# Taking your game online:
## Fundamentals of coding online games

Joost van Dongen

7th July 2005

Website: www.oogst3d.net
E-mail: tsgoo@hotmail.com

## Abstract

This article is an introduction to programming the online functionality for a game. We take a look at the architecture to use, which will be either client/server or peer to peer, and discuss whether to use TCP or UDP as the protocol for sending messages over the internet. We then proceed to show an example of how an online game may work, including what messages to send and which computer is responsible for deciding what. This basic online game is then improved with extrapolation and a more efficient use of bandwidth. Finally, fending off cheaters and the use of libraries to make creating online games easier are briefly discussed. This article does not focus on the low-level technical details of programming online games on a specific platform, nor does it focus on creating fun games or beautiful art.

# Contents

# 1 Introduction

Today many game design companies are creating online games. Doing so is not an easy task: lots of computers must communicate and decide what is happening in a virtual world. Which computer is responsible for what? What messages are sent over the internet to which player? After that has been settled, the real problems begin: the game must run smooth even on relatively old computers with slow internet connections. Taking all of this into account, it is clear that programming an online game is a complex job. In this article we will see the basics of programming online games and the ideas behind it. The focus will fully be on programming and not on creating a game that is fun or innovative. Also, there will not be much technical detail: actual coding is different for each programming language and operating system, so this article takes a look at a higher level of abstraction: what messages are being sent, who is responsible for what, how can things be optimised. However, before diving into online game programming, let's see why so many companies are creating internet games.

### Games are big business

Online games are big business. One of the greatest successes so far is the pc game World of Warcraft by Blizzard, which has over two million subscribers [1]. Unlike most other games, World of Warcraft requires players to pay a monthly fee. This fee varies from eleven to thirteen euro per month [2], earning Blizzard over 22 million euro each month. A recent addition to the world of online gaming is the console market. Playstation 2, X-Box and Nintendo Gamecube have online gaming services since fall 2002 [3]. The successors of these consoles, expected to reach the marked before the end of 2006, will focus strongly on multiplayer gaming. One of the most famous online console games is Bungie's Halo 2 for Microsoft's X-Box gaming console. Halo 2 made one hundred million dollar on its first day [4], which is more than any movie ever made on the day of its introduction. Although Halo 2 has a singleplayer mode, it is intended to be played online.

The facts above all show the same thing: online gaming has grown huge. A lot of money is at stake and publishers will probably demand online gaming as a feature in almost every game being made. Programming such online games is very hard and influences many aspects of how the entire game is coded. If a game is coded without taking online gaming into account, it might take a lot of time to add this feature afterwards. For this reason, every game programmer should have some knowledge about programming for online enabled games.


Figure 1: *Screenshots from World of Warcraft and Halo 2. Both games were very successful and allow many players to play together and against each other over the internet.*

### What we will see

This article is an introduction into the basic aspects of online coding. It starts by explaining the demands for different types of games, followed by the most used architectures, like peer-to-peer and client/server, and an explanation of which internet protocol to use. After that we will see a practical example of how an online game may be structured. This basic example works, but is not really playable, so after that it is improved to become smooth and efficient. Finally, two topics are briefly discussed: preventing cheating and using libraries to ease the job. Especially cheating is a very important subject for online games, because it will make the player's gaming experience much less fun. However, this topic is too large to discuss fully in this article and we will therefore only give a brief introduction into fending off cheaters.

# 2 Bandwidth and latency

Before starting off, two important concepts need to be clear to the reader, as they will be used a lot throughout the rest of this article.

## 2.1 Bandwidth

The first one is bandwidth, which is the number of bytes that can be sent from one computer to another per second. This can be either over the internet or on a local network in a company, although the bandwidth of the internet is usually much lower. Bandwidth is very important, as we cannot exceed the maximum bandwidth and this is a strong limitation on communications for online gaming. If a game tries to send too much data, this might take several seconds and will slow down communications. Therefore limiting traffic is very important. What this maximum bandwidth exactly is, differs from one internet connection to another. Today it usually varies from twenty kilobytes per second to five hundred kilobytes per second [5]. On some connections this may be even more, while older computers might still have a limitation of only five kilobytes per second.

## 2.2 Latency

The other important concept is latency. Latency is the time it takes for one computer to send a message to another computer. If a message is sent over a local network, latency may be as low as five milliseconds, while over the internet it can rise to one second, which is extremely long in the world of computers. As we will see later, it can even rise to as much as 50 seconds, but this is extremely rare. For the internet, we can generally expect latencies of somewhere between 25 en 200 milliseconds. These numbers may be verified by playing first person shooters online, which usually show the latency for each player. In some cases the term "ping" is used in stead of latency. Ping is the time it takes a message to get from one computer to another and back. Latencies tend to vary a lot from moment to moment, so in some cases it is more useful to use the average latency over the last few seconds instead of the latency for a single message. Latency is an important issue when creating online games, as latency makes it impossible to make different computers do the same thing at exactly the same time.

# 3 Identifying the requirements of the game

Requirements are the basis of what is being developed, so the first thing we will discuss is what performance a specific game demands. Is the game fast-paced or actually quite slow? Will there be a few or many messages to send each second? These and other points strongly influence almost all

other design decisions. The first thing to decide is the type of gameplay for the game. Online games can be categorized into three types, each with their own characteristics [6]:

1. Play by mail
2. Turn based games
3. Real-time games

### 3.1 Play by mail

Play by mail means that players act and then press a button to have their moves sent to the other players. This type of game is very slow and a player might actually stop playing and continue the next day. Games of this type typically take days or even weeks or months to finish. An example of this is chess-by-mail, where one player makes a move, sends his move to his opponent using a letter and receives his opponent's next move a couple of days later. If this is the type of game being developed, creating an online game becomes quite easy. There is no need to worry about bandwidth and latency problems; even keeping a connection open with other players will not be necessary. The only thing that does not get easy here is preventing cheating. Unfortunately, in modern gaming hardly any games are played by mail, so the simplicity of this genre is not enjoyed much.

### 3.2 Turn based games

A turn based game is quite a bit like play by mail, only with a much higher speed. Turns may take minutes in some cases, but in most cases only take about ten seconds. Again bandwidth and latency are hardly a problem here, as the player expects the game to play in turns and not continuously. Although most games today are not turn based, quite some real time games may be transformed into turn based ones. Age of Empires for instance seems real time to the player, but internally is actually turn based, which made programming it a lot easier, as is explained by Paul Bettner and Mark Terrano in their article on the professional game design website gamasutra.com [7].

### 3.3 Real time games

The final type of online games is real time games. These are games where the player can take action at any moment and actions must immediately have effect. The pace of such games may differ from the relative slowness of a realistic flight simulator to the incredible speed of a first person shooter like Quake III. As player actions must have effect immediately, latency is a very important problem here and we will have to take measures to compensate for the slowness of the internet. Also, lots of things may be happening at the

Figure 2: *Although in Ensemble Studios' game Age of Empires gameplay seemed smooth to the player, it was actually implemented turn-based with about five turns per second.*

same time, so keeping bandwidth usage as low as possible is very important here. In general, the faster the game, the harder it is to get the online game to run smoothly. In this article we will focus on real time games because they are the hardest and therefore most interesting type of online game to program.

### 3.4 Other requirements

After we have identified the type of online play, some more things will have to be decided in order to know the demands of a game. First comes the number of players that will play simultaneously. The internet does not have a way to do effective multicasting (i.e. sending the same message to more than one computer), so if all players should know about something, we will have to send the same message to every player. This can take much bandwidth if many players are online, so we will need ways to minimize bandwidth.

Another thing that might use a lot of bandwidth and therefore needs to be identified before starting to code, is the number of interactive entities in the world. If the player controls lots of characters, this might take a lot of communication. Finally, we must identify what hardware and internet connection our target audience has. If we know that part of the users still has old 56k modems, this will strongly limit our online possibilities.

# 4 Choosing the best architecture

Once we know what the demands of our online game are, the next step is to choose the architecture for the communications. This can be either peer to peer or client/server. Many variations exist for specific game types with special requirements, but we will only discuss these two here.

## 4.1 Peer to peer

Peer to peer is the simplest architecture to understand. Every player connects to every other player. If an event occurs at one computer, this computer will send a message to every other computer. The idea is very simple, but it results in some serious problems. The greatest problem is that we will have to send each message to every other player. This means that if there are N players, this will require N-1 messages for a single event, which will quickly fill all the available bandwidth. This also requires N2 connections for the entire game, as every computer needs to be connected to every other computer.

If the number of players is low, this may be acceptable, but bandwidth is not our only problem. Which computer is to calculate what? For example, if each computer calculates collisions, then it might be that one computer decides there is a hit and another computer decides there is a miss. This can happen, as latency creates small differences between the states of the world at different computers. If there are different outcomes, which computer is right? This will require some smart design to solve.

Peer to peer does have some great benefits, though. If one player disconnects, the other players can continue playing, as no one player is the leader. Also, calculations can be spread, so no one computer will have to calculate collisions and events for the entire world. If our virtual world is very large, this is a great benefit. Finally, each player is running the game in the same way, so the programmer will not have to write different code for the server and the clients, as is necessary in the client/server architecture.

## 4.2 Client/server

Although peer to peer has some important benefits, client/server is the most used architecture in today's games. In client/server, each player connects to one specific server, which is the leader of the game. This server may be either a normal player whose computer has a special role, or an external server run by the game publisher.

Client/server has a number of benefits, but also some problems. The greatest benefit is that the required bandwidth is low on all the clients, as they will only have to communicate with the server. The required bandwidth for the server however will be quite high, but possibly not much higher than

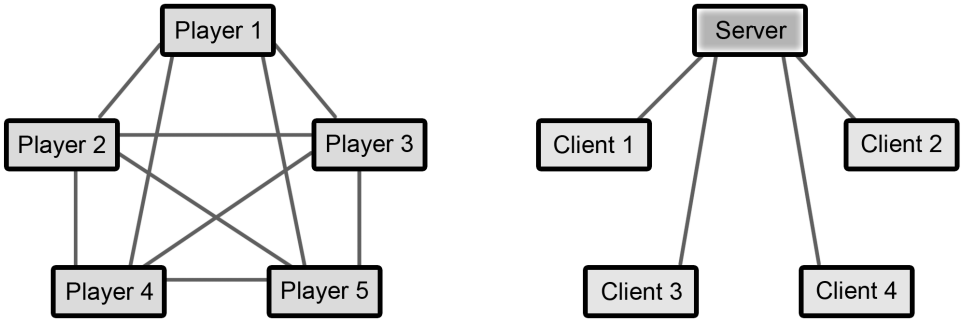


Figure 3: *On the left peer to peer, on the right client/server. Grey lines indicate connections over which messages are being sent.*

for any computer in peer to peer. The smart thing to do is to choose as the server the computer with the highest bandwidth, or have an external dedicated server at the game design company that has a very high bandwidth. Another benefit of client/server is that the server is the leader, so only the server does important calculations like collision detection. The server then just tells all the other computers what has happened and they can update their game state. This way responsibilities are clear: the server is always right.

Client/server has its downsides too. The first one is that it requires a lot of bandwidth at the server, which may not always be available. If the server is run at the game publisher, this is no problem, but running dedicated servers is very expensive and will have to continue for years after the game was first released. If a normal player is the server, there is another problem: when the server disconnects, the game ends. We might solve this by making another player become the server when the server disconnects or his bandwidth drops too much, but this requires some clever coding.

Which architecture is best varies from game to game, although client/server does win in most cases and we will use it in the rest of this article. Choosing the right architecture should be a well considered decision.

## 5 TCP versus UDP: choosing the right protocol

Any online game will have to send messages from one computer to another. Such communication over the internet can be done using one of two different protocols: UDP and TCP. The reason we need a protocol, is that computers will have to communicate with each other and using a protocol makes that possible. TCP and UDP handle getting messages from one computer to another over the internet or a network. Both have different properties, so the right one should be used. We do not really have a choice here as one is

clearly better than the other, but first let's see what the characteristics of UDP and TCP are.

## 5.1 TCP

Let's start with TCP. TCP is the standard protocol for the regular internet and is used, for instance, when a website is viewed in a browser. TCP makes sure that if one computer sends a message to another, this message will reach the other computer.

However, the internet is a big bad place where messages get lost all the time. A message is sent from one computer to another and sometimes somewhere on the road it just disappears. The great thing about TCP is that it makes absolutely sure that if a message is sent, it also arrives [8]. This is done by sending acknowledgements upon receipt of a message and resending information if no acknowledgement is received. This is a great thing, but it may result in extreme latency. For every message an acknowledgement is needed and resending takes a lot of time. Due to this, with TCP we may occasionally get a latency of several seconds when a message needs to be resent several times before it arrives. Very rarely it gets even worse: if the path from one computer to another is clogged somewhere, messages will be lost. Resending messages a lot of times will make the connection even more clogged, so TCP waits a while before resending. This is great for resolving clogged connections, but is terrible for the latency. In his article about creating multiplayer functionality for X-Wing vs. Tie-Fighter, Peter Lincroft reports occasional latencies of up to fifty seconds due to clogged connections [9]. This will simply kill almost any game.

TCP also provides for another great service that is bad for games: it makes sure that messages are received in the right order. So if one message has been delayed for a few seconds, all other messages will be kept waiting until that one message has been received. The last aspect of TCP we will mention here is that it requires a connection before it sends data. So if two computers need to communicate, they first establish a connection and then start sending the data.

## 5.2 UDP

So, now that we know that TCP is useless for games because of the sometimes extreme latency, what is our alternative? In some respects, the alternative is even worse: UDP. UDP does not guarantee us anything at all. It attempts to send the message and if the message reaches the other computer, then that is great, but if it does not, then UDP does nothing at all to resend it. It does not inform the game that the message was lost and it does not resend the message. So if it is really necessary that someone receives a message, we will have to write our own acknowledgement system for that

message. When a game is coded well, however, most messages can be lost without causing problems. For instance if a message contains the position of an object and this position is sent ten times per second, then it is not really a problem if one message is lost: another message comes in shortly after and contains all the information needed. As UDP does not wait for the path to clear and does not try to resend anything, you will never get really big latencies and if your game is coded well, lost packages do not need to be a very big problem.

Because UDP guarantees nothing, it also does not guarantee that messages are received in order. If one message is delayed a few seconds, it may arrive after other messages that were sent later. This may confuse our game, so we should use a counter to recognise delayed messages and be able to delete them.

UDP does not need a connection, meaning that data can immediately be sent. However, this means that anyone can send UDP messages to anyone, allowing for outsiders to interfere in a game. Wrong and fake message sent by hackers need to be taken care of to prevent hackers from taking over.

### 5.3 Headers

For each packet sent, the protocol adds some bytes with extra information. For TCP this is about 40 bytes and for UDP this is about 28 bytes per message [8]. Once a connection has been established, TCP can use much smaller headers. The result is that on average TCP has smaller headers than UDP. The existence of headers also means that sending lots of small messages should be avoided: each one will require a header, strongly increasing the required bandwidth when messages are too small. The solution to this is simple: send more data per message instead of separately.

### 5.4 Conclusion

So, in the end, TCP is not really an option for games. Everyone uses UDP and there is no way around it. We should take with lost or delayed messages, though, as they might spoil our game.

## 6 A practical example

Once we have settled the requirements for our game and we have chosen the basic architecture and the right protocol, actually designing the code for our online game can begin. Exactly which message will be sent to which other computer? Which computer is responsible for what? There is not one right answer to these questions as this differs from game genre to game genre and from game to game. Even if it is exactly clear what the requirements for

a specific game are, specialists might still argue about the exact implementation. In this chapter we will show one way of how an online game can be made. For any specific game the choices made in this chapter should be reconsidered.

The example we will see in this chapter is for a real time game with very fast gameplay. Our game will be played by only a small number of players at the same time: the maximum is 16 players or on very fast connections maybe 32 players. Our game uses UDP and is made using the client/server architecture where one special player is the server. To keep things simple, our game ends if the server quits.

## 6.1 Responsibilities and communication during gameplay

In our example, the server is responsible for everything. If a player tries to fire a rocket, the server checks whether this player has enough ammo to do so and launches the rocket. After this, the server calculates the movement of the rocket and checks whether it collides with another object. If it does so, the server creates an explosion and adds damage to players close enough to the explosion. The server also keeps track of the score and handles starting and ending the game.

While the server is responsible for everything, it cannot handle user input from all players, as only one player is the server. Therefore, the clients will all handle the user input for their players themselves. The actions of the players however are not performed by the clients, but instead are sent to the server, which then proceeds to perform them. So if a player tries to walk to the left, the client he is playing on will send a message to the server and the server will move the player to the left if this is allowed.

To be able to play our game, all players will need the see and hear what is happening in the game world. Drawing the world to the screen and playing sound files is done by the clients and to do so they keep track of the current state of the virtual world. The server makes sure they can by sending messages to all clients about the game world. This includes messages like "player one is walking with speed x in direction y from position z," and "a bonus item has just popped up on position x." When a client receives a message, he updates the state of his virtual game world and uses this to play sounds and draw the player's view.

## 6.2 Reliable and unreliable messages

We send all messages using UDP, but some messages do need reliability. For instance if one player launches a rocket, all players must add a rocket to their game state. If this message is not received, all further messages that update this rocket cannot be processed as there is no rocket to update. This divides messages into two groups: those that we must send reliably and those that

may be lost. To send messages reliably using UDP, we should create our own code for sending acknowledgements upon receipt and resending messages if they did not reach the other computer.

To improve performance, the programmer should take care to send as many messages as possible unreliable. An example of how this can be done is once more the rocket. If we send the movement since the last frame, a lost package will be a problem: the exact position of the rocket is the sum of the starting position and all the small movements. If instead all packages contain the actual position of the rocket, it is no problem if a message is lost. A new message will come shortly after, which will correct the position of the rocket once more.

### 6.3 Finding and starting an online game

We have now settled communications during gameplay, but how to start a game? Players need to know which other players want to play and they should be connected to play together. The simple solution to this is that one player hosts a game and calls his friends to tell them that he has done so and what the IP-address of his computer is. Now the friends of the host player type the IP-address in the game and the game can begin. In the past, this was actually the way starting an online game was done. However, it is very awkward and does not allow players to play with people they do not know.

So, let's look at the solution that is used today: the matchmaking server. A matchmaking server is a server run by the game company that does nothing more than connecting people who want to play with each other. The domain name of the matchmaking server is added to the game during development. When a player starts a server, his game uses this domain name to tell it that a new game has been created. When another player wants to join a game, his game uses the domain name to ask the matchmaking server for a list of all games that can currently be joined. The player now chooses the server he or she wishes to join and starts playing. The only drawback to this is that the matchmaking server must run all the time. This costs money, but fortunately it is not very expensive: the matchmaking server does not have to host any games. It only has to keep track of which games are being hosted at the moment, which takes much less bandwidth and processing power.

We have our basic game working now. In the next chapter we will improve it to decrease bandwidth usage and smoothen the jerking animations due to latency.


Figure 4: *Battle.net is an advanced matchmaking server created by Blizzard for most of their games. This screenshot shows a menu to choose an online game to join in Warcraft III. Battle.net does not only connect players who want to play together, it also automatically updates the game with the latest patches and keeps track of the victories of each player to create a list of highscores.*

# 7 Ways to improve performance

In the previous chapter we have seen an example of how to program the online aspect of a game. Unfortunately, the choices made so far will not result in a playable game: movement is jerky as it only happens when the server sends a message to update the position of an object. Bandwidth usage may also be very high when sixteen players are playing at the same time.

## 7.1 Smooth animations with extrapolation

In our design so far a client updates the position of an object only when it receives a message from the server. The number of messages received each second will usually be somewhere between four and twenty messages per second. This is not enough: to make our game feel totally fluent and fast, the number of times the screen is updated should be somewhere around fifty frames per second. The exact minimum is a point of discussion among gamers, but thirty frames per second is a bare minimum. Therefore, something will need to be done between the receipt of two messages. The wrong solution is to simply send more messages, because this will take too much bandwidth and at times the internet will only pass a few per second anyway.

There is good solution to this, though: extrapolation. With extrapola-

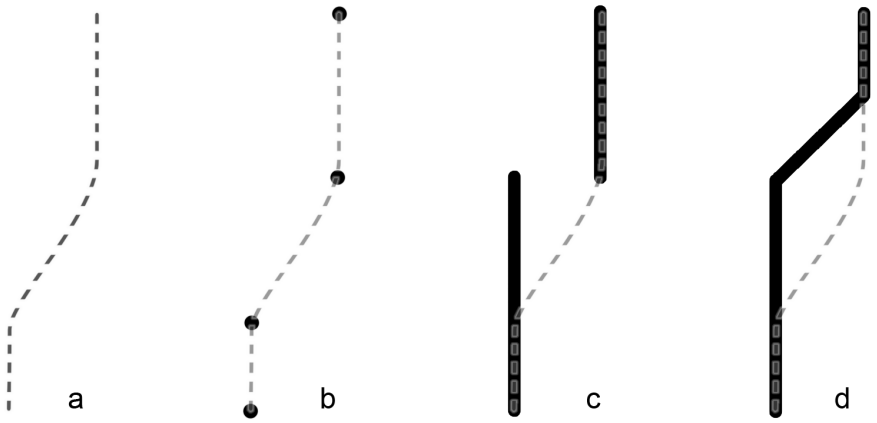


Figure 5: *Extrapolation. The dashed line is the actual movement of the object on the server. a. Only the movement on the server. b. The large dots show where the object stands on the client after a message is recieved. No extrapolation is done, so the object jumps from one position to another. c. Extrapolation is used, so the object moves forward fluently and jumps to the correct position when a message is received. d. Extrapolation and smoothening are used. The object fluently corrects its position when a message is received.*

tion clients update the state of their virtual world many times per second with a guess about what will happen. When a message arrives from the server, the client will correct this guess with the data from the server. For example, if another player is running forward, the client will guess that the player will keep running and will move the player forward a little each frame. At the same time that player really started turning left, so on the server this player turned left. When the client receives a message with this new position, it corrects its guess with the real position.

Our basic implementation of extrapolation will still produce some flickering in the animation: when a guess was wrong, it is corrected at the instant a message is received and this may make an object jump to a new position. We can solve this using smoothing. This means applying the correction in small steps, spread out over a few frames. The flickering will now be completely gone. Also, remember that the interval at which messages are received is very short, so the error of the guesses will always be small.

A more advanced form of extrapolation is dead reckoning, developed by the United States Department of Defence [10]. Here the client does normal extrapolation, while the server does something more advanced. The server keeps track of two versions of the virtual world: how it really is and how a client that uses extrapolation sees it. Only if the real version and the

extrapolated version differ too much is a message sent to the client. So if the standard guess is that a player keeps running forward, the server will only send an update to the client if the player did not run forward. This greatly reduces bandwidth usage, although care should be taken as packets may be lost and therefore the server may have a wrong idea of what the client knows.

### 7.2 Latency on player actions

Our graphics look fluent now, but reactions to user input are still a bit slow. This is due to the fact that when a player acts, his action is first sent to the server, which processes it, then returned to the client and only then is its result shown to the player. When latency is high, the player might notice this. A solution to this is giving the client more responsibilities, although this is a tricky solution, as the responsibilities of the server and each client must be very clear. In the example here we will not do so, as this article is only about the basics. Instead, it is important to note here that the problem is smaller then it may seem: latency is not very high on modern internet connections, so if our game is not extremely fast, players will usually not notice the delay.

### 7.3 Reducing bandwidth usage

So far we have not said much about bandwidth. Some policy and care to keep bandwidth usage low is necessary though. In this section we will see some of the many improvements possible. The first thing to notice here is that messages with updates should not be sent at every iteration of the game loop. The game may draw to the screen up to 80 times per second, while messages should not be sent that often. Therefore, we keep track of the elapsed time and only send a maximum number of updates each second. The number of updates per second will vary for different game types and different internet connections, so an exact amount per second cannot be given here.

Another improvement is to combine several messages into one. Many events can be put into one message instead of separate messages for each event. This will save us a lot of bandwidth as each message has some overhead due to its header.

Many moving objects in our game world are only decorations and have no influence on the actual gameplay. We can use this to optimise bandwidth usage a bit further: such objects can be fully calculated by the clients and need not receive updates from the server, or maybe only very occasionally. Examples of this are the movement of clouds and the waves in the water. Some other things only need to receive an update occasionally. If a plane flies by in the air, this plain should fly by on each client, as players may

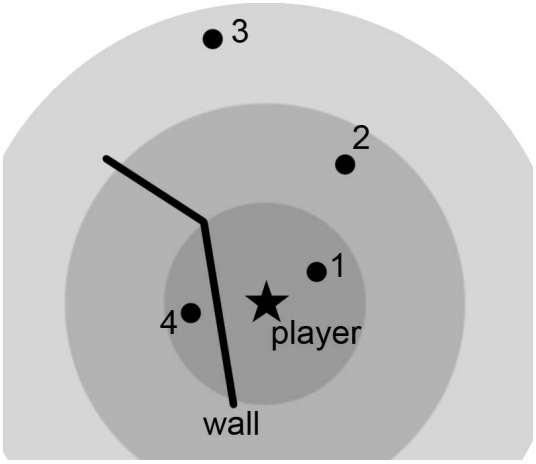


Figure 6: *In large areas, only objects that are close to the player need to be updated frequently. Here object 1 is very close and therefore must be updated most frequently, while objects 2 and 3 may be updated less frequently. Object 4 is behind a thick wall, so the player need not receive information about it at all.*

be sitting in the same room and may notice that there is a plane on one computer and not on another. The exact position however is not important, so one update-message every few seconds is enough. Some other events are triggered, like explosions. The exact animation of an explosion is not important, but the explosion should happen at the same time on each client. This can be done by only sending a message when the explosion starts.

If our virtual world is very large, like in a massive multiplayer game, players will only see a small part of this world. The server can keep track of what every client sees and only send information that is relevant to the player. Information is relevant if the player may see or hear it, or if the player may be hit by it. We can add to this that things that are close to the player should be updated more frequently than things that are far away. Even if the player can look very far, the precision of the objects in the distance need not be high.

One final optimisation is to compress data. We can do this in many ways, for instance by giving frequently used text message a short code that can be sent instead of the actual message. A more sophisticated method is using Huffman encoding, which replaces frequently occurring byte-sequences with shorter ones and less frequent byte-sequences with longer once. This may result in savings of 20% to 90% [11], although 20% will be rare with the relatively short messages in games.

So far we have only discussed a few of the many possible optimisations and improvements to lower bandwidth usage and improve the smoothness of

the gaming experience. Many more exist and can be devised. Programmers are advised to take the time to come up with their own optimisations for specific games, as much performance can be gained here.

# 8 Cheaters and libraries

In the last chapter we have made an online game that works and keeps within bandwidth limits. However, our game is still unfinished, for hackers will be able to spoil our game easily by cheating. Another subject that has not been discussed so far is libraries that help create online games. These topics are both too large to fully discuss in this article, so only a short introduction will be given here.

## 8.1 Beating cheaters

Most online games are based on competition: players play against each other and try to achieve the best score. Competing is only fun when chances are even. If a player always wins because he cheats, his opponents will not enjoy the game anymore and will stop playing. If our game is to be fun, we must prevent cheating as much as possible.

Cheating in games can be done in two ways. The simple way is that a game may contain a bug that can be used to get an advantage. There may for instance be a wall that the player can walk through because collision detection contains an error. This is simply a bug and can be solved by testing he game thoroughly before releasing it.

The other kind of cheating is much harder to beat: hackers. Hackers may change the program, alter parameters, spoil communications or do many other things to cheat. Even if a game is coded perfectly well and contains no bugs at all, hackers may still alter the game to cheat. Defending an online game against hackers is an immensely complex task. For a good introduction into the techniques hackers use to cheat and some defences against them, the reader is advised to read "How to Hurt the Hackers: The Scoop on Internet Cheating and How You Can Combat It" [12] by Matt Pritchard, which is an excellent introduction into the topic.

To show just a little example of the problem, imagine a game where players sneak up on each other in dark dungeons. A hacker may write a small program that alters the video drivers and adds light to this dark dungeon. A player using this cheat will easily win as he will see his enemies coming. The solution to this cheat is to check whether the settings for the video drivers have been changed during gameplay and to reset them if this has happened. However, the hacker may now search for the part of the game that does this check and disable it. Now something will be needed to check whether the video settings inspection mechanism is still working, but hackers may hack that too...

It is not possible to write a game that cannot be hacked: hackers can always find some way to create a cheat. The best we can do is to make hacking as hard as possible and release patches that fix things whenever cheats have been found.

## 8.2 Using programming libraries to ease the work

When we create an online game, we must take care of many things, as we have seen so far. A little help would spare us a lot of time and effort. Fortunately, this help is available and comes in the shape of programming libraries. These libraries offer functionality like sending messages using UDP, both reliable and unreliable, easily creating matchmaking servers, encrypting messages to give hackers a hard time and compressing messages to save bandwidth. Both Playstation 2, X-Box and Gamecube offer programming libraries that help create online games [3]. For the pc there is the open source library RakNet, which can do a lot of work for the programmer [13]. RakNet is free for small products and can be licensed for commercial use. Many other tools and libraries exist to help us and programmers are advised to use them to ease both programming and debugging.

# 9 Conclusion

Multiplayer games are big business and many game design companies are working on them. Creating an online game is not an easy job, as the game must play smoothly without taking too much bandwidth and hackers should be kept off. In this article we have seen the basics of how to program the online functionality of a game. To do so, we started by determining the requirements of our game. After this the basic architecture has been chosen, which usually is client/server. Now we decided which computer is responsible for what and which messages are sent to which player. In almost all games these messages should be sent using the UDP-protocol. When the basics were settled, we made a number of improvements: our game should look fluent, even with high latency, and bandwidth usage must be limited. We used extrapolation, compression and checking what is relevant for which player to do this. Finally, if our game is to be a real success, defences must be added against cheaters.

If all of the above is done well, a great game may be made that will offer players the incredible experience of playing with people from all over the world!

## References


[1] David Jenkins. World of warcraft hits 2 million subscribers. http://www.gamasutra.com/php-bin/news_index.php?story=5696, 2005. (last visited on July 6, 2005).

[2] Blizzard. World of warcraft subscription details. https://www.wow-europe.com/en/requirements/subscription.html, 2005. (last visited on July 6, 2005).

[3] Pete Isensee and Steve Ganem. Developing online console games. http://www.gamasutra.com/features/20030328/isensee_01.shtml, 2003. (last visited on July 6, 2005).

[4] Franklin Paul. Microsoft sees $100 million first day for 'halo 2' game. http://in.tech.yahoo.com/041110/137/2hsn2.html, 2004. (last visited on July 6, 2005).

[5] HCCNet. Abonnementen en prijzen. http://adsl.hccnet.nl, 2005. (last visited on July 6, 2005).

[6] Dan Royer. Network game programming. http://www.flipcode.com/articles/network_part01.shtml, 1999. (last visited on July 6, 2005).

[7] Paul Bettner and Mark Terrano. Gdc 2001: 1500 archers on a 28.8: Network programming in age of empires and beyond. http://www.gamasutra.com/features/20010322/terrano_01.htm, 2001. (last visited on July 6, 2005).

[8] Andrew S. Tanenbaum. *Computer networks*. Pearson Education, Inc., New Jersey, 2003. ISBN 0-13-038488-7.

[9] Peter Lincroft. The internet sucks: Or, what i learned coding x-wing vs. tie fighter. http://www.gamasutra.com/features/19990903/lincroft_01.htm, 1999. (last visited on July 6, 2005).

[10] Jesse Aronson. Dead reckoning: Latency hiding for networked games. http://www.gamasutra.com/features/19970919/aronson_01.htm, 1997. (last visited on July 6, 2005).

[11] Ronald L. Rivest Thomas H. Cormen, Charles E Leiserson and Clifford Stein. *Introduction to algorithms, second edition*. The MIT Press, Massachusetts, 2001. ISBN 0-262-53196-8.

[12] Matt Pritchard. How to hurt the hackers: The scoop on internet cheating and how you can combat it.

http://www.gamasutra.com/features/20000724/pritchard_01.htm, 2000. (last visited on July 6, 2005).

[13] Rakkar (real name unknown). Raknet manual. http://www.rakkarsoft.com/raknet/manual/index.html, 2004. (last visited on July 6, 2005).

The image on the cover is a cartoon version of Halo's Master Chief and is taken from www.halobabies.net