot take the load. Naive reliable protocols will then try to retransmit the dropped packets, creating an even larger congestion.
The main reliable protocol on the internet, TCP, has built-in safety mechanisms to prevent this. Sadly, this also means that in the even of congestion, TCP will become very slow, reducing it's throughput to a mere fraction of the previous bandwidth, which can be a problem for a fast-paced NVE. UDP on the other hand, has no such safety mechanisms. As we will discuss later, programmers will often define their own reliable protocol on top of UDP to create a more flexible solution. This also means that the problem of congestion must be dealt with in this reliable UDP implementation or the NVE may cause certain networks to become congested, which is detrimental to the NVE an all other applications using that network.

So in conclusion, we can state that the available bandwidth on the internet for users is generally limited, but the bandwidth available to servers for the NVE will typically be higher. This bandwidth can also fluctuate considerably and we should take care not to exceed these dynamic limits. Finally, packet loss can not only influence consistency in the NVE but also increase bandwidth usage for reliable streams, and so we should be careful to use reliable protocols when trying to limit bandwidth.

2.3 Game Types, Architectures and Protocols

In this section, we will discuss how modern day smaller-scale games and interactions are usually implemented and try to determine how they could be implemented in an NVE setting. We first discuss three specific types of game and then look at two common concepts in many games and applications. Later in this thesis, in chapter 6, we will actually implement some of the here discussed approaches in an NVE setting.

2.3.1 First Person Shooter game (FPS)

First person shooters are games in which every player directly controls an avatar from a first person perspective (it seems as though the player is looking through the eyes of his character). The player usually has a gun or another type of ranged weapon with which he can kill other players in the game. The action in an FPS is typically very fast-paced and players change position, orientation and behavior very rapidly and unpredictably. This type of interaction requires a large amount of updates per second and a small latency, as even a moderate delay can mean the difference between life and death.

The usual implementation approach for this kind of game is through Frequent State Regeneration [86]: every user sends his keyboard and mouse input a number of times per second. This

information is sent to a server which calculates the new world state and then sends updates to all players. These updates also need to be very high in frequency to support the FPS's fast action and can range from 30 to 100 updates per second, depending on the game [89].
An example of an FPS is CounterStrike. In a study of this game [88], the researchers found that there were on average 437 incoming packets and 360 outgoing packets per second for a single server of 22 players. This is a very high amount of packets per second for that small amount of players. Furthermore, all these packets were relatively small. As we will see later, every packet introduces extra overhead and extra bandwidth usage, so this model is not optimal where bandwidth is concerned.

Lag compensation

In this kind of fast-paced game where a fraction of a second can make a difference, large delays on the network can be detrimental to the gameplay experience. Because of this, advanced lag compensation algorithms are needed.

One possibility is to use an interpolation strategy [58]. In this method, an extra 100ms of artificial lag is added to every incoming packet before it is processed and the results are shown on screen. This ensures that there are always at least two updates between which the client can interpolate the world state, which reduces the effects of jitter. However, this also means that what every user sees on his screen is always a delayed game state. This would mean if they shoot a player, they would never be able to hit them if they shot directly at them. To remedy this, an approach called time-rollback is used to calculate if a bullet hits another player or not.
A second possibility is to use extrapolation instead of interpolation [63]. Here we do not wait until enough data has been received but predict how the state will evolve in the future (see chapter 4 for more details). Here, the local world state will be much more accurate, but users will have to lead their aim to be able to hit others because there is no time rollback. This type of system is also more susceptible to lag and jitter.

Even with these advanced techniques and high update rates, there will still be artefacts in the world state which can lead to illogical world states, such as players walking through obstacles or dead players killing someone. The techniques aim at limiting these artefacts as much as possible however, as they would be much more serious without these systems.

NVE approach

The typical bandwidth usage for FPS games is relatively modest for up to 32 players, but larger numbers will increase the size of update messages as more players mean more changes to the world. But even with 32 players, the bandwidth usage per player is still much higher

than that of existing MMORPGs, a popular type of NVE. The counterstrike example sends approximately 60 packets per second of about 130 bytes to each player. World of Warcraft sends about 6 packets per second at 71 bytes per packet to each player [88]. This means that even with a low amount of players, FPS games use more bandwidth than MMORPGs with much larger numbers of players.
This can lead us to the conclusion that to bring FPS games into an NVE setting, bandwidth scalability is a very important requirement to be able to support more players. Furthermore, latency compensation stays very important and is difficult even with very high update rates, as discussed before. Should we want to limit the bandwidth usage by reducing the frequency of the updates, the consistency might suffer considerably, making it a good example of the throughput/consistency tradeoff.

2.3.2 Real Time Strategy game (RTS)

In real time strategy games, users typically govern a large number of different units as a general and then send these units into battle versus other large groups of units. As these games were already being made at the beginning of the internet, with very slow connections, extensive work has been done in the past to be able to make a simulation of this scale possible. This is because it is impossible to send the position updates for every single unit, as this would require a huge amount of bandwidth, which was and arguably is not available on the internet. The solution developed for the early RTSs was that of deterministic simulations [59]. The idea is that the engine will react exactly the same way if the same input is given at the exact same time. This means that we only need to send the player's commands (mostly just mouse clicks) across the network. If these commands are then delayed a little bit at the sender's end until the other players can also execute them at the same time, the simulation will remain consistent. This way, an RTS becomes very optimal for bandwidth usage. This is possible for this kind of game because the interactions are not supposed to be real-time, as is the case with the previously discussed FPS games. Thus the amount of commands the players give are relatively low in frequency and delays up to 600ms are not noticed by the players [59].

However, this kind of system certainly has its drawbacks. First of all, all network traffic has to be reliable because all actions need to be executed. Otherwhise, some player's simulation would start to differ from that of his fellow players which could accumulate into large errors. This is because there is never any communication other than the commands the players give, so even little roundoff errors can accumulate over time and never be restored.
The second problem is that it does not allow players to join when a game is in progress. This is because the new player would first need to receive the current state of the world, which can be enormous if there are a lot of units present in the world. Then, he would have to synchronize with the game clock to be able to continue the simulation. In theory, it would

be possible to implement late user joining, but in practice this is very difficult and would require a lot of extra work and the starting state transfer would require extra methods and large amounts of bandwidth [59].

NVE approach

The deterministic engine solution is typically only suited for a limited amount of users as all users have to be synchronized perfectly with eachother, which can be difficult with a large amount of users. Also, if one user out of 100 would have a very bad latency, the other 99 players would suffer from this as they need to wait for the slowest player to process all commands. Furthermore, the small roundoff errors resulting in slightly different local states might not be a problem in limited-time RTS matches, but they can be in a large persistent world. These problems in combination with the fact that no users can join once a game is in progress and that all traffic has to be reliable makes it less fit for an RTS in an NVE setting.

This would suggest that for a large scale RTS, we might need to look to the more traditional approach of frequent state regeneration as is used for FPS games for instance. We said that this was not possible when RTS games were first made because of the limited bandwidth, but the bandwidth available to users has increased considerably since then. It is not so that we can just synchronize every unit at a high update rate, but maybe it is possible trough the usage of some smart optimizations to make an MMORTS without using a deterministic engine. This kind of system would certainly use more bandwidth than the normal approach for RTS games, but if the bandwidth usage can be controlled, the approach could be viable. Findings in chapters 5 and 6 support this possibility.

2.3.3 MMORPG

One of our main points we are trying to make is that MMOFPSs and MMORTSs are not yet omnipresent because, among other reasons, they have a high bandwidth usage. However, there are other types of MMOs that are possible today and they are very popular, namely MMO Role Playing Games like World of Warcraft [53], Guildwars [21] and Eve-Online [13].

This type of game generally does not require a large amount of traffic and also the real-time requirements are not that high as is the case with for example an FPS. Six update-messages per second can suffice for a normal MMORPG [88], which we can call Infrequent State Regeneration. Fight-actions are often delayed somewhat and do not take place in real-time on the screen like it is the case with FPSs, giving the servers some breathing room to schedule all calculations.
Furthermore, the existing systems often use a heavy simplification of the one-large-world ideal. World of Warcraft for instance uses sharding [53]: a number of independent, parallell

worlds, each of which supports only a limited number of players. Eve-Online claims to offer one large virtual world but instead uses separate solar systems between which the players have to travel. Recently they also had to apply a player maximum per solar system as the load was too high in the most popular regions [13].

The fact that MMORPGs are possible already shows us that the bandwidth usage depends on the type of NVE we are trying to build. The kind of interaction model used in the NVE will determine how frequent updates have to be sent and also how many objects have to be synchronized. MMORPGs have relatively low amounts of objects (typically only 1 or 2 per player) and have low real-time requirements. This allows them to keep their bandwidth usage relatively low while still providing an interesting gameplay experience to the users.

2.3.4 Physics simulations

Now that we have discussed three different types of game, there are two more concepts that can be present in multiple types of environments, namely physics, discussed in this section and collaboration, which is explained in the next section.
When simulating a realistic world in a computer game or application, realistic looking physics on object are often important to help the users immerse themselves in the world and make it look more real and acceptable. For this we can use actual known physics formulas and effects like gravity, acceleration and other forces. These systems exist but typically require a lot of computations to deliver good results.

In networked games, physics is often used as eye-candy and does not have much influence on gameplay because it is difficult to do truly distributed physics for a couple of reasons. Distributing physics basically comes down to keeping the positions of every object physics has an influence on equal for all users. As has been discussed before and will be discussed later, this consistency problem is not trivial, but mostly it is possible because we only need to keep a limited number of objects consistent. With physics, we would suddenly need to synchronize a lot more different objects, as most complex physics interactions will occur on small objects and each will have a different behaviour. More objects also means a much higher bandwidth usage. In this respect it looks a lot like the RTS games discussed before. Physics can be implemented using a (semi-)deterministic engine (the game Burnout Paradise does this for example [88]) so not all objects need to be sent. But if we look at physics in the context of FPS or collaboration (see further) for instance, real-time consistency is extremely important and this cannot be achieved using a deterministic engine.

Once again as with RTS, we might need to fall back to a send-all-objects/frequent-state-regeneration approach and combine this with advanced prediction and grouping techniques.

In chapter 6 we will discuss our own approach for distributing physics in a large-scale environment. Distributed physics is a very active and important area of investigation.

2.3.5 Collaboration

In RTS, FPS and other games, players do not need to send a lot of info to the server themselves. It is mostly just the position of mouse and keyboard strokes. For truly collaborative and interactive environments, this data can be much larger and more elaborate. For instance, in modern systems there can be different kinds of input devices like force-feedback machines, multitouch tables, virtual reality, etc. This kind of input device will typically output a lot more data than a normal mouse and keyboard setup and thus to send this output to other users will require more bandwidth.
A second aspect is that collaborative environments will probably use audio and/or video communication between users as this can help these users better accomplish their tasks together. These tasks can often include the real-time manipulation of 3D objects or the sending of large files between different users, both requiring large amounts of traffic.

Collaborative environments will typically have different goals then games that are meant for leisure activities and thus they will also produced different kinds of network traffic. This means that they might also require different bandwidth scalability techniques. In chapter 5 and 6, we will discuss these differences further and explain how a single technique for bandwidth scalability can be used in both collaborative environments and games.

2.3.6 Conclusions

The bandwidth usage of an NVE greatly depends on the type of game or application and the type of interaction it uses. Non real-time interactions generally require less bandwidth than real-time applications and true interaction has its price. In order to bring existing small-scale games and concepts to an NVE setting, bandwidth scalability is probably needed. It is even possible that the large bandwidth requirement is one of the reasons why these games have not yet seen large breakthroughs in the MMO market.

2.4 Network Streams

This section discusses the kinds of network streams that are typical for a networked game or NVE. The goal is primarily to identify those streams that use the most bandwidth. This can be easily described by two parameters: average packet size for a stream and the frequency by which the packets are sent. A third parameter we will discuss is the real-time requirement of every stream, as this is important in later sections.

We should also note that every stream will have additional overhead depending on how many packets it sends. There is a certain amount of network-induced overhead per packet that should not be ignored. This aspect is discussed in more detail in the section on aggregation in chapter 3.

2.4.1 User actions

This is the stream to communicate which actions the user is doing to control his avatar(s) and manipulate the world. The way these actions are communicated depends heavily on the type of NVE or game.
One possibilities is to send the position and orientation of the avatar(s) as they are changed by the user (mostly XYZ-coordinates etc.). Another way is to send which keys the user is currently holding on his keyboard and what he is doing with his mouse (FPS). A third way is to only send info if specific actions occur (i.e. a user clicks on something) (RTS). The size of these action packets can vary greatly, depending on the chosen representation.
The frequency of the packets can also differ greatly between interaction models. As discussed in the previous section, RTSs only require a few clicks to be sent every few seconds, RPG's can perform well with around three updates per second [88], while FPSs often require high update rates. These frequency requirements are tightly coupled to the real-time requirements of the interaction model of the NVE in question.

2.4.2 World state

The state of the world is what should be consistent across users and this often requires more than sending user actions to other users. Although in some cases they are very alike, in other cases the separation between user actions and world state is very important, as we will see later.

Object and avatar state

The actions of a user can have serious consequences on the state of the world and other objects in this world. The position and other properties of objects and avatars (like animation state) can change, and these changes have to be sent to the other interested users. The amount of users in the world and the impact of the action determines how much has changed and thus also how much data has to be sent. For instance, a world in which every object is simulated by physics will have a lot more changing positions and orientations for objects than in a mostly static world where only the avatars move and objects stay more or less stationary.
The frequency of the state updates once again depends on the type of NVE and the real-time requirements, like with user actions. Generally speaking though, the update rate should be high to retain maximum consistency.

Full State Transfer

If a new users joins the world, he has to receive the current state. If there are a lot of objects and users present, there can be a huge amount of data to be sent to enable the user to join the world and see all the correct state.
Luckily, this full state transfer only happens very infrequently. Besides the case of a new user joining the world, some NVEs use a subdivision of the world into separate regions (see chapter 4). Then full state transfer is also needed if a user changes regions, but this is still relatively infrequent. Another upside is that the per-region state that needs to be sent is much smaller than the complete world state.
This kind of data is mostly not extremely real-time and often the user can join the world and perform some actions while not all of the state has been transferred.

New content

A lot of modern NVEs give their users the possibility to add new content to the world themselves. Other NVEs are expanded by their creators to add new objects and regions. All this new content has to be downloaded by the other users so they can see it. This new content can be a large variety of things, but most often they are textures and new 3D meshes. This content can be very large in size and quantity.
This kind of new content only has to be sent if the user does not know of it yet and thus it is often considered a part of a full state transfer upon entering the world or changing a region. In any case, also this type of traffic can be considered to be infrequent.
The new content is often purely visual in nature and thus not very time sensitive.

2.4.3 Audio/Video

Some research has been done into the usage of audio and video communication in NVEs and it is possible that these forms of interaction add to the immersive feeling of users and aid them in communicating with others [83].
Audio and video are huge bandwidth consumers. The data is essentially captured multiple times per second (10 - 30 for video, about 20000 samples per second for audio) and especially in the case of video this data is very large.
In addition, this data is very realtime sensitive. Several protocols such as RTP [70] have been developed to ensure the transmission of audio/video across an IP network with a focus on retaining these realtime properties.
These factors make audio/video streams the biggest and most fragile users of bandwidth in the NVE setting.

2.4.4 Low-bandwidth streams

The previously discussed streams often have medium to very high bandwidth requirements. There are other streams in an NVE which are very infrequent or do not use a lot of bandwidth.

Control info

Often NVEs use extra control info besides the world state and user actions to govern the virtual world. We can think about login procedures, network status info etc. . This control information is often small and infrequent. Mostly the info is time sensitive though, as the entities need the info to make their decisions about what to do next or how to do it.

Chat messages

Besides audio and video communication, users often communicate with each other through the use of small chat messages. These message consist of plain text and are often quite infrequent. Also, they are not extremely realtime and delays of a few seconds can be acceptable.

2.4.5 Case-study: Managing stream bandwidth with the NIProxy

As we have discussed here, there are many different types of network streams active at the same time in a NVE. It is therefore needed to have a robust system for managing these streams and, in our case, their bandwidth usage in particular. One existing framework for doing this is the Network Intelligence Proxy (NIProxy) [94], an academic research project led by dr. Maarten Wijnants. The main goal of the NIProxy is to provide network traffic shaping possibilities and it offers a means to do this by using multimedia services like on-the-fly video transcoding. The NIProxy takes the form of a non-transparent server in the network and provides an API through which applications can indicate their specific needs. By doing so, the proxy can adapt the network streams on a per-user/per-application basis, can make the network more intelligent and can in the first place regulate bandwidth usage so networks do not become overloaded.

Examples

A good example to explain the basic idea behind the NIProxy is to think of a central live video streaming server that is streaming high-quality video to its subscribers. When a user with a mobile phone wants to watch this video stream on a mobile network, it is very unlikely that this will be possible if he subscribes directly to the high-quality video of the server. This means that the server would have to send multiple streams, each with a different quality and resolution, to enable mobile users to watch the video stream. However, this requires additional processing and bandwidth overhead for the central server, which will probably already have a considerable workload if it is serving popular content.

This is where the NIProxy comes in. This extra proxy server is placed between the mobile user and the video server. The video server can just continue to stream the one high-quality stream, which is intercepted at the NIProxy. This way, the NIProxy can perform the transcoding of the video to a lower quality for the mobile user before sending the video over the slow mobile network. Now the central server has to perform less calculations and the workload is distributed over a number of different proxy servers.

Whilst this example was for just one video stream, the NIProxy can shape all kinds of network streams and possibly service many users and many applications at the same time with a single machine. Experiments have been performed in which multiple audio and video streams were sent to a single user at once, where the NIProxy had to dynamically determine which quality the different streams should have to stay within the bandwidth limits [94]. Besides video transcoding, the NIProxy has also provided other services, like one which adds Forward Error Correction to a multimedia stream to improve stream reception quality in error-prone networks and hence user Quality of Experience [92].
Other earlier experiments were aimed at NVEs that have to distribute a large amount of geometry (like for instance Second Life, as discussed in chapter 5). The general idea was that the geometry of nearby entities would have to be transmitted prior to that of more distant entities, because the former are likely to be much more important for the user and because this approach will yield the highest visual accuracy. In addition, it was determined that these further away entities could use an alternative, image-based representation with a much smaller transmission size instead of the full geometry. These choices were based on a simple circular AOI model. This way, the NIProxy could determine which entities to send and at which representational accuracy, while still making sure the bandwidth limits were respected [91].

Bandwidth shaping trees

Internally, the NIProxy works with a tree-based architecture to structure bandwidth distribution techniques in a so-called stream hierarchy [93]. Let us say we have a single tree for a single user or network connection. The top node of the tree is given the total amount of available bandwidth to distribute over its children. The leaf nodes, lowest in the tree, represent the actual network streams we want to manage, for instance a number of individual audio and video streams. In between the root and leaf nodes, we have the internal nodes which implement the actual bandwidth distribution strategies.
There are a number of different types of internal nodes implemented and many more are possible. For instance, the mutex node chooses just one of its children and only this child gets bandwidth. This can be interesting in cases where both a low-quality and high-quality version of a single stream are provided, yet only one version should be delivered at a time. A

priority node will grant the child with the highest priority all bandwidth it needs and only if there is bandwidth left after this first node, the next child is given this residual amount. A percentage node has a percentage for each child and each child gets that percentage of the bandwidth allocated to the internal node to use.
Using these basic node types, complex trees can be built for bandwidth distribution where multiple internal nodes are placed after one another, allowing the combination of different strategies. An example of such a tree can be seen in figure 2.2.

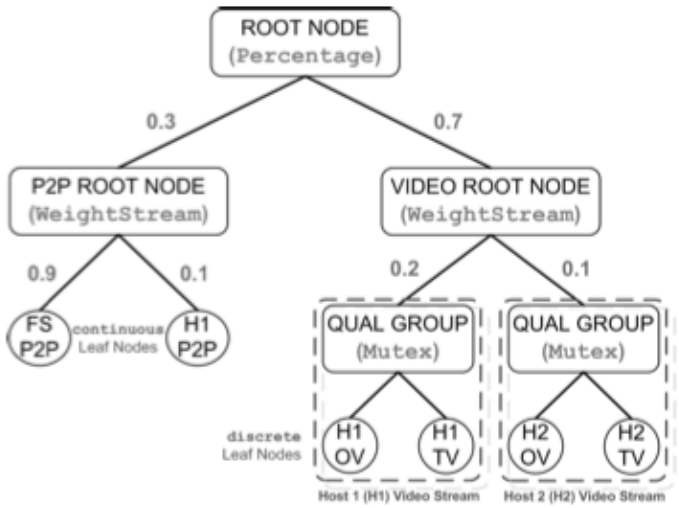

Figure 2.2: Example of an NIProxy tree [93].

This stream hierarchy representation allows for a very dynamic system. If at a time the network connection would have fluctuating bandwidth capacity, only the root node has to be notified, after which the entire hierarchy makes sure that the new bandwidth constraints are being respected. We can also adjust the percentages, weights and priorities for specific child nodes of internal nodes to change the importance of a certain network stream at runtime. Nodes can also be added or removed from the stream hierarchy at runtime.
One of the most important characteristics of the NIProxy is that the multimedia services it provides are coupled to this traffic shaping mechanism. This means those services can change the tree by adding or removing nodes and changing parameters. The stream hierarchy in its turn is able to inform the service about how much bandwidth the stream it manages is allowed to consume, so the service can for instance switch to a lower quality representation when needed. This two-way communication between services and bandwidth constraint management means that the bandwidth is always kept within limits, but that the streams can be kept at the highest possible quality for the user.

2.4.6 Conclusion

Audio/video and world state are the most bandwidth consuming streams we need to deal with. Luckily, world state is not very real-time sensitive, so the transfer can be spread out in time, delaying parts of the update to ensure there is bandwidth available for other, real-time traffic while the world state is being transmitted. In any case, the NVE should be able to deal with these sudden peaks in bandwidth requirements for specific users as they join the world or change region.

Many different streams are active at the same time in a NVE and a system for managing them is necessary. In our implementation, we will use the NIProxy to provide this stream management. This is discussed in more detail in chapter 6.

2.5 Programming Idioms

Downsizing bandwidth also has to do with how the programmer chooses to practically approach the synchronization of different objects across the different users. In the abstract, we want to make an exact local replica of every object in the world on every user's pc. This can be done in a number of different ways.

2.5.1 RPC, Replicas and attribute synchronization

Arguably easiest way to make a distributed application is to think of the world as a single, large distributed database. Every user has local copies of (a subset of) the objects in the world and manipulates these objects locally. Whenever a function-call is made to a local object, this function-call is automatically propagated to the other users holding a copy/replica of the object, ensuring maximum consistency. This could be done using a Remote Procedure Call (RPC) system. This type of system will serialize every function call made to the object across the network. This can be done in two different ways: blocking and non-blocking. In blocking mode, the original caller of the method will wait until the method has been called on all other objects as well. This ensures complete consistency, but is not usable for real time systems. Non-blocking RPCs will allow the application to continue as usual, but can lead to inconsistency.
While this kind of system generally works very well for critical, non-realtime sensitive systems like applications for banks or large corporations (where parameter type checking for instance is very important), the method is not directly usable for games or realtime interactions. It is very unpredictable when data will be sent and the system always requires every parameter of the function call to be sent, often containing unnecessary information.

A variation of this technique is not to send a message whenever a function is called on an

object, but rather only send a message when an attribute of the object changes. For instance, when the position of an avatar changes, the system automatically detects this change and wraps it in a packet to be sent to the other users. This type of state propagation is sometimes called attribute synchronization and is a popular networking model for several small scale systems, as we will discuss in chapter 5.
The main problem with this approach is once more that it can be unpredictable about when packets have to be sent and how these packets are precisely composed, i.e. what data is being sent when. This is especially the case when we would use an existing library or middleware which tries to shield us as much from the networking aspects as possible to make it easier to focus on the gameplay logic. Whilst this might be a good choice for smaller games, for NVE development we will typically need full control about what is sent and when, and RPC and attribute synchronization approaches are mostly not sufficient. Custom message passing on the other hand, is more flexible.

2.5.2 Message passing

As we said in the previous section, it is important to determine what data is being sent at what time to effectively manage bandwidth usage. And so, instead of sending an update every time an object has changed, we can determine ourselves when to send updates and more importantly: in which way to do so.
The message passing approach means that we will create our own set of messages to be sent over the network. These will often contain some properties of objects, but not all, and will be able to group different objects together into one packet. This approach also enables a high-level separation of packets into streams of different kinds of data. As this method is so versatile, it is probably the best choice for a large-scale NVE framework [71].

However, this method requires a lot more work than the other described approaches and is also not generic: it needs to be created and fine-tuned for every application specific. To still provide a good level of maintainability for the programmers, it is often coupled with code generation facilities, which allow the programmer to describe the specific packet structures and protocols in scripts or xml files. These scripts are then parsed into real program code.
In comparison with the previous discussed methods, message passing requires more work but is also more versatile and offers more direct control over the data that is being sent, which is often interesting when we are trying to limit the bandwidth usage.

2.5.3 Command synchronization

A method that is tightly coupled to a deterministic engine (see previous sections on RTS games) is command synchronization. Because the engine is deterministic, there is no need to send the state of the objects themselves, only to send the user's actions, which will have the

exact same effect on every local copy.
This approach is quite different from the aforementioned methods as it does not synchronize the objects directly.

Chapter 3

Protocol Optimization

In order to reduce the bandwidth used by the NVE, we roughly have two possible approaches: we can reduce the size of the data we send or we can send less packets. Figure 3.1 shows how these different techniques relate to each other. This chapter deals with the first possibility (compression and aggregation), whilst the next chapter is about the second (filtering).

Figure 3.1: Three different techniques for bandwidth reduction.

As has been discussed before, every user in the NVE has a local copy of (a part of) the world. Any change to this world by a certain user has to be communicated to his fellow users and

this communication occurs in the form of network packets. These packets have a certain size, expressed in bits or bytes (where 1 byte is 8 bits). The number of bits needed to communicate a certain change to the world can vary greatly, depending on the type of change but also on the representation of the change chosen by the programmer.
This chapter discusses two techniques to reduce the bandwidth usage without actually reducing the packet frequency: compression and aggregation.

3.1 Compression

A lot of techniques are called compression and a lot of different subdivisions have been made. We distinguish three different types, based on when and how they can be used.
In general, compression will try to reduce the amount of bits it takes to represent a given dataset, like a network packet. There are a few different techniques that either try to reduce the size by some knowledge about the high-level meaning of the data (for instance which values a variable can take), by a generic idea that works on a certain low-level structure of the data or by using previously known data. The next paragraphs will discuss these three different types.

3.1.1 Logical compression

The first category of compression techniques is logical compression. This name has nothing to do with logic as in mathematics. With logical we mean to say that most of the methods and examples discussed in this section are obvious to most programmers of online games and worlds. It is logical that they would build their communications this way and often these ideas are even dearly needed to get a reasonable bandwidth usage in the first place. therefore they are mostly the cornerstone of the way changes in the world are communicated and I will discuss many examples of how logical or obvious compression is done.

Please remark that one might say some techniques that will be discussed in chapter 4 are also obvious and logical and are used by almost all games. Whilst this is true, those techniques often filter packets that are already "compressed" by using the methods discussed here and thus actually drop packets with superfluous information.

Send only what has changed

This concept directly shows how obvious and logical compression can be. It is indeed simple to see that an update should only be sent for an entity if it has changed since the last update. A very simple implementation could be to send the full state of an object 30 times a second, whether it has changed or not. This is indeed very easy for the programmer as he does not have to check if anything has changed, but it can lead to a huge amount of unnecessary traffic

for entities that remain stationary and for which it is not necessary to send any updates at all. Furthermore, if we do detect a change and send an update, only the changed data should be in the update, not all the data for the entity. For instance, if an object moves but does not rotate, only the position and not the rotation should be sent.

The main problem with this approach is that in most modern engines, objects can be very complicated, consisting of multiple components, variables and parameters. If we are to send an update only if something has changed, we need to have a way to detect if something has indeed changed and what that change was. With such a complex object, this can be quite the challenge. This is further complicated by the fact that this tracking of changes has to be done per receiving user. As discussed later, it is possible that not all currently receiving users have received all previous updates, so we need to know per user what was their last received update and what exactly has changed. This can cost memory and processing power.

The example of Object Views [75] shows how this approach can be practically implemented. Here, every object has an extra view coupled to it for every user that has an interest in the object. This view keeps track of the changes for the object and determines when to send an update to an interested user and what data should be in this update. On the server, every game object has several object views, one for every player, see figure 3.2. This way, the server can track which was the last update of the object a specific player has received and thus also what kind of new update should be sent at any time. This is even more interesting when we consider more advanced approaches like level-of-detail or dead reckoning (both discussed in chapter 4), where only select updates are sent to the users.

Know your bits and bytes

When describing things like position, orientation, speed, color etc. on a computer system, this is done using so called primitive data-types like integers, floats, strings etc. Each of these types has a certain size in bits. A typical integer is 32-bits to 64-bits, a large floating point number (double) is mostly 64-bits and a string uses 7 or 8 bits per character. These sizes are mostly fixed in the programming language as to enable a wide array of possible values for the programmer. A 32-bit integer for instance can represent every value between -2147483648 and +2147483647. It is very hard to imagine a speed or orientation of an object spanning that complete interval though. Mostly, that kind of data can be represented with far less bits.

When packing data into network packets, the programmer needs to choose the smallest possible number of bits to represent that value to limit the bandwidth usage. This means we need to use binary packets as opposed to textual messages [5]. Some important protocols that are used on the internet, like HTTP, use textual communication to transfer data from and to users. This means that this data is generally in a human-readable form and accom-

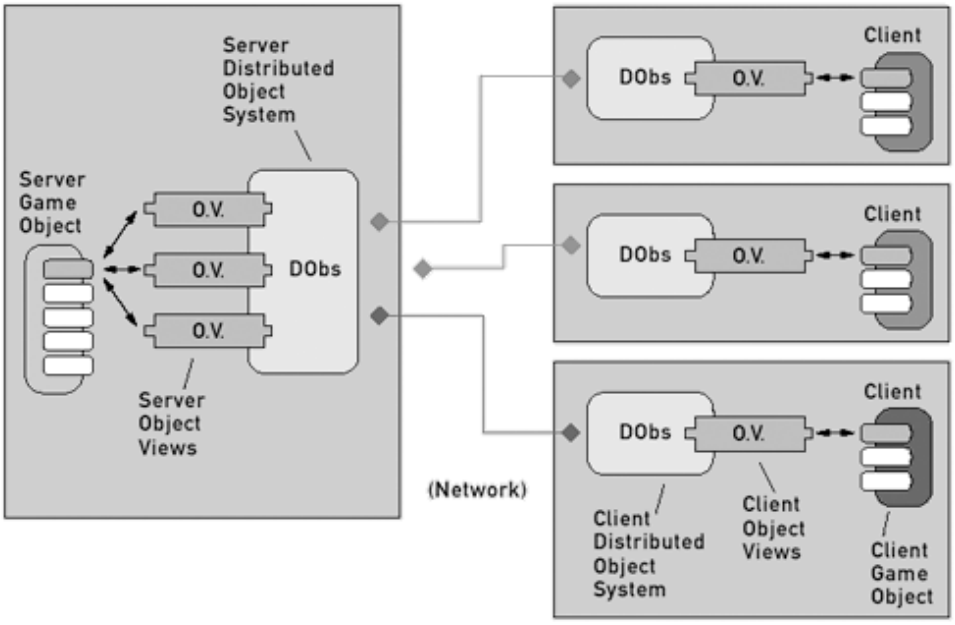

Figure 3.2: Object views can serve as a networking interface that tracks the changes for an object on a per-player basis [75].

panied by extra data that indicates it's structure, as is the case in the XML format [88]. However, this means that all data is encoded as strings, which will often increase the size of the communication when compared to the absolute minimum binary size needed to send the same information. Also, the added structure indicating data can be a large overhead which is generally not important for non realtime protocols like HTTP, but which can be a big problem for NVEs. This is why we have to try to encode the variables we send over the network as efficiently as possible in bit notations.

A simple algorithm to determine the minimal size is to look in what interval the possible values can lie, and only use just enough bits to represent that interval. For instance, a boolean variable can have only 2 values: [0,1]. This means a single bit suffices to represent a boolean in a network packet.
Another example is a rotation around the vertical axis. This rotation is in the interval [0,360]. When we look at the binary system, the closest amount of bits needed to represent 360 different values is 9 bits (as 8 bits can hold 256 values and 9 can hold 512). So when sending the rotation across the network we only really need 9 bits, even though 32 bits are used in the program itself. So while large and fixed size primitive data types are interesting from a programming and computing speed perspective, they are not always needed or fit for transmission over a network.

A special case are floating point values. When the rotation is not limited to integers but can take values in between two discrete values (for instance a rotation of 30.678 degrees), 9 bits will of course not suffice to send all possible rotations. When large accuracy is needed, we have to use more bits to send the rotation. But when this level of accuracy is not necessary, a round-off of the value to its nearest integer value is a very viable technique to reduce the number of bits needed. As we will see later, a distant object in the world will not occupy a large space on the screen, and a difference between 30.678 and 31 or even 35 degrees will not be noticeable to the user. In those circumstances, the precision can be lowered to allow for bandwidth reduction, even if the original values at the computer of the object's owner are very accurate.

A final problematic datatype is a string. The main problem with strings is that they mostly do not have a fixed size. If we would want to send the nickname of the user with every update for instance, so that the receivers would know who sent the packet, it would be unwise to send the nickname in its string-form every time. Some players will have small nicknames, but others may have very large ones, which would lead to a constant larger size of the latter group's packets. A better way to tackle this problem can be applied when the strings are fixed throughout the rest of the game, like is the case with nicknames for instance. When a user first joins the game, a unique user-ID is generated for this user (for instance a 32-bit integer can be used for this). Then this id is distributed together with the nickname to all interested users only once. From then on, every user knows the nickname for every user-id, so the sender can just send his 32-bit user-ID instead of his potentially much larger nickname. While this requires an extra packet to be sent (the user-ID/nickname mapping), in the long run this method is far more efficient [61].
When you must send strings, for example when allowing textual in-game chat, there are still different ways this can be done. Since a string can be of arbitrary length, we must have a way to know how long it is. A possibility is to prepend the length of the string (for instance as a 32-bit integer) so we know how many bits are actually used. Another possibility is to use a special character at the end of the string so that when parsing we know that when we reach that character, the string is at an end. The special character '\0' is often used for these purposes. Note that the latter method only uses 7 or 8 bits, whereas the former uses 32 or sometimes 16 bits, but requires a larger processing overhead as the characters have to be read one by one and evaluated to check for the special character.
In the next section about generic compression we will discuss other methods for compressing strings that can reduce the 7-bit-per-character requirement.

It is sometimes said that modern programmers waste the computer's resources. They no longer think about the data types they are using or how much memory their program consumes when programming for medium to high-end pc systems. For example, programming languages like

Java store the length of each String-instance as a 32-bit integer, as opposed to the 7-bit '\0' alternative of C and C++. So in a way, optimizing the network protocol is a bit like optimizing a program for use in a mobile or console environment. Modern systems like the iPhone, PSP or even the PS3 and XBOX360 do not just have huge amounts of memory at their disposal, and programmers need to be smart about how they store their data to prevent memory shortage. The same logic can be applied to network bandwidth, as this is also in small supply, and programmers need to be aware of this fact and take every possible action to find the smallest possible representation of the data. It is often the case that these systems with lower memory possibilities also have lower bandwidth at their disposal (this is especially true for devices like mobile phones using a wide-area wireless network). This gives the programmer the possibility to tackle two problems at a time: when the memory usage for all the variables is already fully optimized on the local device, one can simply take this representation and send it over the network, without having to copy it to a lower number of bits or perform precision cutoffs.

3.1.2 Generic compression

In the previous section we discussed obvious and logical ways of compressing data. These methods often have to do with what kinds of data are being sent and in what context. The amount of bits needed to represent a value depends on the possible interval that value can be in, precision depends on the context the data is used in, etc.
The methods discussed in this section are more generic, i.e. they work on any kind of data and do not require explicit context to be able to compress the data. If this can be done without losing data in the process, it is called entropy encoding [88]. This makes them very interesting for compressing data at a low-level. These methods can be applied to almost any packet or data you mean to send through, without having to know what kind of data is being sent or what values that data can represent.
Because they are so generic, they are often available in existing (open source) software libraries and implementations, and thus the production costs of using such a method should be very low.

Run-length encoding

Run-length encoding [43] is a very simple type of generic encoding in which runs of data (that is, sequences in which the same data value occurs in many consecutive data elements) are stored as a single data value and count, rather than as the original run. For instance, the string WWWWWWWWWW contains the character W 10 times. If we store a character with 7 bits, this string takes up 70 bits. We can run-length encode this string as 10W, indicating that the value W is repeated 10 times. This only takes up $32 + 7 = 41$ bits.
Another example is when one wants to encode an image which has large areas of the same

color. When there are a large number of black pixels next to each other, the image can be stored efficiently by using run-length encoding.

It is obvious that this encoding only works well if there are large consecutive sequences of the same data. If this is not the case, this method will actually yield larger data sizes. For instance, the string ABCD would take up 7 x 4 = 28 bits normally, but encoded with RLE it would be 1A1B1C1D, yielding 7 x 4 + 32 x 4 = 156 bits, which is more than 5 times the original size.

Huffman encoding

Huffman encoding [23] is another relatively simple algorithm that uses frequency of appearance of separate data values in the total data. Rather then relying on consecutive occurrences of the same data like run-length encoding, huffman encoding tries to encode the data values that appear most in the data with the least amount of bits.

In the example string ababacacacad, the letter 'a' is obviously the one that occurs the most. If we can make it so that the letter 'a' is stored with as little bits as possible, we will have a large profit in size. The way the huffman algorithm works is by making a so-called huffman tree to deduce the binary codes for the different data values. The algorithm for building this tree from a given dataset is given in algorithm 1. To determine the codes for the elements, we simply follow the path from the root of the tree to the element, appending the 0's and 1's during the tree traversal. As we can see from the example in figure 3.3, the most frequent element, a, is encoded with only 1 bit and the longest codes are for the rarest elements in the input string.

Algorithm 1 The huffman algorithm

1. Create a leaf node for each symbol with cost = frequency and add it to a list

while there is at least one element in the list **do**

2.1. Remove the two nodes with the least cost from the list

2.2. Create a new node with these nodes as children and cost = cost1 + cost2

2.3. Add the new node to the list

end while

3. The remaining node is the root of the tree

4. Follow the tree branches down to a symbol, choosing 0 for left branches and 1 for right branches, to calculate the code for that symbol

Once this tree is built and the codes for all the elements are known, we can encode the data by replacing all elements by their codes. The decoding is a little more difficult. First of all it is important to notice that none of the smaller codes is a prefix of the larger codes. This is

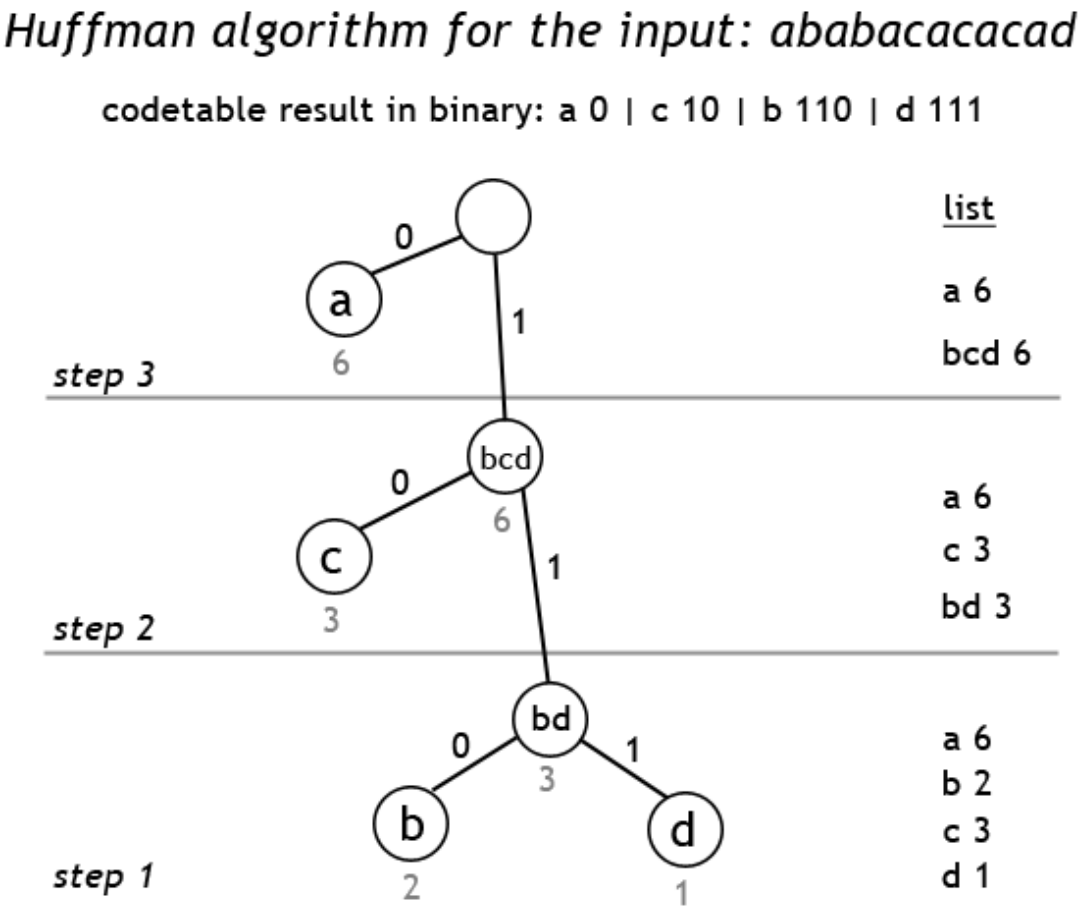

Figure 3.3: Using huffman encoding to encode a string

vital for the decoding to work because otherwise it would be impossible to determine whether a code is complete without adding extra information at the end of each code. Secondly, the decoder needs to know the code-table to know which code is to be translated into which element. This means the code-table either has to be sent along with the data (meaning extra data has to be sent, which can sometimes be quite large) or that we must work with a known and predetermined code-table at both sides.

This last method is especially interesting for text messages in a particular language. One could determine the statistical frequencies of the letters of the English alphabet by analyzing a large number of English texts. Then a general code-table containing the letters could be devised, which could then be re-used when encoding a text. This saves the processing time of building the tree and code-table at run time, but it is possible that this method will not yield the best results. This is the case when a text contains a larger quantity of an otherwise assumed rare character, which is then encoded with more bits then if we would have created the huffman tree for this specific text. Other more advanced methods incrementally update

the code-table at both sides [78], increasing processing requirements whilst possibly providing a better and more flexible compression.

Using huffman encoding, the 12-character example string can be encoded using only 21 bits (see figure 3.3), in contrast to the original 12 x 7 = 84 bits. As the amount of data increases and the elements become more frequent, the measure of compression will increase. Thus this method is good for very large quantities of data with repeating, known elements. Small amounts of data with highly variable data values will be encoded far from optimally, leading to a larger packet size. In general, these algorithms are used primarily when encoding large binary files, textures or images and other large chunks of data. They are generally less interesting to use on single network packets at a time, as they will more often increase the size than decrease it.

Advanced algorithms

The two discussed algorithms are only two examples of a large collection of compression algorithms which have been invented over the years. More advanced algorithms use complicated mathematical constructions or decoding structures to reduce the size even further. Popular compression formats like .zip or .rar use a form of the Lempel-Ziv compression [28]. This type of algorithm uses a library paradigm to replace large chunks of data with their library key. Arithmetic encoding [3] uses a mathematical construct to encode an entire string into one big number. As with the other algorithms, these more advanced algorithms work best on large chunks of data and in many cases prior assumptions about data distribution or structure are made to encode the data.

3.1.3 Delta compression

In literature, the term delta compression is sometimes used to denote a concept that has been discussed in the section about logical compression: only send when something changes and only send what has changed. This makes sense if you know that the word “delta” is often used to indicate the amount of change of a certain variable. The main difference with this previous concept and the current section is that the logically-compressed updates do not necessarily need a previous update to be able to reconstruct the remote state locally. In contrast, the method we call delta compression uses previously received updates as the base for reconstructing the state. The received updates do not contain enough data to be meaningful, but combined with the previous updates the state can be reconstructed completely. This method is sometimes called incremental or differential encoding.

This concept can easily be illustrated for position updates. Imagine an object at location (100,100) in the world at time t. At time t + 1, it has moved to (105,105), so an update

has to be sent. We can now send only the difference between the current and the previous position, namely (5,5), in the update. If the receiver knew the object was at (100,100) before, he can easily reconstruct the correct position of (105,105) from the update, even though it was not completely sent. Now why is this important? As discussed in the section on logical compression, small numbers can be sent using a much smaller amount of bits than large numbers. While we would need at least 7 bits to send the value 105, 3 bits suffice to send the value 5. This might not be a huge difference for these simple examples, but large virtual worlds can be very big, needing 32 or even 64 bits to indicate a position in world-coordinates. The difference between 3 or 32 bits is more than significant, especially for very frequent position updates.

The reliability problem

The main problem with this approach is that the recipient of the differentially encoded information needs to have received the previous update to be able to correctly reconstruct the current state from the packet. This is a problem in the current internet, as there are only two main transport protocols: TCP and UDP [65]. TCP is reliable, meaning that the data is guaranteed to arrive at the receiver's end. UDP on the other end is not reliable, meaning packets could get dropped by the network, which is called packet loss. This would make TCP ideal to transmit the updates as then the receiver would be sure to have all the previous updates.
However, TCP is mostly too slow because of retransmissions needed to guard the reliability and it has other unwanted features like congestion control, in-order delivery and the sending of ACK messages, which increases bandwidth usage. For this reason, the vast majority of highly interactive games use the unreliable UDP to send their state updates. This works well because when the frequency of the updates is high enough, a few dropped updates will not be noticed. But with standard UDP we cannot use differential encoding since there is no way of knowing which update was received by which user.

This is the reason why most programmers usually define a new reliability layer on top of UDP, providing only exactly those features they need for their application [11]. This way, UDP can be used for reliable transmission, but more work needs to be done by the programmers. The way these protocols are usually defined is to use NACK's (negative acknowledgements) to indicate an update has not been received and to request a resend. The detection of a missed update can be done by using sequence numbers for the packets so that when a packet with a sequence number higher than the previous number + 1 is received, we know the other update was lost. These sequence numbers are often needed by the application anyhow, so this incurs no additional overhead. Other more advanced protocols are possible, depending on the application that uses them. A good example of a reliable UDP protocol is Enet [11], providing

most features of TCP with the ability to turn them on/off, making a fine-tuned protocol with specific features possible.

But even a UDP-based reliable protocol often has too much overhead to be used constantly for every packet. This is why mostly, for delta compression, the new updates are not computed on the previously sent update, but on the previously sent *reliable* update. So assume (100,100) was sent reliably, then (5,5) can be sent unreliably . When the object then moves to (107,107), we don't send (2,2), which would be an offset to (105,105), but we send (7,7), still offsetting the updates to (100,100). This way the position (107,107) can still be reconstructed even if (105,105) was not received. Using this method, the differences can become large again over time, for instance when the object has moved to (200,200). Then we can use a simple heuristic to decide when we need to send the next reliable update. For instance, when the differentially encoded positions cost more than 10 bits, we should send a new reliable reference update to be used from then on. This way most of the traffic can still use unreliable standard UDP without overhead, with only an occasional reliable packet. This setup is robust, causes small overhead and can lead to huge bandwidth savings. Figure 3.4 gives an overview of the three possibilities.

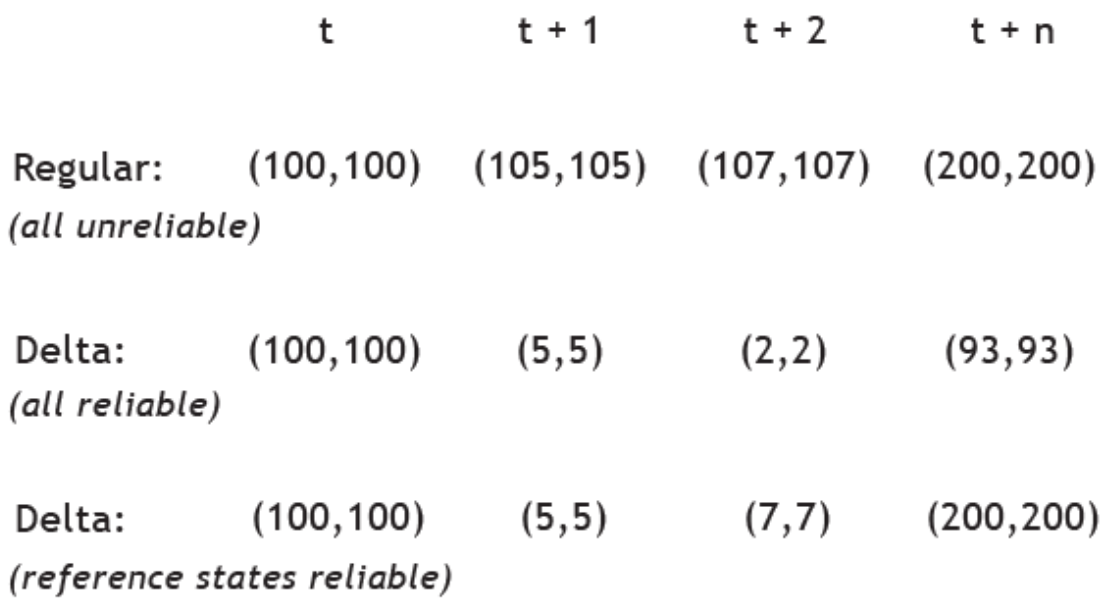

Figure 3.4: Using delta updates to encode positions.

Forward Error Correction

As reliable protocols induce different kinds of overhead like processing time, bandwidth, irregular send rates etc. sometimes it is opted to not send anything reliable. This is the case with for instance audio and video transmission, where in general there is no time to wait for retransmits of dropped packets. However, this means that if some packets are dropped, the quality of the rendered video (or for instance world state) will be of a lower quality. We can better protect ourselves from packet loss by adding some redundancy to the data we send over the network. This generally means we will send additional information with each packet (or in separate extra packets) that enables us to (partially) reconstruct dropped packets or that can help us recover from a dropped packet, if the packets containing the redundant data are received. This method is called Forward Error Correction (FEC) [19]. This can for instance be done by sending exact copies of other packets along with the current packet, but this will essentially increase the used bandwidth by a factor two or more. The use of some smart algorithms make that relatively small error codes can be generated so the redundant data does not require that much bandwidth.
It might seen contradictory to talk about using FEC in the context of this thesis as it will always require additional bandwidth. However, it can have a large impact on the quality of the received data if packet loss is present [92]. This can be especially important if the update frequency is low and a dropped packet means a large gap in the world state. Furthermore, it can be used not as a complete replacement for a reliable protocol, but as a viable alternative if the data has to be semi-reliable but the occasional dropped packet does not have that large a influence. FEC will arguably cause more constant bandwidth overhead then the conditional retransmissions, but this in turn allows us to better estimate this overhead so we can take it into account in our bandwidth limitation approaches.
FEC can be used in many different ways. The most generic way is to have it work at the packet bit-level, so any type of packet can be protected. However, this might lead to data being protected that does not really needs to be protected. Another way would be to calculate which redundant data we want to send on a much higher level in the application. This means we can only send redundant data for certain fields in certain packet types, reducing the bandwidth overhead. An example of this is to use FEC with delta updates.

FEC could be used in combination with delta updates to reduce the need for reliable updates. If we send the position on which the current position is based along with it, we should still be able to reconstruct the current position even if the previous packet was lost. An example would be for the transition from (100,100) to (107,107) via (105,105). If the packet containing (5,5) is dropped, but the packet (2,2) contains a copy of (5,5), the correct position of (107,107) can still be reconstructed without using reliable updates if position (100,100) was received correctly. This extra data requires extra bandwidth, but the total packet is still smaller than

it would be when we would send full updates for every position.
The reader should note that the efficiency of FEC is of course dependent on the actual amount of packet loss. If this loss is low and in very small bursts, FEC can produce good results. When the packet loss is high however and multiple packets in a row are lost, FEC will not perform optimally. More complicated methods of FEC can be used to counter some of these problems (for instance interleaving of FEC packets, which in turn can lead to a higher delay), but sometimes there will still be some reliable updates needed to restore the correct state after a period of high packet loss.

FEC has been shown to produce good results for enhancing video and audio quality [92] and has also been proposed for use in game state protocols [68]. To the knowledge of the author however, the technique has not yet been implemented in a large scale NVE or networking middleware. For the purpose of this thesis, it is doubtful that FEC would fit into an aggressive bandwidth saving strategy. However, it can be interesting for use on networks with high amounts of packet loss (mobile networks for instance) or when it is important that most packets are received but the overhead of a reliable protocol is too large.

PICA

PICA stands for Protocol Independent Compression Algorithm and it revolves around the idea of delta compression [90]. It was originally developed to reduce the amount of data sent in the DIS or SIMNET NVEs. PICA is a little different from the given examples for position updates as it does not use prior knowledge about the meaning of the data. In this way, it is a generic algorithm like the algorithms discussed in section 3.1.2.
PICA does this generic compression by using a sequence of Difference Records (DR's). These DR's each specify a count (number of bytes that were changed) and an offset (where does the change begin) followed by a number of bytes. Notice we are speaking in terms of bytes here, not bits as in the previous examples. These DR's are sent with a sequence number to the other users. When the differences become too large, a new reference state is sent reliably. Thus we see that the PICA method is more or less the same as our last discussed method for sending position updates.

The creators of PICA report a 76% reduction in bit rate for DIS Protocol Data Units (PDUs) [8]. As we will see in section 3.1.4, these PDUs are packets that always have the exact same fields in them, in exactly the same order and thus they are always exactly the same size. This helps explain why PICA obtains such good results. The PDUs have a large overhead of their own because data that has not changed is always sent along in every PDU. It is logical that an algorithm like PICA would remove these fields and thus provide better results. When thinking about using the PICA-method for packets that change size and contents however, it is a lot more difficult to prove it will still yield such positive results. Since the algorithm also

induces extra overhead by the use of the *count* and *offset* fields before every DR, it will only overcome this extra overhead if a lot of the data does not change. But when we do not send the non-changed data in the first place (as discussed in the section on logical compression), PICA will perform much less well. This makes PICA an interesting alternative to logical trimming of non-changed data. Instead of tracking the changes in every variable of an entity, we can just send all data every time, relying on PICA to remove non-changed fields at a packet-level. This reduces the complexity of the logical compression.

In conclusion I would say that PICA is good if packets are always the same size and composition (for instance packets of the same type, in the same stream, ...) but custom delta compression that uses knowledge about the data could get better results if the protocol is more dynamic and flexible.

3.1.4 Case-study: DIS PDUs

The Distributed Interactive Simulation (DIS) [81] is an open standard for interconnecting different kinds of simulators and is mostly used for war and army simulations. DIS allows the simulators to interact using a standard network packet format known as a Protocol Data Unit (PDU) [8]. A PDU describes what data is contained in the packet and as long as all simulators understand the PDUs, they don't need to know anything else about each other.
There are a large number of PDU types but the most interesting one is the Entity State PDU. This PDU is used to send the state of a particular entity to the other users. The structure of this PDU can be found in figure 3.5. It is interesting to discuss DIS here because this PDU is sent as a whole every time an entity update is sent. This means that fields that are unchanged are also sent with each update and that bitwise logical compression is only performed at a very high level. This makes DIS a good real-life example of how protocols can be optimized by using logical compression.
I will not discuss all fields, only those with now obvious compression possibilities. Take the Entity ID Record which indicates the world-wide unique ID of this entity. In a world with a large number of entities, 48 bits may be needed to provide a unique ID and this ID has to be sent every time to identify the entity to which this update belongs. The same is not true for the Force ID, Entity Type Record, Alternative Type Record and Entity Markings [8]. These fields will mostly be constant throughout the entity's lifetime and should only be sent once. This would result in a $8 + 64 + 64 + 96 = 232$ bit profit for most sent packets. The reason that these fields are always present is because when a new user joins DIS, he will only receive updates from the entities to update his world, i.e. there is no full state transfer. Because of this, all updates should contain all information for the new user to be able to know all the data of the entities. However, a better method would be to have the new user request a full state when he receives a partial state from an entity. This will reduce the bandwidth usage

Item Name	Bit Length
PDU Header Record	96
Entity ID Record	48
Force ID Field	8
# of Articulation Parameters (n) Field	8
Entity Type Record	64
Alternative Entity Type Record	64
Entity Linear Velocity Record	96
Entity Location Record	192
Entity Orientation Record	96
Entity Appearance Record	32
Dead Reckoning Parameters Record	320
Entity Marking record	96
Entity Capabilities Record	32
Articulation Parameter Record	128

Figure 3.5: DIS entity state PDU[8].

considerably but will also make the protocol more complex.
In contrast to the previously mentioned fields, the fields for linear velocity, position, orientation and appearance will probably change in every update, but cannot they be represented in a smaller fashion? As shown for a simple rotation previously, orientation and velocity could be represented with far less bits most of the time if we restrict the interval in which these values can lie. The use of three 32-bit floats for both values could be reduced to smaller values if some precision is sacrificed.
The location of the entity consists of three 64 bit floats. This is needed to represent coordinates in a large world. However, as we have seen for incremental updates, differential encoding of these positions can lower these 64-bit requirements to 24-, 10- or even 5-bit entries, making delta compression a good candidate for optimizing the DIS protocol.

If we jump ahead to the next section about aggregation and state that in some cases packets will be grouped together into one large packet, huffman encoding can be used here as well. Most of these simulations have a large number of entities, but many of them will be of the same type (20 tanks, 10 choppers, 50 foot-soldiers etc.). This means that the Entity ID Record and Entity Type Record will mostly be the same across different packets. Using huffmann compression with a code-book tuned for these types and IDs, those fields can be represented a lot smaller, especially when used in combination with aggregation of multiple packets into one.

This DIS PDU is a good example of how state packets for entities in an NVE are usually structured and what kinds of data are mostly present. Our discussion shows that there can be a big difference between the basic version of a protocol (sending all data with every update) and the optimized version which uses logical and more advanced methods of encoding the data. therefore especially an NVE protocol for a large-scale world should be optimized to reduce the bandwidth required because otherwise the NVE will not scale to many users on a real network like the internet.

3.1.5 Case-study: Video compression

A very good case-study on the advantages and usages of compression can be found in the representations of different kinds of media in modern computer systems. Here I will show how video encoding uses the three discussed types of compression and how the JPEG and MPEG [31] standards work.

Video and image compression is based on logical compression and assumptions about how we perceive the world around us [31]. An image can be represented in the so-called frequency domain by using the Fourier Transform. For the purpose of this text, it is not important to understand the exact meaning of this representation. The most important thing is that the Fourier Transform can be done for any image and that from the frequency domain the normal image can be re-aquired.
The use of the frequency domain is logical as low frequencies represent areas with little details in the image, and high frequencies represent the image details. The human eye is - perhaps counter-intuitively - most sensitive to variations in the areas of low detail. Thus if the finer details are too detailed, we can safely remove them from the encoded image, as the human eye will probably not even notice them if they would be there. This process is called quantization and it is an example of lossy compression as certain values (those representing finer details) will be removed to reduce the number of bits needed. The quantization also reduces the amount of bits needed to represent the remaining data by dividing the numbers by a certain threshold and rounding the result. The actual process is a lot more complicated, but this is the basic idea behind it and shows how a notion of human perception can be used to trim unnecessary data from the representation. Later we will see that similar notions of perception can be used for other types of state updates. Further-away entities in the world can be represented with a lot less detail for example, because they will be much smaller on the user's screen.

After quantization, we have matrices of coefficients for every frequency in the image. The higher frequencies will usually have coefficients of zero. Using a method called zig-zagging

the values of the matrix are placed one after the other. This way, the non-zero coefficients of the low-detail frequencies are next to each other, and the zero's of the high-detail frequencies are also consecutive. Differential encoding is used on the non-zero coefficients and run-length encoding is used on the zeros, leading to a more compact representation. Finally, huffmann encoding is used on the differential and run-length encoded sequences. The process is shown in figure 3.6. This shows how generic compression methods can be used in a smart way on specific kinds of data.

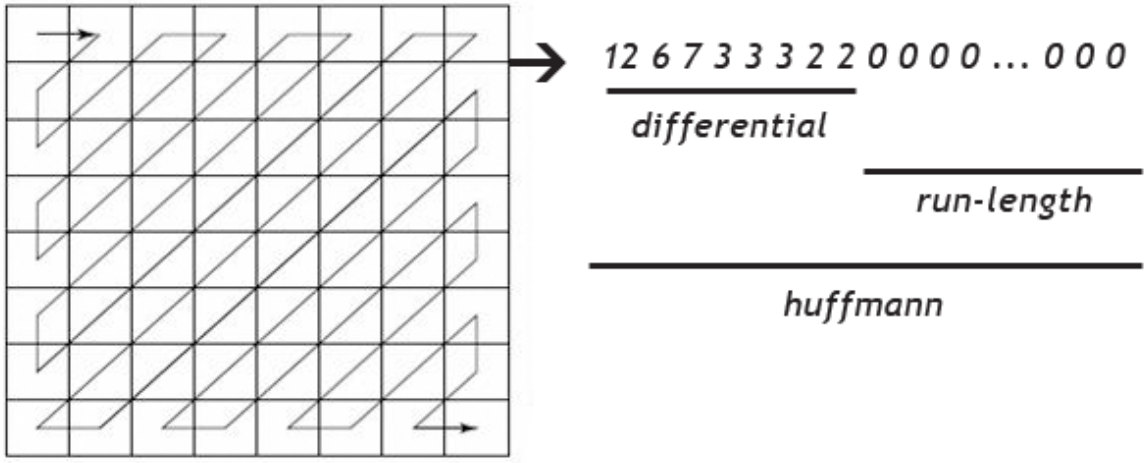

Figure 3.6: After placing the coefficients after each other, the values are encoded with three different algorithms.

Finally, also delta-compression is used in video encoding [31]. If we were to encode every frame of video using the method above, this would still yield too much data to be practical for storage or transmission. Therefor, only certain frames are encoded using the afore mentioned method. These frames are usually called I-frames (intracoded). A number of frames following an I-frame are encoded by referring to data present in the I-frame. These frames are called P-frames (predictive). This is possible because the pixels of a video usually don't change very quickly in consecutive frames. Mostly the pixels of the I-frame are also present in the P-frames following it, be it a little displaced. So if for the P-frames we simply indicate where each pixel (or block of pixels) can be found in the previous I-frame using a so called motion vector, we only need to really store pixels that are not present in the I-frame in this current P-frame. Thus the P-frames use the reference I-frames to encode their information, much the same way we used previous position updates to indicate the current position. Even more complex setups are possible, for instance using B-frames which use information of past and future frames to encode their data.
It should be noted that the problem of reliably sending the reference frames is also present when sending video streams. If the packet(s) containing an I-frame would be lost, the fol-

lowing P-frames would be near useless and the image would be of considerably lower quality. Luckily, there is mostly only a limited time between I-frames and the degradation in quality will not be noticeable for very long. This is why most video transmission protocols like RTP [70] don't resend lost packets but just wait for the next reference frame and in the mean time use estimations of the non-received packet instead. Here, a Forward Error Correction method can be of a great help to improve image quality in the event of packet loss, as discussed by Wijnants [92].

We can make a final notion on the OnLive system [34]. This system allows players to play videogames on remote computers without the need to purchase much hardware themselves. The powerful remote computers render the game and send the output to the user as a (high-quality) videostream, who only needs a way to play this video stream and give basic input to be able to play the game. This system shows that advanced video compression can sometimes be interesting for communicating application state. OnLive is tightly coupled to the idea of remote rendering, which could also be a viable approach to provide mobile users with low-end devices with a high-quality view of the world. These and other aspects of OnLive are further discussed in chapter 5.

In conclusion, we see that all discussed types of compression are used in video encoding and that this compression is very necessary to get an acceptable data size, both for storage and transmission.

3.1.6 Conclusion

The three kinds of compression can yield enormous reductions in the size of the data and thus on the bandwidth usage of an NVE. Many compression schemes exist for much-used kinds of data like video, audio, textures (images), 3D meshes, etc. and for most of these applications existing implementations are available.
When it comes to world and entity state however, there are no global solutions that are optimal for every kind of NVE or simulation. Every game or settings requires different parameters to be sent and other types of data to be synchronized. Because of this, programmers need to optimize their protocol themselves.

For this, using logical compression is a must. Packets should be kept as small as possible and one should only send those things that have changed. Especially this process depends on the type of application and it can take many forms for different kinds of data.
Generic compression is typically more difficult to use on single network packets as the risk exists that they will enlarge the size of the packet instead of decreasing it. However, when

we know the structure of the data or when the packets are very large (for instance by using aggregation, as will be discussed in the next section), generic compression can lead to added data reduction.
Finally, delta compression is a very promising method for reducing the size of packets, but it comes with the reliability-problem. The type of data one wants to send with delta compression often determines what approach is needed for this problem. Video compression for instance often just ignores dropped reference frames, whereas other state could use all-reliable or only reference-state reliable approaches. As we will see in chapter 4, delta compression is a popular method in existing networking middleware and can obtain very good results.

Compression can greatly reduce the size of network packets but it certainly comes with a cost. When using compression, one will need extra processing power and memory as generic compression algorithms mostly use complex computations. Delta compression and only sending what has changed requires that we keep some amount of past state in the memory to serve as reference material. As with most things in an NVE, the use of compression is a trade-off between bandwidth usage, processing power and memory.

As for the usability in a particular network architecture, all compression methods are usable in both P2P and CS settings, albeit not always very straightforwardly. Only delta compression really requires a special note: since reliable multicast can be very expensive in terms of bandwidth [65], it would be very difficult to implement delta compression with reliable reference states using multicast.

3.2 Aggregation

While compression is aimed at reducing the amount of data that needs to be sent and thus in a way create smaller packets, aggregation will try to make as large packets as possible. This may sound counter-intuitive at first, but by grouping a lot of smaller packets together into a single large packet we will actually save bandwidth because the packet headers will be reduced. This principle is further explained in the next few paragraphs.

3.2.1 Packet headers on the internet

Up until now we have acted under the impression that the bandwidth consumption of our networked application was only caused by our own data. This is not entirely true. Every time you send a packet over the internet, so-called packet headers are added for transmission. This is because the internet is built as a layered model and every layer needs extra information to work [65]. A low layer is the datalink layer. This layer adds 14 bytes as a header to the packet if the ethernet protocol is used. The next layer is IP. Here 20 bytes are added for the common IPv4 and 40 bytes for the new standard IPv6. Another layer is the transport layer.

Here, 8 bytes are added if we use UDP, 20 bytes for TCP. When using TCP, this overhead is even worse. This is because each packet has to be acknowledged by an extra ACK packet, inducing even more packets and traffic. One example is for the RPG Lineage II [72], where over 35% of the packets were TCP SYN, ACK and FIN packets.
So in total, this leads to at least 42, at worst 74 bytes of header information for every packet sent. Other networks will add more or less headers, but the basic principle stays: packets use more bandwidth than merely the data you put in them, the so-called packet payload. Since this data is added by the network and lower layers, we cannot influence this data and cannot use the previously discussed compression techniques to trim their sizes. This implies that we are stuck with this overhead.

This means that we have to start thinking about payload/header ratios. If we send a lot of small packets, it is possible that the largest part of our bandwidth is being used by the packet headers, not by the actual payload they transport. The technique of aggregation this section deals with, is aimed at keeping the payload/header ratio as high as possible and thus minimizing the internet-induced overhead. The idea is to combine the payloads of different packets into one big packet and to only send that one packet. The size of these big packets is a maximum of 1500 bytes for the complete packet when using ethernet, also known as the Maximum Transmit Unit (MTU) [65]. If the MTU is exceeded, the packet will be split into separate ethernet frames.

Here we discuss a simple example of how aggregation can help lower bandwidth. Say that we send about 60 relatively small packets per second, each of which contains about 40 bytes of payload data, as is common in fast-paced FPS games like CounterStrike [88]. These games typically use UDP, so the overhead per packet is about 42 bytes. This means the total size of each packet is 82 bytes, of which more than half is network overhead. This comes down to a bandwidth usage of about 4920 bytes per second. If these packets can be aggregated however, we only have the overhead of 42 bytes once. This means the bandwidth usage is reduced to 2484 bytes (two separate packets because the MTU was exceeded), almost less than half the original usage. As we will see later in this section, it will probably not be possible to aggregate this many packets at once in a fast-paced game. But even if we can only group them into packets of 10 messages, we still have a large profit for bandwidth, from 4920 to 2820 bytes per second. It is clear that in this kind of situation, where there are a lot of small packets, aggregation can have a significant impact on bandwidth usage.

Note that when we talk about aggregation here, we are actually aggregating things that would normally be sent as separate packets. One could also do a kind of logical aggregation (just like we used the term logical compression) by grouping information together to be sent in a single update. For instance, an RTS player controlling multiple units at the same time would

logically group the updates for all the units in one packet instead of sending a separate packet for each unit. I consider this type of logical aggregation a part of logical compression. The aggregation discussed here is more generic, usually combining packets with different contents and types. This means aggregation can be done at the packet-level and requires much less knowledge about the data than most discussed compression techniques.

Packet loss

Besides the added network packet headers, the internet has another annoying property: packet loss. When we send less packets and pack all our payload data into less-frequent, large packets, packet loss can become a big problem. It will have a much larger influence on the local state than when only one smaller packet was dropped and care should be taken to prevent large gaps in data received because of packet loss when using aggregation. It should also be mentioned though that through the use of aggregation, the chance for packet loss might actually become smaller as packet loss is often caused by network congestion and aggregation will lower the bandwidth usage, also lowering the chances for congestion.
Possible solutions are to use a reliable protocol or techniques like Forward Error Correction, however they also have their own drawbacks.

3.2.2 Quorum versus Timeout based

When performing aggregation of multiple packet payloads into one big packet, we roughly have two options to decide when the new packet is ready to be sent [86].
The first option is to wait until the size of the packet is as close to a certain desired size (the MTU for example) as possible. Once the packet is as large as it can be, it is sent. This is called the quorum-based method. However, the reader might have noticed that this method can induce a large delay. Indeed, it can take some time before we have gathered enough data to create a large packet and all other data will be delayed until we have the final payload and send the packet.
This gives way to the other option, which sends packets when a timeout has expired. All data available by then will be packed into a packet and sent. This will lower the delay induced by aggregation, but it will also cause non-optimal aggregations to be performed. There is no guarantee that there will be a lot of packets in the send buffer by the time the timer expires, which can cause the aggregation to be nearly negligible.
Another option would be to use a hybrid form, where we use a timeout to determine when the packet should be sent. However, there is also a quorum in effect. When the quorum is reached before the timeout expires, the packet will be sent directly. This option will work best with a rather large timeout and when large amounts of data have to be sent.

So in conclusion, the quorum-based approach can cause large delays for data if there is not

enough data available. On the other hand, the timeout-based approach reduces the delay, but also the effectiveness of the aggregation. Once again, this drawback is largest when there is not enough data available. On the positive side, a hybrid approach can be used with both a timeout and quorum parameter, both of which can be tuned to fit the needs of the application and separate values can be chose depending on the type of data stream under consideration. In any implementation, the use of aggregation will require fine-tuning and testing to obtain the most optimal results.

3.2.3 Where to perform aggregation?

Whether we use timeout or quorum based aggregation is one decision we have to make. The other is where in the network architecture to do the aggregation. As can be deduced from the previous section, aggregation will work best when there is enough data available at all times. When there is a constant, large flow of data present, aggregation will be most effective. When the time between updates is large or the data to be sent is small, aggregation will either induce extra delay or it will not have a large effect on the bandwidth usage. So we are looking for moments or places in a network when large amounts of data come together.

As has been discussed before, users or players often send little amounts of data in most games and applications. It is only when advanced avatar animations are used or when users create new content for the world that the data sent by players is relatively high. This would lead to the conclusion that aggregation on the user-side is only rarely interesting. This means that clients in CS-systems and peers in P2P systems usually have little possibilities for using aggregation. P2P will probably be able to make better use of it when the peers themselves store and distribute some of the world state.
In contrast, places where all kinds of traffic arrive together to be distributed to their destinations would be excellent locations for aggregation. A server in a CS-setup is indeed a very logical place to perform aggregation. Since every player sends his updates directly to the server and the server is usually responsible for other world state as well, it has all information that is destined for one particular user at all times. This opens the possibility to wrap packets destined for this single user together at the server. In fact, this idea of performing server-side aggregation is so logical that there are P2P systems that actually use special aggregation servers [86] to be able to incorporate this bandwidth saving technique. These special servers are for instance deployed at the edges of large P2P LAN networks to facilitate transfer of data over the internet to other connected LANs.

So we see that aggregation is very logical and natural if performed at a server in a CS-setup. P2P systems are much less adept at making full use of this technique. In traditional or multicast P2P, there is no single place where data meets unless it is already at the final

destination. And as we have seen, aggregation at the local side will in general be non-optimal as there are no large amounts of data a user will send directly to one and the same user. For this, P2P systems have been proposed which use special aggregation servers to make good use of the aggregation technique.

3.2.4 Conclusions

Aggregation is not as easily used as compression. It will only perform optimally in server-like entities in a network and a balance has to be made between quorum and timeout based approaches. When using aggregation, fine-tuning is needed to maximize the individual packet size and minimize the extra delay. Care should also be taken to prevent the effects of packet loss. When well deployed, aggregation can mean a large profit for the bandwidth usage, especially when using large packet headers, such as the ones introduced by TCP and the new IPv6.

Chapter 4

Traffic Filtering

Protocol Optimization didn't actually discard information or packets, it just made the information smaller. This chapter looks at techniques that can be used to drop large amounts of packets for specific players to reduce the total needed traffic. Their basic ideas mostly come from the desire to limit the amount of information the user gets and thus focus his/her interest or to improve consistency between users. This makes them very usable to control bandwidth as well and this in a dynamic way. By tweaking how consistent the world is or how many objects of high-interest a user can perceive at the same time, we can filter packets and thus lower the bandwidth.

There are two main techniques that can be used to this end. The first kind is based upon spatial coherence in a virtual world. When players interact in the world, they will most likely do this interaction with objects close to them in this world, which is very much like how we interact in the real world. Far-away objects are thus less likely to receive interaction and by consequence less interesting to the user. This is the basic idea behind spatial subdivision and area-of-interest filtering.
A second kind of technique is based upon predicting the world state in the future. Using (semi-)deterministic models of physics for example, the behavior of certain objects can be predicted with great certainty for a limited period of time. If this prediction can be performed at the receiving user's end, there is only need for information to be sent when the real behavior diverges from the predicted behavior. This principle leads to the Dead Reckoning method and other uses for prediction.

4.1 Spatial subdivision

A very interesting and popular approach to filtering comes in the form of spatial subdivision of the world. This means that there won't be a large, continuous world at all times where all users have all the information about the complete world. There will rather be only parts

of the world users will receive information for, the parts where the user's avatar is. As we will see, this way an NVE can be kept scalable in terms of both processing and bandwidth. I would go as far as to state that without a form of spatial subdivision, a large-scale NVE is simply impossible.

There are a couple of subdivision types possible, all operating at a different level of granularity. Figure 4.1 shows how these different types interrelate.

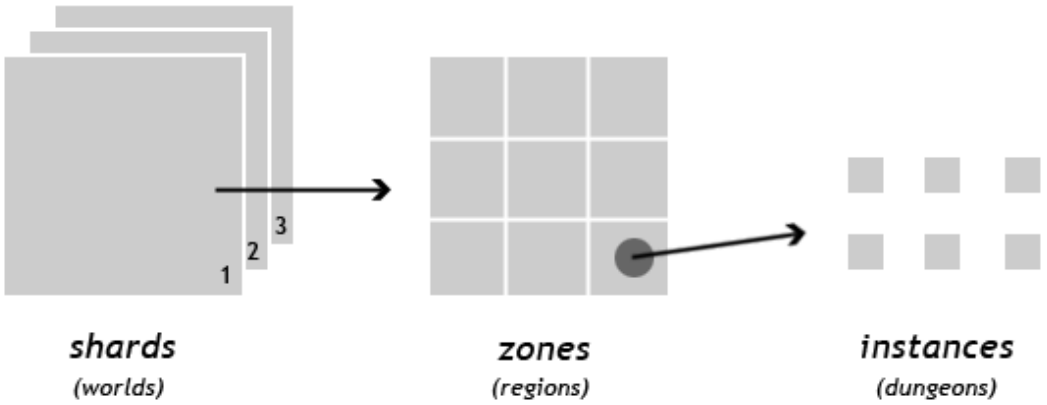

Figure 4.1: Three types of spatial subdivision.

4.1.1 Sharding

This first type of subdivision is very high-level. The idea is to not have one big world where every player plays in, but rather to have multiple parallel worlds with different players in each world. This way not all players playing the same game are able to meet each other in the world, only those players in the same parallel world as their own, called a shard. This sharding model is applied in many modern day MMOG's, including World of Warcraft [53] and Aion [1].

The main reason to do this subdivision is because the worlds can co-exist independently from one another. If a Client-Server model is used for instance, separate servers can control different shards so that when one server goes down, other shards are not influenced by it. Another important reason is the division of the number of players. This way each shard only has a limited number of players and thus less traffic and calculations to perform. Increasing the maximum user count is then achieved by simply adding some new servers to create a new shard.
This is very different from one big world in which every user can meet every other user. That setup will give problems for scalability, as more players cannot simply be supported by adding

some extra servers. For this, we need a complicated and extensive server-architecture that can deal with a large amount of players in one large world, as will be discussed later. Additionally one big world can also be problematic for bandwidth usage as more state is being exchanged between servers in the world.

Another important reason is that the shards are smaller and thus also contain less content. For applications like Second Life [46] where most content is user generated, large seamless worlds are possible. But for games like World of Warcraft [53], where the developers have to create all the world's content, it would be an immense task to create huge worlds. Through the use of sharding, the same content can be re-used in multiple versions of the smaller world, reducing the amount of content the creators need to implement.

Sharding is a popular option for commercial NVE development as it makes the worlds and users more manageable and possible problems are isolated to a single world, where they affect only a limited number of people. One could also see it as developing a relatively small-scale NVE and then copying it multiple times to allow more users to use the application. Different shards can also sport different rule-sets to allow for certain niche gameplay or settings within the same game or application. In practice the biggest drawback is that players on one shard cannot interact with players on other shards directly in game, but this mostly doesn't outweigh the benefits in processing scalability and architecture simplicity for most commercial applications.

4.1.2 Zoning

This second type of subdivision is perhaps the most intuitive, interesting and widespread. As mentioned in the introduction of this section, a user will mostly interact only with those entities close to his avatar in the world. This means he really only needs to receive information from entities in his direct vicinity. Zoning is a relatively coarse-grained method of doing this information management. The world is subdivided into large regions, called zones. A user in a zone will then typically only receive information about the entities inside his own zone, and possibly from neighboring zones but rarely anything past this.

It is clear that this is a very interesting concept for the purpose of this thesis. Instead of receiving all packets from all players in the complete world, a user only receives packets from players and objects in his own spatial neighborhood, removing a huge amount of network traffic. As we will see in the next few paragraphs, the processing requirements are also lowered by zoning. For these two reasons, zoning is an ideal method to be implemented in almost any NVE as it easily enables both bandwidth and processing scalability.

Coupling with network architecture

An interesting property about zoning is that it can be directly reflected by the used network architecture.
When using a Client-Server setup, it is common to have every zone handled by one separate server. This way, extending the world is easily done by adding more servers for the new areas of the world. When using an interconnected P2P network, we can have certain peers act as being responsible for a certain zone. Other peers then only need to connect to this responsible peer to get basic information about the zone, after which he can connect to the other players in his current zone. Furthermore, this also ensures a good processing scalability. The zone servers or responsible peers only need to perform calculations for their own region and only have to deal with the players currently inside this region.
Another possibility is the use of multicast together with zoning. The idea is to associate one multicast group with every zone in the world. This way, users only need to send their data to one multicast group at a time and they can decide themselves which zones they are interested in by subscribing to the correct multicast groups. As discussed before this is in contrast with entity-based multicast, where every entity in the world has its own multicast group. Region-based multicast is a very elegant solution which requires almost no additional processing to route traffic to the correct users.

Zone shapes

When we want to use zoning we need to make a decision about the shape of the zones and which areas different zones will encompass. Traditionally, zoning was used in closed-space maps in for instance FPS games. Here, mapmakers indicated zones themselves (mostly rooms and corridors) and also zone-transition points (like doors or gates). In larger worlds, zones could be made to contain a complete building or another logical entity. In EvE Online for example, an MMOG which is set in space, every solar system is a single zone managed by a single server [14]. However, this mostly requires mapmakers to define these zones for most parts of the world, and this world can be very big for NVEs. In addition, players can maybe add new parts of the world themselves and they will not be able to define optimal zones themselves. The last problem is that most virtual worlds contain large open spaces. If we were to use one region for every large open space, regions could become too large thus reducing the optimizations we expect from zoning in the first place.
It is for these reasons that most large-scale NVEs use a conceptual regular grid for the subdivision. Thus the subdivision is not bound to logical boundaries like buildings or the end of a corridor, but they can run straight through a large open area. This is less optimal as a subdivision, but it is a lot easier to implement and could be combined with limited mapmaker-defined zones [60] in special areas that would benefit from it most. Many possibilities have been proposed, for instance using a simple square grid or hexagonal tiles. More advanced

zoning algorithms use things like octrees or kd-trees to create a more optimal and hierarchical subdivision. Figure 4.2 gives an overview of some possibilities for the definition of zones.
We should note that dynamic subdivision is also possible when using mapmaker defined zones. However, these subdivisions can possibly be more difficult to implement as they should keep in mind the logical boundaries set by the mapmaker, whilst the more generic algorithms can usually subdivide rectangles or hexagons, which is a lot easier.

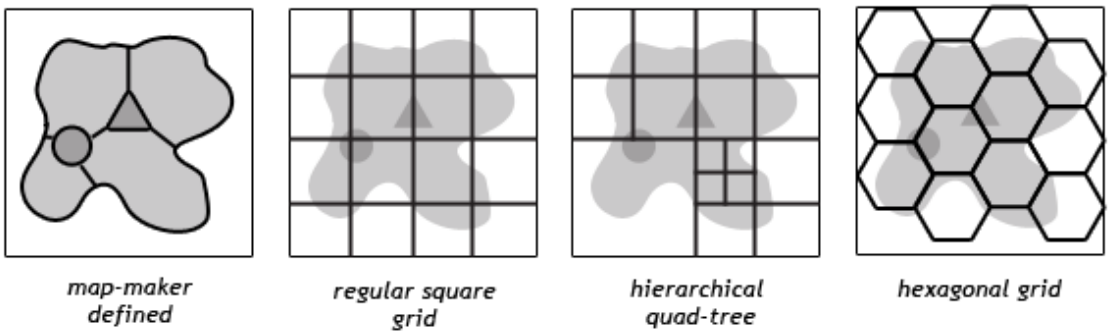

Figure 4.2: Four types of zoning.

Drawbacks

Zoning might seem like the ideal solution for a lot of NVE problems, but there are also downsides. The first problem occurs when there are many players in one particular zone at the same time. Like we said, the reason why zoning works is because a lot of the information about the world doesn't need to be processed by all players anymore. But when a lot of players are in the same area, all players in that area will have to know about all other players there. This causes a larger bandwidth usage but more importantly a larger processing overhead. If only one server or peer is responsible for a zone, this will only work well if there is a limited number of concurrent players in that zone. Compare it to a traditional FPS setup of a 32-player server. That same server cannot handle more than 32 players and the same goes for servers that manage a single zone in a virtual world as well.
This is a large and difficult scalability problem for NVEs that use zoning. In fact it is quoted as one of the most important reasons for the bad scalability of the Second Life simulation [47], as their setup works with fixed-size regions which are assigned to maximum one simulation server. There are roughly two possible solutions: either we limit the number of users that can be in a single zone at the same time (EvE Online only allows 2000 concurrent players in a single solar system [12]) or we can dynamically re-partition the overloaded zone into new,

smaller zones. Through this re-partitioning, more servers will be allocated to handle the area where a lot of users have converged, this way reducing the load on a single server. Dynamic re-partitioning is not trivial however and requires architectural support to make it possible. There is also the issue of state transfer from the already overloaded server to the new servers that will alleviate its load. Active research is being done on this topic, for instance by Cleuren [64].

The second big problem is the crossing of zone boundaries. Especially when working with a regular grid, users are likely to cross a number of zone boundaries as they explore the world. Suppose users only receive information about other users and objects in their current zone. This means that when a new user enters the zone, you will see his avatar appear out of nowhere at the zone boundary he just crossed. The inverse will happen when you exit a zone: players that remain in that zone will see your avatar disappear. This is the main visual problem for the users, but there is also a problem architecturally. If every zone is managed by a server or peer, when users change zones, they also have to change servers or peers. This means all state of the user has to be transmitted to the new responsible entity, possible inducing some delay for the users. When using pure P2P systems, there is also the problem of neighbour discovery to know which peer is managing the new zone, as discussed in chapter 2.

Another consequence of this setup is that cross-zone events are difficult to implement. In an FPS for example we want players to be able to shoot each other over rather long distances, even if the players are not in the same zone. This kind of interaction is not possible when we only receive information about our own zone. When a sniper in zone A shoots a bullet to hit another player in zone B, there should be some kind of communication between the two zones for both the firing of the bullet and the result.

A possible solution to these problems is to have players and zone servers receive all state of any neighboring regions as well. The regions are typically large enough to ensure any visual artifacts from boundary crossing go unnoticed by the players if they have the data of more than their current region. Sadly this solution will require a lot more bandwidth and processing power for both the users and zone servers or peers. Now they are not only receiving all data from their own zone, but also from 3, 4, 8 or more zones surrounding them. This also means users have to maintain connections to multiple zone servers or peers at a time and those connections will change as the users change zones (a large problem in P2P environments as discussed in chapter 2).

A common approach to this problem is to use soft-transitions at zone boundaries. This means not all state from neighbor zones is sent to users in a zone, but only state from the parts close to the boundaries. This will reduce the bandwidth and a lot of processing power related to interpreting the extra state, but it requires extra processing to determine which state has to be sent exactly, i.e. if it is close enough to the zone's boundary. Areas of interest can help to define where the soft boundaries should be, as we will discuss in the next section.

A note should be made on the restrictions of region-based multicast in this matter. When using a single multicast group for every region, it is not possible to filter traffic which is too far from the boundaries. Users can only subscribe to the multicast group or not, there is no middle road. A possibility is to further subdivide the zones into smaller regions with all their own multicast group, but this greatly increases the complexity and reduces the practicality of such a system.

4.1.3 Instancing

The last type of spatial subdivision will isolate a limited group of players in a small piece of the world. This piece is copied/instanced for every group that wants to use it, so that the different users don't interfere with each other. A good example of this concept are dungeons in a typical MMORPG. A group of players has to go inside a dungeon to defeat a boss character and gather new items. These bosses should be available for all players and players should be able to face the same boss multiple times. These dungeons can be implemented as instanced regions of the world. When a group of players enters the dungeon, it is copied for this specific group of players and any action in the instance will not directly affect the larger NVE world. When the mission has finished, the instance can typically be terminated and the players are returned to the larger world. Another example would be for a group of people to come together for a collaboration, video conference or private meeting. They can seclude themselves from the world in separate instances until their meeting is done.
A good real-life example is in the recent FPS game Read Dead Redemption [41]. While this is not a MMOFPS per say, players will arrive in one large world when logging into the game. Once enough players have gathered and have agreed to play a 24 or 32 man FPS match, they are taken out of the world and placed in an instance of a specific map where they can play the match. After completion they are returned to the world so they can search new matches to compete in.

Instances are interesting for NVEs because they actually remove some players from the NVE and place them in a separate place. For this, relatively little interaction should be done between the world and the instance whilst the instance exists. So while the players are in an instance, the larger NVE doesn't have to worry about their constant state updates, zone transitions etc.
In addition, instances can also be run completely separated from the larger world hardware- and network-wise. Since the number of players is typically very low and the instances are not very large, a single server or simple P2P connections will be more than enough to handle the interactions between the players in the instance. This way we are not only conceptually but also practically separating players from the NVE. It would also be possible for a flexible zoning system to incorporate instances as special zones that are created on the fly when

needed. As explained in the section on zoning, a single server can manage a zone. When the instance is needed, a new zone is created and a free server is allocated to manage this zone. This allows for elegant and simple implementations of instancing as extensions of an already existing zoning implementation.
When using separate resources for instancing, we should be careful with our estimates of how much resources will be needed. For instance, how many separate servers will we need to accommodate the largest load? If we reserve too many servers, we will be wasting valuable hardware. If we do not reserve enough servers, the users might notice. Even when we use the same servers for zoning and instancing, we should still keep in mind that every server used for one purpose can possibly no longer be used for the other. Thus when there is a high demand for both instances and zones, it is difficult to decide which should get precedence over the other. All in all, a dynamic system is recommended to prevent wasting hardware resources, but care should be taken in how the resources are assigned to different tasks.
Finally there is another programming aspect to instancing. Simple, low-player count distributed systems are much easier to program and maintain then the larger NVE systems, so instances can considerably reduce complexity for gameplay programming and player interactions.

4.1.4 Case-studies: Second Life, Eve Online and World of Warcraft

Second Life

Second Life [46] is a single-world simulation that tries to mimic the real world as closely as possible. This world is divided into square, fixed size zones of 256m x 256m. The Client-Server network architecture uses one so-called simulator process per zone and one physical server can run several simulator processes. These simulators are only connected to their four closest neighbors. Users connect directly to the simulator servers to exchange state. The client programs are so-called dumb clients: they only render data they receive from the servers but perform no game logic of their own, not even local prediction (see the section on Dead Reckoning).
Second Life uses a separate space server to maintain the regions. Every simulator process registers with this space server and receives the data for its four nearest neighbors to which it has to connect. The open-source variant of Second Life, OpenSim [35] uses a very similar network architecture. For OpenSim there is a plugin available to split regions into smaller regions and merge them to larger regions when needed. As stated before, the lack of such a dynamic region management is one of the biggest problems in Second Life when a large numbers of players congregate in one place.

Eve Online

Eve Online [13] is another single-world simulation which is set in a universe. Here the world is not just one continuous landscape like in Second Life, but it consists of clearly separated solar systems. Every solar system can be seen as a zone and each of them is managed by a so-called SOL-server. The main difference between the architecture of Eve Online and Second Life is that Eve uses an extra type of server: the proxy server. Instead of directly connecting to the zone-servers, users connect to an intermediate proxy server. These proxy servers are then responsible for contacting the correct SOL-servers and to transmit all necessary state to the users. This setup allows users to have just one connection, that to the proxy server, and this connection does not change when the user changes zones (in that case, only connections between the proxy servers and the SOL-servers are changed). This severely reduces the direct connection load on the SOL-servers and the proxy servers can perform other actions like load balancing and traffic filtering in the process. The Eve Online architecture can be seen in figure 4.3.

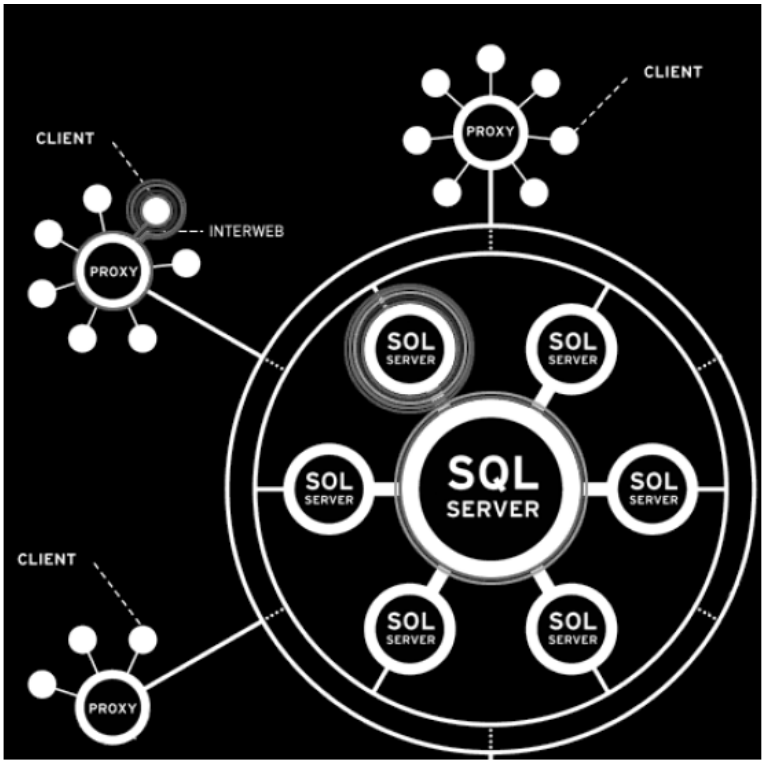

Figure 4.3: The Eve Online network architecture with intermediate proxy servers.

This extra layer of proxy servers solves some of the other scalability problems of Second Life, but there is still no built-in possibility to dynamically divide regions when a large number of players congregates in the same solar system. Recently, this problem has become so large that there is now a limit on the number of concurrent players in a solar system.

The game's setting allows for some interesting scalability improvements as well. First of all, the users don't control their spaceships directly, but by using only their mouse to give orders. Thus only the mouse commands have to be sent in a way very similar to the deterministic simulations for RTS games, discussed in chapter 2. Another fact is that because the universe is so huge, there will rarely be moments where players actually meet during their routine gameplay: most of the interactions are with computer-controlled NPC's. This way, often only little bits of information about other players has to be sent to a player, reducing the bandwidth usage. A third optimization is possible because of the physics-based nature of the game. The spaceships are bound by physical laws like acceleration speed and turning capabilities and when flying towards a certain goal in the world, the user will always follow a straight, predictable path. As we will see later in the section on Dead Reckoning, this can also aid in reducing bandwidth. The last optimization is possible because zones are not connected directly, only through the use of special warp-gates can a player travel to another zone. This way, there are no problems at the zone boundaries or with server-tradeoffs when players travel to another zone.
These improvements and simplifications are only possible because of the specific nature of the Eve Online space-setting and are not as simple to transfer to other games or simulations. For instance, Second Life can use almost nothing of these optimizations because of it's different concept. This once again shows that NVE design is strongly coupled to the specific type of NVE we want to create and that no one solution will fit every purpose.

World of Warcraft

World of warcraft [53] is the most popular commercial NVE. It is set in a fantasy world with large diverse areas like forests and cities. Unlike Second Life and Eve Online, WoW is not a single world. The game has many different shards and worlds, with every shard containing about 2500-3000 players. WoW also uses instancing for the different boss dungeons, which can be battled by small groups of players at a time.

Sadly, little is really known about the network architecture behind World of Warcraft because it is a commercial RPG and the developers have not yet disclosed much concrete information. Luckily, because WoW is so popular, a lot of surveys have been done to research the network traffic generated by the game [77]. From these researches we can deduce some information about the architecture.
It is clear that WoW uses a large amount of servers to run the game on, otherwise this scale would be impossible. From traffic analysis, it can be seen that there are very few (about three) connections being maintained during the game and that most traffic (probably game state) is sent over one connection. Even when we travel to another part of the world or enter a dungeon, this one connection is maintained. This is a strong indication that WoW also uses proxies to manage user connections, analogous to the Eve Online method.

4.1.5 Case-study: ALVIC-NG

ALVIC-NG [85] stands for "Architecture for Large-scale Virtual Interactive Communities - Next Generation", and is an academic architecture for highly scalable NVEs. The implementation part of this thesis will be an extension to ALVIC-NG, so it is a good idea to explain the basic structure behind the architecture here.
This architecture is very much like that of Eve Online. ALVIC-NG too has proxy servers that handle the connections to the clients and the internal logic servers are only connected to the proxy servers, not to the clients directly.

There is one big difference however, and that is that zones in ALVIC-NG are not static, i.e. they can change size, shape and even the server they are managed on. As discussed before, static partitioning of an NVE world can be a large scalability issue when large amounts of players gather at the same place and it is precisely this problem ALVIC-NG tries to tackle. For this, ALVIC-NG uses a Region Management System (RMS) that controls which logic server is responsible for which zone. When a logic server is overloaded, the zone it manages is split into a couple of smaller zones and other logic servers are made responsible for the new zones. When the load decreases once more, zones can be merged again to reduce the number of different zones in the world.
The proxies play a vital role in this dynamic zoning system. They hide any topology change from the users connected to the NVE. If we were to implement a dynamic system like this in the Second Life architecture for instance, the users would have to establish a lot of new connections every time something changed. With the proxies, the user's connections stay the same, only the proxies have to connect to the new servers. This means less connections and less possibilities for problems during zone transitions. The system ensures very good load balancing with as little trouble for the user as possible.
By using this system, we can draw a conceptual parallel between the ALVIC-NG approach and the current trend towards virtualisation and cloud computing in other server systems. With virtualisation it no longer matters on which computer a particular site or application runs: all hardware is a large pool from which the virtualisation layer chooses the correct server for the job. This way, applications and sites can change servers when their load increases or decreases. When a server goes down, its applications can be spread out over other servers and all of this happens transparently and automatically. ALVIC-NG has a very similar concept, except this time not for applications or sites, but for zones in the NVE. Zones can dynamically change servers and be split across different servers as the need arises. This all happens without the user having any knowledge of it, internally in the network infrastructure.

Another difference from Eve Online is that the zones in the world of ALVIC-NG are intended to be seamless like in the world of Second Life, instead of completely separate zones like the

solar systems of Eve Online. This means that the problems of zone transitions and cross-zone events have to be handled. The dynamic regions make the implementation of such a solution a lot harder as regions can suddenly change neighbors, size and players. At the moment of writing, ALVIC-NG does not yet provide a robust solution to this problem. One of the purposes of this thesis is to see how these dynamic subdivisions can influence world state communications and how this in turn will affect bandwidth usage.

4.2 Area Of Interest

Whilst spatial subdivision techniques like zoning are rather coarse for determining which user should receive which information, Area of Interest (AOI) techniques provide much finer grained possibilities for filtering. Zones are fixed and are generally influenced little by the exact positions of individual players or objects. Areas of interest on the other hand are mostly focussed around a single entity, indicating the interests of that entity alone. Therefore it works on entity-level and not on world-level like spatial subdivision techniques. This way the area of interest can change dynamically when the entity moves, moving with it, and provide constant updates on the necessary filtering levels required.

4.2.1 Traditional aura/nimbus model

The simplest representation of AOI is as a circle with a certain radius around an entity. Everything within the circle is interesting to the entity, everything outside the circle is not. The traditional aura/nimbus model introduced by Benford and Fahlen [57] uses this simple representation to describe a very general and fine grained model for AOI management.

The model is based on two main components. The nimbus of an object is the radius of the circle in which an entity can see other entities. In other words, this is what we described above as the area of interest. A small nimbus means we are only interested in entities that are very nearby, a large nimbus means we want to see a lot of entities around us. The aura is in a way the reverse of the nimbus: it indicates from how far other entities can see us. When we have a small aura, we indicate we have little interest in interacting with others and we only want to be seen by those entities close by us. A large aura means we are open to interaction. This way, determining if an entity can see another entity is done by seeing if the nimbus of the first entity intersects with the aura of the second.
The aura/nimbus model is sometimes extended by the use of a focus. This is typically not just a circle around an entity but a triangular shape, usually indicating the view frustum of the entity, although other variations exist. The focus is especially useful when expressing interest in for instance audio. Like in the real world, we can hear audio from all around us (the aura and nimbus) but mostly we will be focussed on the audio that's coming from the person or entity we are currently looking at. This is easy to see if you imagine yourself talking

to a friend across your table in a crowded restaurant. For this kind of situation the focus can help indicate an even more fine grained level of interest that can be easily coupled to the viewing direction, which is another very powerful interest cue next to spatial closeness.

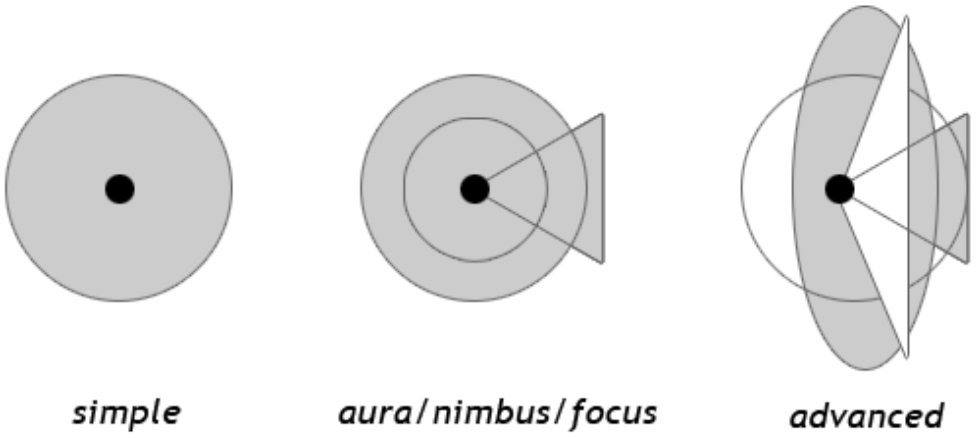

Figure 4.4: The aura/nimbus AOI model, various levels of complexity.

Many variations are possible for this model. The auras and nimbi can be ellipses instead of circles, they can be centered around another point than the actual entity, they can be different for other types of entities etc. Another very interesting quality of AOI's is that they are dynamic. The radii of the circles can change at runtime, indicating a changed interest for the user. The shapes and directions of the auras and nimbi can vary depending on the situations the user is in etc. This is interesting in combination with so-called adapter objects [88], which are entities in the world that cause the AOI's of the users to change. One example would be a large music stage, where the musician's aura would become very large so that everyone around the stage could hear his music. Another example could be a user that wants to play a mini-game inside the larger world and thus the game makes the nimbus of the player very small to enable him to focus entirely on the mini-game.

The aura/nimbus model is even more versatile. There is no need to have the same aura and nimbus for each type of network stream or world events. It is very well possible to have different AOI's for different kinds of information. For instance, we are usually only interested in audio from very close by, but visually we wish to detect entities from further away. We can define separate auras and nimbi for these streams and they can also evolve and change independently. Another possibility is not to have different AOI's per kind of network streams, but per kind of entity in the stream. For instance, as a tank in a war simulation, we want to detect the movements of enemy tanks from a farther distance than we want to detect the movements of foot soldiers. Here the network stream stays the same (stream of entity

positions) but the AOI is different depending on the entity type.

4.2.2 Practical usage

Because the AOI models are so versatile, the possibilities for their usage are endless. Some simple possibilities were already mentioned in the previous paragraphs, but even more extended applications are possible. Here I will focus on the main usages that can help with reducing the bandwidth usage of the NVE.

Level of detail

When we want to use spatial coherence to filter messages, it is rarely wanted that this filtering has a very hard edge. As we discussed in the section on zoning, if we would only receive data inside our own zone, there would be visual artifacts at the edges of zones. The same problems would become apparent if we would only have one level of AOI present. If the AOI would only be on/off for a certain kind of stream, we would see and hear entities suddenly when they entered our AOI, instead of gradually like in real life.
For this, we often desire some kind of level of detail (LOD). This is a concept well known from graphics rendering and it generally means that we are going to degrade the quality of the information gradually instead of instantly at the edge of the AOI. For this degradation we can for example use the algorithms discussed in the section on compression or some kind of variable prediction error as we will see in the next section on dead reckoning. This way further away entities are already displayed on screen in a rather simple representation and this representation will become more detailed as we approach the entity and our highest-LOD AOI envelopes it. This concept is demonstrated in figure 4.5.

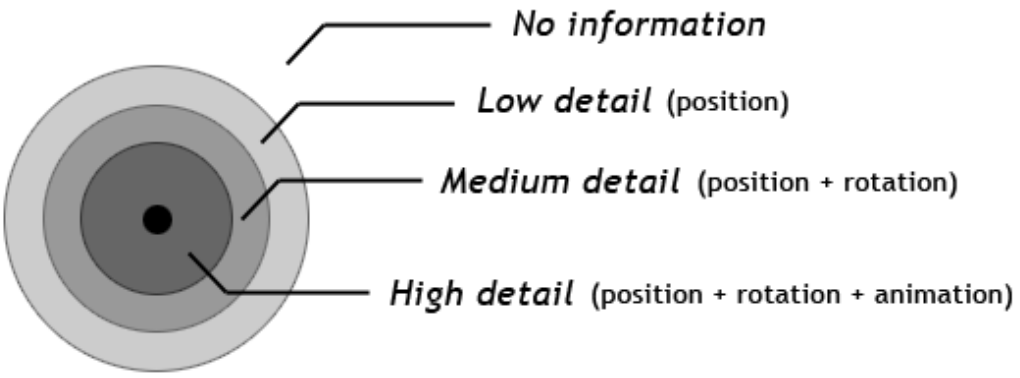

Figure 4.5: Using different AOI's for different levels of detail.

These different levels of detail can be continuous but often it is more practical to have some discrete levels like in figure 4.5. Here we can still have the problem of sudden and large visual artifacts when we change the discrete LOD level an entity is in so care has to be taken to prevent this. Another problem is when an entity is right on the edge of an AOI and is crossing the edge multiple times in quick succession. This would cause the entity to "pop" in and out of the higher LOD layer. To remedy this kind of artifacts, buffer zones or timeouts are often used. When an entity has just crossed into a new AOI it has a minimum time it belongs to this AOI (even if it crossed the edge again) or the edge is moved a little so the entity is no longer extremely near it. This way the popping is less of a problem.

Once again, these LOD boundaries need not be fixed. It is perfectly possible to have them adjust dynamically. One could for instance use certain amounts of entities we want to see in every LOD-level. For example we could say that we want a maximum of 8 entities at full resolution, maximum 32 entities in the medium LOD and 64 in the lowest LOD. This way the distance between the user's avatar an the 8th nearest entity would determine the edge of the first LOD, the 40th nearest entity would determine the next edge etc. When the entities move closer or further away, the edges will move with them, dynamically adjusting the AOI.

Zoning transitions

As we discussed in the section on zoning, a large problem is the transitions between zones and the events that can happen across different zones. A solution is to send information on not only your current zone, but also all the neighboring zones. However, this costs a lot of extra bandwidth and processing. It would be better if there were some kind of soft-transition at the edges, where only necessary information of the neighbor zones is transmitted. The real difficulty then is to determine how large this soft-transition has to be.
This is where AOI comes in. With the AOI definitions for all players on the edges of a zone, we can determine very fine-grained what data from the other zones is really needed. Thus if we already use a certain AOI model in the NVE we can easily use it to improve our zoning implementation. Only the really necessary information is being sent and it is clear how close a user is to a zone boundary, allowing other servers to prepare for a possible zone crossing when they come too close.

Dynamic bandwidth usage

In the interest of this thesis, AOI is probably one of the most versatile techniques to dynamically change the bandwidth usage of the NVE. Even the simplest model with a single circle and radius allows us to adjust this radius depending on the desired bandwidth usage. When bandwidth is scarce, the radius will be small and vice versa.
When combined with more advanced AOI models, like different AOI's for different streams

and LOD's, it is possible to very closely tune the different AOI's depending on the bandwidth requirements. This tuning is interesting because it can be done by tuning parameters that have a logical meaning in the NVE world, like the radius or size of the AOI. Other techniques like aggregation and compression also have parameters which one can alter for different results, but it is not always directly clear how these parameters will affect the logical results. For instance, increasing the timeout of an aggregation algorithm will probably delay the packets more, but what influence will this have on the overall user experience? On the other side, when we decide to reduce the AOI for video because bandwidth is scarce, we will immediately know which entities will stop sending video to us and this selection can be done in a logical manner with a real-world analogy. Note that here we can for instance choose to just stop the video stream but to leave the position streams intact. This is different from a typical zoning-only approach where we can usually only subscribe or unsubscribe from a complete zone and all its data. This could be solved by the use of per datatype zones, which in turn increases the complexity of the zoning and could lead to more calculations to check if an object is in a specific zone or not.
This way AOI makes it easy to manipulate some consistency requirements of the NVE to reduce bandwidth and this in a very understandable way. This is important when trying to maximize the so-called Quality of Experience (QoE) of the users of the NVE, a subjective measure of how well the users can interact in the NVE and if they are not feeling like something is wrong with the program. This QoE is often lowered by lag on the network or when clear artifacts (like popping players at zone/AOI edges) are present because this can ruin the feeling of immersion in the NVEs world for the user. AOI management can help improve this QoE by explicitly defining strategies for determining which streams and types of content in which areas should be transmitted and which can be omitted in the case of a (sudden) bandwidth shortage.

AOI calculations performance

So AOI must look like the ideal solution for bandwidth and interest management now. Sadly all this versatility has its price. A simple and pure implementation of the circle AOI would require n checks for a single AOI, where n is the number of entities in the world (or for instance in the current and neighboring zones when we use AOI in combination with zoning). This means that the complete algorithm is O(n^2) when we compute a single circle AOI for every entity. This can quickly become a serious bottleneck if the number of entities increases and we are only talking about a single circle per entity. When the AOI's are more complex and several AOI's are used per entity, the computation costs will be much larger.
Because of this, the traditional AOI models are rarely implemented in that exact way. Most implementations will use some form of approximation to this model. In fact, the zoning model described before is a very coarse representation of the AOI idea. Other implementations will try to keep the AOI definition more fine-grained. The method suggested by Boulange et

al. [60] subdivides the world into small triangles using a delauney triangulation which can be done as a preprocessing step. In our implementation we will look at other methods for representing AOI's in a computationally attractive manner.
So we see that pure AOI models are probably not very computationally scalable but that approximations that possibly use some form of precomputed data can be used to obtain relatively fine-grained results similar to what a naive implementation might deliver.

4.2.3 Network architectures

While the other spatial methods like zoning possibly had a significant impact on the network architecture of the NVE, for AOI filtering this is usually not the case. AOI's can mostly be used both client and serverside and in peer-to-peer systems, albeit in different forms.
Most commercial Client-Server systems will probably just perform some kind of distance-based AOI with a LOD-scheme. The radii and shapes of the AOI's depend on the currently available resources of the servers and on the current state of the user. This means the server will implicitely change the AOI's depending on the user's role or situation in the world, without specific user interaction or guidance. When we revisit the example of the musician on stage, the server can automatically detect the user is on stage and expend his aura to accomodate for his new role as singer. Based on fixed gameplay rules and knowledge about the virtual world, the server decides what the AOI should be for a particular user.
Other applications can allow the users to indicate some of their specific interests themselves. This is mostly not the case for games as it would be possible for a user to indicate that he wants to see all the enemies across the entire map, while mostly only the enemies that are near and visible should be known to the user. But when we consider collaboration environments where all users have mutual trust, the user-assisted interest management becomes more attractive. Users can for instance indicate when they want to receive video or audio from a person, with which other users they want to interact and in what kind of interaction they are specifically interested, as this kind of data is mostly not just available from gameplay-rules in these environments, in contrast to games. Here it is important to notice that the server is still responsible for the actual filtering of the updates. The clients only indicate their specific wishes, the server has to actively try to make sure these preferences are followed. For the research in this thesis, it can mean that the server cannot comply to the desires of the clients because there is for instance too little bandwidth available to send the video streams of the five users the client has indicated. Then there should be some kind of feedback mechanism to the user informing him that his interest specifications could not be met by the server.
Client-Server setups can also profit from a central bandwidth distribution strategy if other techniques are used to shape the traffic as well. In that kind of setup, AOI filtering is just one of several active methods to reduce bandwidth usage and a dynamic scheme can be used to change the variables for the different techniques to regulate the bandwidth. As we will

discuss in chapter 5 and 6, AOI filtering will be the central idea for our own implementation of bandwidth reduction techniques in ALVIC-NG.

AOI filtering is different in an interconnected P2P setup. Whilst here we can also make decisions on the AOI size and shape based on gameplay or use user-assisted filtering, the peers themselves will be responsible for the filtering. This means peers will have to send their preferences to each other peer close to their location so everybody knows what kind of data the peer is expecting. When working with gameplay-based filtering, the AOI should be large enough to accomodate for all possible gameplay events and this means the number of clients the peers have to connect to can also increase. The peers themselves are always responsible for checking if a certain update they want to send is interesting for a possible recipient, which takes up extra processing requirements at every peer. This also allows for cheating, as peers can refuse to send certain updates to certain players, causing them not to receive crucial information whilst it is in their AOI.

A multicast architecture is not very fit for the use of AOI-based filtering and Singhal and Zyda actually stated it is impossible in a standard setup [86]. As we have seen in the section on zoning, multicast groups are mostly divided so that users in a single zone send all of their updates to that multicast group. This allows for users to indicate their interest on a zone-level, but nothing more fine-grained than that. When we consider using multiple multicast groups to send information about the same area however, there are more possibilities. Imagine for instance a zone stretching from (0,0) to (100,100) in 2D cartesian world coordinates and a single multicast group for this zone. There could be extra multicast groups for the areas from (0,0) to (20,20), (0,20) to (20,40), (20,0) to (40,20) etc. where the peers in that specific area also send their updates to, in addition to the updates they send to the group for the complete zone. This will allow a more fine-grained area of interest to be specificied based on closeness as the peers only subscribe to the smaller groups they need. We can even use a level-of-detail scheme this way by sending full updates only to the groups of smaller regions and sending less frequent and smaller updates to the group of the entire region.
So it's possible to indicate a finer-grained AOI if we use extra multicast groups for further spatial subdivision. But what if we want to perform filtering on specific network streams? Some users won't be interested in the video streams of area (20,20) to (40,40) so we can't use the same group for position updates and video streaming. We can make a new group for the video in this small area to remedy this and some more groups for other types of network streams. Now we also want to filter on entity type: I only want the positions of tanks in area (40,40) to (60,60), not those of foot soldiers. No problem, we just add extra multicast groups for the position stream in that small area for every possible entity type. We could continue like this for a while, extending the setup with extra multicast groups to try and provide a very fine-grained AOI-filtering mechanism, but the truth is that this is not practical. The

number of available multicast groups is very limited and network interface cards can only be subscribed to a limited number of groups at the same time. There is also the problem of subdividing the groups so all peers know which group provides which information and when the world would be extended, more multicast groups would be needed. Multicast simply is not suitable for this kind of filtering. Granted, we could use a different start setup, using entity-based multicast instead of zone-based, but the eventual result will be the same: we need too many multicast groups to use the kinds of filtering Client-Server and interconnected P2P networks can perform with simple, local calculations.
One might say that we do not need this kind of fine-grained interest management. We could opt to send the updates about tanks and footsoldiers in (40,40) to (60,60) in the same group. The peer that is not interested in the foot soldiers can simple drop these packets upon receipt, only really processing the tank updates. This way the endresult for the user is the same as if the updates were never sent to him. This can work well if the main goal is that the user can specify his interest very fine-grained and that this interest is followed. For the purpose of this thesis however, we are specifically looking for bandwidth scalability and then the filter-on-receipt method is actually producing incredible overhead as updates are being transported by the network, consuming bandwidth, only to be dropped at the receiver, rendering them useless. So when using multicast as your NVE architecture it is doubtfull an extended AOI scheme will help you to increase bandwidth scalability.

4.3 Dead Reckoning

The technique of Dead Reckoning (DR) is the main technique for a filtering concept that uses short-term future prediction. The technique of DR is in principle not focussed on reducing bandwidth or even reducing computation power. The main problem DR deals with is the smooth appearance of moving objects in a distributed application. This is because motion is a continuous process but the positions we send over the network are just discrete samples of this continuous movement. If the update rate of these positions is high enough, we will not notice the discrete steps and the object will appear to be moving smoothly. When the update frequency is lower however, entities would jump from position to position, introducing obvious visual artifacts. Another use is the hiding of lag. When we receive position updates from other players, these updates are in fact from the past, as the packet has needed some time to travel over the network. The real position of the entity is already different so we can't just use the position update in the packets because the simulation would always be inconsistent.
This is where the technique of Dead Reckoning comes in. It is based on a very simple physical principe that when we know the current position of an entity, its current speed and direction, we can predict where this entity will be in the future if it maintains a constant speed and direction. This was very important in the first NVEs. Systems like DIS were aimed primarily at the simulation of military entities like tanks and airplanes. Those entities will usually have

a certain goal in the world they need to navigate to and they do this in a predictable and straightforward manner, usually following long straight roads or trajectories. What is more, tanks and airplanes have limited possibilities for quick changes in their movement. A plane cannot make a direct 90 degree turn for instance. This kind of maneuver takes time and there are known physical bounds to the amount the plane can turn. Using this kind of information we can make an accurate physical model for the entities with which we can predict very accurately their movement. This means the update rate for these entities didn't have to be very high in most cases as most positions could be predicted, allowing these systems to scale to several hundreds of entities.

4.3.1 Dead reckoning models

To do an accurate physical prediction of a position, we can use a couple of variables. The current position is the starting point and we can augment the prediction by using direction, speed, acceleration and jerk. These last three factors are all interrelated, as speed is the first derivative of position, acceleration the second and jerk the third. One might think that the accuracy of the prediction will increase if we know all of these parameters, but this is not necessarily true. For the previously mentioned entities like tanks and airplanes, acceleration and jerk are mostly constant or vary slowly and within known bounds. When we look at other entities however, like the avatar of a player playing an FPS game, the acceleration and jerk will vary quickly and with large amounts in very short timespans. This means that unless we can measure these changes very accurately and quickly, the incorporation of these variables in the prediction will lead to big errors and can actually reduce the accuracy of the prediction. Also, because acceleration and jerk will change so frequently and quickly, we would have to send a network update every time they changed, undoing most of the possible bandwidth optimization of DR.

This is why there are several models for Dead Reckoning and the model to be used depends on the type of entity. Highly indeterministic entities like player-controlled avatars mostly use the first order model, which only uses position and current speed. Entities that are more deterministic like tanks often use the second order model, which also uses acceleration. Jerk is rarely used as it contributes little to the accuracy and increases computation and network traffic [88]. We have to note that the simple model of only sending discrete positions (as we have always assumed up until now) can be seen as the zero order model, which is simple to incorporate into a DR implementation to be used in for instance a higher Level of Detail (see the section on AOI above).
For the NVEs that were developed for real-life military simulations often more advanced DR models were used, which not only use speed and acceleration but also the known physical bounds of these variables and sometimes even data about the common ways to control a

vehicle. These models are very entity-type specific however and only work well on these kinds of entities. For most modern-day games and virtual worlds, the simpler forms of DR are very appropriate and usable.

4.3.2 Convergence and consistency

As described in the previous sections we will use DR to predict where an entity will be and this means we can also move the entity to those positions in our local view of the world. When the entity does change its speed or direction however, these parameters will have to be updated by sending a network packet. Through network lag, the local predicted (and thus also rendered) position can be considerably different from the real position when the network packet arrives. This is certainly the case with fast-moving entities like race-cars or airplanes that can travel far in a very short timespan. For instance, a car traveling at 100 km/h will travel about 27 meters per second. This means a lag of even just 200 ms can cause the position of the car to be more than 5m off. This problem is magnified by the fact that the position we receive is actually a position already in the past. It was sent 200ms ago, so the remote car isn't there anymore, it is already 200ms from that position. This means the real current position of the remote car is 400ms different from our predicted and rendered position. Figure 4.6 shows this problem graphically.

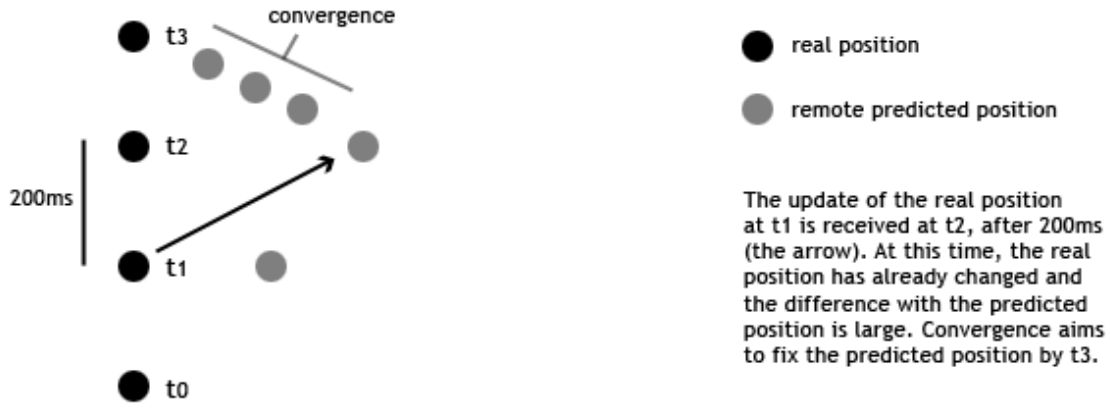

Figure 4.6: The predicted position can change from the real position under influence of lag.

This means that when we do receive the update, we have to find a way to correct the predicted position and make it coincide with the real position. Note that this is not the real position as received in the update (as this is already in the past) but the real position at the time of receiving which can be extrapolated using the previous update and the currently

received update. This process of adjusting the predicted position to the new path is called convergence. There are several possibilities to do this.

Convergence models

The first method is called snap-convergence: we just change the position of the entity to the newly predicted position at the time of receiving the update. This is a very easy scheme but it can cause serious visual artifacts as the entities can jump around on the screen if the real position is significantly different from the predicted position. In figure 4.6 this would mean we immediately change the predicted position to the newly predicted position at t2 upon receiving the update. Figure 4.8 also gives an example of snap convergence after receiving a position update.
The second method is linear convergence. Here we will use a linear interpolation between the current predicted position and a newly predicted position in the future. This will cause the positions to gradually become correct but this takes some time (whereas the snap-convergence will correct itself immediately). In addition, as the entity might have to travel a long way to the new corrected position, it could be we need to increase its speed to make sure it arrives there at the correct time. This can be seen in figure 4.6 where linear convergence is shown in the image, as well as in figure 4.8. Upon receipt of the update at t2, a corrected position will be predicted for t3 and the entity will converge to this position along a linear path. This method is somewhat better than snap-convergence for visual appearances but it can still cause sharp corners and unnatural direction switches.
The third method, cubic or quadratic convergence, tries to make the convergence as smooth as possible. Just like linear convergence it will try to converge in the future, but instead of using a linear path it uses a cubic spline which is fit through the known and predicted points to create a smoothly varying path.

The decision of which method to use depends on the type of entity and of the situation. If we are still sending relatively frequent updates, snap-convergence might be a good choice as the position will not differ by that much. The same goes for entities far away (if used with a LOD scheme for instance). Linear convergence is used when the positions differ too much and cubic interpolation is for entities that should not have a rough change in direction or speed (like airplanes, tanks, race-cars etc.).
Other parameters can be used to decide which convergence scheme to be used. Singhal et al. [87] used a method called Position History Based Dead Reckoning. Here they look at the recent position history of the entity. If these positions are smoothly changing, cubic interpolation is used. If they are not smooth, linear interpolation is used. This is because when the object is not moving smoothly, the linear interpolation will probably be a better approximation to the correct path than a cubic interpolation.

Another parameter can be the computation needs for every scheme. Cubic interpolation will require a lot of resources whilst snap convergence is very cheap. The computational requirements for convergence are often one of the largest costs for using DR and they can be quite significant when dealing with a large number of entities.

Consistency problems

The user should note that through the use of DR and the different convergence algorithms, there is an inherent tradeoff between network traffic generated by position updates and the consistency of these positions at the remote stations. The predicted positions can be significantly different from the real positions and under the influence of network lag, these differences are even enlarged. When using linear or cubic interpolation, there is even more time where the positions differ as the convergence takes a certain amount of time to be performed. This means that there can be large periods of time where the positions will not be the same, especially if the entities are moving at fast speeds or if their movement changes frequently.

This can be a significant problem in games like an FPS where the positions should be very consistent to ensure the gameplay has a certain logic and fairness to it. This can be illustrated with the following problem, illustrated in figure 4.7. Suppose a user A is walking in a straight line. The predicted positions for user A seen by user B are on this straight line. However, right before user A passes a wall, he changes his direction and makes a right turn. Since it takes time for this direction change to be sent over the network, user B continues to render user A on the straight predicted path, causing him to see user A passing the wall while user A shouldn't be visible for user B because he's behind the wall. This situation is further complicated in an FPS because user B could have shot user A before the update has arrived. Then it is very difficult to return to a correct state. Either user A will die and it will seem like user B has shot him through the wall, or user A will live and user B will be confused as he thought he clearly saw and hit user A. Resolution of this kind of game-logic altering anomalies is very difficult and is often just ignored by most implementations, causing possible artifacts which are indeed seen in many games. Other problems are described by Mauve [79]. For the consistency to be at an optimum, the lag between sender and receiver should be relatively well known when using convergence. This is because when we receive an update, this was the position at some time ago, more precisely the lag time. So if we have a delay of 200ms on the network, the received position will be 200ms in the past. For convergence to work, we need to first predict where the entity is at this moment and then predict a new position in the future to which we want to convert. An accurate estimate of the lag can help with predicting where the entity is at the exact time we need to know its position. If we don't know the network delay, there is no way of knowing when exactly in the past the update was sent and thus also no way of knowing where the entity is at this very moment.

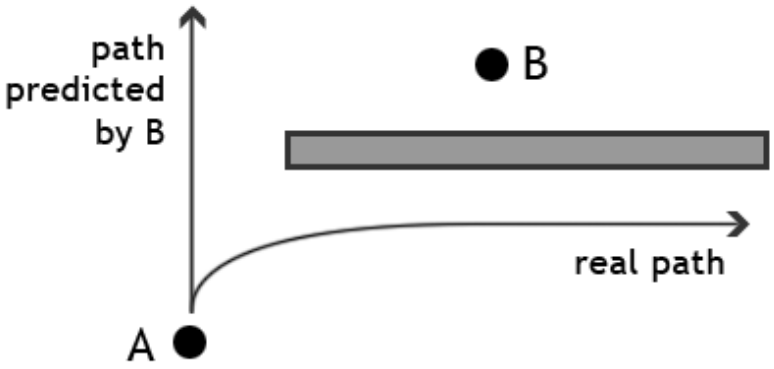

Figure 4.7: Consistency problem with Dead Reckoning.

Dead Reckoning is a so-called extrapolation approach. This means we try to extract new information from known and previous information, in this case mostly positions in a 3D world. The other approach is to use interpolation. This means that we are going to introduce an artificial playout-delay [88] on the receiver side. So when an update is received, it is delayed for about 100ms before it is used to update the local state. This ensures that there is always at least one extra update received between the receipt of the first update and the actual local state adjustments. Using these two received updates, one can calculate a so-called interpolated state somewhere in between these two states. So instead of predicting what is going to happen, we wait until it has happened and we have heard about it and only then do we update our state. This will in general lead to more consistent and logical states but it also introduces an extra delay and implementation difficulties. The local state will never be the exact state of the world and some of the problems of Dead Reckoning can still occur, like dead-man-shooting. However, for fast-paced action games like FPS's where DR is in general a less interesting method because of the fast changes in movement, interpolation can be a good choice. In combination with a time-rollback mechanism hit-detection for bullets hitting other players can still be done correctly, even though the players are shooting at slightly delayed entitties on their screens. Valve uses an interpolation model in the source engine [89] whilst Quake World uses a Dead Reckoning approach [63]. Both of these are FPS games or engines. The simplest consequences of these methods is that with interpolation with time-rollback you can shoot a player where you see them on your screen to hit them. When using Dead Reckoning, you have to aim a little in front of the moving entity you are trying to hit, called "leading" your aim. The amount of leading you need to do to hit the entity is directly coupled to your latency to the game server.

4.3.3 Dynamic bandwidth adjustment

In this section about Dead Reckoning we have put a large emphasis on problems with consistency and the convergence algorithms needed to be able to use Dead Reckoning. This is because the previous methods were often simple in their tradeoff between consistency and bandwidth and about how they would influence world state. As we have discussed for Dead Reckoning it is not this simple. Depending on the type of application the errors introduced by Dead Reckoning can be very serious and they can have a huge impact on how the user experiences the virtual world. Thus before discussing the practical applications of Dead Reckoning for bandwidth scalability, a word of warning. Dead Reckoning allows you to trade consistency for bandwidth in a very direct and logical manner, but the balance between the two is often very difficult to find and it will always depend on the specific situations in the virtual world. When used incorrectly it will produce negative visual and logical artifacts and even when used correctly the gained bandwidth savings might not be consistently large for every kind of entity and application.

This fickle nature of Dead Reckoning will become even more apparant when you understand the method for using it to dynamically change bandwidth usage. The basic idea is that Dead Reckoning will retain a good consistency if the remotely predicted positions don't diverge too much from the real positions. Mostly those two paths can differ a little bit, as long as the general movement remains the same and the eventual endpoint is correct. Remember that in the beginning we said that DR would help if entities retained the same speed and direction and that we only had to change updates if those changed? Now this isn't even true anymore. As long as the local changes in direction and speed are small enough to retain a relatively good prediction, there really is no need to send those small changes, further increasing the possible bandwidth profits. Now there is only the question: how large can these differences become until we need to send an update?
The answer to that question is to use a certain error treshold around the real path that indicates how much the real path can differ maximally from the predicted path. Only when the predicted path exceeds this error treshold should we send an update to recover the predicted path. There is a problem with this approach however, as the sender of the updates only knows about the real path and not the remote predicted path. Luckily, this can be easily remedied by having the sending user perform a local prediction based on the updates he has sent to the others. As this is also the only information the remote users use to perform their prediction, we can reconstruct their calculations perfectly on the sender's side. This way, the local user can perfectly know how the other users are viewing his avatar and thus also when the remote predictions will be too different from the real path. This concept is demonstrated in figure 4.8 where the predicted path reaches the error treshold at time t1, at which an update is sent. In this example we ignore network lag, so the update is received at the remote users at t1

as well. The figure shows two different possibilities for convergence to the new path by the remote users.

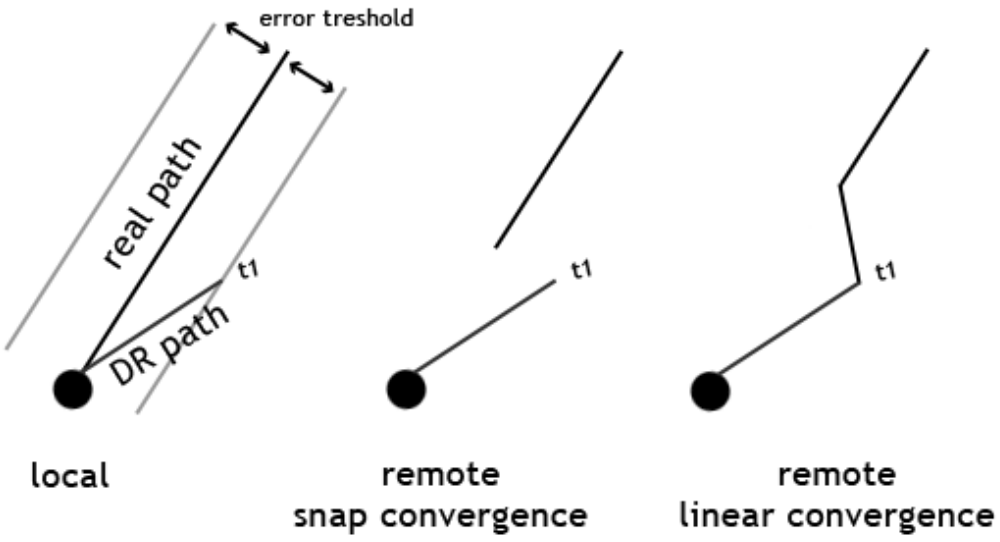

Figure 4.8: Dead reckoning with local prediction and error treshold, disregarding network lag for convergence.

This approach becomes even more interesting when we use a dynamic error treshold. This way we can dynamically change the consistency-throughput tradeoff generated by our use of Dead Reckoning. Larger tresholds will allow the paths to diverge more widely from eachother, sacrificing consistency for bandwidth. On the other hand, small tresholds will make sure the consistency is quite high, but will lead to more updates being sent. The decision on which treshold to use depends on the situation and expected results. If we want to use it as a direct bandwidth limiter, we can simply adjust the treshold as bandwidth increases or decreases so that we stay within the alloted limits. Another option would be to use it in conjunction with a Level of Detail scheme as we dicsussed in the section on AOI before. Entities further away could have a large error treshold as the accuracy of their positions will typically be less important. Entities closer by will have a much smaller error treshold, resulting in a more accurate position. This approach will ensure lower bandwidth usage in general but it isn't as tunable to the current bandwidth limit for the connection as the simpler method.
This way we see that dead reckoning, a method originally intended for lag compensation and smooth movement, can be used as a dynamic bandwith regulator by changing the consistency of the virtual world on a per-entity basis.

The reader might have noticed that we are sending more data than just the world position

when using dead reckoning: we also send speed, direction and possibly acceleration. This might seem like a paradox as we are trying to limit the amount of data we send, not increase it. This is also directly oppositie to the method of logical compression discussed before. However, even though we increase the payload size per sent packet, the actual amount of packets we send is decreased severely. The larger payloads are only a problem if we would work with a high frequency of updates but DR reduces this frequency and it will thus reduce the bandwidth usage as a whole. Furthermore this will reduce the packet header overhead as discussed in the secion on aggregation. A less interesting aspect of this approach is that packet loss can be detrimental to the resulting consistency. This is why a reliable protocol is mostly needed for dead reckoning, especially if the error treshold is large and the update frequency is low.

One additional advantage of using prediction is that the predicted trajectories can be used for other means as well. It enables us to pre-empt on the user's actions and this in turn can help us achieve better bandwidth usage. A good example is that it helps to determine if a user is going to cross a zone boundary. If a user is walking in the same direction for a longer period of time, the probability that he will continue on his path is quite large. We can then see if this predicted path crosses zone boundaries. If this is the case, we can already start sending some state information about the neighbouring zone before the user has crossed into it. This will spread the state information transmission of the new zone over a longer time-period, reducing the bandwidth load when the actual crossing occurs. This also ensures that the user will already have some visual data like textures and geometry when he enters the new zone so he can start interacting very quickly, something other environments like Second Life [46] often lack. Without this form of preloading, the data is only coming in when it's actually needed and when it's a lot of data, multiple things will be severely delayed, leading to large artifacts and delays in the interactions. A form of entity-based priority can further help to determine which entities should be preloaded as soon as possible.

4.3.4 Network architectures

Dead Reckoning is an interesting method because it can help reduce both upstream and downstream bandwidth usage for a Client-Server system. Most of the previously discussed methods are only performed on the server and will help in reducing the data the server sends to the client. Techniques like zoning, AOI and aggregation have no possibility to reduce the amount of data the client has to send to the server. Only compression and dead reckoning can possibly help reduce the data the client has to send. This can be important when the client's upstream bandwidth is low, as is the case in mobile cellphone networks for example. When using dead reckoning if the client is sending position updates to the server, these updates can be cut down by using the dynamic error treshold model described before. This can be even

more important if the client has to send updates for more than one object, as can be the case for an RTS game.
We should note however that this approach is only to be used when it's really necessary for the bandwidth usage. Normally we want the server to have as much information as possible because it is always possible to remove some accuracy but it is difficult or impossible to add accuracy to the user's actions if the user is only sending some of his actions. If we take two players next to eachother for example, one of them on a mobile network and the other on a cable network. The player on the cable network will want to receive the best quality from the other player because they are so close, but the server is unable to comply as the mobile player is only sending dead-reckoned positions to the server and the server cannot improve the accuracy without risking some extra consistency loss. This is why in most cases it is best to leave the client-to-server traffic untouched by filtering approaches and only use compression to reduce the bandwidth usage.

When looking at P2P systems we once again have to differentiate between interconnected peers and multicast. For interconnected peers dead reckoning is as usable as the other described methods in this chapter. Every peer has to decide by itself how it sends its updates to the other peers and it can choose to use dead reckoning for this to limit its own bandwidth usage. Once again like with the client-server above however, this can be dangerous as the other peer might want a higher level of accuracy. Here there is no central server that might provide this, so peers are totally dependent on eachother.
Multicast is quite alike this setup when using normal dead reckoning. We can just send dead-reckoned updates to the appropriate group depending on our bandwidth preserving needs. Multicast becomes more interesting when using DR in order to obtain different levels of detail, as discussed in the section on AOI's. We can easily say that a larger error treshold means a lower update rate and a lower level of detail. This can be interesting in multicast environments. For instance, a peer could send updates with a small treshold to its current group but he can also send updates with a higher treshold to the neighbouring regions. This way, the peers in the neighbouring regions will already receive some information about the movements of the peer and the zone-transitions are less direct (which is an important problem in multicast networks as discussed previously). Another complementary approach is to have different multicast groups for the same zone, each representing a different level of detail. Peers in a zone send accurate updates to the high LOD group and DR updates with a large treshold to the low LOD group. The other peers can then decide to which LOD-group they subscribe for this region. Note that these methods willl increase the upstream bandwidth usage of the peers as they have to send multiple versions of the same packet, but the downstream bandwidth of the other peers can be regulated much more closely by themselves.

A final usage of DR is to help in reducing server-to-server traffic. We have not yet talked about

this topic in this text but it will be more important in the next chapters and this can serve as a short introduction to the issue. When we have an advanced network architecture with a lot of servers like for instance the proxy-logic server setup of ALVIC-NG discussed before, these servers will usually have serious inter-server traffic. This traffic is needed to enable cross-zone events, user zone crossing and server migration. For example, a proxy server has to receive data from multiple logic servers to accomodate even a single connected user. As discussed in the section on zoning, to ensure cross-zone events the proxy needs not only data of the current zone, but also from the close areas of neighbouring zones. This means that the logic servers responsible for these neighbouring zones also have to forward their information to this proxy, increasing their outbound traffic. As there is mostly no guarantee that users in the same zone are also connected to the same proxy, the problem can get worse because the logic servers have to send their information to multiple proxy servers, in fact sending duplicate updates.
These large amounts of traffic are usually not a big problem as most of the commercial systems are hosted by a professional company which ensures the servers have high-bandwidth links to eachother [13]. There is recent research however that indicates that proxies could benefit from being located closer to the users [66], for instance at a hub of an Internet Service Provider (ISP). This severely reduces the distance to the user ensuring lower ping times. This would mean that the proxies will probably no longer be connected to the logic servers with very high-bandwidth links and the extra traffic can become a problem. This is where LOD and DR techniques can also help moderate this traffic. When sending updates to proxy servers that don't directly have a user in the zone the logic server manages, the logic server can use DR for example to reduce the traffic needed to keep the proxy up to date.
Other more advanced techniques are needed to reduce this traffic even futher and the exact usage of zoning and the network setup can have a big influence. This issue will be further explored in the next chapters and will be a part of a theoretical overview of the bandwidth scalability of ALVIC-NG.

Chapter 5

Real-life examples and implementation recommendations

In the previous chapters we have discussed different methods that can be used in the development of scalable NVEs and that can provide (dynamic) bandwidth scalability. Now it is time to see which of these methods are actually being used in real-life systems that have been successfully deployed. This will hopefully give a good indication of which techniques are important and should be implemented in a new NVE framework, as is the goal of the implementation part of this thesis. We first look at some networking middleware for both small-scale and large-scale setups and after that we describe how some contemporary Massively Multiplayer Online Games (MMOG's) and virtual environments work.
It should be noted that for commercial systems, often there are few details available about how they work internally. We can however deduce some information from traffic analysis or the scarce information provided by the developers on their Internet pages. Whilst most of these sources are not scientific in nature, they can still provide valuable information on which techniques are interesting for practical usage in real situations.

After we have discussed these real-life examples, we can draw the final conclusions of the first part of this text before discussing the implementation in chapter 6.

5.1 Engines and middleware

In this chapter we will look at networking middleware and MMOG engines alike and we will try to deduce for each of these platforms which choices they have made for bandwidth scalability and overall setup of the implementation. There are many different engines available and so it is difficult to only discuss a limited amount. The different frameworks discussed in this chapter were chosen because they are popular and often used for real applications, because they made an interesting and divergent choice for a specific technique, because they give practical tips for

NVE development, because they describe themselves as complete NVE-development engines or a combination of these factors. This should give us a good indication of which engines are being used and of the functionality that they provide in terms of bandwidth scalability; it will also illustrate what engines that market themselves as "perfect NVE solutions" choose as techniques.
A large group of middleware discussed in this chapter offers a lot of functionality for game development in general. For the interest of this thesis however, things like graphical world design or an integrated all-in-one editor are not that interesting. This is why we will discuss primarily those aspects of the middleware that have direct ties to possible bandwidth usage and that give a good insight in the concrete possibilities and limitations of the system in this respect.
We start by discussing the simpler, more low-level systems and continue with the larger, NVE- and MMOG-centric frameworks.

5.1.1 RakNet

RakNet [40] is an open-source, "cross-platform C++ game networking engine" that is used by many large commercial game studios. Its major features are: secure connections, autopatcher, remote procedure calls, voice communication, NAT punchthrough, a robust communication layer and object replication. The latter two points are of special interest to us here.

The robust communication layer [1] is a relatively low-level interface that wraps basic networking classes like sockets and addresses. Next to this it provides a complete BitStream class to make the serialization of objects easy. This allows for all kinds of variables to be written to a compact bitstream where every variable is stored as efficiently as possible. Lastly it defines it's own UDP-based protocol to provide a series of different options to developers. Some of these options are similar to features from TCP, like congestion control and configurable reliable and ordered delivery. In addition, it provides sequence numbers and timestamps which are not only used by the communications layer but can also be used by the application developers for event ordering, synchronization or lag estimation.
RakNet sends its packets on set times, known as ticks. Messages that are ready between ticks are buffered and are packed together into a single network packet. Thus we can say RakNet performs a type of timeout-based local aggregation with a short, fixed timeout (the standard value for this timeout is 10ms). The other possibility is also available: if a single message exceeds the MTU size and can't be transmitted in a single packet, RakNet will automatically split it over multiple RakNet-packets as needed.

ReplicaManager3 [2] is an optional plugin to RakNet that helps developers to create simple

[1] http://www.jenkinssoftware.com/raknet/manual/systemoverview.html, 12/08/2010
[2] http://www.jenkinssoftware.com/raknet/manual/replicamanager3.html, 12/08/2010

networked games. Developers extend their own C++ classes from RakNet's base classes and object creation, sharing and deletion can be performed automatically by RakNet. RakNet provides two possibilities for object serialization. The standard approach is to serialize the complete object when anything changes. As discussed previously in the section on logical compression this is very easy to program but it will waste a lot of bandwidth. The other approach is to only send the variables that have changed. RakNet allows developers to do this by either setting "dirty flags" themselves or it can automatically keep track of the variables' changes, requiring more processing and memory. All in all the ReplicaManager3 in RakNet is an easy to use option for developers that need to synchronize a limited number of objects across different users.

Thus RakNet is focussed on the low-level aspects of networking and protocol design and provides options for easy logical compression, basic aggregation and a UDP-based reliable protocol. However, it is certainly not a complete large-scale framework. RakNet does not provide any high-level features like advanced network architectures, zoning or even dead reckoning. The object replication plugin is good for small-scale straightforward games but not a good fit for large numbers of distributed object that need to be created and destroyed dynamically across many users.

5.1.2 ReplicaNet

ReplicaNet [42] is a commercial OO C++ library that completely revolves around the idea of object replication for building distributed applications. So where RakNet was mainly a low-level networking library with a possible ReplicaManager plugin, ReplicaNet focusses on these replicas and tries to hide the networking aspect of multiplayer games from the developers so they do not have to worry about it [3]. This way developers can just create objects in C++ and change their variables without having to deal with how it will get transferred to the other users.
ReplicaNet is very similar to the RakNet ReplicaManager3 discussed previously. The variables are automatically tracked for changes and these changes are propagated over the network as needed. There are also a number of differences however. ReplicaNet uses a version of the LZMPi generic compression algorithm for example to reduce the packet sizes.
Another important concept is the use of the Replica Object Language (ROL). This special purpose scripting language has a simple syntax and allows for developers to indicate in a simple way which variables should be synchronized over the network and in which specific way (reliable or unreliable). These ROL scripts are then automatically compiled to C++ code for use in the application. This allows for practical separation of gameplay and network programming, a feature often wanted for smaller games.

[3] http://www.replicanet.com/about.html, 12/08/2010

What is most interesting about ReplicaNet for this thesis is that they also provide some bandwidth scalability and traffic filtering options. ROL gives the possibility to provide some protocol optimization by indicating how every variable should be sent in bit-form. It is also possible to indicate a certain variable has to be dead reckoned and to set the maximum local error threshold at compile time.
The standard way of using ReplicaNet is that it will perform simple distance-based filtering (the simplest form of AOI) with the standard distance being positive infinity (so all objects are transmitted to all other objects). This means we can easily change the radius at runtime to reduce the bandwidth usage. ReplicaNet also offers the possibility to turn off this distance based filtering and define your own filtering method. This means the developer can specify which users should receive updates for a specific object. This could be done using other high-level mechanisms like zoning as discussed previously. However, ReplicaNet does not provide an implementation to any of these other methods, only for the simple distance-based filtering. It also means that the built-in distance filtering cannot be used in conjunction with other, self-defined methods.

The last interesting aspect of ReplicaNet is that it provides a form of load-balancing. Every object has a so-called master that is authoritive about the changes that can be performed on the object. When the load for a particular object becomes too large and the current master cannot handle this load, the system automatically assigns another entity in the network as the new master. This also adds a good fault tolerance as other entities can take control over objects when a master crashes or loses network connection. This way the NVE experience can just continue even if servers or peers fail.

ReplicaNet focusses on usability for developers and thus sacrifices some configurability. ReplicaNet also does not provide any advanced high-level possibilities like Level of Detail or extended networking architectures. Because ReplicaNet is closed source and the documentation is scarce, we cannot be sure about all the details, but the overall concepts described here show that it is an interesting option for smaller-scale applications and that it provides some interesting options like basic distance-based filtering and dead-reckoning. This makes it a more interesting and more usable framework for this type of application than RakNet, especially from a bandwidth point-of-view.

5.1.3 Quazal Eterna

Quazal Eterna is a commercial OO C++ networking middleware framework aimed at supporting MMOG's. In many ways Eterna is similar to ReplicaNet. Eterna also uses an object-duplication system that can automatically propagate changes to variables on networked ob-

jects to other nodes in the network. This way, Eterna aims to make networking a high-level function which is easy for developers to use without having to worry about the low-level networking details. Eterna also uses an extra scripting language, called the Data Definition Language (DDL) to easily specify which attributes should be synchronized and in which way. This is similar to the ROL used by ReplicaNet.
It is important to note that while ReplicaNet is not developed with large-scale NVEs in mind, Eterna is and this means there are some differences in their implementations.

The first big difference is the central notion of so-called duplication spaces in Eterna. A duplication space is "a space where a duplicated object can discover or can be discovered by another duplicated object. This allows the developer to control where and how many duplicas of an object are published and when they are deleted". These duplication spaces are thus a very general concept that can be used to represent many of the filtering approaches we discussed in chapter 4. Simply speaking, every object has a masternode in the network. This master decides when the object is to be duplicated and thus seen by other users. The master uses the definition of the duplication space to make this decision. Eterna does not provide the implementations for the duplication strategies, so developers can fill these in themselves. An example would to create an AOI-like system where the duplication space is centered around a user's avatar and the master decides which objects the user should see depending on the AOI size and shape.
Eterna is also optimized for dividing the masters over a large number of different nodes or servers. The duplication spaces are managed in different so-called cells, which can be seen as a type of zoning, where each cell is managed by a different server. Cells can hold duplicates of objects in other cells, making zone-transitions and cross-zone events easier. Eterna provides automatic routing, even using simulated multicast schemes between servers, to distribute the update messages across the servers to the duplicates that need them.
Eterna also uses a load-balancing mechanism, similar to that of ReplicaNet, that makes it possible to change the master of an object. Note that this is different from a load-balancing approach like the one ALVIC-NG [85] uses. There, zones are split up and the smaller zones are managed by new servers. In the Eterna setup, individial objects can be moved to different servers without the need to actually split up zones. This requires a more robust routing system but also reduces the anomalies that can arise when moving the management of a zone to a different server.

The second big difference with ReplicaNet is that Dead Reckoning and state extrapolation is a very important part of the way Eterna works, much more important than it is for ReplicaNet. This is their main approach to keep the NVEs bandwidth usage scalable. Next to very extensive configuration options for the DR error thresholds, they also provide ample implementations for convergence and error correction so the consistency is kept as high as

possible. From the provided documentation it is not clear however if these DR configurations are only possible at compile time through the DDL-scripts, or if they are also changeable at runtime, i.e. if they are usable for dynamic bandwidth scalability. Next to this they also provide optional generic compression with a variety of different algorithms and also aggregation possibilities.

Whilst Eterna is similar to ReplicaNet in setup, it is clearly more aimed at large-scale development. The options for multiple-server support are much larger and the concept of duplication spaces and cells make it easier for developers to implement their own algorithms, but they still have to implement them themselves. The strong focus on Dead-Reckoning and the different variable propagation options provide enough configuration to have a relatively high control over the bandwidth usage of the NVE whilst not forgetting the consistency requirements.

Quazal Net-Z

Sadly, Eterna is no longer available or supported by Quazal. This is not because the system did not work but simply because Quazal decided to focus their efforts on the 2-32 player market. For this, they developed the commercial framework Net-Z, which is in essence a subset of Eterna and which has been used in several other large game engines. Net-Z runs on the same object-duplication engine as Eterna and is also focussed around DR and extrapolation. The big changes from Eterna are that there are no longer duplication spaces or cells (as the systems are supposed to run on a single server or via a P2P setup) and no load-balancing or object migration. This clearly shows that these techniques are especially suited and needed for large-scale worlds and not for simpler, smaller-scale applications. The use of DR as an important base concept is still interesting for limiting bandwidth but it is now more important for retaining consistency in fast-paced games like FPS.
Another change is that Net-Z supports a fully deterministic game engine for the development of RTS-like games. This fact shows two important things. First, that an object replication strategy can be used for the development of such a game. Second, that a deterministic engine is less likely to be suited for large-scale worlds as it was not present in the Eterna middleware but was added in the Net-Z middleware to support small 2-8 player matches.

5.1.4 XNA and XBOX Live

XNA [54] is a full game development suite offered by Microsoft which is free to use. It allows for the creation of complete games for Microsoft platforms like XBOX and Windows. The networking part of XNA is specifically focussed on match making between players of equal skill levels, lobbying, voice chat and other player management, which is a large part of the provided services of the XBOX Live network. This network is used by every multiplayer game on the XBOX360 game console and some games on other platforms as well.

This XBOX Live network has a lot in common with similar systems like Steam [4], Battle.Net [5] and Playstation Network [6].

XNA and XBOX Live are interesting to mention in this chapter because they actually provide very little networking possibilities out-of-the-box whilst being a very big and important platform for multiplayer gaming. This means that developers have to do large amounts of work for even simple small-scale multiplayer games. There are only some very basic interfaces for the usage of sockets and bit arrays. A basic reliable UDP-protocol is also available, as well as standard bindings for simple client-server or P2P setups. But beyond that, there is nothing to help developers. No protocol optimization, no filtering and even no simple replication manager like RakNet provides.
Another peculiar fact is that all traffic from XBOX360 games has to use the underlying XBOX Live network, a Microsoft-controlled network setup that connects all XBOX consoles. This means that, in general, dedicated server machines are not allowed and thus network architectures are limited to player-controlled servers or P2P networks. Furthermore, games that are submitted to Microsoft for approval are checked to see if they don't use too much bandwidth. The recommended bandwidth usage is generally very low, about 9 kbps down and 13kbps up for a single user that is not a server, if the built-in voice chat is used in a game with 16 players [55]. This ensures that most server consoles will be able to handle the traffic and that the LIVE-network doesn't get overloaded by greedy games and applications.

What XNA lacks in network implementations, it makes up for in network tools, documentation and debug configurability. It has built-in latency and packet-loss simulators as well as a simple loopback possibility so one can test on a single pc or XBOX360. They also provide the NetGrove tool suite which can capture and analyze network traffic in many different ways. Finally, there are a lot of tutorials and documentation available online that show how to do the more advanced implementations, even without actually providing them in the framework.

In conclusion, even though XNA and XBOX Live are very large and important frameworks, they provide very little possibilities for actual multiplayer game development. It is interesting however that such a limited networking implementation is supported by such a wide array of optimized tools and documentation to help developers optimize their bandwidth usage and network protocols. But even with these documentations it is clear that this platform is not directly suited for large-scale development.

[4] http://www.steampowered.com, 29/08/2010
[5] http://eu.battle.net/en/, 29/08/2010
[6] http://be.playstation.com/psn/, 29/08/2010

5.1.5 Unity

Unity [50] is a complete, free to use game building platform which provides tools for just about every step of the development process, from graphical design to networking. Unity is not aimed at developing large-scale environments but it is interesting to discuss here nonetheless because it markets itself as supporting a number of different platforms, including contemporary mobile phone systems like iPhone and Android and it also offers a web client to be used in any web browser. As we have discussed before, mobile networks are likely to have limited bandwidth and also high latency. Because of this, it is interesting to see how an engine like Unity provides options to deal with these problems.

At its core, Unity uses RakNet which we discussed previously. However, this RakNet functionality is shielded from the user. Instead, Unity provides so called NetworkViews which can be coupled to different components and objects and act as the networking interface for that object or component. Unity uses two mechanisms for network communication: RPC and state synchronization [7] and both are provided by the NetworkViews. The latter is their version of attribute propagation, a method we discussed previously. The interesting thing here is that Unity lets the developers choose from two different options to perform this state synchronization: reliable delta compressed or unreliable. When you choose unreliable, all variables are sent over the network with full precision at set time intervals, whether their values have changed or not. The other option will do about the opposite: all messages are sent reliably and thus the variables are delta compressed, based on their previous values. Variables are also only sent when they have changed and only those variables that have changed are sent (Unity uses a single bit and a fixed variable order to indicate if a variable is present in the network update).
Whilst the second option is certainly interesting for bandwidth usage as we have discussed, Unity takes the all or nothing approach here. There is no in-between setting or the possible use of reliable reference states for the delta updates.

When we look at other possible techniques for bandwidth scalability, we see that Unity offers very little. There is no basic DR, no AOI, no advanced network architectures and no spatial subdivision like zoning. One could say that this is normal for an engine that aims to create games for 2-32 players, but we should not forget this engine is supposed to run multiplayer games across mobile networks on smartphones. As we have discussed previously, these networks often have low bandwidth, too low even for normal multiplayer games.

When we compare Unity to other middleware with similar networking goals like Quazal Net-Z, we see that it has far less extensive options. The standard protocol is an all-or-nothing choice

[7] http://unity3d.com/support/documentation/Components/net-MinimizingBandwidth.html, 12/08/2010

between two options and there are no other techniques implemented for more advanced setups or games. This means that Unity might be able to run on a smartphone graphically but it will still require considerable work from the developer to create a working multiplayer game and a large-scale NVE is definitely not supported, even when we leave the mobile platforms out of the picture. This is remarkable given the large attention Unity has received in the recent past and the high number of developers that use the engine to create their games. Extra techniques for bandwidth limitation and more flexible protocol options could certainly help make Unity a more general-purpose game engine.

5.1.6 DoIT

The Distributed-organized Information Terra Platform (DoIT) [71] is an academic project whose primary goal is to create a practical middleware for MMOG's which addresses some big issues. In their paper about the system, Tsun-Yu Hsiao et al. describe these issues and explain some important characteristics a usable middleware should have.
Because they want to define a practical middleware system, they put their focus on subjects like ease of development, deployment, maintenance and change. The middleware should make it easy to program and change the networked gamecode. Whilst these concepts are important in the larger picture, they are not for the purposes of this thesis. Two other points in their system however are interesting.

First of all they state that any middleware for MMOG's that wants to be prepared for the future needs to provide the possibility of a seamless world and scalability solutions to make this possible. A generic n-tier architecture with load-balancing is their most likely solution to this problem. This means that a setup like that of ALVIC-NG [85] with proxies and dynamic re-allocation of zones to servers is a good base for a usable MMOG middleware. In their research they found that the proxies can be a big performance bottleneck if care is not taken to prevent this. Their test setups produced the best results with 15 proxies and only 3 world servers to accommodate 10.000 players. Where ALVIC-NG states that the proxies should not perform too much processing on their own, DoIT uses the proxies to do cheat detection and prevention, as security is also a big issue for MMOG's according to the authors.

The second point they make is that a message-oriented communication structure is the best choice for an MMOG and this in contrast to a RPC approach. Message are a better match for the event-driven games, can process multiple events at the same time (in contrast to a blocking RPC call) and handler implementations can change independently on client and server, whereas with RPC a complete recompile of the complete system would be required if an interface changes.
They use a simple code-generation tool to make it possible for developers to easily describe

their content protocols. XML descriptions of the protocol are compiled into concrete protocol definitions. They also support the dynamic re-ordering of fields in the protocols. These updates in the protocol are done automatically after a certain period and help protect the system against cheating and message faking.
This code-generation system can also generate the callbacks for every kind of message that needs to be sent, making it simple for the developers to react to an incoming network packet. These callback implementations can also be swapped at runtime. This is not just interesting for quickly fixing bugs but also to change the behaviour of the callback. In our case this could for instance mean a different callback implementation can be loaded when the available bandwidth changes dramatically and the new implementation uses more or less bandwidth.

A discussion of this paper might seem out of place in this chapter and indeed it does focus more on the practical aspects of NVE programming and deployment than on algorithms for load-balancing or bandwidth scalability. This shows clearly that there is a lot more to developing an NVE than the issues discussed in this thesis and gives us good guidelines for the next chapters and the implementation of the bandwidth scalability techniques. It shows that message-passing is a good method for a NVE, that we should be careful not to overload the proxies, that code-generation tools should be used and that they should support the implemented techniques and finally, that the ability to change the NVE behaviour at runtime is a powerful feature.

5.1.7 NetDog

NetDog [32] is an OO C++ networking engine built with large-scale NVEs in mind. It aims to be very performant in terms of processing required so a single server can support a large number of concurrent users. NetDog provides all the basic provisions of a game networking engine like socket wrappers, reliable UDP possibilities, time synchronization of events, event callbacks and object synchronization.

NetDog is special in that it uses a very flexible network architecture where any node can act as a server. This means that just about any network architecture is possible, from simple P2P to multi-tier client-server setups. There is no clear subdivision of server roles so developers are not bound to using a proxy-model like discussed previously. Instead, NetDog works with so-called channels which determine to which server content is being sent. This way, some content can be sent only to certain servers (for instance proxy-like servers) and other content directly to other servers (for instance the logic servers that manage the world). This allows for very specific and dynamic networking architectures while still making the standard setups possible. Because of this flexible setup however, NetDog does not provide a simple mapping of servers to for example zones, shards or instances. Neither does it provide a good load-

balancing algorithm to assign new servers to overloaded areas. Developers that want to use NetDog in combination with a seamless world divided into zones will have to program this themselves, but the NetDog framework certainly allows it.

Next to the special architecture approach and focus on processing performance, NetDog also claims to be very efficient in terms of bandwidth usage. As can be read on their site, even fast-action, FPS-style games are possible. They make this statement because they feel that using UDP instead of TCP already warrants a high per-client bandwidth. As we have discussed before, it is doubtful that this fact alone will make for a possible MMOFPS. Nevertheless, in their documents we can find evidence that they understand the bandwidth problem and that they are willing to provide solutions to it. The newest version of NetDog supports automatic aggregation of packets (though they don't specify if it is quorum or timeout-based) and they say they plan to include Dead Reckoning and Culling (simple distance-based AOI filtering) in future releases.

So while NetDog is a networking engine with support for NVEs, it certainly lacks a large number of implementations to make it usable out-of-the-box. The flexible architecture is definitely an interesting tool to easily try some extra optimizations and setups but the lack of support for load-balancing and spatial subdivision algorithms in this architecture is a big drawback. For the purposes of this thesis they indicate a number of bandwidth scalability approaches but they have actually implemented only a few of these methods. All in all NetDog is an interesting concept to start from but it does not offer a lot of special features for NVE developers.

5.1.8 Ryzom engine

The Ryzom engine [45] is a full-fledged, open-source engine aimed at MMOG development. It was first developed for a commercial game called Ryzom and made open-source some time thereafter. The makers also provide a detailed explanation of their choice for algorithms and techniques in the engine [61], which makes the engine very interesting to discuss here.
Ryzom uses a service-based server architecture where different services regulate different aspects like collision checks, gameplay rules enforcing, player login, etc. These services are run on a number of different servers. Separate front-end servers act as the interface for the services for the client processes, much like the proxies in other systems we have discussed, and route the messages to the correct services.

For the purposes of this thesis, the Ryzom engine is very interesting because it was made to work on very low-bandwidth dial-up connections, with a maximum bandwidth usage of only 13 KB/s in mind. They also put great emphasis on lag compensation and reducing CPU

usage.
The first interesting choice they made was not to use Dead Reckoning in favor of an interpolation strategy. This was because they felt DR was not fit for rapid player movements and the updates would still be frequent, reducing the bandwidth savings.
The second interesting choice is the use of a prioritization system based on entity-distance from the player. Firstly they set a maximum to the number of entities (x) a player can receive updates for at one time (for example 64 or 255) and find the x entities closest to the player. Then the entities that are closest are assigned higher priorities then entities farther away. These priority values determine the update rate with which updates for those entities will be sent to the player. Closer entities will have a much higher update rate than far away entities, effectively creating a LOD system. This is kept CPU scalable by splitting up the computations to multiple game cycles and threads, making it possible for a single front-end server to serve up to 1000 users.
The third aspect which requires attention is the usage of smart logical compression algorithms to reduce the packet sizes. Positions of entities are differentially compressed on the viewer's position to reduce their sizes using a smart algorithm, changing the size from 32 to 10 bit, without sacrificing a lot of accuracy. Next to this, any string communication uses a unique integer identifier for the strings so they only have to be sent once.
A final optimization is that the entire engine's update rate for sending updates is managed by a central service that controls these update rates for all servers. When one server is under heavy load and threatens to cause network congestion by sending too much data, the central service can decide to lower the update rates for all services, this way preserving the bandwidth limit and preventing congestion. This will cause extra lag and lower the consistency but it can prevent some more serious issues that can result from using too much bandwidth.

The Ryzom engine is an excellent example of an advanced system for NVE development which deals with both load-balancing and bandwidth scalability. The exhaustive description of their algorithms is a good source of inspiration for any NVE developer and shows that using even only some of the algorithms discussed previously in this text, bandwidth usage can be kept at very low levels while still providing a good consistency to the users. It also shows a successful NVE can be developed which uses only a very small amount of bandwidth, making it possible for mobile users or other users on slow networks to connect to the NVE.

5.1.9 BigWorld Technology

BigWorld Technologies [4] provides a complete commercial MMOG development solution. All tools are included in a single development pipeline, from the graphical engine to advanced networking and many monitoring applications. BigWorld is quoted as being the most complete solution available today [71]. For this thesis, BigWorld is interesting to discuss because

it provides one of the most extensive and complete networking solutions for large-scale NVEs.

The BigWorld network architecture is aimed at flexibility and has an important load-balancing component. The architecture can handle any setup, from seamless worlds to multiple shards, zoning and instances. All of this is dynamically load-balanced across the different available servers when the need arises. This is all hidden from the clients by using the same 4-tier architecture with proxies we discussed multiple times before. BigWorld goes even further, allowing multiple shards and even multiple completely different NVEs to run on the same server cluster. This is a very far driven form of load-balancing and it is even more similar to modern generic virtualisation approaches taken by cloud computing setups then other previously discussed systems. Because BigWorld is a commercial product, it is sadly not clear how the load-balancing algorithms work, but it is clear that they see that this is big problem in practical NVE deployment and that they can solve it efficiently and economically.

Next to the performance load balancing, BigWorld also provides bandwidth management techniques. They have possibilities for variable transmission rates per player connection but their biggest power is an adaptive Level of Detail and data prioritization approach that is used on all sent updates. Even though they do not provide a large amount of details for these systems, they claim they use high-end algorithms that are very optimized through years of use. The fact that they put forward LOD as their main bandwidth saving technology is a strong indication that this is an interesting technology for implementation in this thesis and in any NVE that seeks to reduce it's bandwidth usage.
Like Unity, BigWorld also advertises it's availability on the iPhone, but they are more realistic in this respect. They provide API's to create a more simple mini-game on the mobile platforms, not a semi-full fledged client like Unity. This reduces the possibilities but also offers a more reachable goal of integrating mobile platforms with an NVE.

BigWorld is a game engine that aims to be a solution for large-scale MMOG development and it succeeds in this aim where the network is concerned. Not only does it use a very flexible server architecture with advanced load-balancing options, in line with this thesis it also provides advanced traffic filtering options in the form of Level of Detail to reduce the bandwidth usage of the NVE.

5.1.10 HeroEngine

HeroEngine [22] is another commercial engine that is specifically focussed on the development or MMOG's. It is very similar in concept to BigWorld, offering a complete solution for every aspect of the development and deployment process. It generally provides the same high-level networking options as BigWorld, but with some small differences.

First of all HeroEngine also support almost all possible world setups, from sharding over zoning to instanced parts. They also have their own load-balancing system to re-assign servers to areas with high load. The main difference with BigWorld (besides probably the exact algorithms used to actually perform the load-balancing) is that HeroEngine does not support the running of multiple shards or different NVEs on the same server cluster. This extra high-level load balancing which BigWorld offers can be very interesting to optimize the use of the cluster's hardware, especially when a single shard or application is underloaded and can lend processing power to the applications that need more resources.

The other difference with BigWorld is that HeroEngine does not explicitly mentions a LOD scheme or controllable transmission rates. Instead, it offers a spatial awareness system to regulate which objects are aware of which other objects and this awareness can be adjusted by a dynamic awareness range. It is clear that this incurs some form of a dynamic AOI system. Next to this, they also use a Character Position Interpolator to "reduce or eliminate overhead of distant characters" for which they also scale down or eliminate animation, controller updates and collision checks. This can mean that they use some form of LOD based on distance, where this LOD is accomplished by using a combination of DR, logical compression and ignoring collisions to speed up calculations. From these two descriptions it seems they use two separate systems, one distance based for determining which entities should be sent and another one which is also distance based to determine the level of detail for the entities. In the next chapter about the implementation of this thesis, we will see that these two systems can be combined into one large AOI-based filtering setup.

While HeroEngine offers the same large concepts for MMOG development as BigWorld, the actual implementations of the algorithms used to accomplish these concepts can still differ largely between engines. This shows once more that NVE development and scalability is not a simple issue and that there is not one straightforward solution that solves all problems. It also shows that two of the most important MMOG engines employ techniques for both performance and bandwidth scalability, empowering the statement of this thesis that the bandwidth usage is also an important parameter of effective NVE design, next to performance scalability.

5.1.11 Other systems

In this chapter we have discussed only a few of many engines and networking middleware available. In this section we will look at some other engines and middleware shortly and quickly touch their main points, to show that there are even more possibilities.

The first NVEs

First we should mention the first important NVEs that were developed: DIS, NPSNET,, PARADISE , DIVE and MASSIVE. Many of the technologies and techniques we have discussed in the previous chapters were first introduced and used in these academic NVE systems. DIS (Distributed Interactive Simulation) was a descendent of SIMNET, a system developed by the military for enabling networked war simulations. As we have discussed before in chapter 3, DIS made heavy use of dead reckoning to predict the motion of the vehicles in these simulations. However, we have also discussed that DIS was very bandwidth unfriendly, sending a lot of duplicate information in it's PDU's.
PARADISE introduced the practical usage of AOI filters through specific AOI servers in an otherwise entity-based multicast system. These AOI servers monitored the position of entities and notified other entities when they should subscribe to a multicast group to receive information about new entities in their AOI. PARADISE also supports multiple information flows per object so others can choose which level of accuracy they want about the object.
DIVE (Distributed Interactive Virtual Environment) was aimed at collaboration and interaction. It used a dynamic distributed database with reliable multicast algorithms, causing it to use a large amount of bandwidth and so it was only scalable up to 32 players. The interesting thing about DIVE (and later MASSIVE) is that they implemented the aura-nimubs model for AOI management. MASSIVE also used extensive options for aggregation of information into higher-level streams.
These systems were among the pioneers of NVE development and introduced a number of important techniques that are still used today. They also all used multicast, which is one of the reasons they were not covered in more detail in this thesis, as our focus is on a real-life NVE for which multicast can not be used on the contemporary internet. The other reason is that they are all discussed to great lengths in other books and papers, so the user is referred there for extra information [86].

Quick looks

Here we will have a quick look at some other real-life contemporary engines and their characteristics.
The Unreal Engine is another very well-known commercial engine for game development. The developers have recently introduced Atlas [51], a MMOG system for use in conjunction with the Unreal engine. Atlas uses a 4-tier server setup, divided into multiple clusters with load-balancing and a master server per cluster that keeps track of the positions of all entities in the world. Their site does not give any information on their use of bandwidth limiting techniques.

Icarus [25] is another commercial engine that is aimed at MMOG development in particular. The interesting thing here is that they provide a cloud-based solution for developers who use

their system to be able to develop and test the games on the cloud servers of Icarus studios. This way, the Icarus engine is not only designed to run on multiple servers when deployed, during development the entire pipeline can exist on the cloud-systems, taking virtual computing and NVE development to a whole new level.

ICE (Internet Communications Engine) [26] is a commercial networking middleware and on their site they make an interesting statement: "The design philosophy for the Ice encoding is to optimize performance at the expense of bandwidth, that is, to use as few CPU cycles as possible during marshaling and unmarshalling". This to show that the tradeoff between CPU usage and extensive bandwidth optimization is an important aspect to keep into account when choosing a bandwidth scalability technique.

Finally we mention the commercial service of OnLive [34] again (see chapter 3). OnLive replaces all server-client traffic with a video stream of the world, calculated on the server. The client only sends his input state to the server, where the server gives this input to the game, renders the new images, and sends the video stream to the client. It may seem weird, but in some cases this video stream actually uses less bandwidth then when we would send the updates ourselves. This can certainly be the case in environments with a lot of user-created content that would otherwise have to be sent to all clients.
The video streams can also be downscaled to a lower resolution or frame rate to dynamically limit the bandwidth usage, making mobile gaming more possible as the small phone-screens can only handle a limited resolution natively anyway. This is an approach also taken in one of the test setups for the NIProxy, which we will discuss in more detail in the next chapter.
Finally, this approach eliminates the need for the developers to write complicated networking code for single-server games at the expense of needing more and more powerful servers to run the game and render the different video streams at the same time, whilst most modern server system do absolutely no rendering on the servers themselves. For more complex systems, like NVEs, networking code is still needed for communication between the different servers.
Video streaming systems are certainly an interesting option for making network communication for graphical applications like games easier. This is all done in a simple, straightforward way with only a few directly adjustable parameters such as resolution and bitrate, that have direct influence on the bandwidth usage, consistency and arguably QoE of the game for the player.
There is much debate about the lag OnLive will introduce between the user's input and the changes in what he sees, because local lag compensation algorithms are no longer possible when using this system. The results of OnLive for these and other factors will help determine if this kind of technology will be used for NVE development in the future.

5.2 Games and Virtual Worlds

In the previous sections we have focussed on engines and middleware that enable developers to build their own MMOG or NVE by using those systems as the basis. Now it is time to discuss some real-life MMOGs and NVEs that have actually been deployed on the internet. The items discussed here do not use any of the aforementioned middleware or engines but rather choose to use their own proprietary technology. For the commercial systems, this means there are not a lot of details available, as is the case for the engines discussed previously.

5.2.1 Second Life, There and World of Warcraft

Second Life [46] and There [48] are two commercial NVEs that attempt to simulate the real world in their virtual environment. Both of them focus on the social aspect of an NVE and allow players to create new content for the world themselves, like objects and avatar clothes. World of Warcraft [53] on the other hand is the most successful MMORPG [30] and is set in a fantasy world. Unlike Second Life and There, WoW does not allow its users to create new content for the game themselves. WoW also uses shards, where Second Life and There are large seamless worlds.
The network architectures behind Second Life and WoW were discussed in chapter 4. For There, there is not much known about the underlying architecture. There uses a large number of connections to content servers (and audio streaming services for voice chat). Most traffic comes from a single server, which is most likely streaming the environment to the user, whilst a number of smaller streams from other servers probably send information about dynamic objects in the current zone (as the servers this traffic comes from depend on the part of the world the player is in) [77].

What is most interesting for this thesis is the difference between a completely dynamic, user-generated world and a world with fixed content. The latter option will remove the need to send all but a small amount of world information, automatically reducing the potential bandwidth usage, especially when crossing zone boundaries. User-generated content on the other hand introduces an extra network stream with (very) high bandwidth requirements. The management of this data becomes as important as the management of user movement and event data and is crucial to bandwidth scalability for these systems.
Both Second Life and There use a continuous streaming system to deliver the world content to the user. This way the world can be completely dynamic as all data is sent to the client program only when it is needed. As a result, the client program is little more than a simple viewer/renderer for Second Life and only contains basic geometry and texture information in There. All the world data is cached locally on the user's computer to prevent unnecessary retransmits if the data does not change.
This streaming approach means that there is a constant (potentially high) usage of bandwidth

when connected to these worlds and that visual artifacts can occur as it is possible that not all data for a certain part has been sent when the user moves to this area. The user will then have to wait for all data to arrive to be able to fully interact with and immerse himself in the world.

To make this streaming approach possible in terms of bandwidth usage, advanced techniques are used, most notably compression and LOD [77]. For instance, There uses three LODs for all geometry in the world and Second Life incrementally improves texture quality by first sending a very coarse version to quickly provide the user with an initial view, after which extra information is sent incrementally so that the quality of the texture becomes better over time [77]. Second Life also uses a form of AOI where large and close-by objects are streamed first to make sure that the user has that information as quickly as possible.
To give a full overview and discussion of these techniques is not necessary here as this kind of user-generated content is not present in every NVE and because most of the common techniques for reducing the bandwidth used by streaming the world data have been discussed in this text. The exact implementation of the (incremental) compression and LOD management of geometry and textures is more a subject for computer graphics experts, as they are also interesting for graphical rendering systems.

But even with these optimizations, Second Life is only available for users with a broadband connection and There also uses a considerable amount of bandwidth, which makes it difficult to access these systems on slower network links. However, we can say that this behaviour is expected. Seeing the amount of data that has to be sent over the network, it is already impressive that these NVEs are possible without extremely high-bandwidth connections. This is in contrast with World of Warcraft, which is playable using a low-bandwidth connection, due to pre-made content and because content patches are distributed separately as big downloads, not streamed to the client while playing.
A possible conclusion from this section is that, to achieve a low bandwidth usage, an NVE should use as little user-generated content as possible. When user-generated content is required for the NVE concept, advanced techniques for compression, streaming and LOD management are needed to keep the bandwidth usage feasible on contemporary internet connections, but even with such techniques in place the NVE will use a large amount of bandwidth.

5.2.2 MMOFPS and MMORTS examples

As we have discussed in chapters 1 and 2, one of the goals of this thesis is to see if large-scale MMOFPS and MMORTS games are really possible. As discussed, these games could possibly need huge amounts of bandwidths if there are a large number of people in the same place in the virtual world; they consequently require good bandwidth scalability implementations.

MMOFPS

Lately there have indeed been some games that try to bring the FPS genre to an MMO setting. One of the first real MMOFPSs is PlanetSide [37], a futuristic shooter set on a distant planet. PlanetSide offers large battles with foot soldiers and many different vehicles. Sadly, no real numbers are available on how many players could be in a single battle at the same time and there is no information about the used network architecture or bandwidth regulating techniques. It is known that PlanetSide had severe problems with lag however, often causing player disconnects by timeouts [38].

Recently, Massive Action Game (MAG) [29] impressed gamers by allowing up to 256 players to compete in an FPS game on the PlayStation 3 console. While MAG is not a persistent, seamless world MMOFPS per se, it does offer large scale combat with a large amount of players. Once again, little is known about the real inner workings of MAG, just that they use a "new networking architecture" [18] and that lag is not a very big problem. It is also true that there will rarely be more than 64 players in one area at the same time, so the game setup allows for techniques like zoning to be used.
Another new MMOFPS is All Points Bulletin (APB) [2] which is set in a contemporary city where policemen fight against criminals. APB has a strict subdivision of the game environment in the form of different districts. Action districts allow for up to 100 players in the same instance and this is where the FPS-style gameplay occurs. Social districts allow for up to 250 players and are designed for player interaction and feature a lot less action-packed gameplay. Finally, one world with several different districts in APB can contain up to 100.000 players and there are several worlds available.
Other MMOFPSs exist or have been announced. Global Agenda [20] couples instance based matches at key locations with a persistent world system. Huxley [24] will feature 120 player battles in separate instances. Lastly, Fallen Earth [16] offers specific Player-versus-Player zones in a larger world where FPS-style combat can take place between players.

The games marketed as MMOFPSs are rarely built as a large seamless world but instead make use of instancing and separate zones (sometimes with a maximum user count) to handle scalability issues. But even in these restricted environments, bandwidth scalability can be important when all players would join a small area in the zone/instance and bandwidth limitations might very well be one of the reasons these restrictions are used in the first place.

We should make a sidenote here on the usability of a 4-tier networking architecture with separate proxy or front-end servers and internal game servers for MMOFPS games. This is because FPS games are very sensitive to lag [84]. Using an extra server in between the player and the actual game server that performs the action calculations, can possibly increase this

lag; the data namely has to first be processed by the intermediate server, is then sent over another link to the game server for processing, after which the inverse process takes place. This extra stop in the network is not present in the smaller-scale FPS setups and even there lag problems are sometimes quite large.
On the other hand, we can also say that the 4-tier setup can help reduce lag. If the proxy servers no longer just serve as neutral intermediaries and connection points for players, but instead actually take gameplay decisions without waiting for answers from the internal game servers, the results of the player actions can be sent to other players much more quickly. Game servers can then still be responsible for the more important decisions in the game, but things like player movement could be handled by the proxy servers directly [56]. When users are connected to a proxy server that is deployed close to their geographic location, the lag will be low. Note however that this will also increase the processing requirements on the proxy servers, possibly decreasing the number of players they can handle at the same time. This is a problem we will also encounter during our implementation of bandwidth scalability techniques.

MMORTS

There are only a few true MMORTSs which are set in a large continuous world or that support a large number of players that compete with each other directly at the same time. Most MMORTSs do provide a persistent system where your achievements and items are being tracked, but the real matches are mostly limited to 10 players or less, which comes down to a small-scale RTS game which can be created by a synchronized simulation as discussed in chapter 2.
If we would define MMORTS as a game set in a large world where a large number of players can meet each other in battle at any moment, we will not find many true MMORTSs. It is true that the same can be said for the MMOFPS games discussed previously, but they at least offered larger amounts of players in a single match than do their normal multiplayer counterparts, whereas a common MMORTS does not. Whether this is because of technical problems or gameplay design choices is however unclear.

Recently, End of Nations [10] announced that they will have a possibility for up to 51 players per instance, with the possibility for players to join a game that is already in progress (which, as we have discussed, is difficult to achieve in a synchronized simulation). Because of this, End of Nations is close to a large-scale MMORTS. Interestingly enough, in the FAQ of the developer's site, we can find that a high speed broadband connection is needed to be able to play the game. Because a traditional synchronized simulation does not use that much traffic, even with 50 players, and given that there will be no support for user-generated content, we may assume that End of Nations will not use a synchronized simulation but a per-object replication system in combination with bandwidth scalability techniques to make this amount

of players possible.

5.2.3 Browser-based games

Finally, we will look at NVEs where there is a large, seamless world and a high number of users, but where the interactions are not real-time or time critical. This is the case for many browser-based games where users put their actions in so-called queues and those actions are executed automatically by the system after a set time. Note that this is different from a browser-based client to for example an FPS, like Quake Live [39]
A good example of this kind of game is Travian [49]. Every player has a village in a larger world and can erect buildings and train units in his village. Players can send units to other players' villages to attack them, but this attack is not done in real time and neither is it graphically visible to the player (besides from a small textual message in the graphical user interface). It takes some time for the units to travel to the to-be-attacked village (typically tens of minutes to several hours) and the battles are executed automatically after this time has passed.
This kind of NVE is quite interesting because it offers a persistent world and interesting gameplay, so it also attracts a large number of users. Another example would be FarmVille [17], a game where players manage a virtual farm on the social networking site Facebook. Here, players can share virtual items and animals and can help each other out if they have neighboring farms. A last example is the very popular free MMORPG RuneScape [44]. RuneScape uses a java-based client in the player's browser and even does 3D rendering of the virtual world. Players are confined to a checkerboard pattern for movement, which is probably one of the reasons the game is scalable enough for access from a webbrowser.

From a technical point of view however, these NVEs are quite different from the ones we have discussed before. There is a very small amount of network communication needed, all of which is done through the use of HTTP, the standard protocol for the transmission web pages. The lack of most graphical feedback means that no separate unit updates have to be sent. As far as server architecture goes, the different actions of users are kept in a database on what is mostly a typical webserver and so-called cronjobs run at set times to process all these actions and generate the results for the users.
This means that this type of NVE is very scalable, both in terms of processing power and certainly bandwidth usage and it shows once more that the characteristics of the NVE can have a huge impact on the techniques that are needed for ensured scalability.

5.3 Choosing a technique for implementation

In this section, we will create a summary of all the conclusions that have been drawn in the previous chapters and try to indicate the points that are important and that can influence

the choice of techniques for bandwidth scalability of an NVE. These conclusions will guide the implementation part of this thesis, where the goal is to implement a few bandwidth scalability techniques for a NVE middleware engine. The initial ideas and the outline for the implementation part will be treated in chapter 6.

5.3.1 Which techniques to use?

The techniques one can and should use are very dependent of the type of NVE we are trying to build and other decisions we make with regard to gameplay elements, networking architecture, user heterogeneity and practicality of implementation. There is no single technique which performs best in all situations, all techniques have their benefits and drawbacks. The real challenge is to carefully trade-off different aspects of these techniques to best suit the specific purpose of the system you are trying to build.
The conclusions, advantages and disadvantages for the different techniques have already been given in their own sections, so in the current section we will focus on how to choose the most suitable alternatives and which parameters can affect our decision.

Dependent on NVE type

As we have discussed many times before in this text, every NVE is different and has different needs. Second Life and There need to stream their world dynamically to the user, requiring good compression and LOD management. MMOFPS games are very fast-paced and should have as little latency on player movement as possible. MMORTS games and physics simulations will need to synchronize a huge number of objects, which will be even larger than the number of players in the game. Collaborative environments will benefit from audio and video communication.

The type of NVE will also help determine some of the more logical optimizations we can use. A good example is the partitioning into separate solar systems in EVE online, as discussed in chapter 4. This is possible because the concept of the game allows it, while in a world like Second Life such artificial partitioning would be illogical in a world that is meant to look like our own. Another example is that Dead Reckoning methods will only really shine in an NVE with a lot of vehicles or entities whose movement only alters slowly. Battling a boss with a small group of people in an instanced dungeon is the right fit for a game like World of Warcraft, but for a game like Global Agenda we want the outcome of the battle to be persistent and since the winning group gains control over the area, multiple instances of the same area are not possible.

Table 5.1 gives some suggestions which one could possibly follow when trying to determine the optimal techniques for a specific type of NVE. This table is in no way exhaustive or perfect,

but gives a good indication of how different techniques can benefit different NVE types in a significant way.

NVE Type	Recommended techniques	Less suited techniques
MMOFPS	- Aggregation (many small packets)	- Dead Reckoning (non-deterministic entities)
MMORTS	- Dead Reckoning (predictable movements)	- Aggregation (lower packet frequency)
MMORPG	- Instancing (gameplay encouraged)	- Aggregation (lower packet frequency)
Collaborative	- Compression (reduce userdata size) - Area-of-Interest (fine-grained interest management)	- Sharding (users want to meet)
Physics	- Dead Reckoning (predictable movements)	- Area-of-Interest (many small objects)
Browser based	- Sharding (simple scalability)	- Most other methods (not needed)

Table 5.1: Recommended techniques for different kinds of NVEs

Dependent on network architecture

The choice of the underlying network architecture also limits or enables some techniques for bandwidth scalability. As the network architecture is mostly chosen for reasons other than bandwidth scalability (for example processing scalability, manageability, ease of use, etc.), sometimes this choice is not optimal for the bandwidth requirements of that system and trade-offs will have to be made.
A good example is that with region-based multicast, it is difficult to create an effective and dynamic LOD scheme, whereas this is much simpler with a central entity like a server. On the other hand, when a server-based architecture has to perform a lot of calculations to ensure bandwidth scalability, there will be more problems with processing power requirements and load-balancing will be needed, whereas in a P2P system every connected client automatically provides new processing power to the system.

Dependent on the user

One of the points made in the introduction and in the second chapter is that there can be a lot of heterogeneity between users of an NVE. This is especially reflected in the network connection that they use to connect to the NVE. Fast, high-bandwidth connections with low packet loss can appear side-by-side with slow, mobile networks with large amounts of packet loss and possibly even dropping connections. What is more, bandwidth is not constant and can fluctuate considerably, even for good wired network connections.
This means that we have to choose bandwidth scaling techniques that can deal with this heterogeneity and that can possibly even adapt their behaviour dynamically at runtime to adjust for bandwidth fluctuations. A technique for such an NVE should for instance be able to lower bandwidth considerably for mobile connections and by only forwarding the most necessary data, while it should at the same time be able to offer a lot of information to users with better connections.
This implies that techniques like Dead Reckoning and AOI management are more interesting and needed when supporting low bandwidth connections than for instance compression and aggregation, as the latter are much less dynamically adjustable and generally do not offer the possibility to improve the information quality by changing their parameters.

Dependent on practicality

Finally, the choice of which techniques to use or implement can also be practically motivated. Many of the discussed techniques are quite advanced and need an extensive implementation to work properly with the intended results. A good example is zoning for a seamless world. Here care has to be taken to provide possibilities for zone crossing, cross-zone events, zone splitting and merging etc. Another example can be found with Dead Reckoning, where developers often just ignore any consistency problem that has occurred because they are tedious and difficult to resolve in a correct way, as we have discussed in chapter 4.
We have also seen that there are a lot of techniques that require some kind of reliable algorithm to function optimally. This can be a problem in multicast-based networks where reliable delivery is not trivial. It can also be an issue on mobile networks where packet loss is large and retransmits will use more bandwidth and can slow down the NVE.

5.3.2 Choice for implementation

Now that we have discussed a large amount of different techniques for bandwidth scalability in an NVE, it is time to see how some of these techniques can be practically implemented in a real-life setup. The goal is to extend the ALVIC-NG platform [85], developed at the Hasselt University, Belgium, which has been introduced in chapter 4.
ALVIC-NG uses a 4-tier network architecture involving proxy servers. The most important aspect however is its dynamic load-balancing system which will split up and merge zones

when the individual internal zone servers are respectively over- and underloaded. While this network architecture exists and the load-balancing algorithms are operational, there is not yet an implementation for a bandwidth scalability technique. As the goal is to use this framework in real-life NVE deployments, it is important that the framework is extended with one or more techniques for bandwidth scalability.

Now we are posed with a problem however. As we have said in the previous section, it is impossible to provide a single solution that works optimally for all types of NVEs. As the goal of ALVIC-NG is to be a generic middleware for NVE development, this means we would have to implement a large amount of different techniques. But as we have seen from the discussion of real-life middleware with similar objectives, this is certainly not the approach taken by many other implementations. Most of those systems choose one or two main techniques and focus their implementations around those chosen techniques. In a way this is logical, as they also only support a single network architecture. There is no commercial middleware that supports the creation of server-based, P2P and multicast based architectures and likewise there is also no middleware that provides all aspects of compression, aggregation, zoning, AOI and dead reckoning. This focus on a limited number of techniques is also practically motivated, as it is already difficult to implement and thoroughly test one technique, let alone multiple. This means that some types of NVEs will run slightly better on some systems than other types, depending on which bandwidth scalability technique is present in the framework. We do see that nearly all commercial systems are based on a zoning subdivision of the world. We also notice that all those systems use a client-server based architecture. ALVIC-NG fits right in those contemporary systems, which already reduces our possible choices and helps us to focus on a 4-tier architecture with a zoning approach. This way, we do not have to worry about making the implementation work for other architectures like P2P or multicast. It however also implies that we should take care not to overload the servers in the ALVIC-NG framework and that our implementation has to work harmoniously with zoning.

It is impossible to implement every discussed method and thus, like the other platforms, we have to choose which ones to implement. We have opted to implement a dynamic and extensive Area-of-Interest technique with Level-of-Detail provisions (chapter 6 explains more details). We believe this is the most interesting candidate for implementation because it works well with other methods like DR, zoning and (logical) compression and because it has a strong focus on dynamically adjusting the properties of the specific AOI, this way enabling dynamic bandwidth management. AOI also allows such fine-grained interest management that it is possible to indicate very precisely which data is important for which user, which is ideal to help with aggressive bandwidth limitation.
The main issue with this technique is that it can be very processing intensive. As the plan is to execute this filtering on the proxy servers of ALVIC-NG (chapter 6 provides detailed

explanations for this choice), we will need to be careful that the AOI calculations are not too complex. This would cause the proxy servers to overload quickly on a limited number of users, while their purpose is exactly to support a large number of users at the same time.

We believe a generic AOI/LOD implementation can provide the basic possibilities that can be benefitted from by most types of NVE and, more importantly, that it can be easily extended with specific other techniques to fit the needs of any type of NVE. This will also be shown in the following chapters, where we will demonstrate the power of the AOI/LOD implementation for a number of different NVE concepts and explain how an AOI-centric approach can help to make good use of DR and compression techniques, whereas an implementation focussed around DR for instance would probably not be so easy to combine with a secondary AOI system.

Chapter 6

Implementation

From this point on, we have a clear goal: implement a dynamic AOI/LOD technique focussed on bandwidth scalability for the ALVIC-NG framework. In this chapter we will discuss how we approach this implementation and the choices made in this approach. The next chapter will then discuss how we used this implementation to obtain answers to our research questions through several simulations that were designed specifically to test every implementation detail.

6.1 AOI-API

As we have discussed in chapter 4, the usage of an Area of Interest mechanism can take many forms, from a simple circle-based representation to complex overlapping AOIs for different streams and different Levels of Detail. As the goal of ALVIC-NG (and most other NVE middleware) is to support many different kinds of NVEs, we need to ensure that the implementation is robust and extendable and that it can be used to represent many different situations in different types of NVEs.

To this end we developed the AOI-API, a small and to-the-point library that allows for advanced AOI creation, usage and manipulation. The AOI-API is built around the concept of a hierarchy of entities that together define a fine-grained interest specification. At the highest level we have the InterestManager. This entity maintains all AOI-related data for a single user/player. Its most important method returns a LOD level for a given object. This allows the calling code to easily check if an object is within the AOI specification for that user and at which LOD. If the object is not in the AOI definitions, an LOD of -1 is returned. The InterestManager contains multiple InterestDefinitions, which are primarily a way to group certain AreaDefinitions. These AreaDefinitions are the atomic parts of the interest specifications and each of them has a specific LOD and other InterestFlags assigned to them. This way, we can create InterestDefinitions with various AreaDefinitions that work concurrently.

The InterestDefinition can also utilize a caching scheme so we can save on some checks to see if an object is contained within a certain AreaDefinition. For example, we can keep a list of objects and timestamps for when they were last checked. Only if it has been long enough since the object has been checked, added calculations are needed to see if it is still in this AOI.
The AOI-API structure is visible in figure 6.1.

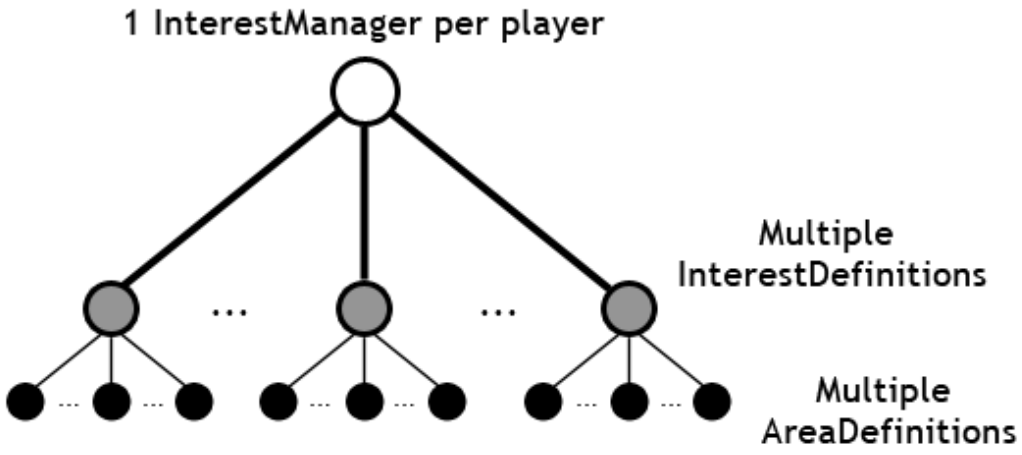

Figure 6.1: The AOI-API structure.

This structure gives us a very flexible way to define interest. InterestDefinitions can be disabled or enabled at will (as can AreaDefinitions for even finer-grained adjustments). We can also have multiple InterestDefinitions per player, for example each pertaining to a specific situation the player can be in and for which different filtering is needed. The LOD and InterestFlags assignment per AreaDefinition allows us to very meticulously filter certain traffic in certain areas.

Through this structure, the AOI-API provides the following feature set:

- Dynamic AOIs with different shapes
  Circle-, polygon- and wedge-shaped regions with methods that allow for easy manipulation of the shapes at runtime.
- Different AOIs for different LODs
  Every AreaDefinition also has an associated Level of Detail which can then be used by the calling code to determine how to send a packet that has been assigned a particular LOD.
- Different AOIs for different network stream and packet types and contents
  For every AreaDefinition we can define which streams and packets it is interested in

and only use it to filter that type of traffic.

- Multiple sets of AOIs per user
  Every user can have multiple sets of InterestDefinitions, which can be dynamically switched on or off. Multiple definitions can be active at the same time.

- API-centric design
  Care was taken not to make the AOI-API dependent on ALVIC-NG but rather to leave it as open-ended as possible to allow for usage in other frameworks as well.

- Extendability
  The AOI-API was made with application-specific extensions in mind. Users can easily use traditional inheritance-based approaches to define new areas and functionality. Most of the internal classes also offer generic data fields that can be used for adding application-specific logic without having to create new classes.

6.1.1 Dynamic shape definitions

Shapes inherit from the abstract IAreaDefinition class. They provide a specific implementation of the containsObject(object):bool method, which can be used to check if an object is inside this particular area. This way, users of the AOI-API can easily implement their own AreaDefinitions and shapes.
The AOI-API comes with three different types of shapes: circle, wedge and polygon. Each of these use highly optimized code to do containment checks and their parameters can be adjusted at runtime. The circle has a radius, the wedge 2 angles and 2 radii between which the wedge is defined and the polygon has a number of corner points.
In addition to specific parameters per shape, each area can be centered on an object in the world (for instance a player object) so that it automatically moves along with this object, around a group of objects (where it uses the point of gravity of the group to center itself) or it can be fixed around a set point in the world to allow for AreaDefinitions that encompasses a particular area, independent of player movement. Finally, each AreaDefinition can also use a specific offset from the used center point, to allow for even more customization.
Besides these shapes, there is also one special AreaDefinition that allows us to specify interest in a specific object or player. This AreaDefinition does not really perform a calculation but rather checks if the player or object ID is of interest. This allows for very quick processing and for specification of interest on a per-object instead of per-area basis. This can for instance be used to set up video/voice-chat with a group of friends, no matter where they are in the game world.
The AreaDefinitions are not directly tied to a specific InterestDefinition, which means they can be used by many InterestDefinitions at the same time, saving memory and instantiation. When we combine this with a caching system, we can dramatically reduce the amounts of

checks that have to be performed for the same objects if an area can be shared by a number of players. This is because if an object is in a given area for player 1, it will still be in that area for player 2 if player 1 and player 2 share the AreaDefinition, and thus we can skip this calculation for player 2.

6.1.2 LOD assignment

Every AreaDefinition has a specific Level of Detail assigned to it. This way, when we use the containsObject():bool function and it returns true, the caller knows that the object should be sent to that user using this optimal LOD. Internally, these LOD levels are of a very simple representation (an integer), as the actual adjustment of the traffic depending on the LOD is not done in the AOI-API itself. This also means new, application-specific LODs can easily be introduced into the AOI-API.
The calling code will rarely call the methods of the AreaDefinition directly. Mostly, they will call the methods of the InterestManager or InterestDefinition. The InterestDefinition will loop over all its AreaDefinitions and find the highest possible LOD for an object, as AreaDefinitions can overlap. The InterestManager does the same, but by looping over all its InterestDefinitions.
By slightly changing these schemes, we might obtain serious profits in speed. For instance, we can stop looking for a LOD as soon as we have found one (even though it might not be the highest), possibly preventing a lot of checks if there are many Area- or InterestDefinitions. This can be viable in certain situations or if CPU usage has to be kept very low and visual consistency is of a somewhat lesser concern. This can be combined with a sorting scheme so that the highest LODs are checked first.
Another approach is to sort the AreaDefinitions by shape. For some area types, it takes longer to determine whether an object is inside them than others. By first checking the areas with low computational cost, we might avoid more complicated calculations. This can be important if we have a high traffic rate for many clients or many different areas.

6.1.3 Interest flags

Besides a LOD level, each AreaDefinition can also have specific InterestFlags associated with it. These flags allow us to very flexibly disregard certain types of traffic for that particular AreaDefinition. A good example would be an area that is only interested in a video or audio stream and not in position updates. Without InterestFlags, the AreaDefinition would do a calculation for each method call, even for the position updates. However, when we specify the correct InterestFlags, it will first check whether they match with those of the current method call and only then will it calculate the containment.

The InterestFlags are not limited to the stream type however. The AOI-API allows for user

implementations. This means we can also filter on packet types within a given stream (only specific position/state updates for example) and even on the packet contents (e.g. only if a state update is playing a certain animation). This could also be used to pass some of the larger program-state into the AOI-API, which would allow for fine-grained conditional checks whether an Area should be used or not.

6.1.4 Multiple sets of AOIs per user

The InterestManager maintains a list of InterestDefinitions per user. This means that a user can not only have multiple AreaDefinitions, but can also group them together in multiple InterestDefinitions. This way, we can enable/disable/adjust groups of AreaDefinitions at once. This setup can be used to have a different InterestDefinition for a particular in-game situation and to dynamically enable one or more depending on the current situation. The simulations described in the next chapter clearly exemplify this possibility.
This approach also allows for world-types where users have multiple objects that they control and follow at the same time, for example Real Time Strategy games. This way, each unit (or group of units) can have its own InterestDefinition with specific AreaDefinitions.

6.1.5 API-centric design

The API-centric design allows for the AOI-API to be used in frameworks other than ALVIC-NG. It provides a set of ready-to-use classes, especially for commonly used AreaDefinitions, and at the same time allows for extensive customization through inheritance and also parameter passing (InterestFlags and LOD for instance).
The API depends on the concept of an Object/Entity in the world that has a specific 3D-location (x,y,z). The abstract class IObject provides the interface that is used throughout the AOI-API. Adopters of the AOI-API should implement the IObject interface, either by inheritance or by encapsulation (so-called proxy object). In our implementation, we decided to use the encapsulation option because we did not want the ALVIC-NG specific WorldObject class to have to inherit IObject directly. The AOIObjectProxy implements the IObject interface and has a WorldObject member and acts as the glue/translation between the two.
The main functionality has also been kept as simple as possible. After setup, a single method-call (InterestManager.getObjectLOD(object, InterestFlags):LOD) suffices to use the AOI-API and to obtain the interest calculation results in the form of a LOD level. This LOD information can then be responded to in an application-specific manner.

Listing 6.1: AOI-API simple example

```
// 1 InterestManager per player (use player ID)
aoi::InterestManager* manager = new aoi::InterestManager(1);
aoi::InterestDefinition* id = new aoi::InterestDefinition(1);
```

```
StreamInterestFlags flags(STREAMTYPE_VIDEO);

aoi::CircleAreaDefinition* area = new aoi::CircleAreaDefinition();
area->setReferenceObject( gameObject );
area->setRadius(50);
area->setLOD(LOD_3);
area->setInterestFlags(flags);
id->addAreaDefinition(area);
manager->addInterestDefinition(id);

...


StreamInterestFlags positionFlags(STREAMTYPE_POSITION);
StreamInterestFlags videoFlags(STREAMTYPE_VIDEO);

// will always return -1 because the stream types do not match
int lod = manager->getObjectLOD(object, positionFlags);

// will return the correct LOD
int lod = manager->getObjectLOD(object, videoFlags);
```

6.1.6 Extendability

The AOI-API provides a number of so-called interface definitions to enable programmers to create their own specific implementations of their functionality. This is most important for the IAreaDefinition, which allows for the creation of entirely new area types by implementing the containsObject() abstract method. These interface classes usually provide certain built-in and re-usable functionality (such as object caching) to prevent double work.
Next to this, most of the base classes contain generic variables, for example an integer field named type. This field is not used within the AOI-API, but instead enables programmers to use the field for their application-specific logic. For most simple cases, having a couple of these generic fields is enough, and it alleviates the need to create a custom inherited class just to add this kind of simple functionality.

6.2 NIProxy

In the previous chapters, a lot of attention was paid to the concept of a dynamically adjustable bandwidth usage for an NVE. While the AOI-API provides the tools to determine

if objects and entities are within a player's interest and at which LOD, this is not entirely sufficient to be directly used in a bandwidth shaping system. To provide more fine-grained dynamic control of the bandwidth usage, we opted to re-use concepts of the original Network Intelligence Proxy (NIProxy) [94] project, which was discussed previously in section 2.4.5.

The integration of the NIProxy provides interesting options for all three of the different systems we are trying to combine (AOI-API, NIProxy and ALVIC-NG). The NIProxy itself will be extended and tested on new kinds of data, as we are focussing primarily on position updates, a type of traffic that has not yet been heavily explored in NIProxy research. Furthermore, the NIProxy will be tested in a multi-user large-scale setup, which is one of the large open research questions for this framework [93]. ALVIC-NG on the other hand will gain most of the already existing functionalities of the NIProxy; the NIProxy will serve as a good starting point for the integration of various bandwidth shaping techniques into the ALVIC-NG architecture. Finally, the AOI-API can be coupled to ALVIC-NG in a very transparent way and the NIProxy can be used to tweak its parameters as needed. It will be possible to objectively measure the impact the AOI-API has on bandwidth usage when certain parameters are used, which will certainly help to determine its usefulness in a NVE for providing dynamic bandwidth scalability options.

In this section we will first discuss the opportunities and difficulties the NIProxy brings for its integration into ALVIC-NG. We then explain how we try to minimize these issues in our implementation and motivate our choices for the integration.

6.2.1 NIProxy limitations and issues

As discussed, the NIProxy architecture provides a very flexible and powerful way to represent different traffic streams and their interdependencies and manipulate them at runtime to obtain optimal bandwidth usage. However, there are many implementation details that hinder the direct usage of the NIProxy in our implementation.

Implementational limitations

Firstly, the NIProxy codebase was not designed to be used as a library in a different project. It consists of many different heavily interdependent projects and smaller libraries that make its integration into a different build system difficult. In addition, many of the higher-level classes assume a very direct knowledge of the connection to the user, the networking architecture and require the use of the NIProxy service subsystem. This means they would need to manage their own sockets, protocols and traffic adjustment implementations and it is difficult to ensure interoperability with the ALVIC-NG implementations for these aspects. A suggested approach was that we could run the NIProxy as a separate entity next to the ALVIC-NG

servers in the network (as opposed to integrating the codebase as a library into the ALVIC-NG server processes), but this would induce yet another layer of abstraction and possible point of failure. A further discussion of why this is not optimal will be given in a coming section.
Secondly, the method used for the building and maintenance of the bandwidth shaping trees is based on the assumption that the tree is first constructed client-side and then transmitted to the server, which replicates the clients' commands. This is a good approach for video streaming, but not for game-like environments where we want full control on the server (to prevent cheating or to allow for a dumb client for example). The direct impact of the client-side approach is that all tree operations are performed using extensive string parsing instead of via direct invocation of appropriate methods or class constructors. This makes the code more difficult to understand and adjust to our specific needs.
Thirdly, the NIProxy's class structure is heavily dependent on inheritance and templates. Some of the node classes for the bandwidth shaping trees have an inheritance depth of 6, of which 4 intermediate classes are template-based. In theory, this allows for a very extensible and re-usable architecture. In reality however, it becomes very difficult to locate specific functionality and determine how it can be adjusted. A component-based approach (as opposed to inheritance-based) would have been more interesting in this case, as new nodes would have been easier created by the assembly of simple components rather than adding extra inheritance layers to obtain more complex behaviour.

Conceptual limitations

Next to these implementational issues, there were also some conceptual problems. In the original NIProxy test cases, all the leaf nodes of the bandwidth shaping trees represent separate network streams (i.e., one leaf node represents a single video or audio stream). In addition, many of the experiments have a leaf node per user for these network streams, which means there are a lot of leaf nodes when dealing with a large amount of users.
This presents a few problems. First of all, it would be more practical for game creators to specify bandwidth usage based on the InterestDefinitions as a whole, instead of separately per network stream. For example, in a simple video scenario, the NIProxy would be able to transcode a video stream to a lower quality to reduce its bandwidth consumption. With the AOI-API however, it makes more sense to be able to, for instance, shrink the radius of a circle-shaped area to reduce the bandwidth in this way. Other interesting operations could be lowering the LOD for a specific area or disabling a complete InterestDefinition or single AreaDefinition. This means it makes more sense for the leaf nodes to represent a single InterestDefinition instead of a single network stream (as an InterestDefinition can contain multiple different network streams). This makes the bandwidth shaping trees more comprehensible and logical to create and manipulate in our scenarios. However, the NIProxy did not directly allow for this special usage and we had to create our own leaf node classes to

provide this new type of behaviour.
Besides this, having a leaf node per stream type per user is not very scalable. For example, if there are 50 players in our AOI, each with a position stream, there would be 50 leaf nodes in our tree. In addition, every time a user enters or leaves the AOI, the tree would have to be updated, which requires additional processing power and memory. In the original NVE example [91], this approach is taken, but for a very low number of users and it was not tested for performance. It makes more sense to just have a single leaf node for the entire AOI where the different streams attribute to an aggregated bandwidth usage for the entire AOI, instead of operating on a per-player and per-stream basis. This gives us less fine-grained control possibilities in exchange for a much simpler representation and serious performance and tree maintenance gains. In addition, it is unlikely that the adaptation of a single position stream of the 50 users would lead to much bandwidth usage difference in this larger-scale setup. It is better to, for instance, change the LOD for the entire AOI to get a larger and more directly measurable bandwidth adjustment.
Lastly, the NIProxy's bandwidth distribution scheme makes heavy use of predicting how much bandwidth a particular situation will use. For instance, we can very accurately predict how much bandwidth a transcoded video with a certain codec and resolution will use. The NIProxy can then use this knowledge to decide to which quality level it should switch to when the available bandwidth suddenly changes. For the AOI-based approach, this is a lot more difficult. Firstly, in a fast-paced game, it is likely that many objects will enter and leave the AOI in short intervals. Therefore it is difficult to predict what the bandwidth usage of a given configuration will be, even if we have the measurement for the current frame. In addition to this, it is difficult to predict how a certain configuration change will actually influence bandwidth usage. For instance, we can shrink the radius of a circle shaped AOI to half its size. However, this will not necessarily mean the bandwidth usage will also be reduced by half. If most of the players are close to the circle's center, the actual bandwidth change might even be 10% or less. This means we have to constantly re-evaluate our actual bandwidth usage and take further actions to try and get the desired bandwidth usage. This also means that it can possibly take much more time to arrive at a steady bandwidth distribution than with the original NIProxy experiments. It furthermore implies that serious spikes in the bandwidth usage are possible, which might lead to exceeding the available bandwidth, with bad side effects as a consequence (e.g., packet loss and increased delay due to overloaded network links). One possible solution would be to be very conservative and for instance set the distributable bandwidth budget to just 60% of the actually available bandwidth. This will prevent spikes having detrimental effects (as they are absorbed by the unexploited bandwidth), but also makes for non-optimal, non-complete bandwidth usage.

Another conceptual issue was that, in the original experiments, every NIProxy user had his own bandwidth shaping tree that worked separately from the other users. The available band-

width for that user was set at the root node of his tree and the calculations affected the nodes to obtain the traffic shaping. For ALVIC-NG however, it would be interesting to be able to give certain users more bandwidth than others. This could be achieved by just defining 1 huge bandwidth shaping tree where the single root node represents the total bandwidth available to the server. This root node's children would then be the nodes that represent a player (the separate player trees are thus subtrees of the main tree). Thus, by using for example a priority node for the main root node, we could make sure that some players receive more bandwidth than others. This approach can be interesting in non-entertainment applications where certain important meetings/interactions need absolute priority or for quality of service provisioning.
The main problem here is that the root node would represent the server's total available bandwidth, while we also need to account for the bandwidth of the clients. For example, assume the server has a capacity of 5000 Kb/s. Say we have 5 clients; each would be allotted 1000 Kb/s from the server if everyone had the same priority. However, one of those clients could be connected through a slow mobile link, with a maximum downstream bandwidth of 500 Kb/s. This means we would waste 500 Kb/s that could possibly be used for other clients. Thus, we need to not only be able to account for changes in the available server-side bandwidth, but also the client-side bandwidth, and make sure excess bandwidth is redistributed. There was originally no direct provisioning in the NIProxy framework to deal with these two separate types of bandwidth constraints, so we had to find a way around this limitation.

These practical and conceptual factors indicate that the NIProxy is much more difficult to use in the ALVIC-NG setup and use cases than it is in the original NIProxy experiments. It was interesting to see if it could be used at all for these purposes and which of these factors would have the biggest impact. In the next section, we discuss the choices made in our implementation to address the discussed problems and try to use the NIProxy as dynamic bandwidth shaping framework for ALVIC-NG.

6.2.2 Practical use of the NIProxy

As we are using the NIProxy in a completely new context, it is normal that some parts will be unusable, while others will require adjustments. Our difficulties should not be seen as direct criticism on the NIProxy project but rather stem from this entirely new context. This section aims to explain some of our encountered challenges in this regard.

Code approach

The interest to use the NIProxy as a part of the ALVIC-NG processes instead of as a separate entity led to the necessity to separate out large parts of the more high-level classes and implementations. This includes most of the NIProxy packet processing chain, Bandwidth- and

StreamManager and the multimedia service subsystem. In essence, only the classes used for creation and maintenance of the bandwidth shaping trees was used as-is, in addition to some of the smaller utility libraries. Even though this allows us to use the most important parts of what we need for our experiments as a library in ALVIC-NG, it still does not circumvent the heavily inheritance based architecture and string-parsing based tree manipulation. We briefly considered implementing a new system to provide the same functionality but via a simpler API and with higher usability. However, due to time constraints and the sheer amount of complexity in the system, we decided to keep the existing codebase and rather provide wrapper classes with convenience functions to more easily invoke the functionality.
In addition to this, we made our own implementations for the Bandwidth- and StreamManager entities, re-using some code of the original versions but adjusting it to our specific needs and making up for the lack of a stand-alone NIProxy entity.
An important consequence is that we sadly were not able to retain the extensive service subsystem in its current form. One of our original motivations for the use of the NIProxy was that we would get options for video transcoding and FEC protection "for free" through the existing implementations, which could be interesting for more serious NVEs. We do think that these implementations could be re-used within ALVIC-NG and our adaptation of the NIProxy quite simply by writing some intermediate code and taking care to properly translate the concepts from the NIProxy main use case to the ALVIC-NG scenarios. This is however left as future work and is not part of our implementation.

Conceptual approach

Our decision to re-use the bandwidth shaping tree implementations instead of making our own version meant that we would need to find a way to make a leaf node represent an InterestDefinition rather than a single network stream. This makes it easier and more logical to construct trees and in addition removes the need to have a leaf node per-player and per-stream. In practice, this was not as difficult as we first imagined. For this, we made smart use of the concept of a discrete leaf node. This type of node represents a stream with a number of discrete quality levels. The simplest instantiation is a node representing a stream with 2 levels: zero and one, to turn the stream on or off respectively. In this most straightforward approach, there is hence no notion of multiple quality versions. Another possible representation is a video stream with 4 levels: 1080P, 720P, PAL, off. At any time, only one of the levels is active and the tree chooses the level with the most appropriate bandwidth usage with respect to the current bandwidth constraints.
We can then represent the different adjustments to a given InterestDefinition as a list of discrete levels in a leaf node. For instance, level 1 corresponds to the original InterestDefinition, level 2 has a lower LOD for a specific AreaDefinition, level 3 has a smaller radius for a circle area, level 4 disables the video transmission, etcetera. When the available bandwidth changes, one of the levels can be chosen as a possible solution. However, as we have discussed,

we cannot be sure of the concrete bandwidth influence of our decision (as opposed to the video quality levels, where we were very sure).

There are multiple approaches to try and alleviate this problem. Firstly, we could calculate the bandwidth usage for all the levels by actually checking the traffic for all the different InterestDefinitions, even if they are not actually active at that moment. This would give us an idea of the bandwidth usage for that specific option, especially when we would use an historical average of for instance the five most recent measurements. This does not help for a lot of entities suddenly entering/leaving the AOI, but does make it much more probable to select an appropriate option for a given bandwidth change. The downside is that it uses more processing power as all the options have to be calculated, even though they are not used. Some smart optimizations here help reduce this overhead however. For instance, adjustments to LOD or InterestFlags do not require a recalculation of shape containment (arguably the biggest performance factor) as they only influence the packet size. Only shape changes would have to be recalculated. If we take care not to include these in high numbers or for example only calculate them for larger intervals (i.e., once every 3s instead of every 500ms), this overhead can be kept low too.
A second approach would be to iteratively adjust the current option based on measurements. When the bandwidth changes, we choose the option we think is most appropriate (for instance shrink the radius by half) and we wait until the next bandwidth measurement to see if it indeed had the desired effect. If not, we can decide to try a different option or even to combine options to get a more complex behaviour. This approach reduces the need for extra computations as only the bandwidth consumption of the current configuration has to be calculated (which we have to do anyway to perform the filtering). However, it induces a possibly big delay to find the most optimal option for a given situation. If the measurement interval is too big, it can take multiple seconds for the correct option to be found, during which the bandwidth can be severely under- or overused. This method could be optimized by doing lots of simulations during development to determine the optimal configurations and a heuristic of how much they will typically change the bandwidth. This heuristic can then be used to come to the best option much more quickly, but it requires added development time and effort.
In a more extensive implementation, we would opt for a hybrid approach that combines both methods. Depending on the performance parameters of the server, we could switch the calculation of all options to the iterative approach when the performance begins to suffer. We could then use any calculations made in the first approach to serve as a heuristic for the second. This allows us to trade performance for bandwidth fidelity when needed, while possibly reducing the need for development time based heuristic construction (however a combination could of course be made and would be advisable for extra fidelity).
In our implementation, we opted to use only the iterative approach without advanced heuristic calculations. This is mainly because we had severe time constraints and this approach

was a lot easier to implement than the first method. Because we mainly used very controlled simulations and game logic (see the next chapter) for our experiments, we could provide a good heuristic and option balancing through design. This helps prevent severe bandwidth anomalies in the experiments, making it easier to evaluate this method when used with a fitting heuristic.

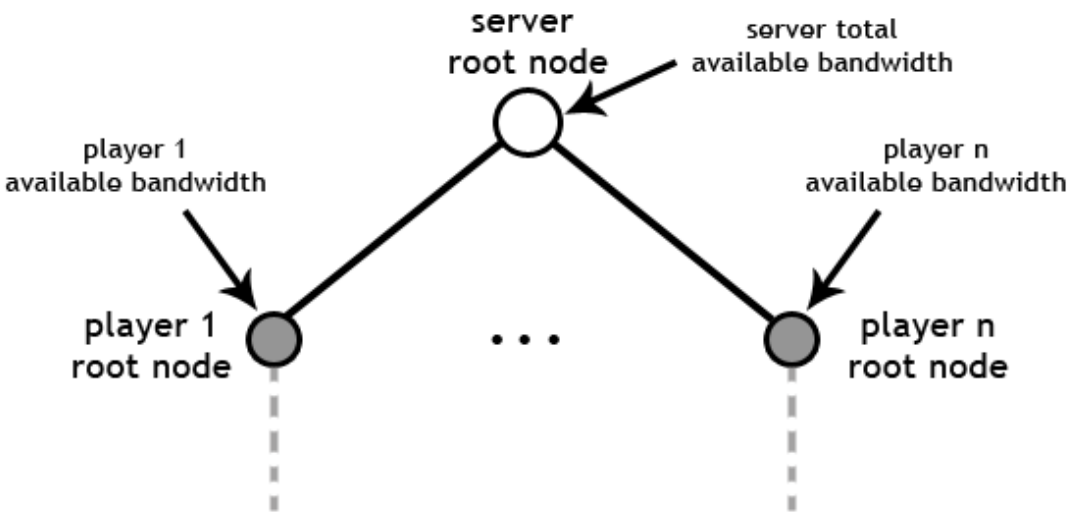

Figure 6.2: Bandwidth shaping tree that spans the entire proxy server. Individual player trees are coupled to a central server root node.

The other main conceptual issue was that we wanted to connect the different bandwidth shaping trees for the players into one larger tree to provide high-level bandwidth distribution. This also turned out to be easier than originally estimated. The proposed solution is to create a new type of intermediate node that can take two different bandwidth constraints as input and uses the smallest one as the actual budet (i.e., min(server allotted bandwidth, client available bandwidth)). This can be seen in figure 6.2. If the client bandwidth is smaller than the portion given by the server, the excess bandwidth can be re-used by other clients through the built-in feedback and redistribution implementations of several of the internal node types. Thus, if we just use this new type of node as the root for each player subtree, we get the required behaviour. The only drawback of this approach is that we would have to provide this functionality separately for each type of intermediate node (i.e., percentage, priority, weight, ...) because of the inheritance-based architecture.
In our implementation we decided not to implement this option as it would be simple enough to add later and did not offer a lot of extra functionality for our envisioned experiments.

Conclusion

In hindsight, the bandwidth shaping part of the NIProxy framework was a lot easier to adapt to our needs than we had originally anticipated. Most of the conceptual issues were solved by introducing a new type of node for the tree, where we could often base the implementation on existing source code.
The main problems were the implementation details, that made creating these new types of nodes more of a hack or "smart use" of concepts earlier used for something entirely different. In addition, they made it difficult to retain many of the other aspects of the NIProxy besides the bandwidth shaping tree part.
This all shows that the ideas and concepts behind the NIProxy can be used for all kinds of traffic shaping approaches and thus that the framework is flexible and robust and forms a valuable inclusion into ALVIC-NG. A new, more general implementation can help alleviate most issues we have seen here.

6.3 Integration into ALVIC-NG

Now it is time to discuss how we can properly integrate the AOI-API and NIProxy into the existing ALVIC-NG framework. First, we discuss where in the network we can possibly integrate the filtering and motivate our choices. After this, we discuss how our implementation helps provide low-coupling between the existing ALVIC-NG systems and our filtering implementation.

6.3.1 Where to perform the filtering?

There are roughly four places we could incorporate the NIProxy: inside the internal logic servers, inside the proxy servers, inside the client program or as an extra server somewhere in between these three entities. All of these positions have their advantages and disadvantages. To determine the optimal location, we should first elaborate on the common traffic flow in a 4-tier system like ALVIC-NG. It is so that the users generally only send a relatively small amount of data to the servers, containing their commands. The traffic from the servers to the users on the other hand is typically much larger. Not only does the server have to send a (limited) copy of the world state to each user, it is also responsible for sending full state when a user enters a new zone, and possibly also user generated content. Most of this traffic will be generated on the logic servers and is then sent to the proxy servers, who in turn deliver it to the users. Figure 6.3 gives an overview of the traffic patterns and how the different possible placements in the network can influence these patterns.

Each of these possible deployment locations has its advantages and drawbacks. The best place would be in the logic servers, as these will have to send and receive most of the traffic.

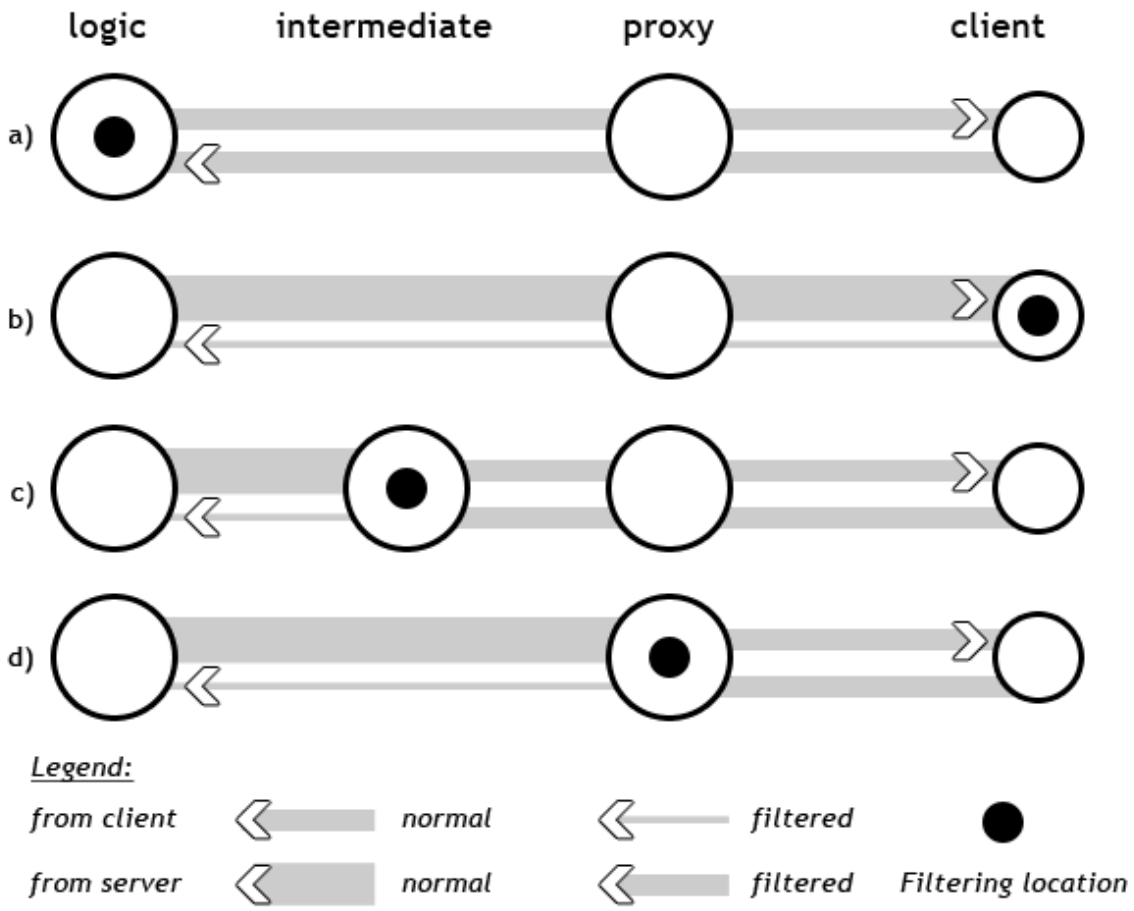

Figure 6.3: Four possible locations of filtering in the ALVIC-NG network.

This placement can be seen in figure 6.3, point a. If the filtering can be performed at this point, most of the unnecessary traffic to the proxy servers will be eliminated and by consequence also the unnecessary traffic to the users. The biggest drawback is that the filtering will require a considerable amount of processing power. Since the logic servers are meant to calculate the updates to the game state and perform things like physics simulations and gameplay checks, they will already have high processing workloads. Adding the filtering here would mean that every server could only manage a relatively small zone, increasing the number of zone-crossings and by consequence all the traffic that is needed for those zone-crossings.

Another possible location is at the user side. This placement can be seen in figure 6.3, point b. This will effectively help manage the user's upload traffic, which can be interesting for networks with low upload speed. However, this would be the only thing the filtering would provide: the downstream bandwidth consumption behaviour would remain unchanged. There

is also the problem that there are only a few methods that can be effectively used for limiting the data a client has to send, as we have discussed before in chapter 4. For instance, zoning is not possible and neither is an extensive form of AOI. Only Dead Reckoning will be usable and this will limit the options of the server to provide other players with more detailed information about the player. On the other hand, as most clients will only run a limited simulation on their local computers, there will be typically excess processing power available to perform the filtering. This would bring the system closer in abstract concept to a P2P setup, where every new player adds extra resources to the NVE, in this case to perform the filtering.

Another option would be to perform the filtering in between the logic and proxy servers, or in between the proxy and the user, as part of a separate entity. This placement can be seen in figure 6.3, point c. This adheres more to the original concept of the NIProxy: to be a network intermediary. A deployment in between the logic and proxy servers would help to limit the traffic bidirectionally, both relieving the logic server of some of its pressure and limiting the updates the proxies receive and thus have to send to the users. This would seem like a good solution, but it would require a new type of server in an already complicated networking architecture. It would be difficult to manage and deploy these extra servers, to make sure that all connections are properly set up between the different tiers and to provide failsafe mechanisms in case one of the servers goes down. Furthermore it would require extra hardware, increasing the cost of NVE deployment, and it would require much game-related into to be present on the NIProxies as well, increasing implementation complexity.

The final option is to integrate the filtering into the ALVIC-NG proxies. This placement can be seen in figure 6.3, point d. This way, we still have the basic purpose of the NIProxy as intermediary, but this time between the logic servers and the user. As this is already the basic function of the ALVIC-NG proxy as well, it makes a lot of sense to incorporate the NIProxy at this place. The ALVIC-NG proxy primarily serves as a connection point for the users so that the logic servers do not have to manage a lot of connections and so that any topology change in the internal network is hidden from the users. Because of these limited responsibilities of the proxy servers, they have lower processing requirements and can serve a large number of users at the same time. This means that there is the option to trade some of this processing power (and also users per proxy) for filtering possibilities in the proxy servers. Furthermore, this still allows the possibility to limit traffic to both the logic servers and the users, which was one of the main advantages for the previously discussed approach. Because this placement is so logical, we have chosen to integrate the filtering into the ALVIC-NG proxy servers.

This approach also has some disadvantages however. As we have said, there will be a trade-off between the number of simultaneously supportable users and the complexity of the provided

bandwidth management. If the number of users which a single proxy can support becomes too low, the primary advantages of the proxies in the network are lost and we will still need extra hardware to support the filtering. This means that the filtering implementation has to be maximally efficient to be able to serve as many users per proxy as possible.
Secondly, there is the problem that we are focussing primarily on the bandwidth stream from the proxy to the user. As we have said, the traffic from the users to the logic servers (and thus also from the proxies to the logic servers) is relatively small and difficult to filter. It is the traffic from the logic servers to the proxies and from the proxies to the users that will be large, especially if the world is very dynamic. When we perform the filtering in the ALVIC-NG proxy, we can effectively manage how much of this traffic actually reaches the user, but there will still be a large amount of (potentially unnecessary) traffic from the logic to the proxy servers, which cannot be filtered. To add to this traffic, proxies might serve multiple users who are not necessarily close to each other in the virtual world. This means a proxy will potentially receive updates from a number of different logic servers (which each manage a different zone) and will hence effectively receive many data streams; a lot of logic-to-proxy-server data will be filtered and will hence not reach the user, which implies that this data is actually being sent unnecessarily.
This traffic is generally not a big problem when the logic servers and proxy servers are located close to each other and connected via very high-speed links, as is usually the case in commercial systems. It only really becomes a problem if the proxies would be placed far from the base of the network and are connected to the logic servers through slower links. If we would need to filter this internal traffic, we would possibly need other methods than we would likely use to limit the traffic on the sometimes called "last mile" between the proxy and the user. Furthermore, it would also require filtering implementations and computations on the logic servers.

For these reasons, we have decided to focus on reducing the traffic between the proxy servers and the users and not between the logic and proxy servers. Our filtering implementation could possibly be used for reducing this latter type of traffic as well, but for the rest of this text we will assume the proxy and logic servers are connected through a high-speed network with sufficient bandwidth to support the logic-to-proxy traffic. This means we can also always be sure that the proxies will have all information about the zones its users are in, which will make the filtering implementation easier as we just have to filter a complete set of data, instead of an already incomplete set, which would be the case if the logic servers would also perform filtering.

Finally we should mention that even though we have opted for a single filtering position in the network, it is equally possible to perform the filtering on multiple places at the same time. For instance, we could do filtering on the client side for Dead Reckoning calculations, on the proxy servers for limiting the traffic that goes to the user and another, possibly lightweight

implementation on the logic servers to filter the largest chunks of unnecessary data. Given the very loosely-coupled nature of our implementation (see the next section), these extra filtering locations can easily be added later with small effort.

6.3.2 Separation via plugin-like system

Regarding the actual integration into the ALVIC-NG proxy, we decided to completely separate the filtering behaviour from the core systems by using a plugin-like system. This allowed us to work independently on the filtering implementations while the other system of ALVIC-NG could evolve without being hindered. This also makes it possible to completely enable or disable the filtering at runtime, by just setting the plugin to (non-)active.
The concrete implementation is through the FilterManager. Every network packet is sent through a list of different filters, which can decide to block, pass or adjust the contents of the packet before passing it on to the next filter. Only when the entire list of filters passes the packet, it is sent to either the logic server or client. The AOIFilter maintains an NIProxy bandwidth shaping tree per user, checks whether a packet is in the user's AOI and can change the packet contents, for instance to adjust for different LOD levels.

The main drawback of this highly separated integration is that it is more difficult to use the AOI-API's area definitions in the zoning system directly. If a user is close to the border of his current zone, the ALVIC-NG proxy will automatically connect to the logic servers of the neighboring zones in order to obtain their info. This is currently done using a very simple circle-based AOI: when this circle intersects a certain zone, its data is requested. The eventual goal is to use the InterestDefinitions we use for filtering to also check which zones we should get data from.
This also stems from the fact that we have added the possibility to have multiple InterestDefinitions per user, where ALVIC-NG was built to only deal with one object/location/AOI per user. Changing this would have far exceeded the scope of this thesis and was left as future work.

Finally, we would like to make a small remark on the current implementation of how ALVIC-NG communicates world state. Normally, the logic servers would receive the user input, update the world state, and send this world state to the proxy servers for distribution. These world state packets would contain information on a number of objects in the scene. In the current implementation however, the logic servers act just as an echo server: every packet they receive is sent as-is to the interested proxy servers. This mimics a server that authorizes every user movement and does not check game logic or performs physics calculations. This for instance means we do not deal with aggregated world state packets but with smaller PositionPackets. This makes it easy to perform the filtering on a per-update basis, but this

situation is not very realistic for real game systems. It also means we are sending a lot more smaller packets from the logic to the proxy servers than would normally be the case with the larger, more aggregated world state packets.
When the logic server changes its behaviour, the implementation of the filtering will also have to be adjusted. World state packets can contain information on multiple objects, including some which might not be in the considered user's AOI. The filtering would then need to be able to re-assemble world state packets based on the user's interest, as the packets coming from the logic server have to be applied to the proxy's world state, after which a new packet has to be constructed for the client. This can increase the complexity considerably. For our tests however, this world state setup does not directly influence the evaluation of the AOI-API or even NIProxy usage. It does have some impact on CPU consumption and the actual sizes of the packets will probably be quite different in a more realistic implementation, but the relative bandwidth gains should be obvious nonetheless.

6.4 Simulating client behaviour

In this section we first discuss how we automatically simulate client behaviour for thousands of clients. Afterwards, we look at some implementation details that make the deployment of a large number of clients possible.

6.4.1 Behaviours

The simplest way of testing a game would be to set up a server process on a server machine and run a client application on your local computer and perform the necessary interactions yourself using mouse and keyboard. When testing a multiplayer game, multiple developers might run their own client applications on different computers. However, this approach requires the developers to invest time for tests and it is impossible to make tests that are exactly the same every time they are run, which might be interesting for objectively checking code change impact.
In addition, ALVIC-NG is a system for large-scale networking environments. This means we need to simulate thousands of individual clients, so it is impossible to rely on developers testing everything "live".

A better approach to testing is to devise an automatic system that can run any number of instances (clients) and that can simulate independent behaviours for each of them. A way of simulating these behaviours would be to record real player movement and then later play back that movement from a file. This way, we would only need developers or playtesters to play the testing scenario once, after which we can automatically replay it. However, this method is not very flexible, as a client will probably not perform the exact same actions as another client in a real-life situation. This means that we need a separate recording for every client

as we cannot simply use the same recorded movement for more than 1 client. For a 32 player FPS this is quite possible after a large playtest with the entire team. But for thousands of players, this is a lot more cumbersome.
A more interesting way is to use Artificial Intelligence (AI) techniques to try and obtain realistic movement. Here, no recorded movement is used to simulate a client. Instead, algorithms figure out what the most likely behaviour would be, based on a number of clues, such as world geometry or other players' positions. This means that, with one algorithm, we can simulate as many clients as we want, as the algorithm will adapt itself to different situations and starting parameters automatically. These algorithms often work with random numbers and as such a pseudo random number generator with a certain seed can make sure these behaviours are also exactly the same across different runs, as was the case with the previous method. The biggest drawback here is that to obtain realistic player movement, we would need a very advanced AI algorithm.
Of course, a hybrid of the two methods can be used where we start with recorded movement and introduce random deviations with AI to make recordings more re-usable across different clients. However, this would also require a lot of effort to get a realistic system.

For our experiments, we decided to use various simple AI algorithms to simulate player movement, as they are adjustable, easy to use and require little up-front work. As discussed, it can be argued that these algorithms do not produce realistic player movement. This is further discussed in the next chapter where we look at the results. For now, we limit ourselves to the discussion of the various implemented behaviours.
The simplest behaviour makes a client move in a circle around a given center point at a certain speed. This has very few ties with normal player movement but is very tweakable and requires few CPU resources for calculations. This is normally only used for quick tests that are not tied to a specific experiment. It does have the interesting property that the behaviour repeats itself. The clients keep moving in the same circles, so after they complete their 360 degrees, the same cycle starts over. Over long enough periods of time, this leads to a periodicity in the events in the world, which can be useful for seeing how a system holds up during recurring events and stress testing.
The second type of behaviour is flocking. The original boids research paper [**?**] describes a simple yet elegant AI algorithm to simulate flocks of birds. Each individual animal evades obstacles, stays within an optimal range of other birds and aligns itself to the general movement of the flock. These simple rules make for a good simulation of group-based player movement and they have been used in previous ALVIC-NG research papers [**?**, **?**]. This behaviour was used for most large-scale experiments.
The third type of behaviour started out as a straightforward purely random movement, where the client would change direction every couple of seconds. This did a good job of being purely random and is thus theoretically a semi-usable approximation of player movement, but it was

not very tweakable or controllable. This behaviour was reworked to a waypoint behaviour, where the client moves in the direction of a given waypoint. By randomly moving the next waypoint, the same random behaviour can be obtained. However, we could also use lists of waypoints that are not random but controlled by a use case designer. This way the behaviour can be used to meticulously direct client movement. This in turn can help to simulate player movement between important landmarks in the game, for example a player moving from one quest area to another in an MMORPG. The waypoint behaviour can easily be combined with the flocking behaviour to make sure the clients follow a given path or a random target, but still exhibit the flocking rules.
These behaviours all control the position of the player in the world, but not necessarily the rotation. For this, we have also included some smaller options. For instance, a client can be made to change his rotation to look in the direction of another client or the rotation can fluctuate to simulate a player looking around.

Using these behaviours, we can design a wide array of different experiments and test cases. The exact contents will be discussed in the next chapter, but most test cases are similar in setup. Every client is given a certain start location and a behaviour is attached to it. Different clients in the same test case can have different simulations affecting them.
For the larger scale simulations, we use the random versions of the behaviours, where most parameters are chosen at will from for example a range of options. For the more specific experiments, very fine-grained behaviour can be defined to create very specific environments. We also included a way of changing parameters and even behaviours during the runtime of the system, so that we can simulate events in the world at given times.

6.4.2 Many clients in one process

In the implementation of the client simulations we used a clever trick to make everything more manageable. The behaviours enabled us to automatically simulate thousands of clients, but this would not matter much if we would still need a separate computer for each client. It would be possible to run multiple client processes on a single computer, but this has a lot of overhead and makes it a lot more difficult to control cross-client behaviour. It would also take a long time to start and later stop all the independent client processes.
Thus we decided to simulate multiple clients through a single process. Every client still has a separate socket connection to the server, but many aspects of the process are shared between them, for example the GUI, the currently running simulation and world state.
Without this setup, development and testing would have been more cumbersome and much more difficult to debug. It also makes it a lot easier to run experiments on a single local computer (where this computer runs all servers + the client process). Where we would normally have problems running 100+ processes on a limited virtual machine setup on a medium-

performance home computer, our approach makes local testing possible, saving deployment time to the server cluster and thus speeding up iterations.

6.5 Graphical user interface

The servers and client processes of our implementation can be run perfectly as console applications. In fact, this is the method used for deploying and testing our experiments on a larger server cluster, see the next section.
However, especially for debugging purposes, console applications are often difficult to work with, as you only have textual output to go on. A graphical user interface (GUI) can offer a lot more options and is a lot more powerful in this regard. We chose to implement an extensive GUI system that provides many graphical debug options and runtime parameter tweaking. This GUI system is completely separated from the main implementation and primarily uses polling and object inspection methods to be able to represent the application state. A simple C++ #define directive can be used to compile the systems with or without GUI support.

Historically, the GUI was exactly the same for both the Proxy server (where our filtering integration into ALVIC-NG took place) and the client process. Its main function was to show a top-down overview of the game world with images representing the individual clients and their position/orientation in the world. Next to this, the GUI also rendered all the AreaDefinitions. This makes it a very powerful tool for testing and debugging new shapes and checking if the calculations to see if an object was inside a specific shape were correct.
However, after some time in the project, the GUIs started to diverge, as both proxy server and client required different functionality and debug data to be shown. The most important adjustment was to no longer show any AOI data on the client side. In the earlier implementations, we kept the clients in sync with what the proxy server was doing with the AOI definitions, but when we started integrating the NIProxy and dynamic AOI adjustments, this became too cumbersome. From then on, the client GUI shows the viewpoint on the world-state for a single client, where the client can be switched at runtime. The proxy server GUI shows the complete world state and we can choose for which user(s) we want to show the AOI definitions. This difference can be seen in figure 6.4. On the client side, we only see the world state for a single selected client. Red entities indicate other users that have been out of the current client's AOI for less than 2 seconds.
The full GUI for the proxy server side can be seen in figure 6.5.

Next to this main function of graphically showing world state, both GUIs offer some interesting extra functionalities. Through the proxy server GUI, we can enable/disable filtering, see the current bandwidth usage for each client (textually and plotted in a graph) and see a complete tree of AOI definitions with relevant parameters. An interesting addition would be a visual

Figure 6.4: Comparison of the different GUI views on the world.

representation of the bandwidth shaping trees of the NIProxy, which did not make it in due to time concerns.
The client GUI offers the option to pause the current simulation so we can check world state at a given time. It also gives us statistics on average round-trip-times to the server for each client and allows us to directly control a single client (or group of clients) through the use of keyboard and mouse (as if it were a normal GUI for a RTS game for instance).
Some of these extra functions and how they are visualized can be seen in figure 6.6.
Through these options, the GUIs allowed us to visually debug and test various elements of the implementation and thus catch and repair actions very quickly. However, they also came with the drawback that a lot of extra effort had to be put in their development. For example, the rendering of a wedge shape using the Qt graphics library [**?**] proved quite challenging and in need of serious debugging of its own. This is one of the main drawbacks of using a more graphical way of debugging: if there are bugs in the graphical representation, you might draw the wrong conclusions about the underlying data it is supposed to convey.
All in all, we can see the added development cost of a GUI as a good investment, as it will usually pay itself off in later stages of the project, as was also true in our case when using it to create and test our experiments before deploying them to the server cluster.

A last advantage of the GUI is that it makes the experiments a lot more understandable and tangible to outsiders. It is easier to explain what a given algorithm or experiment does if you

Figure 6.5: The GUI for the proxy server viewpoint.

can show it to the interested people graphically. In addition, if they have a good conceptual overview of the situation, it is often easier to understand or automatically deduce the more intricate implications of a given system. This is why we took our GUI a step further and also tried to make it visually attractive and place it into a recognizable context, as this is an area were we often find other research projects lacking.

IT students or researchers are rarely graphically talented and most research projects will feature unattractive ad-hoc graphics that are just enough to make the point, if even that. While fellow IT aficionados might have an easier time understanding the meaning of 100 green and red moving circles on a screen, people outside our area of expertise will have a harder time imagining that those are supposed to represent separate players in a World of Warcraft type environment, viewed from the top down. While one can argue that most research is not intended for outsiders, we believe more efforts should be made to make research accessible to a broader public, and better graphical representations can go a long way in aiding this. In

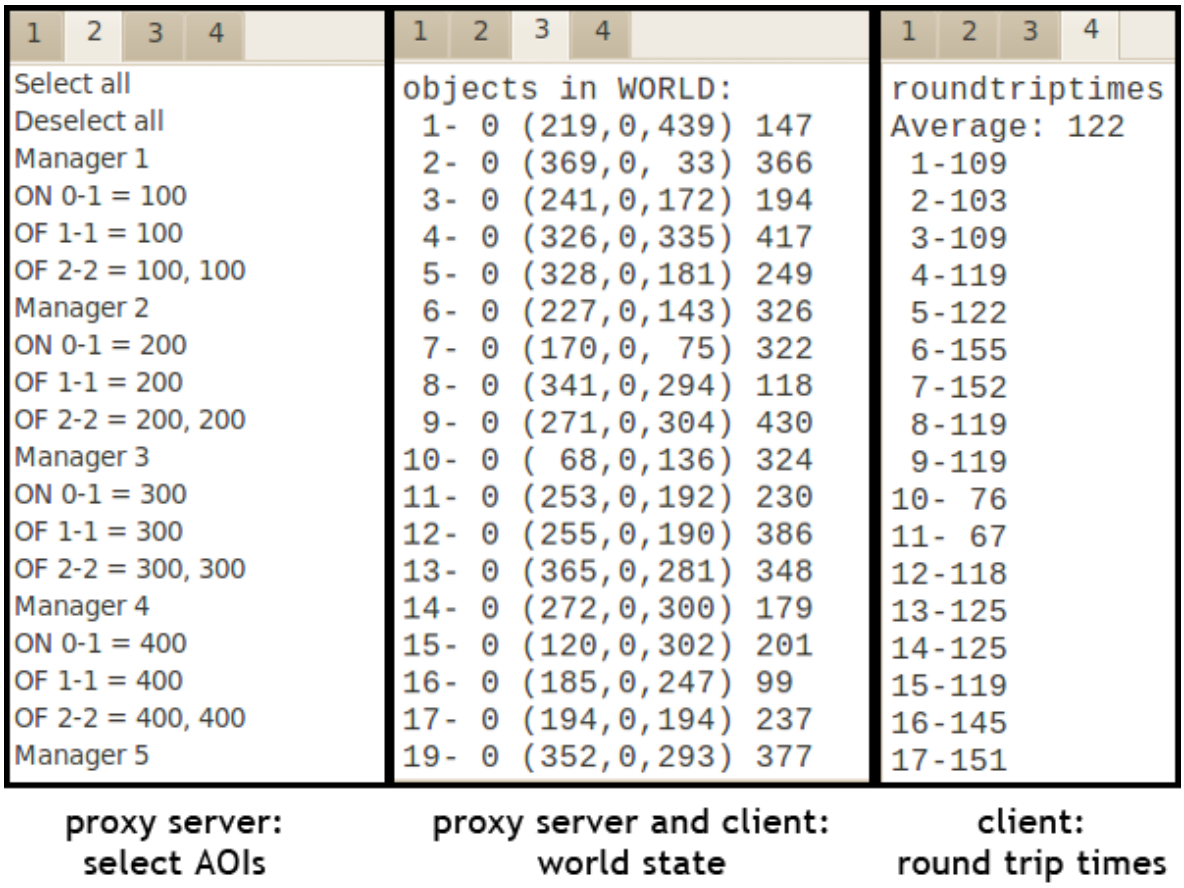

Figure 6.6: Extra functionality in the proxy server and client GUIs.

addition, a more contextual representation of the world can also benefit insiders, as seeing groups of tanks flocking together or watching a sniper following a distant target is still a lot faster to grasp than simpler visuals.

6.6 Deployment and result analysis

6.6.1 Deployment to a server cluster

As mentioned in the previous sections, we had two main ways of testing and running our implementation during this project. One was to run everything local on one machine with GUIs. This is mainly for debugging purposes and to enable quick iterations. Once the implementation was found working locally, we would move to deployment, testing and eventually result extraction on a larger cluster of servers. Depending on the experiment, from 3 to 9 servers were used, where each server hosted a single process, i.e. either a region management server, a logic server, a proxy server or a client process (that in turn simulated many clients). This setup is close to what the setup would be in an actual deployment situation for a com-

mercial system, except for the client processes, and allows for a good basis for our experiments.

As we also discussed, all the processes could run perfectly without a GUI, so that was not a problem when running the setup on the cluster. The first challenge here was to transfer the latest files and dependencies to the server machines. Both locally and on the servers, we used the same Virtual Machine image of an Ubuntu linux distribution to remove the need to recompile the code remotely on the servers and enabling us to just copy the files. For this we created a script that uses the rsync command to recursively sync all needed libraries and binary files to the servers.
The second challenge was then to quickly run the experiments on the servers. We could use ssh to connect to each of the servers individually and start the correct processes by hand, but this would be cumbersome, especially seeing as we were planning to run a lot of experiments. We decided to make use of the PHP scripting language to create a script that, through making ssh connections automatically, starts the necessary processes on the different servers in the correct order. It also pipes the runtime output of all the processes back to the local machine for easy monitoring. This process can be seen in figure 6.7. At the end of the simulations, the script copies all the outputted log files from the servers to the local computer for further processing (see the next subsection).

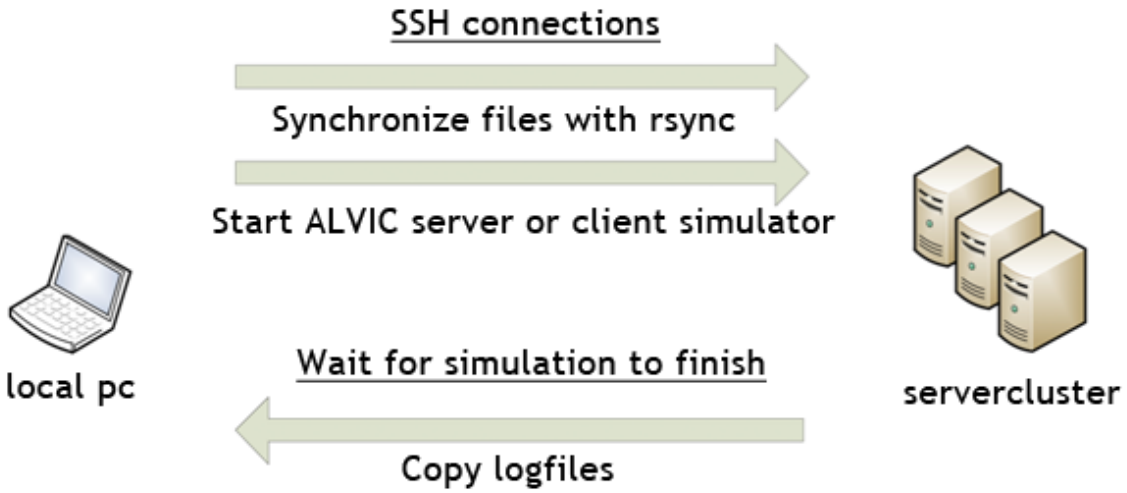

Figure 6.7: The various steps involved in automatically running a simulation on the servercluster.

The use of these scripts allowed us to very quickly deploy and test the implementation on the server cluster, where the biggest time overhead was actually using the syncing script. Once the servers had the latest versions, running a simulation was just as easy as starting a local test case. Good tools like this really help shorten development time in the long run, but as

was the case with the GUI, they require some up-front investment of time and effort to create them. For this, we would like to thank Jimmy Cleuren, who developed the basic versions of these scripts for his own research project on ALVIC-NG [64], to share his experiences and allow us to build upon and expand the functionality.

6.6.2 Result data gathering

As discussed, we had an extensive tool suite to deploy the experiments to the server cluster. However, those experiments would be useless without output data from which we can extract objective results with which we can answer our research questions. Once more, we leave an in-depth discussion of the contents for the next chapter, but here we discuss the ways we might obtain the needed data and how we chose to do it.
As we will see, the main things we wanted to research were bandwidth and CPU usage. The thesis focusses on bandwidth scalability, so that speaks for itself. The CPU data is important because we can hypothesize that the AOI implementation has a severe impact on CPU usage and thus also on the amount of concurrent clients a single proxy server can support, which is an important parameter in the viability of the ALVIC-NG system for real-world deployment.

Bandwidth

There are plenty of options for measuring bandwidth. One of the most versatile and commonplace options is to use the linux utility tcpdump, which allows you to make a filecapture of all packets sent over the network. This gives a very thorough set of results. However, tcpdump works at the lowest level, only dealing with raw packets. For our experiments, we are only interested in the positiondata ALVIC-NG uses, not all the other data that is being sent (for example control data or communications for zoning). Using tcpdump, it would be more difficult to eliminate these extra packets. In addition, it would also be more difficult and cumbersome to determine how much bandwidth had been used by every client individually. There are also other properties of tcpdump that make it less suited for our purposes. In general, it is too low-level and it provides too little context for the more high-level analysis we want to perform.
This is why we decided to go with our own implementation, embedded into the ALVIC-NG proxy server. We already had to do bandwidth measurement to be able to provide the bandwidth shaping techniques, so it was a small step to write those measurements to an output file. This allows us to capture only the data we are manipulating (removing any noise) and this per user. It also allows us to easily write extra metadata, such as the moment when a certain change in the game logic happens, so that analysis of the files becomes easier afterwards. Our implementation is a lot less fine-grained than tcpdump, only writing the average bandwidth consumption per second. However, as our bandwidth shaping also only

works every second, this loss of detail does not cause us to loose significant data points and actually makes the data easier to interpret afterwards by focussing on the overall fluctuations instead of the tiny details.
The proxy server writes a single log file per user, outputting a tuple of (timestamp, number of objects in AOIs, packets sent, bandwidth used) every second.

Listing 6.2: Part of a bandwidth log file

```
time,timems,objects,packets,bw
...
0:42:48,42048,5,100,35100
0:43:63,43063,5,100,35100
0:44:3,44003,5,95,33145
0:45:5,45005,5,98,34398
0:46:19,46019,5,97,34247
0:47:18,47018,3,100,24276
0:48:25,48025,1,100,7232
0:49:31,49031,1,100,3020
0:50:35,50035,1,100,3020
...
```

CPU usage

To obtain a measurement of the CPU usage there are many options as well. Here we could also use a standard linux command, "top" for example. This gives us an overview of all kinds of statistics for each running process. This output would require extra parsing however. Another approach would be to measure the CPU usage in the process itself through a C++ API. However, it is difficult to find a good cross-platform solution to this.
A more versatile option would be to use the Simple Network Management Protocol (SNMP) protocol. SNMP is an extensive toolsuite to obtain runtime information on a huge number of parameters of machines running in a network. SNMP can for example also be used to capture bandwidth usage. For our experiments, SNMP was already installed on the virtual machines (through previous work performed by Jimmy Cleuren [64]) so we tried to re-use this option. The SNMP data that is sent out by the servers can be received on a local computer by the program Cacti (http://www.cacti.net/). This is a very extensive browser-based tool to parse and more importantly, visualize various SNMP data streams. However, because of the way SNMP and cacti work, the smallest time interval for data capture is 1 minute, in which the measurements are averaged. As most of our experiments only last for a couple of minutes, this is not fine grained enough to extract our needed performance information from.

We finally chose a less direct way of observing server performance. This can be understood if we ask ourselves what the direct impact on a client is when the server CPU usage exceeds its capabilities. Certain operations will be postponed to the next ticks and eventually to the next second(s), which means packet processing times go up as packets stay in buffers for longer periods of time. This in turn means that the so-called round-trip time for packets will rise if the CPU usage rises (i.e. the user experiences lag). A simple approach to get an indication of CPU usage and the impact it has on a client's experience is then to capture the round-trip times for all packets a client sends. This allows us to test the maximum number of clients a proxy server can handle in a very contextual and high level manner: put an upper limit on the round-trip time and slowly increase the number of connected clients until we hit that limit. If we then perform an experiment with filtering and after that without, it is easy to check how the filtering impacts the round-trip times and the "practical" CPU usage and how it impacts the maximum number of clients connected. Note that this approach is only possible because the logic server just acts as an echo server instead of creating new packets (as discussed in more detail in a previous section). This way, we can just check on the sequence numbers, which is quite easy. Should the logic server be changed, it might be more difficult to measure exact round-trip times per packet.
The client process writes a log file per user, outputting a tuple of (timestamp, packet sequence number, round trip time) for every packet received.

Listing 6.3: Part of a round trip time log file

```
time,timems,seqnr,roundtrip
...
00:00:10.302,10302,198,25
00:00:11.314,11314,219,18
00:00:12.316,12316,238,27
00:00:13.325,13325,258,20
00:00:14.356,14356,278,30
00:00:15.362,15362,299,19
00:00:16.457,16457,320,23
00:00:17.475,17475,341,16
...
```

6.6.3 Result visualization and analysis

In the previous paragraphs we discussed how and why we obtain certain data output from our experiments. But it is difficult to draw good conclusions from raw, textual data. As was the case with the GUI that helps to better understand the world state, we also need a graphical

representation for our results analysis.
The standard way of doing this is to generate graphs using Microsoft Excel or a similar spreadsheet application that allows for importation of logfiles and can make various kinds of visualizations from it (barcharts, piecharts, linecharts, etc.). This works well if all the data is contained within the same file and if the graph is not too complicated. For our data however, we had a single file per client, which often meant having over 100 files per simulation. It would be possible to write a spreadsheet macro to automatically import these files and format them in the tables, but this has been proven quite difficult and not very powerful or flexible. In addition, we would like the ability to generate quite complicated graphs, for example overlaying multiple datasets to see how they differ, and this in a dynamic manner.
Because we had a lot more experience with PHP and other web technologies such as JavaScript and HTML5, we decided to move away from the spreadsheet idea and implement our own visualization suite. We use a couple of simple PHP scripts combined with the RGraph JavaScript library (http://www.rgraph.net) to automatically read a complete directory of files, convert the data from .csv to json format, and generate various types of graphs with it, as can be seen in chapter 7. A simple HTML interface, as seen in figure 6.8, allows us to filter on specific clients or to aggregate the results to an average. It also allows us to overlay different kinds of data: for example we can plot the average bandwidth usage together with the number of packets sent, to clearly see the correlations between those parameters.
These tools were easy and quick to implement and offer options that are completely tuned to the data that we are trying to analyze. Trying to obtain the same output images with a spreadsheet would have been possible, but much more difficult and time-consuming. As demonstrated before in this chapter, sometimes it is good to invest some time and effort in the creation of specific tools to further benefit the project down the road.

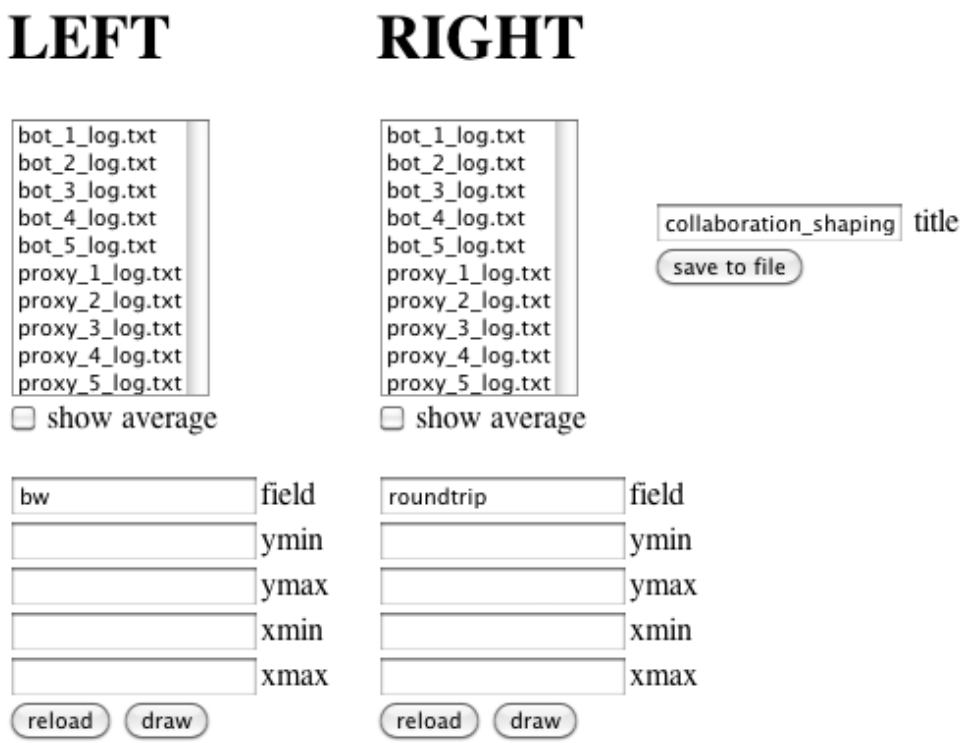

Figure 6.8: The visualization tool allows us to overlay two different datasets and give them their own vertical axis, limit their range and show their averages.

Chapter 7

Results

In this chapter we discuss our results of the implementation and the various experiments we used to assess how well it aids us in limiting bandwidth consumption in networked virtual environments.
The experiments and simulations were designed specifically to help answer a set of research questions that adhere to common problems or pitfalls when using bandwidth limiting techniques such as Area of Interest. These research questions are such:

- How versatile is the technique?
  As discussed in section 5.3, not every bandwidth shaping technique is appropriate for every game type or virtual environment. We chose Area of Interest because we theorized it was still very usable in many situations. It is important to see how the technique reacts in situations where it is traditionally less used and if it is still usable in these situations and to which degree.

  We also have to demonstrate the flexibility of the implementation through the experiments. This holds for instance for the AOI-API in situations where we have multiple objects per player or multiple Areas of Interest per player. For the NIProxy, this means we need to discuss how the bandwidth shaping trees can be built to accommodate a particular situation that was not originally considered in the NIProxy design.

- How effective is the technique in limiting bandwidth?
  There are two sides to this question. Firstly, we can ask how big the bandwidth gains are in a particular situation, i.e. how much difference does it make for the bandwidth usage if we enable or disable bandwidth shaping. This can be quite easily tested by performing the exact same simulation twice, once with and once without filtering enabled.

  The second side is more complicated, as we can also ask how well the technique can be used to enforce a particular bandwidth limit. In many cases, the maximum bandwidth usage is determined by an external factor, for example the speed of the connection between server and client. If the bandwidth shaping technique fails to stay under this

bandwidth limit, it is in fact little better than if we would use no filtering at all. This means the implementation needs to not only be able to make sure the limit is respected, but also that it needs to be able to deal with situations where this limit suddenly changes.

- How well does the technique retain consistency?
  When using a technique like Area of Interest, it is not very difficult to adjust bandwidth usage, for instance by shrinking a circle-shaped AOI so less objects are in it. However, the real question is then: how does this affect the overall consistency of the world state as seen by the user? For example, if we shrink the circle too much, certain objects that we would expect to see will disappear, which can lead to deteriorating gameplay or loss of interaction possibilities. This means we should not just look if the technique can adhere to a bandwidth limit, but also how this affects the user's view on the world and if this view is at all times sufficient to allow expected interactions with the world.

  Testing this consistency is a lot less straightforward, as it depends on the type of game or environment. As such, we will approach this question in a less empirical manner, but instead try to logically assess the impacts on player interaction and consistency.

- How does the technique affect the performance of the proxy servers?
  One of the most important aspects of the ALVIC-NG architecture is the usage of intermediate proxy servers to lower the load in the logic servers. However, this only works well if the proxy servers are able to serve a lot of clients simultaneously. As the amount of clients a proxy can serve is strongly tied to its CPU usage, it is important to look at the impact of the bandwidth shaping techniques on this performance and how much it limits the number of players a proxy can handle.

  In addition, the NIProxy had not before been used on a large scale and its techniques had not yet been tested for computational performance. It is interesting to research how the bandwidth shaping trees hold up in a fast-paced, large-scale environment and how they affect performance.

Many of the experiments and simulations test multiple aspects of these research questions. As such, when we discuss the experiments, we will indicate which research aspects were involved and the conclusions we can draw based on that experiment. In the final section of this chapter, we will re-visit the questions and summarize the individual results to derive a global conclusion for every question.

The following sections describe seven different experiments. Experiments one through three focus specifically on the versatility of the AOI-API and show how it can be used in diverse situations and how the technique can be coupled to logical parameters of the virtual environment. Experiments four and five deal specifically with the integration of the NIProxy, how we

can build effective bandwidth shaping trees and how we can use heuristics to predict bandwidth usage for a given configuration. Finally, experiments six and seven are experiments on a larger scale than the others, looking at performance and bandwidth usage in a more fast-paced environment.

In many of the experiments we will simulate a drop or rise in the available bandwidth for the NVE, causing our traffic shaping algorithms to perform the necessary adjustments to for instance the AOI shapes. In real scenarios, these fluctuations typically arise when switching to a different type of network (from cable to wifi or from wifi to a mobile internet connection etc.) They can also occur when a router or link in the network goes down, causing the traffic to follow a different, possibly faster or slower link. For some internet technologies, the available bandwidth is also influenced by what other users on the network are doing. If a fellow user suddenly starts streaming a high-definition video stream, this can possibly lower the available bandwidth for the user connected to the NVE.
In our experiments, the available bandwidth is artificially controlled and not directly tied to actual network conditions at the time of the simulation. This allows us to always run the experiments in the same setup and also to look at best and worst cases.

Please note that in all the graphs there are sometimes tiny bumps or valleys in what should be a constant horizontal line. This is due to the fact that we use our own bandwidth measuring mechanism and sometimes it measures one packet too much or too little during a given measuring interval. As such, these very small fluctuations can and should be disregarded and treated as if there is a constant bandwidth usage.
Secondly, also note that many of the bandwidth values might seem quite low or unrealistic for an entire NVE. We keep the number of players and the packet sizes artificially small to prevent large fluctuations in the bandwidth due to many users entering/exciting the AOIs at the same time, which allows us to better evaluate our implemented techniques.
Finally, please note that, unless otherwise specified, all graphs have bandwidth in bytes/sec on their left y-axis and time in milliseconds on the x-axis. Bandwidth graphs always show the outgoing bandwidth, from the proxy server viewpoint, so in the direction of the connected client(s). In other words, it shows how much bytes/sec were sent to a user.

7.1 Experiment 1 : First Person Shooter

This experiment was designed to showcase most of the different options of the AOI-API and especially how different shapes and InterestDefinitions can be used in the same game or environment for different purposes. In addition, the First Person Shooter game genre is traditionally hard to bring to a large-scale setting because of the fast-paced action and high update rates (see chapter 2). As we hypothesize that a good bandwidth limiting scheme

can help bring this type of game closer to being do-able in a large scale environment, it is important to test the AOI-API in an FPS scenario and see how it can relate to the FPS concepts.
The most important concept here is that we can tie the radius of circle- or wedge-shaped areas to the effective range of the gun the player is using. If the gun is only accurate to about 50 meters, it is not necessary to receive high-quality information from outside those 50 meters. Similarly, if the accuracy drops with distance (which is a realistic scenario), the more distant entities can be sent at a slightly lower Level of Detail. This logic is the rationale behind the AOI-API exploitation in this experiment and is used to shape the different areas for different player types and guns.

7.1.1 Description and setup

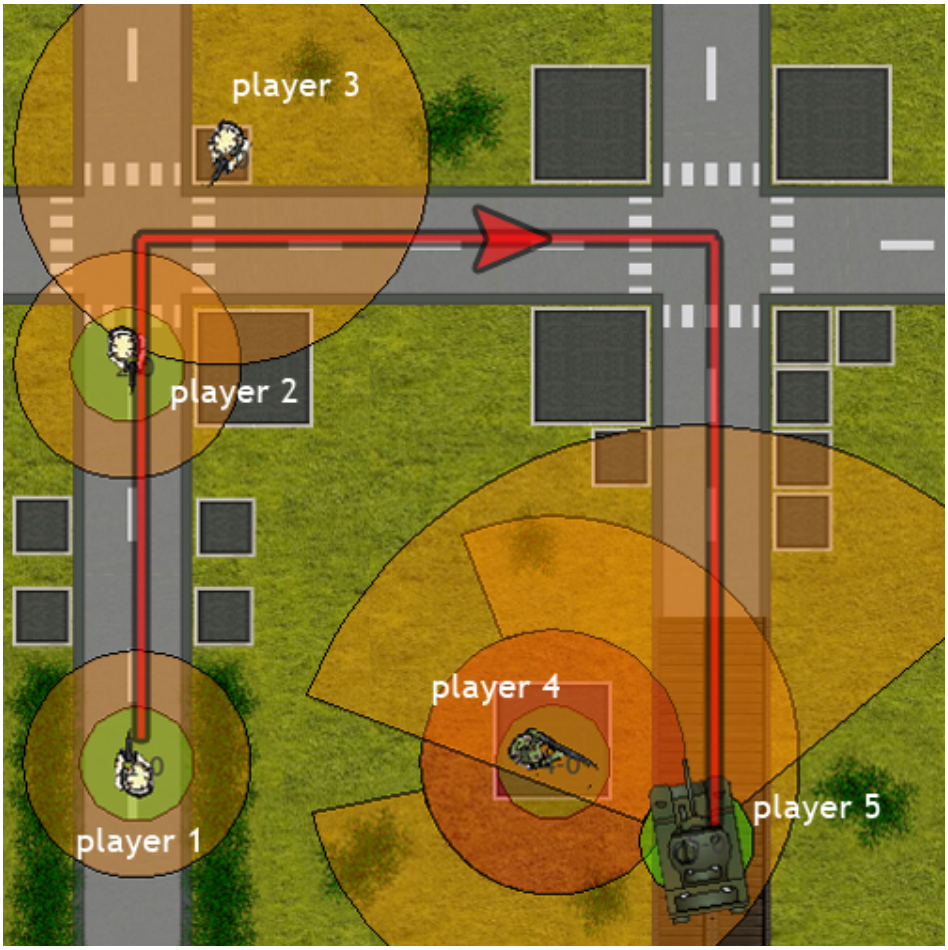

Figure 7.1: The start of the FPS scenario. The red line shows player 1's movement in the world.

The scenario consists of five independent game entities, each having their own game rules and

movement. Entity 1 (a soldier) will move across the map, passing in order entity 2, 3 (also soldiers) and 5 (a tank) before finishing next to entity 4 (a sniper). Entity 2 will move to the south of the map, while entity 5 moves in a northern direction. These entities and their starting AOIs can be seen in figure 7.1.
Entity 1 and 2 are normal FPS players that run around the game with their guns ready. These guns are very accurate close by, but a little less accurate further away. This is represented through two circle-shaped areas, centered around the user, where the smaller circle has a higher Level of Detail. As the two soldiers come closer and become visible to each other, they will switch to a so-called "iron sight" mode of the gun. This means they look down the gun, which increases their range and accuracy. This is modeled by changing the radii of the circles to make them larger when the players enable iron sight aiming. Player 1 shoots and kills player 2, and remains in iron-sight mode as he progresses because he thinks other enemies might be near. This AOI change (and the changes we will discuss next) can be seen in figure 7.2.

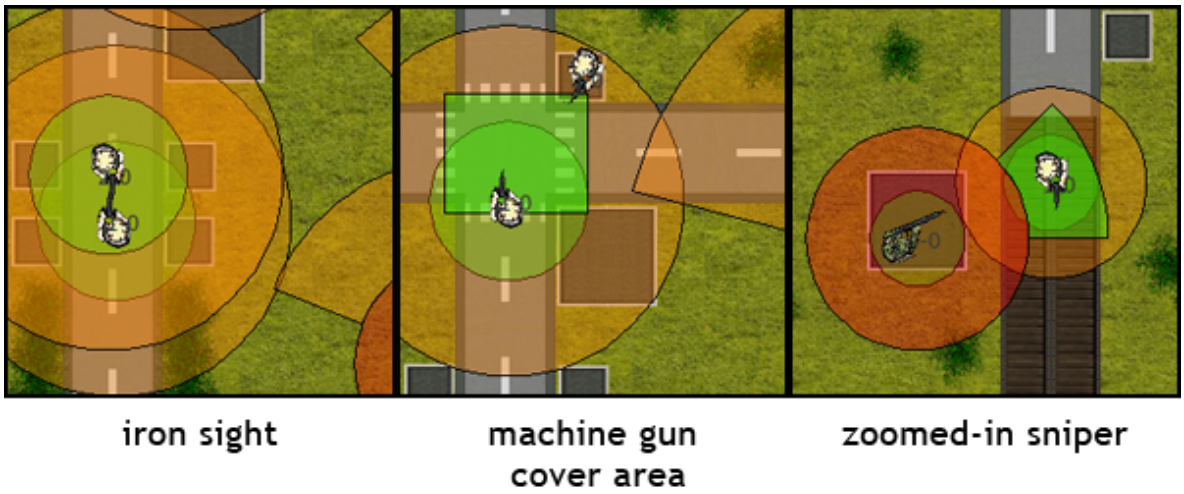

Figure 7.2: The various AOIs for the different stages in the simulation.

Player 3 is a soldier sitting in a small building behind a mounted machine gun. This is a very powerful gun with a limited range of movement and limited shooting range. At first, player 3 is on the lookout using his binoculars (high range but medium LOD). As soon as he sees player 1 coming close, he switches to his machine gun, causing him to change his AOI. Instead of a circle, he now has a rectangular shape on the road before his building (indicating the limited range of movement of the gun) but with a high LOD (note that we could have also used a wedge-shaped AOI for this purpose). When player 1 enters the rectangle, he notices player 3 and they start shooting at each other. Player 1 wins and decides it is safe, so he switches back to his normal state, leaving the iron sight mode.

At the same time, player 5 (the tank) is moving in northern direction, seen from a distance by stationary player 4, the sniper. A tank has a big cannon on top, which has a good range and medium accuracy. A tank is also very slow to turn. This means we can use a wedge shape (with a far away outer radius) to represent his AOI, as he cannot instantly turn to see whatever is behind him (as is possible for the soldiers). The sniper on the other hand has a long-range weapon that can be zoomed in. This means we want him to receive information from his full zoom-range, which is quite a lot larger than the range of the other players. This might lead to a sniper consuming substantially more bandwidth. In order to tackle this, we use a big circle with medium LOD for when he is not zoomed in. Outside this circle, we also have a wedge shape that indicates his optimal range when zoomed in. This wedge is of a low LOD. This allows the sniper to roughly pick up on possible targets, even when not zoomed in.
When player 1 enters this wedge, the sniper notices him and zooms in. This causes his circular AOI to become of a lower LOD and the wedge will become of high LOD, but shrink to cover only 45 degrees, centered on player 1's position. This should make sure the sniper continues to use approximately the same amount of bandwidth as he did when not zoomed in, while still getting enough information on entities that might be between his zoomed in range and his current position. The sniper shoots and kills player 1 and thus ends the experiment.

In this scenario, there is also a link between player 1 and player 5. Player 1 has an ObjectAreaDefinition for audio on player 5. As discussed in chapter 6, this means that he will receive all audio from player 5, even if the latter is currently not in one of player 1's other interest areas. This allows us to simulate that these two players are on the same team in the game, and that player 5 is the commander who gives orders to his troops through audio communication (in other games, it could also be used for normal voice chat between friends). As an extra, we disable this link between the players when Player 1 goes into iron sight mode. This will ensure he is not distracted by the audio while fighting and is also because the iron sight might use more bandwidth than the normal mode, so we prevent sudden big spikes in the bandwidth by disabling the audio link temporarily.

7.1.2 Results

With this scenario, we ran two different simulations. The difference between them is that in the second simulation, we disabled the audio link from player 1 to player 5 completely for the entire simulation, not just during the iron sight part. This helps us show that, depending on the exact setup, the bandwidth usage will be very different and will require different approaches to bandwidth shaping.
Please note that these simulations did not use the NIProxy to perform the AOI adaptations. Most of the simulation is scripted so that we force a certain AOI to be chosen. Also note that

the simulated audio packets are of the same size as the normal high LOD position packets. Every entity sends about 10 updates per second.

Simulation 1 : Remote audio enabled

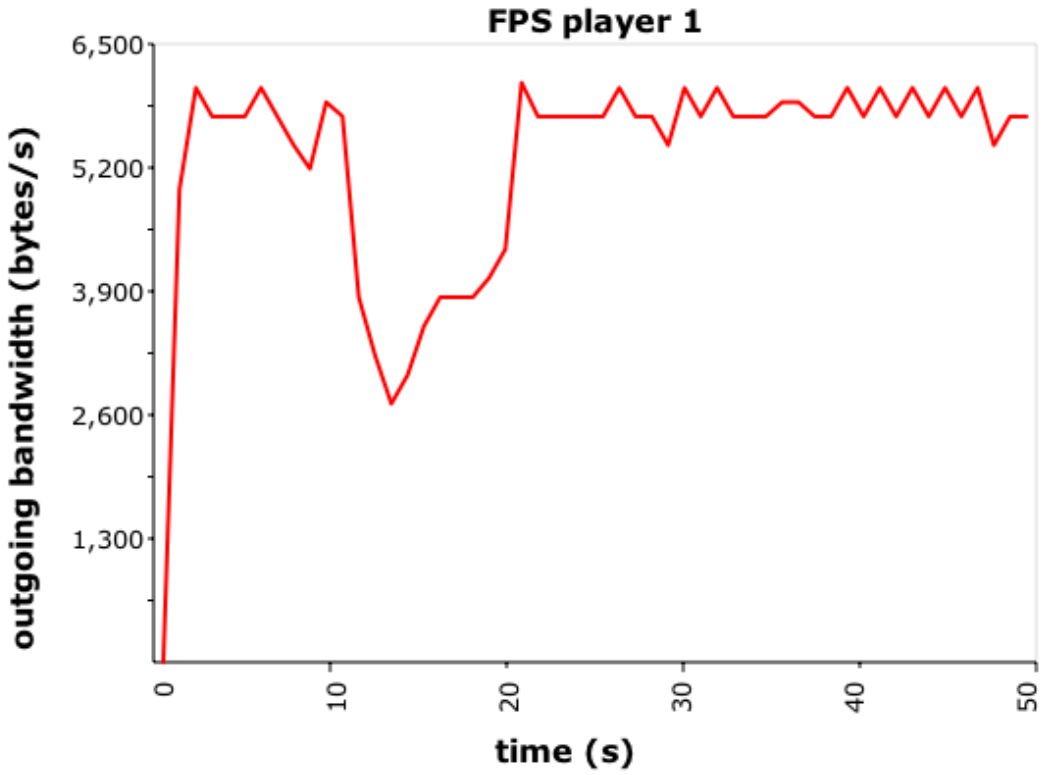

Figure 7.3: FPS experiment: Bandwidth usage of player 1

In figure 7.3, we can clearly see that player 1 sees player 2 around 8 seconds into the simulation. At this point, the audio connection to player 5 stops. At first, player 2 is in the outer circle AOI, and the total bandwidth usage drops slightly. Then we have a spike when player 2 enters the high LOD inner circle. After this, around 15 seconds, player 1 moves past player 2, causing the bandwidth to drop considerably as there are no other entities in the AOI, but player 1 is still in iron sight mode, which means the high BW usage of the audio communication is absent.
Around 16 seconds, player 1 and player 3 see each other, causing player 1's bandwidth usage to rise slightly (as player 3 is only in the outer, lower LOD). Player 1 moves past player 3 and disables his iron sight at 21 seconds. At this point, the audio communication sets in, and the bandwidth usage goes back to the same level as at the beginning of the simulation.

Figure 7.4 shows the bandwidth usage of player 4 (top line) and player 3 (bottom line). For

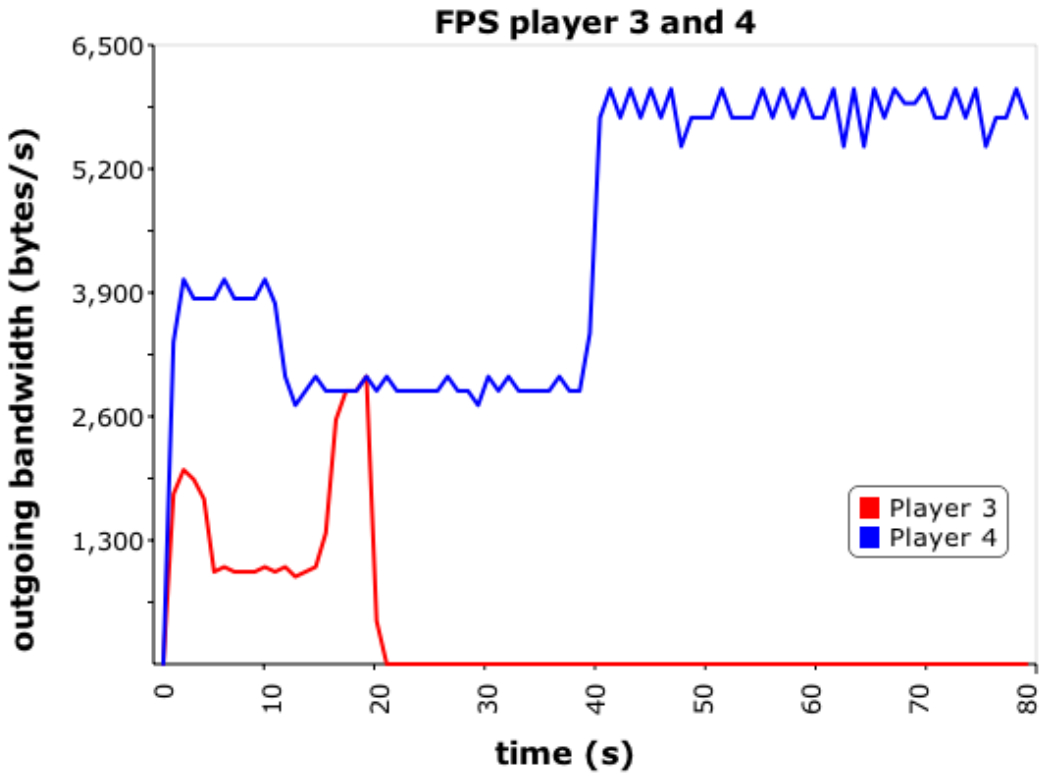

Figure 7.4: FPS experiment: Bandwidth usage of players 3 and 4

player 3, we can clearly see that he watches player 2 walk by at the beginning of the simulation (the big spike). After this, he is the only one in his AOI for a while, leading to a constant bandwidth usage as he only receives his own position info. At the 16 second mark, player 1 becomes visible and player 3 switches to the rectangular AOI with high LOD. Note that player 3 himself is not in this high LOD area. This causes his bandwidth usage to increase significantly. When he dies, the bandwidth usage drops to zero for the remainder of the simulation.

For player 4, we see the same spike at the beginning, but for a different reason. For player 4, this means he sees player 5 (the tank) driving by. The tank is in his wedge-shaped AOI with medium LOD because the sniper is not zoomed in. The bandwidth usage then becomes more stable as he is the only one in his AOI for a while. Notice that his idle bandwidth usage is a lot higher than that of player 3. This is because his inner circle AOI is of the highest LOD, where player 3 only has medium LOD close to himself. Near the end of the simulation, player 1 enters the AOI of player 4, who immediately zooms in. This causes his wedge AOI to switch to the highest LOD. We can see that this uses a lot more bandwidth than in the beginning, where the tank was in the same wedge, but the sniper was not zoomed in.

Simulation 2 : Remote audio disabled

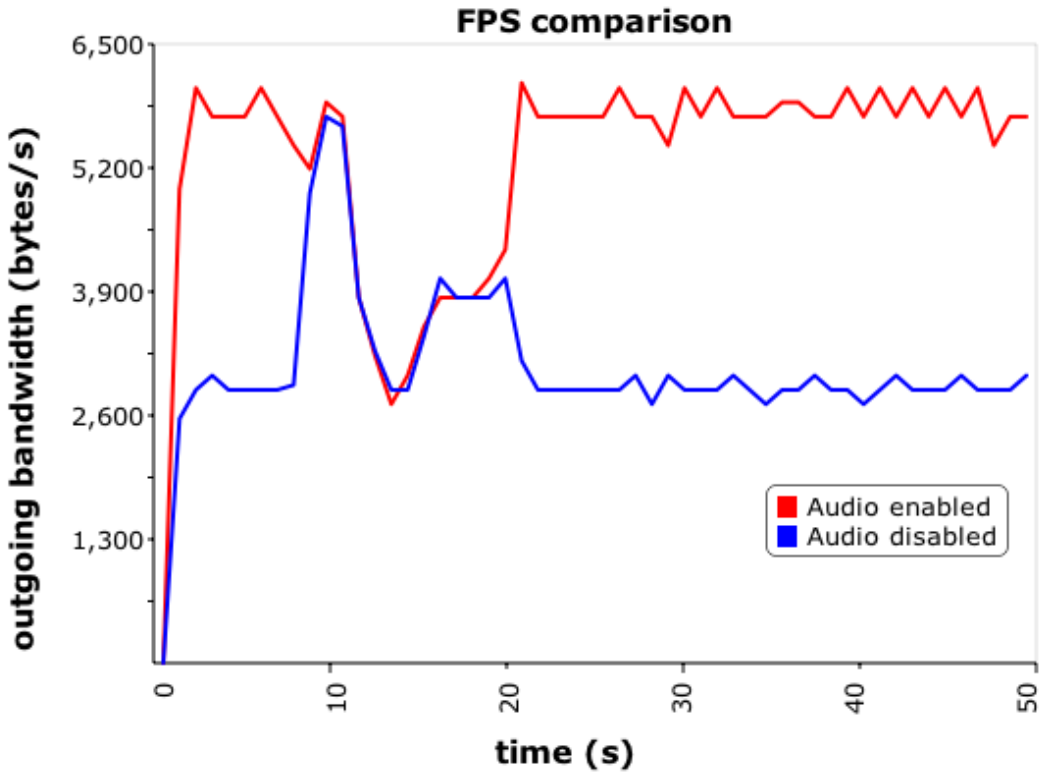

Figure 7.5: FPS experiment: Comparison of bandwidth usage for player 1 with audio enabled or disabled.

Figure 7.5 shows the results from both experiments in one graph. The top line is the same as in figure 7.3, the bottom line is the bandwidth usage when we have no remote audio coupling to player 5 at all. We see that, through this single adjustment, we get a radically different bandwidth usage pattern. Only the parts where player 1 is aiming down the iron sight are clearly the same. However, in the top graph, this constitutes a period of below-normal bandwidth usage (overall speaking), whereas in the bottom graph they represent (big) spikes in the bandwidth consumption. This is a big conceptual difference between the two situations.

7.1.3 Discussion

The simulation and results clearly show the versatility of the AOI-API. Many of the typical FPS situations (iron sight, sniper zoom, machine gun ambush) and player styles (soldier, sniper, tank) can be accurately represented by the appropriate use of areas of interest.
However, we can also that see these representations are maybe not entirely sufficient to represent an FPS scenario. For instance, player 1 only sees player 2 when he is in his AOI. However, they are both on a single street without obstructions. Logically, they should be

able to see each other all the time, instead of suddenly popping into each others view halfway on the street. This could be solved by making a rectangular AOI with low LOD on the street or by using a long-range wedge shape that follows the player's viewing direction. The first option is very dependent on the world/level and would be difficult to generalize. The second option can make for some sudden bandwidth spikes if the user would do a rapid movement (for instance if he suddenly looks behind him). This means both options are not ideal and that, even though the AOI-API can be used to represent many real-life situations for an FPS, in this experiment it is not entirely sufficient to provide the full play experience we would get with filtering disabled.

The experiment also shows that by dynamically adjusting these AOI definitions, we can have a measure of control over the bandwidth usage by using very logical and real-world applicable parameters. For instance, using the more intense iron sight AOI while disabling remote audio, allows us to make sure the bandwidth usage does not increase too suddenly and is also strongly tied to game mechanics and meaning. The same goes for the small high LOD wedge for the zoomed-in sniper who loses some LOD in his more immediate surroundings.
However, it is difficult to perfectly control the amount of bandwidth used here. While we can be relatively certain that a specific configuration will use more (or less) bandwidth than another one, this is not true in 100% of the cases (depending on world state, for example a lot of entities suddenly come very close) and there is also no clear way of knowing how much less or more bandwidth will be consumed after the switch. This can be solved by using a heuristic in combination with the NIProxy, which we discuss in section 7.4.

Lastly, our second run of the simulation (with the audio disabled) makes it clear that the actual bandwidth consumption patterns depend heavily on exact situations in the game world, even though both simulations are for an FPS game. This will affect the way algorithms for bandwidth shaping and bandwidth limiting work and how they should be configured. With audio enabled, the algorithm might need to redistribute the newly available bandwidth to other players. With audio disabled however, the algorithm needs to account for the sudden spike. This means the algorithm needs to be very robust and capable of dealing with these various situations. In section 7.4 and 7.5 we discuss our algorithm and see how robust it is.

7.2 Experiment 2 : Collaborative Environment

This experiment also shows the versatility of the AOI-API by using it in a more serious, collaborative environment. Here, we focus more on user-location independent AOIs and the effects of moving to a completely different situation or location in the game world and how it reflects on the behaviour of AOIs. A collaborative environment is notable for its requirement of highly detailed interactivity (and thus bandwidth usage), but typically all this activity is

within smaller groups.
The scenario is inspired by earlier experiments by PhD student Anastasiia Beznosyk. In her simulation, users would work together in different smaller groups and would be confined to a single building in the world. Users in the same building could interact directly through audio and video, while interaction with users in other buildings was limited to text chat. If one group had finished their task early, they could move to the other building to help the second team.

The scenario is built to reflect this type of environment, where one can intensely cooperate with local users, but still interact in some way with more remote users. Thus, the AOI definitions allow for a more fine-grained definition of interest, where it would even be possible to let the users themselves decide which types of data they are interested in from which sources.

7.2.1 Description and setup

In this scenario, we have 9 different users, each in a single building or room in the world. Users receive video from all users in their building or room, while they receive text messages from other buildings if they have an interest in them. In figure 7.6, the AOIs for player 1 and player 8 are superimposed. Player one only has interest in the two top buildings, while player 8 only has shapes defined in the bottom building. The green areas indicate a high LOD and also the receiving of video, while orange areas indicate text and medium LOD.

Where in the previous experiment 7.1 the AOI transitions were completely scripted and not based on game logic, in this simulation the AOI adjustments were enforced dynamically and automatically depending on which room the player was in. This means we can also directly take control of one of the users should we want to and move him around any way without having to worry about the AOIs changing in the correct way, which was not the case in the previous experiment.

7.2.2 Results

In figure 7.7 we can clearly see the different steps in the simulation. After 10 seconds in the top left room, user 1 starts moving out of his building, which he exits a couple of seconds later. At this point, he transfers to a completely new circle-shaped AOI and disables his previous AOIs. This circle shape is of high LOD and allows the player to receive text messages from players it overlaps. This allows him to see where people might need his help or where activity is going on. As he is no longer receiving video, the bandwidth usage goes down considerably. Then, user 1 goes into the top right building at approximately 27 seconds. He starts receiving video from the other users there and receives text from the top left building from which he

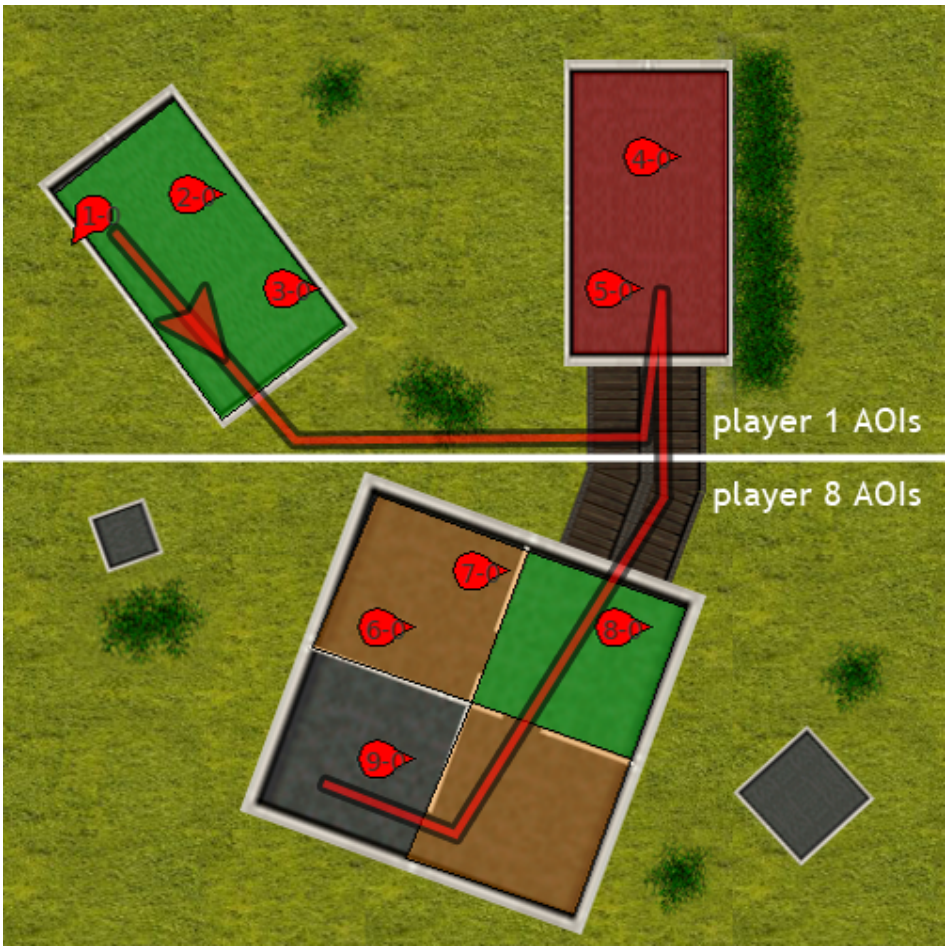

Figure 7.6: The start of the collaboration scenario. The red line shows player 1's movement in the world.

came. As both buildings contain the same number of users, the bandwidth consumption is equal to what it was at the beginning.

35 seconds into the simulation, user 1 starts moving to head for the bottom building. He once again transfers to the circle-shaped AOI on the outside. When he enters the bottom building, he adopts the AOI scheme for this building, which is: video for your current room and text for a neighboring room if there is a door. He stays in the top right room for a while with player 8 (spike in the bandwidth) before proceeding to the empty bottom right room. After being there for a while, at around 52 seconds he finally moves to the bottom left room, where he starts getting video from player 9, causing his bandwidth to go up again. Note that his bandwidth usage is slightly less than when he was in the top right room. This is because then he had text from players 6 and 7, which he does not have now as there is no door between the bottom left and top left room.

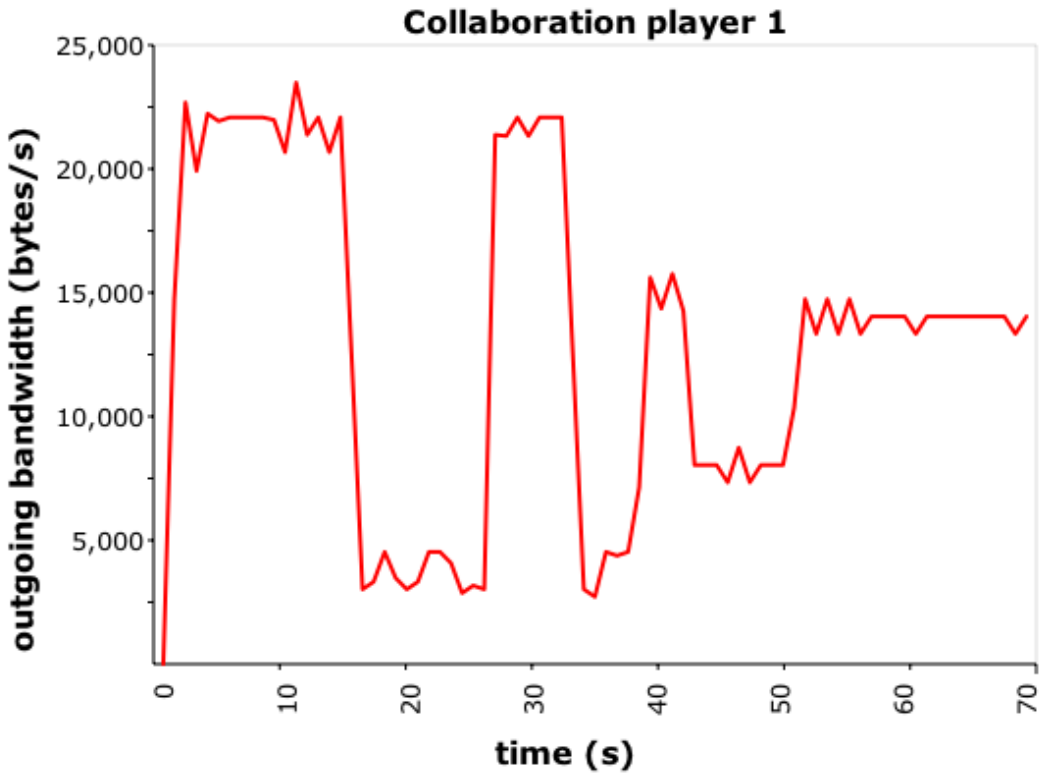

Figure 7.7: Collaborative experiment: Bandwidth usage player 1

7.2.3 Discussion

The simulation shows how the AOI-API can be used for situations that are quite different from the normal usage forms, i.e. centered on the player. It also shows once again that the AOI definitions can be tied to logical elements in the virtual environment, in this case indicating which group we are most interested in working together with.
Next to this, it also shows how AOIs can change radically and suddenly when the situation changes. When the user leaves a building, the AOI shape changes and the type of data we receive is different. This is effectively an example of how the world state can change very quickly and that the AOI-API can handle such changes.

In contrast to the previous experiment 7.1, this simulation does have optimal consistency and good world state to allow the player to perform all necessary operations and interactions. This primarily follows from the fact that most interactions are contained within well defined borders (the buildings/rooms). However, we can say that this is a realistic setting for a collaborative environment, and as such our implementation can be used for collaborative environments while retaining maximum consistency, which was not always the case for the FPS use case.

What is similar to the previous experiment however, is the fact that we can know certain AOIs will use more/less bandwidth than others, but not exactly how much this difference will be. Experiment 7.5 adds considerably to this scenario by incorporating actual traffic shaping and considering changing bandwidth limits, helping to resolve this problem.

7.3 Experiment 3 : Real Time Strategy

This third experiment is the last one that was designed specifically to show the versatility of the AOI-API in different situations. The Real Time Strategy genre is special in the way it does multiplayer across networks. Instead of sending position updates for every entity, the game sends only the user input. This is combined with a fully deterministic simulation where it is made sure that the user inputs are processed at the exact same game time across the distributed players. This guarantees full consistency while drastically reducing the bandwidth usage. However, this method also brings some limitations. For example, it is very difficult to allow users to join after the game has started and if the deterministic simulation diverges even a little bit, the consistency can be broken without a way to repair it. This al is discussed in more detail in chapter 2.

It would be interesting to try a different approach to MMORTS games by dropping this method and trying to use the same method we used for the FPS and Collaborative experiments: sending position updates for every entity, several times per second. Traditionally, this has been very difficult because of the enormous amount of individual units in an RTS game. If we would just send the position updates without any filtering, even a good zoning technique would not be able to sufficiently limit the bandwidth consumption.
This means the RTS scenario is a good research subject for the AOI-API to see if our implementation could help make this alternative method for online RTS games viable. Sadly, due to time constraints in the project, this scenario was never fully finished and not tested, even though the implementation contains most of the necessary building blocks. Since it is such an interesting use case, we discuss our views on it here, even though it was not tested.

7.3.1 Description and setup

Our setup for the simulation featured 2 players, each controlling around 20 individual units on the battlefield. Each of these units would be part of a specific group (i.e. each player would have 2 groups of units, 1 of 15 units, 1 of 5 units). These groups would have a single circular AOI around them, using the group-based AOI feature of the AOI-API (see section 6.1). This allows us to reduce the computation times by only having to check a shape once for an entire group of units. This setup resonates with the common concept of “fog of war” for RTS games. This fog of war means that most of the game map is invisible to the player. Only in the direct environment of his units can he see other players’ units moving. This

allows for tactical maneuvers in the game and encourages exploration. By tying the AOI definition to this fog of war concept, we once again get a very logical resonance with the game mechanics for the AOI, which will in turn result in good consistency and for the preservation of interaction possibilities.
These unit groups would engage each other in the scenario, where instead of checking if a unit is in a particular AOI, we can check if two of the group AOIs intersect. This once again cuts down on computation and makes sure that if we see one unit of a group, we immediately see all the units. Depending on the game, this can be wanted or unwanted behaviour, but for our scenario this was the best route.
A side-effect of working with groups of units would be that we can use a form of delta compression to cut down on the size of the position updates (for more information on delta compression, see section 3.1.3). Instead of sending every position update in full world-coordinates, we can send them relative to the group's center point or to another unit in the group. This can help cut down the packet sizes and incurs a form of aggregation (section 3.2) if we send all the position updates for a single group in a single packet, instead of a packet per object.

To see how we could dynamically adjust bandwidth usage in the case of changing bandwidth capacity, we made use of the current user viewing position. An RTS game takes place on a big map and the user can only focus on one section of the map at a time. Normally, all data for all our units would be sent, even if the player is not observing them on his screen. Should the throughput go down however, we can make smart use of this viewing position and first change the AOIs for the units he is not observing (primarily by switching to a lower LOD). This will help retain high consistency for his current viewing area. As the logic server still receives all the updates from all players, the global game state is consistent at all times, even though the user does not receive full updates of everything. When the user would suddenly shift his viewing position, the proxies are notified and can immediately switch the AOI definitions to accommodate for the change.

Next to this, it would be interesting to have the unit groups form and disband automatically based on the distance between units. As soon as two units are close enough to one another, they would be automatically grouped and their AOIs would merge into a larger one. This is interesting for the traditional way in which RTS games are played, namely that the user does not control each unit individually but rather issues orders to entire groups of units at once. In the same way the user sees the units as a group, the AOI-API sees them as a group, further connecting the AOI to the real game concepts.

Finally, we could use rectangular AOIs to be put around stationary buildings which users can build in their bases. This way, we can simulate the line of sight of these buildings without needing to have units next to them.

7.3.2 Discussion

Even though we did not execute the experiment, we can still hypothesize about its usability in a real-life RTS game setup.

Firstly, we can argue that the consistency is well kept if we tie the AOI definitions to the fog of war / line of sight concept. Any actions happening outside this line of sight are invisible because of the game logic anyway, so not receiving updates about them does not affect the local consistency for the player.
It is more complicated when we have to start dealing with bandwidth limits however. If even the bandwidth savings of the AOIs are not enough, we are going to need to use the viewing position method to bring less relevant AOIs to a lower LOD. This should not have a big impact on the global consistency, as the logic servers still receive all updates. If the user still receives enough important information (for instance: a team in the other corner of the map is under attack), local consistency should also be fine.

The problem comes when the user changes his viewing position. Even though the proxy server can pick up on this relatively quickly, there will probably still be a delay in receiving the more up-to-date information. However, as an RTS is generally not an extremely fast-pace game, this can be dealt with by using appropriate interpolation techniques for the visuals on the user's computer to hide most of this delay. The game state will not be 100% consistent this way, but still enough to provide a normal RTS experience to the player.
In a more extreme approach, we could even use an algorithm to predict user camera movement and build a probability distribution function to try and guess where the player will look next (for instance based on his previous patterns or thinking that he will soon focus on a group under attack etc.). This would allow us to preemptively put some AOIs in a higher LOD in the expectation that the player will focus his attention on them in the near future.
Another important factor could be that the movement of the units in an RTS game are usually quite predictable. Units in a group will oftentimes move in the same direction (if the user has set a waypoint etc.) and we can use additional Dead Reckoning techniques effectively to help reduce bandwidth usage, but also to try and predict which enemy units will soon be in the line of sight and preemptively start preparing for encounters with LOD adjustments.

It is important to notice that this setup will work well in the beginning of the game, where large parts of the world are still covered in fog and we do not have large amounts of units with line of sight. As the game progresses however, larger areas of the map start to become visible at all times (more units, higher line of sight etc.). Thus, we could doubt the scalability of our approach for the later stages of a game match. In this case, the LOD management for the AOIs will have to be even more flexible and keeping good local consistency will become

more difficult. The exact problems that arise can probably only be assessed when running the necessary experiments, but we can hypothesize that our method will need to be supplemented by other algorithms or it will need to be very flexible to be able to handle the later game situations.

Our final remark is that this RTS scenario has a lot of overlap with the problems associated with networked physics simulations. If we would like a physics simulation of many objects to have a real impact on the game state, this means we also need to transmit the positions of all these objects to all players involved. For this reason, most physics simulations in games are just for the visual aspects or quite limited in how many objects can be affected and how. Yet, this is quite similar to an RTS game. They both have large numbers of objects they need to transmit, objects can be grouped together based on physical closeness and most of the movement is quite predictable. While it is a lot more difficult to tie the physics simulations to concepts such as fog of war or viewing position, many of the conclusions drawn from an RTS experiment could be applicable to distributed physics setups as well, which is still an area in full development for which much research is being conducted.

In conclusion, this experiment is certainly one of the most interesting options for future work with the AOI-API to evaluate if the bandwidth usage can indeed be cut down enough to make this method usable for an MMORTS, with a focus on the late game with thousands of units and large areas of the map visible.

7.4 Experiment 4 : Sniper

This scenario was designed to show how the NIProxy can be used to model bandwidth shaping trees and strategies that can help shape bandwidth usage when for example the available bandwidth for a given client changes. The scenario was deliberately kept small and relatively simple to unambiguously evaluate the exact workings of the NIProxy integration.
This is important as one of the main elements is the usage of a bandwidth heuristic method (see section 6.2.2) to try and estimate how much bandwidth a given adjustment to an AOI will consume. To properly evaluate and demonstrate how this method works, a simple test case is better than a more extensive example. Experiment 5 is a lot more complicated and is designed to show how the system works in more extensive situations.

7.4.1 Description and setup

For this scenario, we have only 3 players in the world. Player 1 is a sniper who sits on top of a building in the center of the map. The 2 other players walk around him in circles, each at a different radius from the sniper. The sniper tracks the outer player, which causes him to constantly update his rotation to face in the direction of the outer player. This can be seen

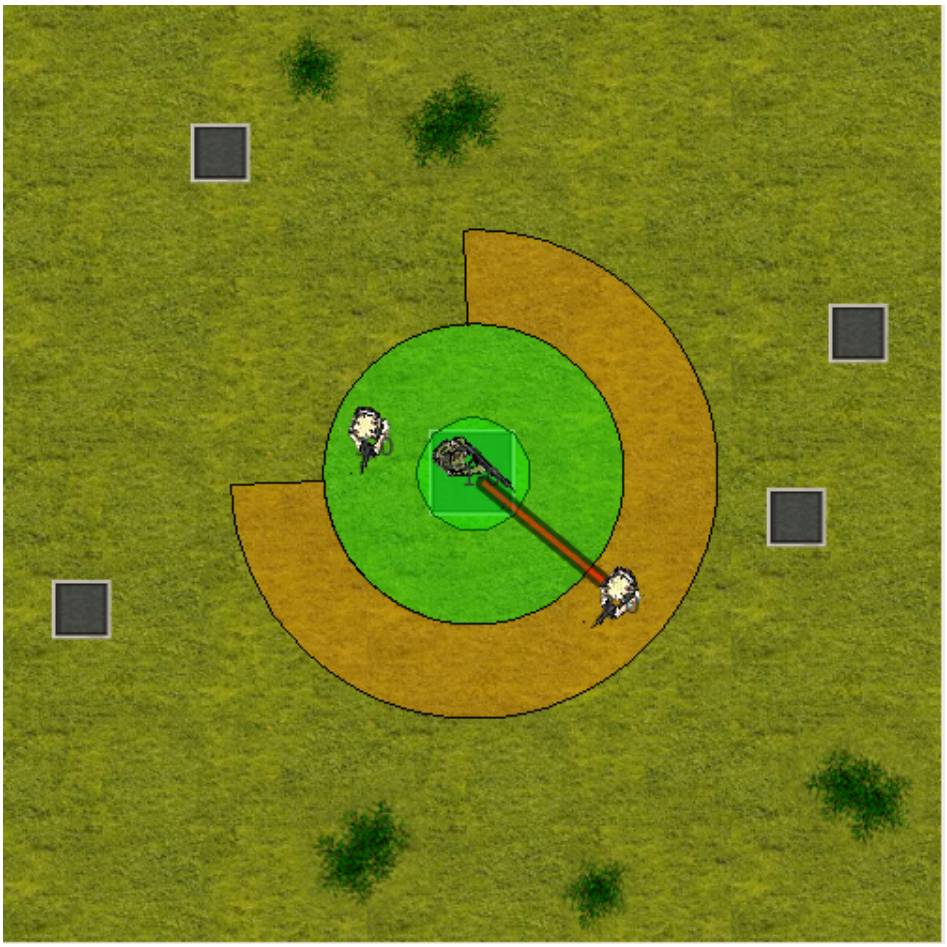

Figure 7.8: The start of the sniper scenario. The red line shows that the sniper is tracking the farthest player.

in figure 7.8. As the wedge shape is tied to the sniper's rotation, this will also move along. This is not a very realistic situation, but it is kept simple to demonstrate the workings of the bandwidth shaping.

At given times, the available bandwidth for player 1 will change. This will prompt the NIProxy to take over and select a new AOI definition that it hopes will make sure the bandwidth limit will not be exceeded. The NIProxy bandwidth shaping tree is shown at the left in figure 7.9. As you can see, this tree is exceedingly simple. The root node is a priority node (although this does not really matter for this scenario) and it only has a single child leaf node, in charge of manipulating a single InterestDefinition.

Note that this tree could have been built quite differently. Most importantly, we could have opted to have multiple InterestDefinitions (and thus child leaf nodes) to represent the various AOIs. In that case, only one InterestDefinition would be active at a given time and switching to a new AOI would be done by enabling the correct child leaf node. This situation is visible

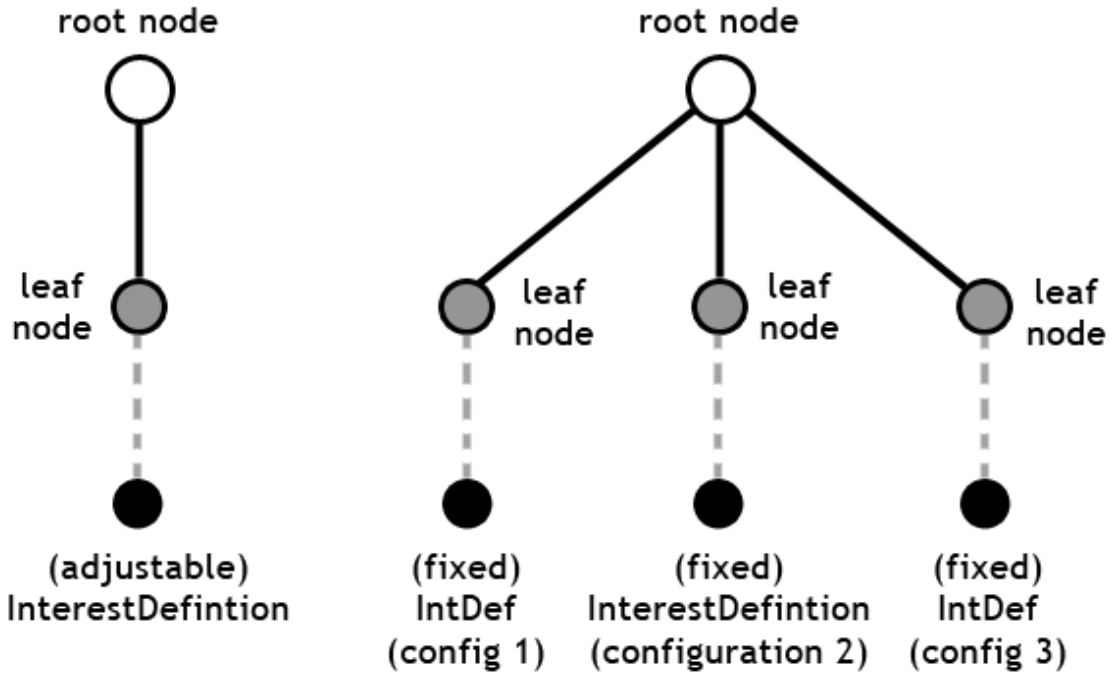

Figure 7.9: Possible NIProxy bandwidth shaping trees for the sniper scenario

in figure 7.9 on the right. We opted to use this option in experiment 5 (see section 7.5) and keep the tree as simple as possible in this explorative experiment. We will later discuss and compare the two alternative approaches.

When the available bandwidth changes, the root node is notified. He in turn tells the leaf node how much bandwidth it can use. It is up to the implementation of the leaf node to decide which adjustments to the LOD will lead to an optimal bandwidth usage. For this, the leaf node has a list of possible adjustments to the single InterestDefinition, called configurations. For each configuration, we have a heuristic tied to it, that should give an indication of how much bandwidth this configuration would use if enabled. The leaf node chooses the configuration with the most appropriate expected bandwidth usage (the highest under the limit) and tells the AOI-API to make the necessary changes to the InterestDefinition, in this way applying the new configuration. The configurations and heuristics for this scenario can be found in figure 7.10.
Note once again that this would be different if we would use multiple leaf nodes, one for each individual configuration. Then, the InterestDefinitions would not have to be adjusted by the AOI-API, the NIProxy would just enable the correct leaf node and by extension the correct InterestDefinition.

7.4.2 Results

For this scenario, we ran two different tests. The first one uses good starting heuristics that give a realistic indication of how much bandwidth a configuration will use, i.e. a best case scenario. The second test shows the worst case: what if the heuristics are completely wrong? Will the implementation be able to deal with this and which errors will occur? Figure 7.10 shows these two different setups.

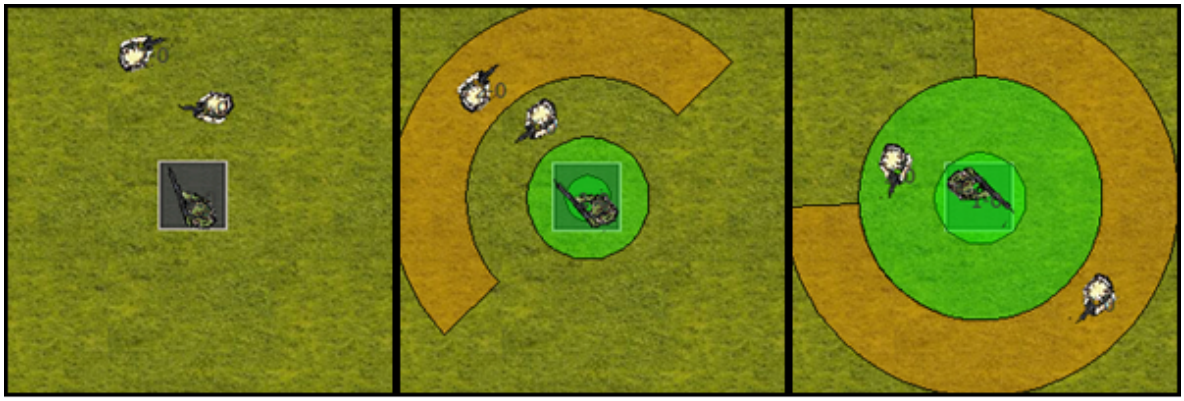

Figure 7.10: Different configurations and their bandwidth usage heuristics for the sniper scenario

In figure 7.10 we see how the different configurations tie to specific AOI shapes and LODs. Configuration 3 is the most extensive and has high LOD with large circle and wedge shaped areas. Configuration 2 is designed to leave out the inner opponent so the sniper can focus on his current target, i.e. the outer opponent in his wedge area. Configuration 1 drops the outside areas and only sends the position updates of the player himself. This is quite radical but indicates a situation where the bandwidth becomes extremely low and the game will be nearly unplayable, but we still want the game to feel a little responsive to the player. Configuration 0 disables even these player updates should the bandwidth drop below the absolute limit of 510 bytes/sec (the amount a single player's position updates use).

Simulation 1 : Best Case

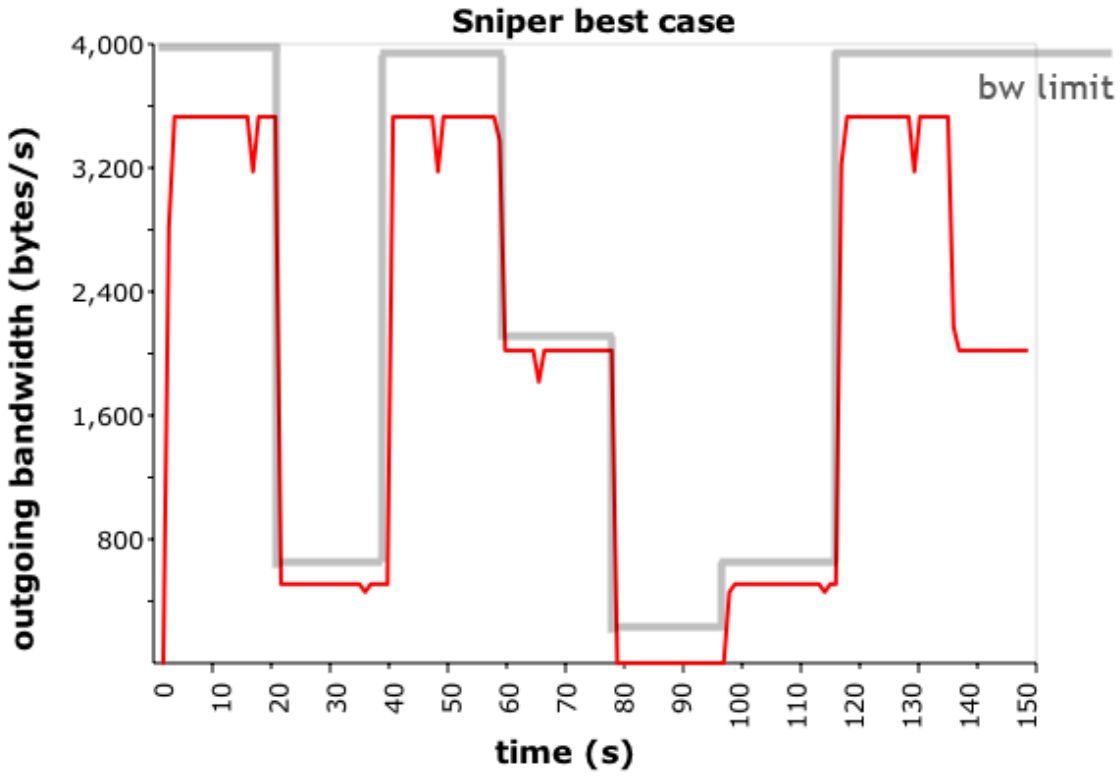

Figure 7.11: Sniper experiment: the best case scenario for player 1 (i.e. the heuristics are accurate)

In figure 7.11 we can clearly see the different bandwidth limits the simulation goes through. The scenario starts at a very high limit of 10000 bytes/sec, so configuration 3 is chosen. At 20 seconds, the limit drops to 540 bytes/sec. This causes the chosen configuration to switch to number 1 and the AOI is adjusted accordingly, so only the positions of the sniper himself are sent. At 40 seconds, the limit is once again put high to 10000 bytes/sec and configuration 3 is enforced once more. At 60 seconds, the limit becomes 2300 bytes/sec and configuration

2 is appropriately chosen. After 80 seconds, the limit goes down to the very low level of 400, which is below even the bandwidth usage of configuration 1, so all traffic is effectively shut down. At the 100 second mark, the simulation wraps, starting at the 540 bytes/sec limit and proceeding to 10000 bytes/sec, 2300 bytes/sec etc.

As we can see in the graph, due to the heuristic's accuracy and this approach, the bandwidth limits are not exceeded at any time and the most appropriate AOI is chosen at all times. Of course, this is because the simulation was specifically designed to show that, given a good heuristic of the bandwidth usage, the NIProxy bandwidth shaping approach can be effectively used in this virtual environment setup. The next simulation makes it clear that the results are quite different if the heuristics are not realistic.

Simulation 2 : Worst Case

Figure 7.12: Sniper experiment: the worst case scenario for player 1 (i.e. the heuristics are wrong)

In this simulation, the bandwidth limits fluctuate in exactly the same way as they did for simulation 1. However, we see that there are some distinct differences in the resulting bandwidth graph. This is completely due to the use of a different heuristic for the different configurations, as no other adjustments were made to the simulation.

At first, the limit is 10000 bytes/sec. According to the heuristic, configuration 1 uses the highest amount of bandwidth, so it is selected. After one second of measurements, the heuristic is updated to reflect the actual bandwidth usage (which is, as we know, actually just 510 bytes/sec). This causes the implementation to switch configurations to the now most appropriate configuration, number 2. This can be seen at the beginning of the graph: there is a very small (1 second wide) hump of 510 bytes/sec before switching to the higher usage of 2020 byte per second. The most important thing to note here is that, even though the bandwidth limit is respected, we are using the available bandwidth sub-optimally. The optimal usage would be for configuration 3 (as we can see in the best case scenario). However, because the initial heuristics for configuration 3 are very low, it is never even considered.

When the limit switches back to 540, configuration 1 is appropriately selected. Remember that its heuristic has been updated after the measurements in the first second, so that the heuristic is now correct. When the limit switches back to 10000, configuration 2 is once again selected, continuing the sub-optimal bandwidth usage for this limit. The switch to the 2300 bytes/sec limit at approximately 60 seconds also does not result in a configuration switch.
At the end of this interval, when the limit switches to 400, something interesting happens. The heuristic for configuration 3 is still 300, which is as we know much too low for its actual usage. Still, the algorithm selects configuration 3 and for a second it is allowed to be used. This makes for the huge bandwidth spike to the highest possible bandwidth usage of 3550 bytes/sec, while we are actually supposed to use only 400 bytes/sec. Luckily, the situation is quickly adjusted after the bandwidth measurements made during this one second. The heuristic for configuration 3 is updated, causing the implementation to choose configuration 0 because all heuristics are now above the limit of 300.
Now that every heuristic has been updated, the simulation will continue to run normally with optimal bandwidth usage.

Note however that if there would have never occurred a limit below 510, the wrong heuristic for configuration 3 would never have been detected and the bandwidth usage for higher limits would always be sub-optimal. This means wrong (initial) heuristics can not only cause sudden bandwidth spikes, but can also have severe consequences that lead to continued sub-optimal bandwidth usage. Let us for example think about what would happen if an intermediate level would have too high of a heuristic, while the higher and lower levels have a correct heuristic. When the limit becomes lower, the simulation will switch from the higher to the lower configuration because it does not know the intermediate configuration has the wrong heuristic and that it is actually the most appropriate option. This can prompt very big swings in consistency and AOI shape, which are of course to be avoided.

7.4.3 Discussion

This comparison between best and worst case gives us some very interesting conclusions, even though the testing scenario itself was very simple.
First of all, we see that, with the exception of a few spikes, the bandwidth limits are maintained very well. The spikes that occur due to incorrect heuristics only last for a second and are quickly detected and repaired. So even though there might be a few of these anomalies, the algorithm works relatively well compared to what could happen if we would not work with a heuristic or estimation and just try every configuration until an appropriate one for that moment was found.
This is an important achievement because there was a doubt that the NIProxies bandwidth shaping strategies would be usable in cases where predicting bandwidth usage is difficult. The NIProxy was mostly tested with video (and other) streams of which the actual bandwidth usage under different configurations was perfectly predictable. Our solution using heuristics is not foolproof, but it is a good approximation and if the heuristics are good, the method uses the optimal configurations while staying under the set bandwidth limit, just as is the case in the earlier NIProxy experiments.

Heuristics algorithm

There is however the important problem that it is possible that some configurations are never visited and as such wrong heuristics can go undetected for a long time, leading to big shifts in AOI shape and bandwidth movement. This problem can be resolved in a number of different ways, which all come down to the notion that we will have to perform measurements for non-active configurations to (periodically) update their heuristics.
The methods differ in how often they would update the heuristics of non-active configurations. In 6.2.2, we suggested a method that would check all configurations for every packet. This would keep the heuristics perfectly updated, but would also require large amounts of processing power. We could instead opt for a method that checks the configurations above and below the currently selected configuration. This is better for performance if we have a lot of configurations, but there is still a (small) risk that a configuration will not be checked for a long time.
Another approach would be to periodically check a non-active configuration (for example every 5 seconds we do a 1 second measurement for 1 non-active configuration). If we make sure the configurations that have not been selected in a while (and as such are most likely to have outdated heuristics) are probed, we can make sure the heuristics are kept relatively up to date without requiring too much extra processing power. Of course, combinations of techniques and approaches are also possible and extensive testing would be needed to detect the worst cases for each approach and see if they are acceptable enough for deployment.

Also, as we stated before in 6.2.2, the frequency of these checks could be tied to the actual CPU budget available. This means that if there are few players, more regular heuristic updates are possible. When the number of player rises (and by consequence the CPU usage), the frequency of the checks can drop in order not to overload the CPU. This allows for a flexible and tunable trade-off between performance and bandwidth usage.

Note also that good initial heuristics can work, but that they are not necessarily sufficient. In this very simple example the factual bandwidth usages for the configurations will never change, but imagine a larger world where players are moving in and out of AOIs all the time. An AOI that used 500 bytes/sec 5 seconds ago can now use 10 times as much if there are more objects nearby. This means the regular heuristic updates are not only important for preventing sub-optimal bandwidth usage: they help prevent some sudden bandwidth spikes by making sure the expectations for bandwidth usage of a given configuration are as accurate as possible.
Another approach that could give better results would be to take the average over time of the measurement results, instead of just using the latest measurement as new heuristic. This would make sure unusual situations will not radically influence the heuristics and make for a better flow of the AOI transitions. This could lead to longer small spikes in the bandwidth usage, as they are not responded to as quickly, but overall it would prevent too many configuration switches, which can help with consistency and performance.

In conclusion for the heuristic algorithm, we can state that the current algorithm is not sufficient and should be supplemented with provisions to (periodically) recalculate the actual bandwidth usage of non-active configurations to keep the heuristics up-to-date.

Bandwidth shaping tree structure

As discussed in 7.4.1, there are multiple ways to build a bandwidth shaping tree for the NIProxy. Figure 7.9 shows two possible approaches for the sniper scenario, while Figure 7.13 shows our approach for the collaboration scenario.
In chapter 6 we already explained that our way of using the NIProxy was not consistent with its original usage scenarios. In previous NIProxy experiments, extensive bandwidth trees were used, where there would for instance be a leaf node per stream type or individual stream or a node per other connected player. Depending on the scenario, these trees were updated and maintained often, with complicated and possibly expensive tree operations.

Our first mission was to see if we could use InterestDefinition-based leaf nodes instead of stream-based leaf nodes. As seen in figure 7.9, this is very well possible but it still gives us plenty of choice on how to actually approach it. The two options here are either to have the AOI-API deal with the different configurations directly (by just having a single

InterestDefinition and leaf node with multiple discrete levels) or, in the other case, to create a leaf node for each configuration separately and switch not through the AOI-API but by just enabling a new InterestDefinition and disabling the old one(s).
The first option gives us a very simple tree and makes good use of the built-in functionality of the AOI-API to adjust areas on the fly when needed. It uses some CPU resources when doing these adjustments though, which can be noticeable if there are many configuration switches. The second option's tree is a lot more complicated (especially as the number of configurations rises), takes up more memory and possibly requires more tree maintenance/updates. The cost of switching to a different AOI is somewhat lower however, as we only need to select the appropriate leaf node.

Both approaches thus have advantages and disadvantages, but it can be argued that both are quite usable. It depends largely on the personal preference of the tree builder (which method is most logical, usable to the programmer) and the situation it is used in. For example, in the next experiment, we use a combination of the two, where separate leaf nodes represent InterestDefinitions that indicate a different game situation, while every leaf node in itself still has separate configurations for the manipulation of that InterestDefinition. This will be discussed in more detail in section 7.5.

7.5 Experiment 5 : Collaborative Environment version 2

This scenario is a reprise of experiment 2 (see 7.2). The main difference is that for experiment 2, we selected the appropriate InterestDefinition manually when the player's position changed and without using the NIProxy. In the original experiment, we also did not include variable bandwidth limitations.
For this experiment, we wanted to try a more complicated structure for the NIProxy bandwidth shaping tree and verify whether these NIProxy, in combination with AOI-API, can hold up in a complex scenario of changing conceptual situations and external bandwidth limits for multiple traffic stream types. This scenario is more alike some of the original NIProxy scenarios than the previous sniper experiment. However, the bandwidth shaping tree is completely different from the NIProxy experiments (where traffic streams were used as leaf nodes as opposed to our InterestDefinitions), making it interesting to see a new way of approaching a similar problem in a different conceptual context and with a different leaf node implementation.

7.5.1 Description and setup

The setup of the experiment is almost identical to that of experiment 2, which can be seen in figure 7.6. The main difference is that player 1 only moves from the top left building to the top right, instead of later continuing down to the bottom building.

Player 1 will remain stationary while the bandwidth limits change (see next section) and will then move outside of the top left building to finally end up in the top right building, where he once more remains stationary while the bandwidth limits alter.

As said, the bandwidth shaping tree used for this scenario is more complicated than the simple tree for the sniper experiment. The tree has 3 different leaf nodes, each representing a location in the world. The first leaf node is for the top left room and is rectangular in shape. The second leaf node is for the circular AOI when player 1 steps outside. The third leaf node corresponds to the top right building. Individually within the leaf nodes, we still have the same kind of configuration/bandwidth heuristic pairs that we used in the sniper experiment. This setup can be seen in figure 7.13.

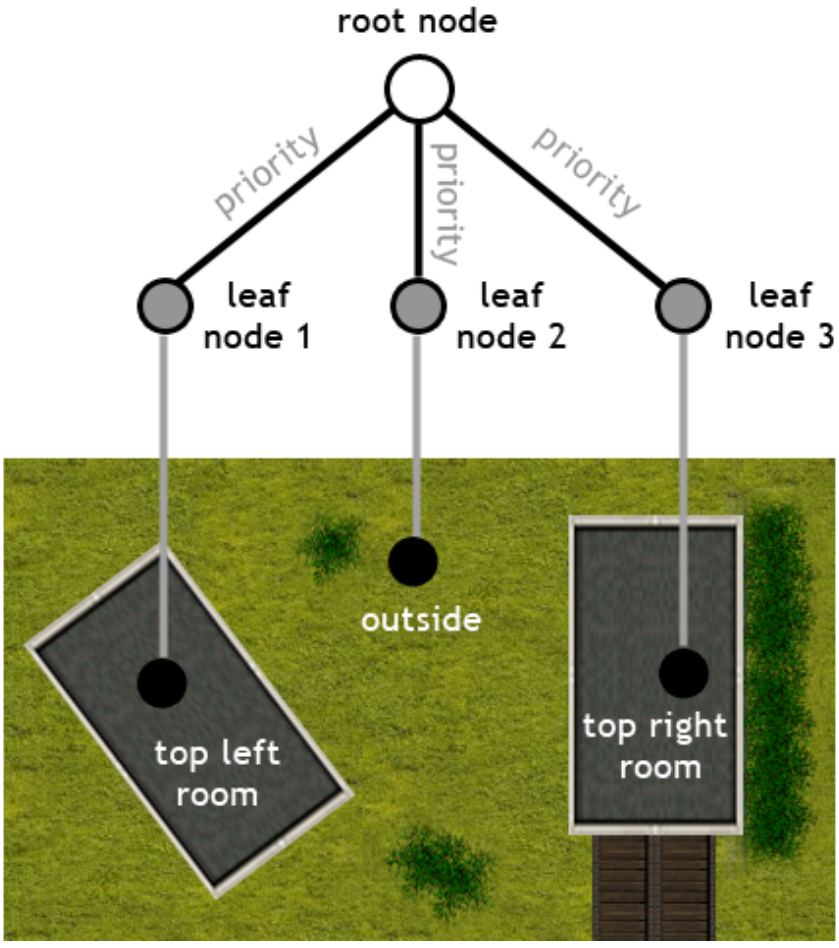

Figure 7.13: NIProxy bandwidth shaping tree for the collaboration experiment

In the sniper experiment, the root node did not do all too much as there was only a single child. Here, it is more important, as it will dictate which configuration each of the leaf nodes

will choose. The root node works with priority values for its children. The child with the highest priority gets the first chance to use as much bandwidth as possible within the limit. It calculates the bandwidth it will use (through the heuristics). This way, the root node can see if there is still some bandwidth left to be distributed. If this is the case, the leaf node with the second highest priority is given the opportunity to use bandwidth, its limit being the original bandwidth limit minus the consumed bandwidth of the highest priority leaf node. If there is still bandwidth left, the next leaf node gets a chance at exploiting this residual capacity etc.
This allows us to indicate which location or states are most important to the player in a very simple and logical way. If the player is in the top right building, the third leaf node will get the highest priority. As we are also interested in what happens in the other building (in experiment 2 we would get text messages from the opposite building), the first leaf node gets second highest priority.
When we are located in one of the buildings, we are not interested in the outside area. Thus, we can assign leaf node 2 a priority of 0, meaning that it is not active. The inverse logic can be applied when the user goes outside: leaf node 1 and 3 become inactive by setting their priorities to 0. Thus the priority system does not just allow us to indicate which AOI should get the most bandwidth, but also which AOIs should be (in)active, allowing us to model complicated situations and conditional scenarios.

When the bandwidth limits change, the priorities automatically aid in providing the most logical bandwidth usage. For example, if we are in the top right building and the bandwidth limit is lowered, the priority root node will make sure that the top left room is first disabled before shutting down video for the top right room. This is arguably the desired behaviour if the player is actively collaborating in the top right room. But this can go a lot deeper. If the top left room is disabled but there is also not enough bandwidth available to keep the video for the top right room, the top right room will switch to only audio or even just text. It is possible that this leaves enough excess bandwidth to make it possible to re-enable text communication with the top left room. This way, the bandwidth is always optimally used and as much data as possible is delivered to the user. Note that this all happens automatically if the priorities are set correctly and the heuristics for the leaf nodes are filled in properly. This means the tree structure requires very little maintenance at runtime, which is in turn better for performance and implementation complexity.

On the other end, there might be applications that would rather switch off video while maintaining text communication with the other room when the bandwidth budget drops. While this is not easily done with the priority node directly, a programmer could implement his own node type that deals with this kind of situation explicitly. This is perfectly possible by extending one of the NIProxy's base classes for tree nodes. This means that, even if a given

situation is not directly supported by the current implementation, it is easy to develop a new node type that delivers the required functionality.

Note that this tree setup is different from the bandwidth shaping tree on the right side in figure 7.9. There, the individual leaf nodes were fixed, i.e. they did not change configurations or use heuristics, but each configuration had its own individual leaf node. Here, we have a leaf node for a world situation or location and multiple configurations within that leaf node. This shows that there are many different ways of constructing the bandwidth shaping tree, depending on user preference and ease of use. This in combination with the extendibility through implementing new node types, makes the NIProxy approach very flexible, versatile and usable in many situations.

7.5.2 Results

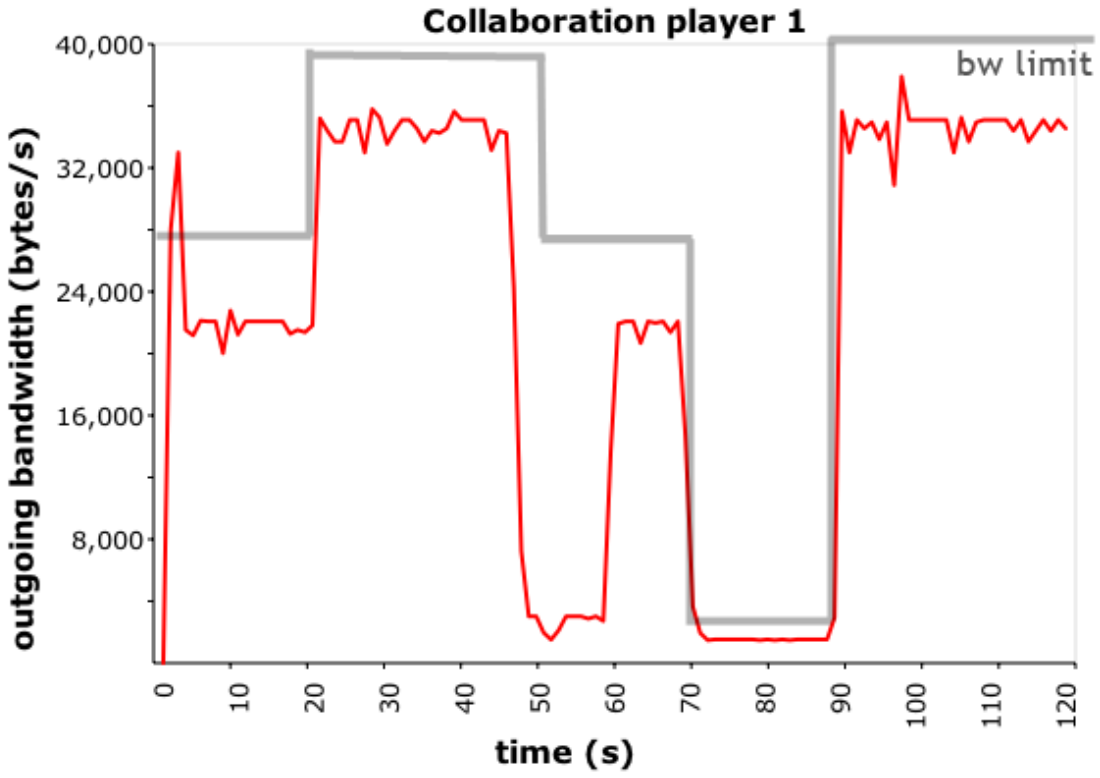

Figure 7.14: The bandwidth usage for player 1

The scenario begins by giving player 1 a bandwidth limit of 27000 bytes/sec, which is enough to be able to receive video from the top left room and textual messages from the right room. Even though we mostly used realistic heuristics, the initial estimate for the highest configuration for the rooms is (intentionally) a little off, which causes the bandwidth spike at the very

beginning of the simulation because video for the top right room is also enabled, as seen in figure 7.14. The system quickly recovers and sends video for the top left room only. At the 20 second mark, the bandwidth limit changes to the 80000 bytes/sec, which is sufficient to allow video from both rooms. The tree automatically makes sure that the highest configurations are chosen for both rooms, resulting in an optimal bandwidth usage.

After 40 seconds, player 1 starts to move outside of the room and into the outside area. When this happens, only the leaf node that represents the outside region becomes active. The priorities for the rooms are assigned 0, effectively disabling them. As the bandwidth limit is still 80000, the outside AOI is set to its highest configuration. This uses considerably less bandwidth than the limit, but this is due to the game logic and semantics (when outside, the player is not interested in the rooms) and not due to the bandwidth shaping approach.
While outside, the limit changes to 27000 bytes/sec again. As the bandwidth usage of the outside AOI is still plenty below this limit, no changes occur. Then, player 1 enters the top right room. This causes the leaf nodes for the rooms to become active again (the top right room gets highest priority, the top left room second highest) and the outside AOI is disabled. As the current limit only allows for video from one room (as was the case in the beginning of the simulation) player 1 receives video from the top right room and text from the top left room.

After 70 seconds, the bandwidth limit is set to the low value of 2000 bytes/sec. This is only enough to allow for textual communications with one room. Automatically, the leaf node for the top right room disables video and switches to an alternate configuration as to not exceed the bandwidth limit. This leaves insufficient bandwidth for the left room to send any data, so it automatically switches to configuration 0. Note that the bandwidth usage in this stage is slightly lower than when the user was outside, as the textual message stream uses less bandwidth than the outside position stream.
Finally, after 90 seconds, the limit is reset to 80000 bytes/sec and both rooms can start sending video to player 1 once more.

7.5.3 Discussion

This experiment clearly shows the flexibility, extendibility and versatility of the NIProxy bandwidth shaping trees and that, in combination with the AOI-API, they can help ensure that bandwidth limits are respected.

It is clear that there are many ways to design the bandwidth shaping tree structure. Depending on the logical structure of the game and the different situations a player can be in, the tree can consist of many combinations of individual nodes, coming together to provide complex behaviour. Important here is that the tree structures are logical and that its parameters

and concepts are easily mapped to concepts in the environment. For example, the priority root node allowed to specify which part of the virtual world was most important, depending on the user location. It also allowed to disable certain InterestDefinitions if they were not desirable during a particular world state. Next to this, the fact that the leaf nodes represent individual AOIs or locations, makes it very easy to assess which ones should be (in)active during a particular situation. The ease of use for the programmers is thus large.
Perhaps even more important is that, by tweaking these simple parameters, the complex bandwidth shaping behaviour is automatically provided without having to change the tree structure or to add specific implementation code for each different situation. If the tree is configured appropriately at the start of the program, bandwidth limit changes will automatically propagate and solicit an appropriate change in the AOIs. This also means that time and care should be spent in designing and testing different bandwidth shaping trees and configurations for every specific virtual environment in order to obtain the most optimal functioning of the system when deployed.

7.6 Experiment 6 : Medium Scale

As the previous experiments all involved fewer than 10 simulated clients at a time, we have not yet tested the scalability of our implementation or approach in a large scale virtual environment. It is important to see if the bandwidth shaping also works in these settings and also that the technique does not incur too high performance penalties which lead to a lower supported number of clients on a single proxy server.
Originally, we conducted ?the large scale experiments on a separate server cluster with large amounts of clients. Because of our findings in these experiments however, which were unexpected and will be discussed in more detail in section 7.7, we decided to also perform tests on a more medium number of clients. This allows us to explain the approach we took for simulating player movement more in detail while also showing how the technique could be used in a smaller scale setup, for instance on a single dedicated FPS server, to provide extra bandwidth scalability.

7.6.1 Description and setup

For these medium scale simulations, we connected 20 clients at once to a single proxy server. These clients are controlled through one of two different movement algorithms: either circle or flocking. The circle algorithm has a client moving on a single circle-shaped path, while the flocking algorithm uses more complex behaviours of separation, alignment and cohesion. As discussed before in section 6.4.1, these methods do not simulate very realistic player movement, but they come close enough to being usable in these scenarios. The reason we

use two different movement algorithms is to be able to investigate which impact this has on overall bandwidth usage and bandwidth scalability. This helps us form a conclusion on the hypothesis that, depending on the type of virtual environment (and thus also type of player movements), different bandwidth shaping provisions must be addressed.
Note that these simulations are indeed quite different from the previous 5 experiments. Now, the client movement is somewhat random, causing clients to enter and exit each other's AOIs with a much higher frequency and in a much less predictable fashion than in the controlled environments of the previous experiments. Where the former experiments delivered easily interpretable results, the measurements for these simulations will be more prone to interpretation and they warrant closer inspection.

For the simulation runs, 6 different setups were considered: 1 time with filtering disabled, 1 time with no change in bandwidth limits and one time with changing bandwidth limits. Additionally, they were tested with both movement behaviours. From these setups, we are able to draw conclusions on performance (through the measured round trip times), the fluctuation of bandwidth usage and the amount of possible bandwidth savings from the technique as opposed to using no filtering at all. The setups with changing bandwidth limits allow us to see how well bandwidth constraints can be satisfied through simple adjustments to the AOIs in this more chaotic approach.

7.6.2 Results

Figure 7.15 shows the bandwidth measurements for the two different movement behaviours in a simulation where the bandwidth limits are not adjusted over time and with filtering enabled. Please note that these results are the averaged measurements of the 20 clients, that both datasets were gathered in independent simulations and that both vertical axes indicate the outgoing bandwidth. Also note that the graph is lacking the averaged measurements for the disabled filtering. As this bandwidth usage is relatively constant at 30200 bytes/sec and because it would make the differences between the two behaviours less clear, we decided not to include it in this graph.

Figure 7.15 is interesting because it shows that the flocking movement behaviour not only consumes more bandwidth overall than the circular movement pattern, it is also far more irregular. Further inspection of the results confirms that this is strongly tied to the average number of objects in the client's AOI. The average number of clients in the AOI of a client moving according to the circle behaviour was 4,5 with a low variance. For flocking, this number was 8, with a high variance. As the same size and shape of AOI was used for both simulations, this indicates that the flocking behaviour leads to more clients being within an AOI on average. This can be explained by the fact that flocking has a cohesion component,

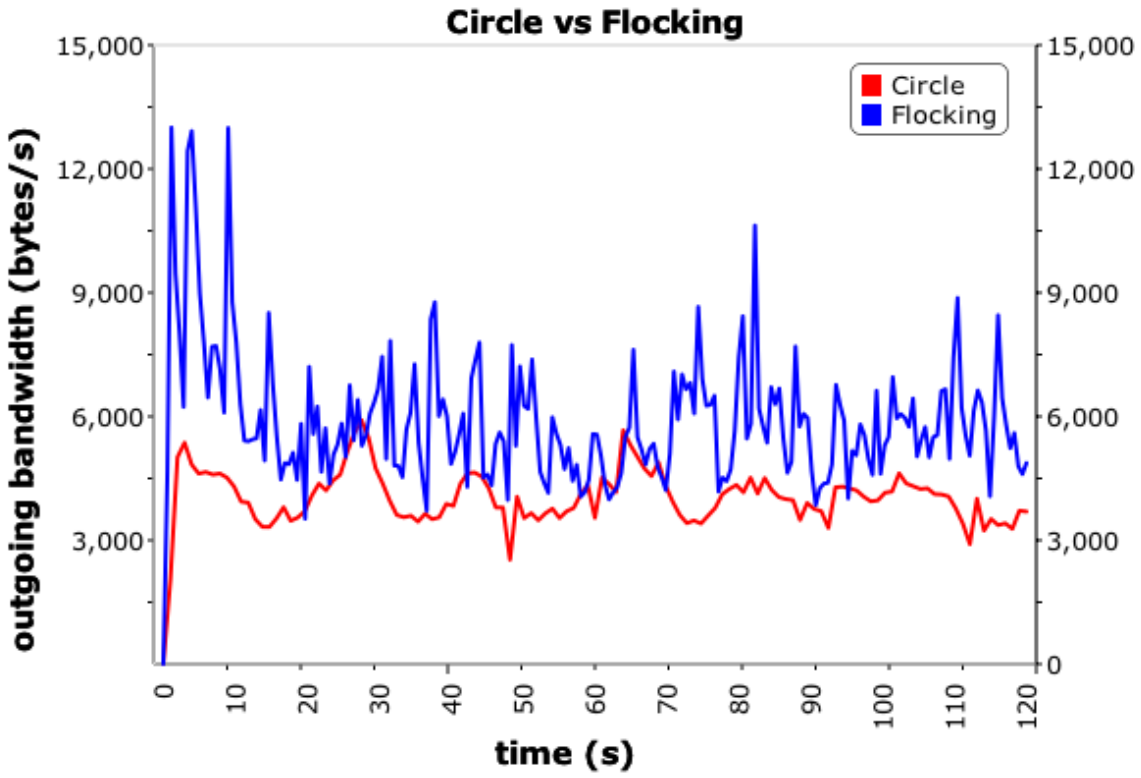

Figure 7.15: The bandwidth usage of the circle and the flocking movement behaviours (filtering enabled)

which tries to maintain an optimal distance between the clients, something which is not present in the circle behaviour.
These results indeed suggest that, depending on the exact movement behaviours of players in the virtual world, the bandwidth patterns will be different, which in turn leads to the need for individually tailored bandwidth shaping approaches per kind of movement behaviour.

Figure 7.16 once more compares the two different movement behaviours, but this time we clearly see that the bandwidth limits. With them, the AOI definitions have changed at set times during the simulation. The simulation begins with a large circular AOI with a high LOD. At 60 seconds, the bandwidth limit goes down and we switch to a much smaller circular AOI with a lower LOD. This is clearly visible in the graph as the bandwidth usage goes down considerably. At 135 seconds, the bandwidth limit goes up slightly. With this, the AOI switches to a dual-circular shape, consisting of a small inner circle with high LOD and a larger outer circle with medium LOD. It is a little more difficult to see, but there is a medium bandwidth usage during this interval, up to 200 seconds, where the bandwidth limit becomes high again and we switch back to the large circular AOI with high LOD. Note that these bandwidth limits and AOI switches are done manually and that the NIProxy was not

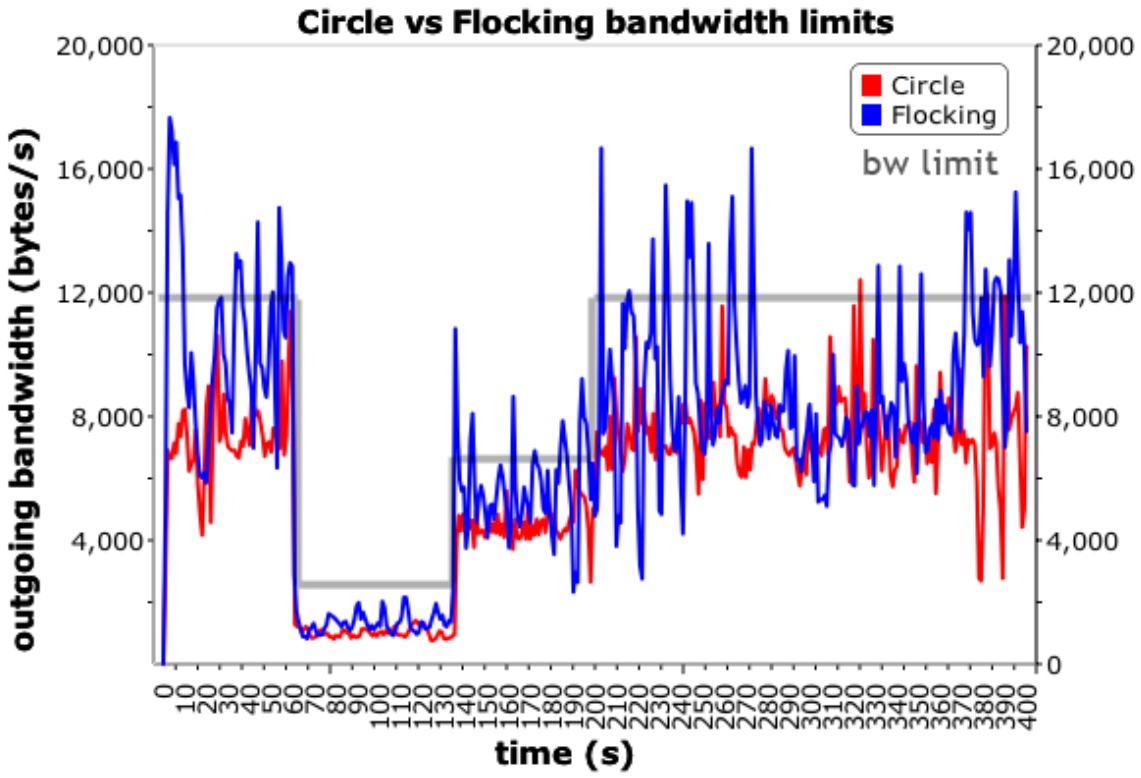

Figure 7.16: The bandwidth usage of the circle and the flocking movement behaviours with changing bandwidth limits

used in these simulations.

The measurements shown in this graph confirm our earlier findings. Even though the circle-based movement has a lot more variance in this graph, as opposed to the graph in figure 7.15, the variance in the flocking behaviour is still considerably higher and the bandwidth spikes and gaps are consequently larger. It is significantly less clear though that the flocking behaviour uses more overall bandwidth than the circle movement. This could be explained by the fact that different AOI shapes and LOD were used for this simulation when compared to the simulations in figure 7.15. The first simulations use a smaller AOI than the bandwidth-limited ones, so it can also be reasoned that clients that were just outside of this smaller AOI in the first simulations, will be just inside the AOI in the second ones. If we assume this is less the case for the Flocking behaviour, it can help explain the smaller difference in bandwidth usage between the two movement patterns in this second graph.

In figure 7.17 we can see a comparison between the round trip times for two simulations, one in which the filtering was disabled and one in which it was enabled. Logically, we would expect the simulation in which the filtering is enabled to perform less, as it needs to do a lot

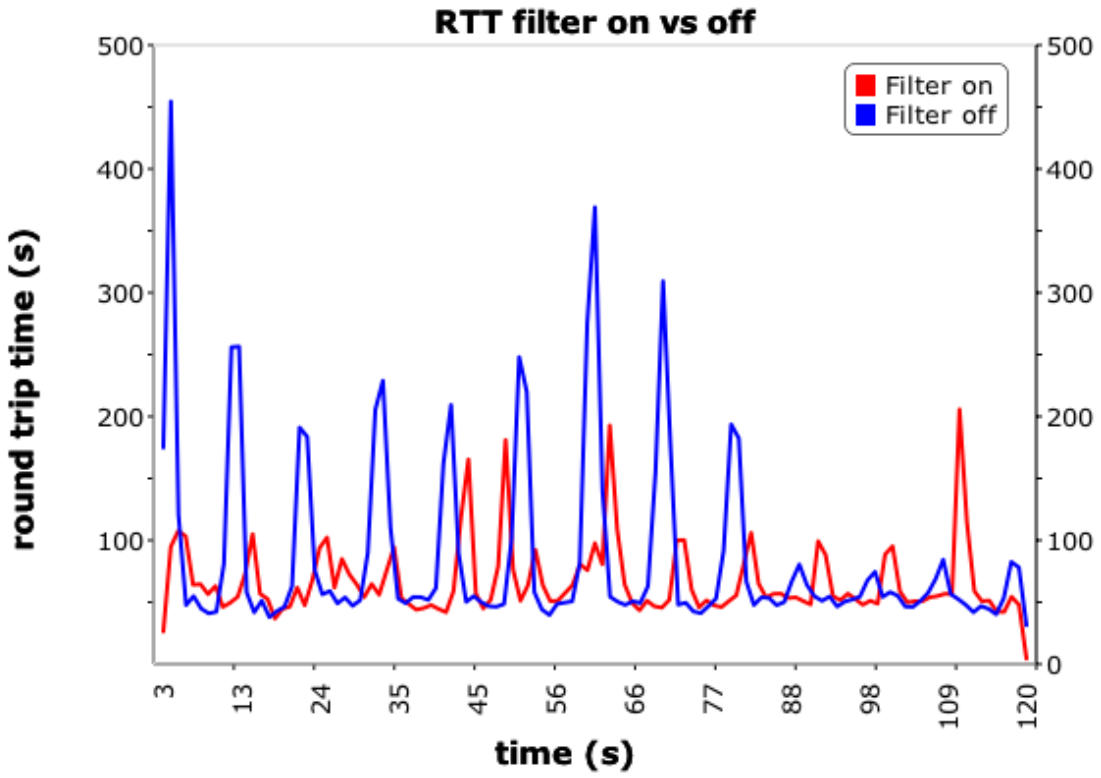

Figure 7.17: Round trip times for enabled and disabled bandwidth filtering

more calculations to determine which objects are in which AOIs. However, in figure 7.17 we can see that the version without filtering has higher peaks for the round trip time than the version with filtering, which is the opposite of what we would expect. It should also be said that the difference is not very striking and that the measurements are close enough to say that there are no truly significant differences. This graph was chosen as an example, as we found that these small differences were true for each simulation with 20 clients, and as such this graph is indicative of the overall measurements.

There can be multiple reasons for the small difference and unexpected measurements. ALVIC-NG was not entirely built with performance in mind, and things such as a custom memory manager are not implemented. NetworkPackets that arrive on the proxy are copied multiple times and sent through many functions before they are sent to the client. It is our belief that this packet pipeline incurs a lot of overhead because of inefficient memory handling. If this overhead is indeed large, it will in fact be reduced by the filtering implementation, as this will severely cut down on the number of packets that is actually sent from the proxy server to the client. This way, the extra CPU power required by the filtering is offset by the memory lookup savings we get because of dropped packets, thus resulting in a slightly lower overall RTT time for the simulation with filtering enabled.

A less plausible explanation could be that it is due to the fact that these simulations were largely executed on a single virtual machine and not on the server cluster. Virtual machines are known not to provide full performance and to cause performance fluctuations since other actions on the hosting operating system could influence CPU usage and thus also round trip time. However, this is less likely as the same trend manifests itself in over 10 different simulations performed. Another explanation could be that we need to draw more graphical entities on the GUI if the filtering is disabled (as all clients will see and thus also render all other clients). However, this extra number of rendered clients is not high with 20 clients total, which also makes this theory less likely.

7.6.3 Discussion

When looking at the different graphs, it is clear that there is a difference between the bandwidth use and fluctuations of the circle and flocking movement behaviours. This confirms our hypothesis that individualized bandwidth shaping strategies might be needed on a per NVE-type basis to provide optimal bandwidth usage.
When we look at figure 7.16, we can also see it is much more difficult to ensure a certain bandwidth limit in these types of situations. Even though we did not use hard bandwidth limits in the simulations (we just changed the AOI shapes to show the relative differences in bandwidth usage), it is clear that especially for the Flocking movement it will be difficult to at all times retain this bandwidth limit, as there are many high spikes in the bandwidth usage. For the circle movement, this is a lot better, as it shows a lot less variance overall, but there are still plenty of bandwidth anomalies that could potentially cause problems. Overall, a more extensive simulation is probably needed, possibly with the integration of the NIProxy and appropriate bandwidth shaping trees, AOI configurations and heuristics. With this limited data, it is difficult to conclude how well bandwidth limits can be satisfied using our techniques.

The same is true for the research question about performance and how it is affected by the filtering implementation. The results are inconclusive and even indicate an inverse of the expected result that filtering would require more CPU power than no filtering. Because the differences between the datasets are not all that large and there are many factors that might influence this performance, it is needed to perform further testing on a larger scale to see if these trends continue

Finally, since we use movement behaviours that are not necessarily realistic and because the used AOI shapes and player interactions are not based on game logic (as they were in the other experiments), we cannot make any statements about consistency within these experiments. This is because consistency is strongly tied to the game logic and what measure of consistency

is needed for the human player to be able to play the game without noticing errors. A more rigorously designed experiment, set in a real game scenario with realistic player movement is needed to judge the measure in which we can retain consistency. Overall however, we could say that consistency retention is relatively bad for this type of scenario, especially with low bandwidth limits, as the number of clients in the AOI can drop quite low, limiting the client's interaction options in the environment.

7.7 Experiment 7 : Large Scale

These final experiments were aimed at simulation large scale situations for virtual environments with thousands of users. However, all these experiments use only 100 clients and 1 server of every type (1 proxy, 1 logic server, 1 region management server and 1 client controller). This is because we found that the experiments with 100 were already difficult to execute and that the data collected from them is largely inconclusive, due to various factors. As such, it is not useful to test with even more clients, as the results will be of doubtful quality.
In this section, we will discuss our simulations for 100 clients and investigate the results.

7.7.1 Description and setup

The executed simulations are very similar in setup to those for experiment 6, discussed before. We once again perform simulations with filtering disabled and with filtering enabled but with no change in bandwidth limits. The simulations with changing bandwidth limits were not executed on this scale because the results of the other simulations were already of doubtful quality.
The main difference is that these larger scale simulations are done on a separate server cluster and that instead of 20 clients, we simulate 100 clients at a time. In the previous experiment, the clients all connected at the same time. Here, the clients connect with an interval of 1 second (i.e. normally all clients are connected 100 seconds after the simulation starts). This allows us to better observe the number of clients a proxy server can comfortably support before we notice severe increases in the round trip times (which indicate too high CPU usage on the proxy).

7.7.2 Results

Filtering and 10 updates/s

First, we compare the results for 10 updates/s per client for the two movement types. We plot their outgoing bandwidth next to the observed round trip times, to see if there is a connection between bandwidth usage and changes in round trip time.

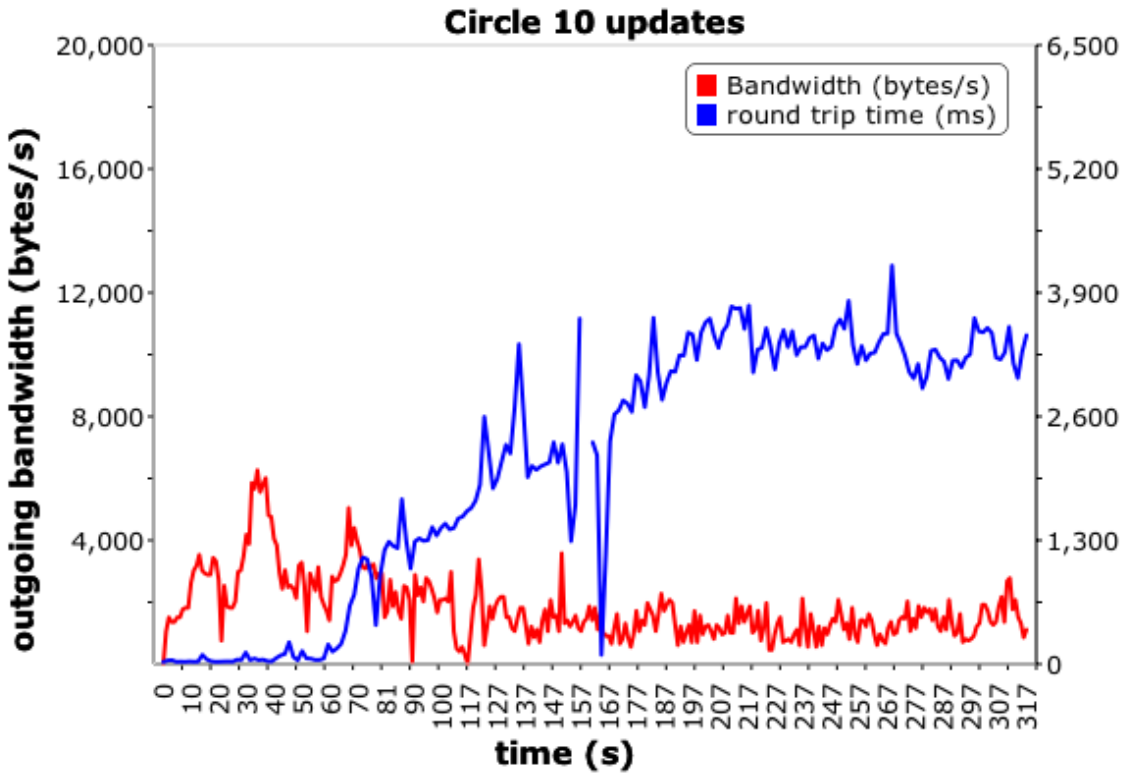

Figure 7.18: The bandwidth usage and round trip time (in milliseconds, right vertical axis) of the circle movement behaviour at 10 updates per second per client

Figure 7.18 shows the results of the simulation for the circle behaviour, while figure 7.19 shows the results for the flocking behaviour. At first sight, they seem to have much in common. For both simulations, the round trip time starts going up considerably round the 50 to 60 second mark. The round trip times keep climbing steadily until they become more or less stabile again after 160 to 180 seconds (the point at which the clients have stopped spawning).
In figure 7.19 there seems to be a clear connection between the round trip times and the outgoing bandwidth: as the round trip time goes up, the bandwidth usage goes down. This trend is less obvious in figure 7.18, but we can still see that there is somewhat less bandwidth usage after the 120 second mark than overall before. This connection between bandwidth and round trip time could possibly be explained by saying that the round trip time rising so high is a clear indication that the proxy server was having a hard time dealing with all the traffic. As the proxy servers becomes overloaded, it starts dropping packets (as we use the UDP protocol), which never get distributed back to the clients. These dropped packets cause the measured bandwidth to go down.
The graphs also confirm the previous experiments by showing that the flocking behaviour uses more overall bandwidth than the circle movement and has a larger variance.

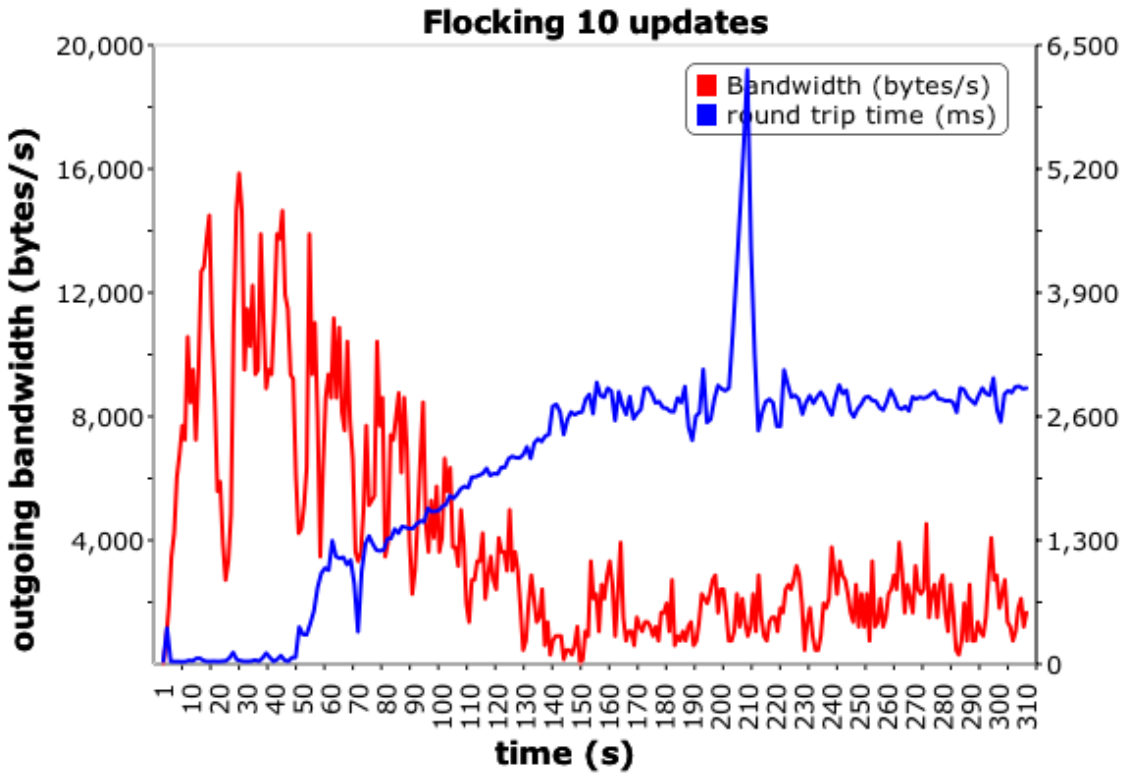

Figure 7.19: The bandwidth usage and round trip time (in milliseconds, right vertical axis) of the flocking movement behaviour at 10 updates per second per client

However, despite these seemingly logical results, we noticed some anomalies during the simulations. Firstly, all clients should have been spawned at the 100 second mark (as we spawn one client every second). However, we noticed this process received serious delays, resulting in the total 100 clients only being spawned around 160 seconds. We thought this could be due to an overloaded bot simulator (as that is the only possible factor that can delay the spawning process). However, when looking at the CPU usage of the client simulator (which was run on its own machine as only process), it was clear it never went over 50% CPU usage and that it usually even only used around 10%. As such, we have been unable to find an explanation for this delay in client spawning. The simulations where the filtering was disabled gave even larger delays, as we will discuss in the next section.

Filtering and 2 updates/s

After the two previous simulations, we wanted to see if lowering the update rate to 2 updates per second would have an influence on the bandwidth/round trip time connection and in which way.
As we can clearly see in figure 7.20 and figure 7.21, the moment at which the round trip time

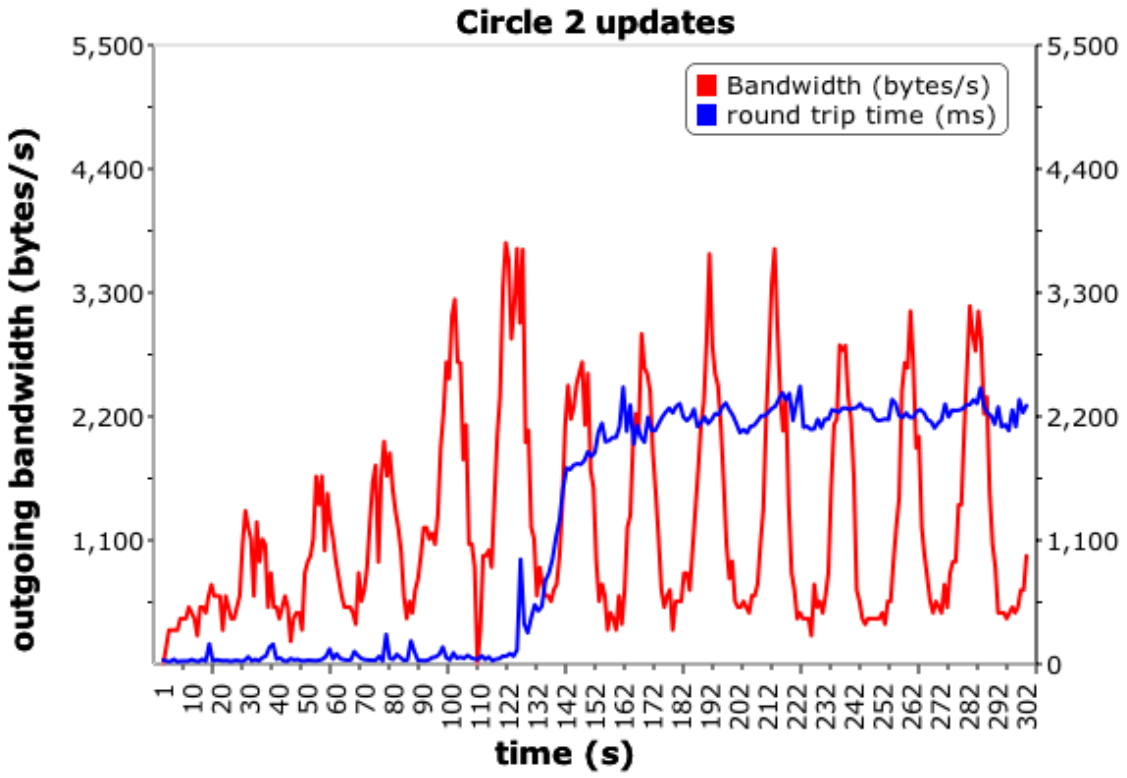

Figure 7.20: The bandwidth usage and round trip time (in milliseconds, right vertical axis) of the circle movement behaviour at 2 updates per second per client

starts going up is no longer at 60 seconds, but after 120 seconds. This means that the proxy server gets overloaded a lot later and that the amount of updates per second indeed has a serious influence on how many clients can be connected to the proxy server at the same time. We can also see once more that the flocking behaviour uses more bandwidth overall than the circle movement. At the same time, figure 7.20 seems to contradict the previous observations that flocking has a higher variance than the circle movement, as we can clearly see the very periodic bandwidth spikes for the circle movement. However, this can be explained by the nature of the circle movement. The client will always follow the same circle and such, at some point, he will pass his starting point again (and again later etc.). This explains the periodicity. The fact that this was much less obvious in the other graphs is because the initial positions of the clients and the radii of the circles is entirely random and not the same for every simulation with the circle movement. For this particular simulation, the spawning positions were so that most clients were in a semi-circle around the center of the world map. As such, when they moved away from this center, less other clients were in their AOI and the bandwidth usage goes down. When they move back in the direction of the center however, more clients come in their AOI (as all clients are gathering in the center instead of being more dispersed on the edges of the map) and the bandwidth goes up again. Because of the

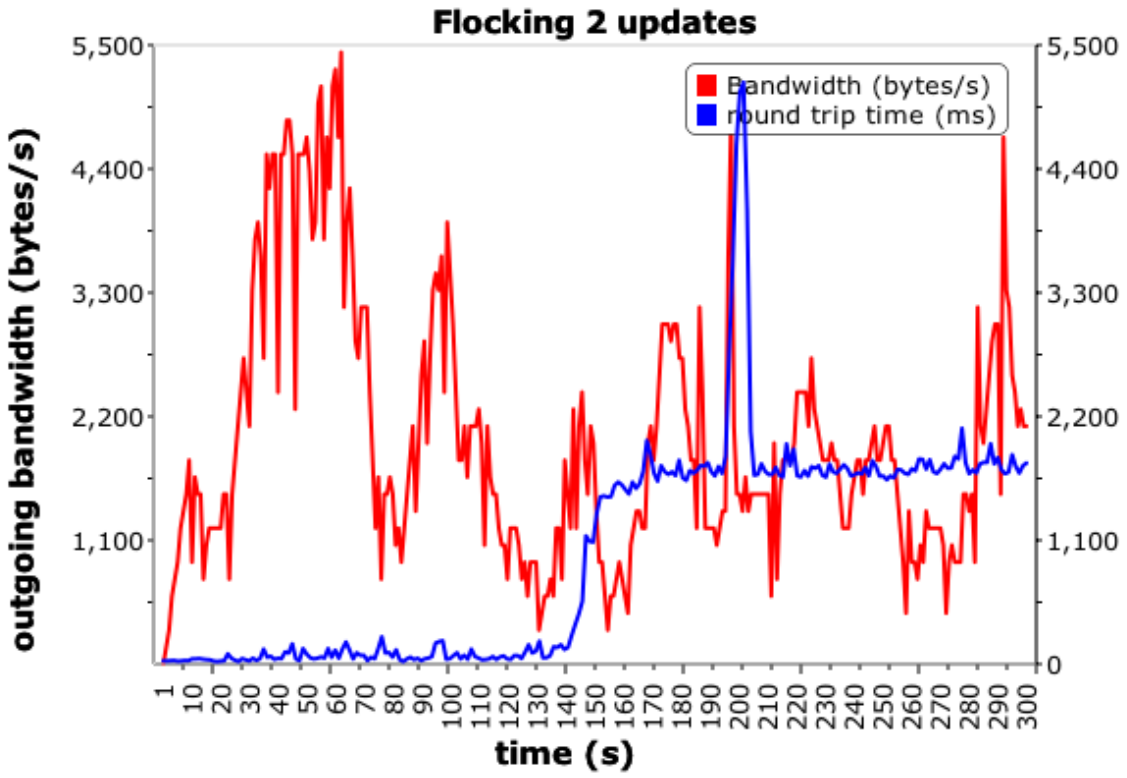

Figure 7.21: The bandwidth usage and round trip time (in milliseconds, right vertical axis) of the flocking movement behaviour at 2 updates per second per client

periodicity, this manifests itself as a series of bandwidth spikes.

A more important observation here however, is that the previously observed connection between bandwidth usage and round trip time seems to be completely non-existent. Especially in figure 7.20, we see that the bandwidth spikes and bandwidth usage is still at a very similar level as before the 120 second mark where the round trip time goes op. This is less clear in figure 7.21, but here we also cannot say that we see a clear connection between the round trip time going up and the bandwidth going down. These results are very unexpected and remain without an explanation.

This is coupled to the very sudden rise of the round trip time, as opposed to the very gradual ascent in the simulations with 10 updates per second. Here, the round trip time goes up from a very low level in about 20 seconds, where the previous simulations did the same in 100 seconds. The fact that the rise is short before stabilizing is normal, as it only starts 40 seconds before the spawning of the clients ends and we converge to a stable round trip time (where in the other simulations, it was 100 seconds before the spawning ended). However, the strange thing is that the rise in round trip time is so high for the latter experiments. We would expect the rise to continue for 20 seconds before becoming stable at maybe 500 to

1000ms. However, it rises much higher, almost to the same level as for the simulations with 10 updates per second.
This can indicate that the rise of round trip time is not (completely) coupled to the CPU usage because of client traffic. If this would be the case, the rise of round trip time should be a lot less high for the simulations with 2 updates per second. This in combination with the very sudden rise in the round trip times makes us think the performance deterioration might have something to do with other side effects, such as memory leaks that reach a critical point around 130 seconds, buffers that are completely filled around that time or thread pools that are depleted. This is only a guess and remains untested. Much deeper analysis of the exact resource usage of the proxy servers is needed before we can create conclusive simulations that can give us an idea of how the filtering affects the performance and also how different update rates affect this performance.
This hypothesis of a deeper underlying cause to the strange performance measurements is deepened by the results for simulations where the filtering is disabled, as we will discuss next.

No filtering

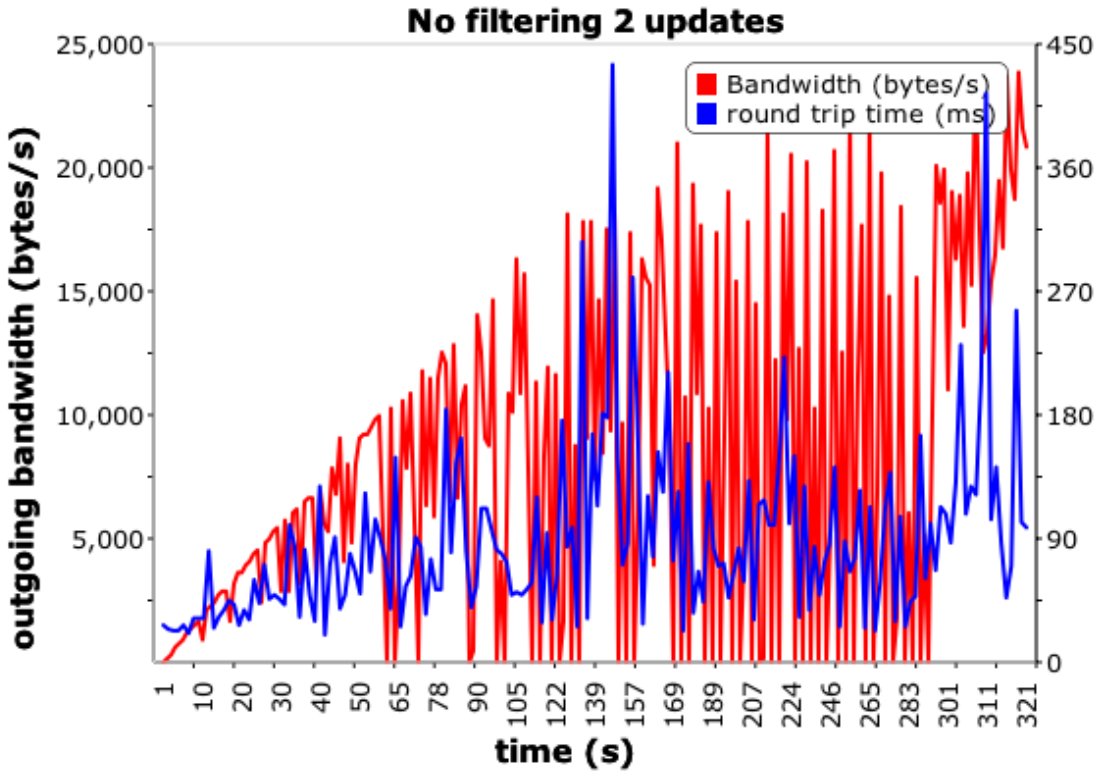

Figure 7.22: The bandwidth usage and round trip time (in milliseconds, right vertical axis) of the circle movement behaviour at 2 updates per second per client with filtering disabled

Finally, we perform the same simulations with the filtering disabled. This means we use the original ALVIC-NG implementation for packet handling with only the bandwidth measurement code of ourselves as an extra step. We would expect this to give us results similar to the original ALVIC-NG scalability experiments [85], where a single proxy server was shown to be able to handle up to 625 simultaneous clients at 3 updates per second.
However, the actual measurements are very strange and do not directly support this. The weirdest things are the extremely fluctuating bandwidth measurements, where there are plenty of measurement that show zero bandwidth usage. This could indicate very high CPU usage (causing threads to be slowed down for example), but the round trip time measurements do not support this hypothesis. In fact, the round trip times are a lot lower than those of the previous simulation measurements (as can be expected), which makes the bandwidth measurements even stranger.
Because these measurements were so strange, we performed them multiple times, each time resulting in similar measurements. Another weird anomaly that we noticed was the spawning delay that we also noticed in the previous simulations. We would expect all clients to be spawned after 100 seconds, where this took about 160 seconds for the previous simulations. For these simulations with the filtering disabled however, we saw that all bots were only spawned after about 300 seconds. This adds to the idea that there is a deeper reason for these weird measurements that we have been seeing for the large scale experiments, as this spawning delay is completely inexplicable and warrants further investigation.

Even though the simulations with 2 updates per second gave us these weird results ,we decided to also do simulations with 10 updates per second with the filtering disabled. Sadly, we cannot present those results as they were completely unusable. As soon as the simulation had spawned about 20 clients, the simulation started behaving very strangely. For example, the output files for the round trip time measurements weren't even being created, let alone written, for many clients. Instead of 100 measurement files (as we had gotten from the previous simulations), this simulation only resulted in 26 files. 20 for the first 20 clients and then sporadically a file for a random client.
The bandwidth measurements were also off. When looking at the files, we could see that there were gaps in the bandwidth measurement: i.e. there was not a line in the file for every second of the simulation, which was the case for all the previous experiments. Note that these gaps are not the sudden gaps of 0 bandwidth in the graph: those are actual written bandwidth measurements in the file.

7.7.3 Discussion

The main conclusion for the simulations in this experiment is that we can present no conclusive results on the scalability characteristics of our implementation or the normal ALVIC-NG

implementation in general. Where the first simulations looked like they could provide us with logical measurements and following conclusions, these conclusions were later contradicted by other, very similar simulation results.
The most important question, namely how does the overhead incurred by the filtering implementation influences the maximum number of clients a proxy server can support, could not be answered. Looking at the measurements, one could say that the simulation without filtering clearly shows that it has no increase in round trip time, as opposed to the simulations with filtering, where the round trip times went up spectacularly after a low number of clients. However, given the other anomalies in the measurements and the observed behaviour while performing the experiments, it is very difficult to just draw this conclusion. We hypothesize that there are deeper, underlying reasons in the lower layers of the implementation of the ALVIC-NG and possibly AOI-API codebase that lead to these strange measurements. As such, we cannot formulate a clear answer to this research question based on the results and say that a deeper investigation into the implementation is needed.

7.8 Conclusions

After describing the various experiments and simulations that we performed, it is time to discuss how they have helped answer our various research questions that we posed at the beginning of this chapter.

- How versatile is the technique?
  Through the various experiments, we have shown that the implemented techniques are very versatile and that they can be applied to a wide array of different type of virtual environments. Even traditionally difficult game types, such as an FPS or RTS, can be modeled through the AOI-API, where it is very important that logic from the game world can be coupled to logic in the AOI definitions. This is for example visible in the coupling of AOI radius with weapon range in an FPS and line of sight in an RTS. The simulations also show that using these models, the player's expectations for the virtual environment can be largely retained while effectively cutting down on bandwidth usage.

  In addition, we have shown that the basic principles behind the NIProxy, and the bandwidth shaping trees in particular, are very flexible and extendable. Even though most of our scenarios were never envisioned in the original design of the NIProxy, it was relatively easy to incorporate our own game logic into the existing implementation. The experiments show that the bandwidth shaping trees can be successfully used to manage bandwidth streams in a virtual environment, while also being very logical to build and maintain when coupled to game concepts and player state.

- How effective is the technique in limiting bandwidth?
  Firstly, we want to know how big the total bandwidth gain is that can be obtained

by this technique. While depending on the specific game or environment and specific AOI definitions used, throughout the experiments we have seen that there can be huge differences between the bandwidth usage with filtering enabled or disabled, while for the most part keeping consistency intact.

The second aspect was if we could also make sure the bandwidth usage stays below set bandwidth limits. In many of the simple, controlled simulations this was definitely the case. Especially when coupled with good bandwidth heuristics and knowledge about the game logic, any bandwidth limit can be respected given enough up front configuration. Next to this, the implementation is very fast in detecting errors in its bandwidth shaping, which reduces the risk of precarious bandwidth situations becoming even more harmful.

For the less controlled and larger scale experiments, this aspect of the technique was less clearly defendable. This is partly because we work with somewhat unrealistic portrayals of user movement, but also partly because the AOI definitions were not founded in a particular game logic or environment. As we have seen in the earlier experiments, game logic is very important to make an accurate model of the AOIs that are best fit for a particular game or environment. Only if we have the correct AOI model and player movement can we accurately assess if various bandwidth limits can be respected.

Based on these simple larger scale experiments however, we can say that it is likely that bandwidth limits can also be kept in larger scale situations, but that there will most likely be more sudden spikes or gaps in the bandwidth usage. Luckily, it was shown that our techniques can deal with these changes quickly and effectively, which strengthens our belief that the technique is usable for fine-grained bandwidth shaping in large scale environments.

- How well does the technique retain consistency?
  The retention of consistency is, much like the previously discussed adherence to bandwidth limits, very game or environment dependent. It is very difficult to think of an objective measure that can be used to track consistency for all types of game logic. Depending on the game, it will be very harmful to the player if certain elements are invisible, or not harmful at all.

  For the simpler experiments, this consistency was preserved relatively well (even though we did make some remarks for the FPS experiment, see section 7.1). This can be explained once more by the fact that the AOI definitions and as such the bandwidth shaping strategies were all built starting from a good conceptual knowledge of the possible world states. This can severely help limit the chances of certain AOI definitions having a negative impact on the user's experience in the game, even when the bandwidth limits change.

  As the larger scaled experiments primarily use algorithms that are not completely real-

istic or based on actual player movement observations, it is much more difficult to give a conclusion for the consistency retention capabilities of our technique for larger scale environments. When looking at the smaller experiments however, it is likely that these concepts can be transferred in some way or the other to the large scale equivalent and as such that a good measure of consistency will be kept in those situations as well.

The most important thing here is the versatility of the AOI-API and the bandwidth shaping techniques. It is very easy and quick to change to a different AOI definition, which means that should the consistency start to deteriorate, we can switch to a new model in which the consistency is kept better in that particular situation. There will always be a trade off between this consistency and the bandwidth usage, but with our technique this trade off is very controllable, especially at runtime.

- How does the technique affect the performance of the proxy servers?
  This last question remains unanswered by our simulation results. As discussed in experiments 6 and 7, we ran a large number of different simulations to assess not only the bandwidth characteristics of a large scale setup, but also how it would affect the proxy server's performance and eventually the observed round trip time (or ping) for the users, which is a very important measurement for the application to feel responsive and interactive.

  The simulation results were inconclusive. While with medium scale simulations it appears that there is relatively little impact from the added filtering implementation (see figure 7.17), the larger scale measurements contradict this in many ways while showing several other anomalies as well. Because of these anomalies, our hypothesis is that these performance issues might have a deeper origin within the implementation and are not just dependent on if we use our filtering implementation or not.

  As such, a deeper investigation and possible re-implementation of certain parts of the ALVIC-NG, AOI-API and NIProxy frameworks are needed before running these simulations again and drawing conclusions from their results.

Chapter 8

Conclusion

A couple of important conclusions can be drawn from the first part of this thesis.
Firstly, that dynamic bandwidth scalability is needed and useful in nearly all forms of NVE development and deployment. While zoning is the prevalent technology for this purpose, we find that additional techniques for bandwidth shaping are needed to make specific types of NVEs possible. Game types such as First Person Shooters and Real Time Strategy games or more serious applications such as collaborative environments often have specific bandwidth usage characteristics that cannot be managed by using only zoning. For these types of NVEs, a more fine-grained technique for bandwidth control is needed.
Secondly, after examining various other techniques and determining which techniques are used in other existing systems, we came to the conclusion that not one technique is optimal for all NVE types. Depending on the exact game or application logic and even deployment characteristics, there are different expectations for consistency and bandwidth usage. The most flexible of the researched techniques is Area of Interest, as it can be used to model a wide array of different situations and can be coupled directly to the NVE logic to provide a meaningful connection between bandwidth and consistency that can be used when determining the best trade-off between those two parameters.

In our implementation of an Area of Interest system for the second part of this thesis, we put a strong emphasis on this versatility to make sure our technique would be usable in as many types of NVE as possible. Our AOI-API for instance includes Level of Detail management, multiple AOIs per user, different AOI shapes and dynamic AOI shape adjustments. We couple this AOI system to the bandwidth shaping tree technique originally used in the NIProxy project [93]. This allows us to provide very fine-grained bandwidth control that is flexible and able to quickly react to changes in bandwidth availability. We implemented and tested various use cases and experiments that show the power and flexibility of our technique and its ability to provide good and adaptable bandwidth shaping at runtime.
These experiments also showed once again that the exact type of NVE we use the technique

in can have a large impact on the obtained results. As such, the large versatility of our implementation does not only allow us to create many different types of NVE, it also allows us to provide very fine-grained modeling of specific situations for each of these NVEs and thus obtain the optimal AOI definitions and bandwidth shaping for many situations.

The research question on the usability of the proposed technique for larger scale deployment remains open. Several (external) factors contributed to the non-conclusive results, including performance issues in the underlying implementation and the absence of realistic player movement in combination with a conceptual world logic on which to base the AOI definitions. With additional effort however, these issues can clearly be resolved.
Future work is needed to investigate the exact usability of our technique for the envisioned applications, such as MMOFPSs or MMORTSs. But the high versatility of the implementation and good test results for smaller scale scenarios make it likely that it will be possible to use our technique in these challenging large scale settings to provide a flexible way to tune the consistency/bandwidth tradeoff.

Chapter 9

Future work

There are various aspects about our technique that can be improved or extended upon.
Firstly, it is important to review the ALVIC-NG implementation and find ways to optimize its performance or at least identify the source of the issues we encountered in our large scale experiments. Optimization of this implementation, for example through explicit memory management or by using less threads for the network communication, will help improve performance even if the AOI techniques are not used, which will in turn increase the number of clients that a single proxy can support. This leads to a lower overall number of deployed proxy servers in the network and thus cost and maintenance.
Secondly, we should look into a more integrated solution for the coupling between ALVIC-NG and the AOI-API/NIProxy implementation. Up to this point, the versatility provided by adding our techniques via a plugin-like system allowed us to work without worrying about the effect or our implementation on ALVIC-NG while other people were working with the same code base. However, now that the implementation is completed, we would like to use it in other areas of ALVIC-NG as well (for instance to calculate which neighbouring areas should be requested by the proxy server). An integration of the AOI-API in the more basic layers of ALVIC-NG gives more modules access to it and extends the capabilities for developers that use the system.
Thirdly, the implementation of the RTS experiment and scenarios can prove to be a very interesting use case for our suggested techniques and help determine if a large scale RTS is possible without using the traditional lockstep/fully deterministic simulation. In addition, it can provide some insight in possible approaches to networked physics, which is an important area of research, even for smaller scaled games.
Fourthly, the NIProxy can be adapted to be of a more modular nature and thus more parts of it can be fit into ALVIC-NG. This would not only benefit the NIProxy's re-usability in other projects but also enhance ALVIC-NGs capabilities with for example video transcoding services, which makes it more attractive for collaborative environments.
Lastly, a bigger project can be started in which we first gather realistic movement data from

players in an existing NVE. After this, we can run extensive experiments, using various AOI models based on the NVE logic, to see how different AOI models impact the consistency and bandwidth usage for this particular game. We can also use those experiments to find the best way of constructing fitting heuristics for the bandwidth shaping system. Only when a pipeline has been devised for iterating towards the optimal AOI models and bandwidth heuristics, can we think about integrating this system as an important part of a (commercially) deployed NVE.

Bibliography

[1] Aion official homepage. http://uk.aiononline.com/, August 2010.

[2] Apb: All points bulletin. http://www.apb.com/intro/en/, August 2010.

[3] Arithmetic coding. http://en.wikipedia.org/wiki/Arithmetic_coding, August 2010.

[4] Bigworld technology. http://www.bigworldtech.com/index/index.php, August 2010.

[5] Binary packets. http://wiki.gamedev.net/index.php/Binary_Packet, August 2010.

[6] Bittorrent : P2p filesharing. http://www.bittorrent.com/, August 2010.

[7] The cost of creating and maintaining an mmorpg (2010). http://www.thamelas.com/2010/02/12/the-cost-of-creating-and-maintaining-an-mmorpg/, August 2010.

[8] Dis data dictionary-pdu data (2006). http://www.sisostds.org/dis-dd/pdu/index.htm, August 2010.

[9] Distributed hash table. http://en.wikipedia.org/wiki/Distributed_hash_table, August 2010.

[10] End of nations mmorts. http://www.endofnations.com/en/game/faq.php, August 2010.

[11] Enet. http://enet.bespin.org/, August 2010.

[12] Eve online developer blog. http://www.eveonline.com/devblog.asp, August 2010.

[13] Eve online official homepage. http://www.eveonline.com/, August 2010.

[14] Eveonline sol architecture optimizations (2005). http://www.eveonline.com/devblog.asp?a=blog&bid=286, August 2010.

[15] Everquest official homepage. http://everquest.station.sony.com/, August 2010.

[16] Fallen earth. http://www.fallenearth.com/, August 2010.

[17] Farmville facebook game. http://www.farmville.com/, August 2010.

[18] First look at mag. http://blog.us.playstation.com/2008/07/22/first-look-at-mag/, August 2010.

[19] Forward error correction. http://en.wikipedia.org/wiki/Forward_error_correction, August 2010.

[20] Global agenda. http://www.globalagendagame.com/, August 2010.

[21] Guildwars official homepage. http://www.guildwars.com/, August 2010.

[22] Heroengine. http://www.heroengine.com/, August 2010.

[23] Huffman encoding. http://en.wikipedia.org/wiki/Huffman_coding, August 2010.

[24] Huxley mmofps. http://www.huxleygame.com/huxley_faq.php, August 2010.

[25] Icarus studios. http://www.icarusstudios.com/, August 2010.

[26] Ice internet communications engine. http://www.zeroc.com/ice.html, August 2010.

[27] Ipv6 multicast support. http://www.tcpipguide.com/free/t_IPv6MulticastandAnycastAddressing.htm, August 2010.

[28] Lempel-ziv encoding. http://en.wikipedia.org/wiki/Lempel%E2%80%93Ziv, August 2010.

[29] Massive action game. http://www.mag.com, August 2010.

[30] Mmog active subscriptions 200000+ (2008). http://www.mmogchart.com/Chart1.html, August 2010.

[31] Mpeg-2 video encoding. http://en.wikipedia.org/wiki/MPEG-2, August 2010.

[32] Netdog networking engine. http://www.pxinteractive.com/, August 2010.

[33] Network address translation requirements for udp. http://www.ietf.org/rfc/rfc4787.txt, August 2010.

[34] Onlive game streaming. http://www.onlive.com/, August 2010.

[35] Opensim load balancing and region splitting. http://opensimulator.org/wiki/OpenSim_Load_Balancing_and_Region_Splitting, August 2010.

[36] Pastry : P2p dht system. http://www.freepastry.org/pubs.htm, August 2010.

[37] Planetside mmofps. http://planetside.station.sony.com/, August 2010.

[38] Planetside wikipedia entry. `http://en.wikipedia.org/wiki/PlanetSide`, August 2010.

[39] Quake live. `http://www.quakelive.com/`, August 2010.

[40] Raknet networking engine. `http://www.jenkinssoftware.com/`, August 2010.

[41] Red dead redemption : multiplayer features. `http://www.rockstargames.com/reddeadredemption/features/multiplayer`, August 2010.

[42] Replicanet networking middleware. `http://www.replicanet.com/`, August 2010.

[43] Run length encoding. `http://en.wikipedia.org/wiki/Run-length_encoding`, August 2010.

[44] Runescape. `http://www.runescape.com/`, August 2010.

[45] Ryzom engine. `http://dev.ryzom.com/`, August 2010.

[46] Second life official homepage. `http://secondlife.com/`, August 2010.

[47] Second life: Scalability through per-resident subdivision of the grid. `http://wiki.secondlife.com/wiki/AWG_Scalability_through_per-resident_subdivision_of_the_Grid`, August 2010.

[48] There official homepage. `http://www.guildwars.com/`, August 2010.

[49] Travian browser game. `http://www.travian.com/`, August 2010.

[50] Unity game engine. `http://unity3d.com/`, August 2010.

[51] Unrealengine 3 atlas technology. `http://www.epicgameschina.com/tech/tech-atlas_overview.html`, August 2010.

[52] Vast : scalable p2p network for nves. `http://vast.sourceforge.net`, August 2010.

[53] World of warcraft official homepage. `http://www.worldofwarcraft.com/`, August 2010.

[54] Xna creators club. `http://creators.xna.com/`, August 2010.

[55] Xna netgrove traffic analyzer. `http://creators.xna.com/en-US/article/gdc2009presentations`, August 2010.

[56] Sudhir Aggarwal, Justin Christofoli, Sarit Mukherjee, and Sampath Rangarajan. Authority assignment in distributed multi-player proxy-based games. In *NetGames '06: Proceedings of 5th ACM SIGCOMM workshop on Network and system support for games*, page 5, New York, NY, USA, 2006. ACM.

[57] S. Benford, J. Bowers, L.E. Fahlén, and C. Greenhalgh. Managing mutual awareness in collaborative virtual environments. In *Virtual reality software & technology: proceedings of the VRST'94 Conference, 23-26 August 1994, Singapore*, page 223. World Scientific Pub Co Inc, 1994.

[58] Y.W. Bernier. Latency compensating methods in client/server in-game protocol design and optimization. In *Game Developers Conference*, 2001.

[59] P. Bettner and M. Terrano. 1500 archers on a 28.8: Network programming in age of empires and beyond. *Presented at GDC2001*, 2:30p, 2001.

[60] J.S. Boulanger, J. Kienzle, and C. Verbrugge. Comparing interest management algorithms for massively multiplayer games. In *Proceedings of 5th ACM SIGCOMM workshop on Network and system support for games*, page 6. ACM, 2006.

[61] Olivier Cado. Propagation of visual entity properties under bandwidth constraints in the ryzom engine (2007). `http://www.gamasutra.com/view/feature/1421/`, August 2010.

[62] L. Cardelli. Abstractions for mobile computation. *Secure Internet Programming*, pages 51–94, 1999.

[63] John Carmack. Quakeworld network implementation. `http://www.fabiensanglard.net/quakeSource/johnc-log.aug.htm`, August 2010.

[64] Jimmy Cleuren. Schaalbare genetwerkte virtuele omgevingen. Master's thesis, Hasselt University, 2011.

[65] G.F. Coulouris, J. Dollimore, and T. Kindberg. *Distributed systems: concepts and design*. Addison-Wesley Longman, 2005.

[66] J. Diaz Pineda and C. Palau Salvador. On using content delivery networks to improve mog performance. *International Journal of Advanced Media and Communication*, 4(2):182–201, 2010.

[67] N. Ducheneaut and R.J. Moore. The social side of gaming: a study of interaction patterns in a massively multiplayer online game. In *Proceedings of the 2004 ACM conference on Computer supported cooperative work*, pages 360–369. ACM, 2004.

[68] Dan Esbensen. Online game architecture: Back-end strategies (2005). `http://www.gamasutra.com/view/feature/2242/` GDC 2005 Proceedings, August 2010.

[69] B. Ford, P. Srisuresh, and D. Kegel. Peer-to-peer communication across network address translators. In *USENIX Annual Technical Conference*, pages 179–192, 2005.

[70] Network Working Group. Rtp: A transport protocol for real-time applications. `http://www.ietf.org/rfc/rfc1889.txt`, August 2010.

[71] T.Y. Hsiao and S.M. Yuan. Practical middleware for massively multiplayer online games. *IEEE Internet Computing*, pages 47–54, 2005.

[72] J. Kim, J. Choi, D. Chang, T. Kwon, Y. Choi, and E. Yuk. Traffic characteristics of a massively multi-player online role playing game. In *Proceedings of 4th ACM SIGCOMM workshop on Network and system support for games*, page 8. ACM, 2005.

[73] B. Knutsson, H. Lu, W. Xu, and B. Hopkins. Peer-to-peer support for massively multiplayer games. In *IEEE INFOCOM*, volume 1, pages 96–107. Citeseer, 2004.

[74] E. Ko, D. An, and I. Yeom. Dealing with sudden bandwidth changes in tcp. In *Proceedings of IEEE international communications conference (ICC)*, pages 3007–3011. Citeseer, 2008.

[75] Rick Lambright. Distributing object state for networked games using object views. `http://www.gamasutra.com/view/feature/2948/distributing_object_state_for_.php`, August 2010.

[76] W. Lamotte, P. Quax, and E. Flerackers. Large-scale networked virtual environments-architecture and applications. *Campus-Wide Information Systems*, 25(5):329–341, 2008.

[77] Wim Lamotte. Networked virtual environment course. `http://didactiekinf.uhasselt.be/gvo/`, August 2010.

[78] Wim Lamotte. Technology and tools for multimedia systems course. `http://didactiekinf.uhasselt.be/tms/`, August 2010.

[79] M. Mauve. How to keep a dead man from shooting. In *Interactive Distributed Multimedia Systems and Telecommunication Services*, pages 741–779. Springer, 2000.

[80] B.A. Nardi, S. Ly, and J. Harris. Learning conversations in world of warcraft. In *40th Annual Hawaii International Conference on System Sciences, 2007. HICSS 2007*, pages 79–79, 2007.

[81] D.L. Neyland. *Virtual Combat: A Guide to Distributed Interactive Simulation*. Stackpole Books, 1997.

[82] Tateru Nino. Mmogs and virtual worlds: Hidden costs (2008). `http://www.massively.com/2008/10/22/mmogs-and-virtual-worlds-hidden-costs/`, August 2010.

[83] P. Quax, T. Jehaes, P. Jorissen, and W. Lamotte. A multi-user framework supporting video-based avatars. In *Proceedings of the 2nd workshop on Network and system support for games*, pages 137–147. ACM, 2003.

[84] P. Quax, P. Monsieurs, W. Lamotte, D. De Vleeschauwer, and N. Degrande. Objective and subjective evaluation of the influence of small amounts of delay and jitter on a recent first person shooter game. In *Proceedings of 3rd ACM SIGCOMM workshop on Network and system support for games*, pages 152–156. ACM, 2004.

[85] Peter Quax, Bart Cornelissen, Jeroen Dierckx, Gert Vansichem, and Wim Lamotte. Alvic-ng: state management and immersive communication for massively multiplayer online games and communities. *Multimedia Tools Appl.*, 45(1-3):109–131, 2009.

[86] S. Singhal and M. Zyda. *Networked virtual environments: design and implementation*. Addison-Wesley Reading, MA, 1999.

[87] S.K. Singhal and D.R. Cheriton. Exploiting position history for efficient remote rendering in networked virtual reality. *Presence: Teleoperators and Virtual Environments*, 4(2):169–193, 1995.

[88] Anthony Steed and Manuel Fradinho Oliveira. *Networked graphics: building networked games and virtual environments*. Morgan Kauffman, 2009.

[89] VALVe. Source engine multiplayer networking. `http://developer.valvesoftware.com/wiki/Source_Multiplayer_Networking`, August 2010.

[90] D.J. Van Hook, J.O. Calvin, and D.C. Miller. A protocol independent compression algorithm (pica). *Advanced Distributed Simulation Project Memorandum 20PM–ADS–005, MIT Lincoln Laboratories, Lexington, Massachusetts*, 1994.

[91] M. Wijnants, T. Jehaes, P. Quax, and W. Lamotte. Efficient transmission of rendering-related data using the niproxy. In *Proceedings of the IASTED International Conference on Internet and Multimedia Systems and Applications*, pages 162–169. Acta Press, 2008.

[92] M. Wijnants and W. Lamotte. Fec-integrated network traffic shaping using the niproxy. In *2009 First International Conference on Emerging Network Intelligence*, pages 51–60. IEEE, 2009.

[93] Maarten Wijnants. *Service Quality Improvement and User Experience Optimization by Introducing Intelligence in the Network*. PhD thesis, Hasselt University, Expertise centre for Digital Media (EDM), May 2010.

[94] Maarten Wijnants and Wim Lamotte. The niproxy: a flexible proxy server supporting client bandwidth management and multimedia service provision. In *Proceedings of the 8th IEEE International Symposium on a World of Wireless, Mobile and Multimedia Networks (WoWMoM 2007)*, Helsinki, Finland, June 2007.

[95] CK Yeo, BS Lee, and MH Er. A survey of application level multicast techniques. *Computer Communications*, 27(15):1547–1568, 2004.