

# Peer-to-Peer Online Games

Seminar Massively Distributed  
System

WS 2006 / 2007  
RWTH Aachen University

Hendrik Thüs  
hendrik.thues@rwth-aachen.de

## Abstract

The current Multiplayer Online Games are suffering from the high number of participating players. The most used approach for Multiplayer Games is the standard client-server system. In this paper, some other systems are presented that are more flexible and scalable than known client-server approaches.

The presented architecture Solipsis does not need any servers. It is very scalable, theoretically it can handle an unlimited number of players, but has other disadvantages.

There are also some approaches that have most of the positive characteristics of the client-server architecture and Solipsis. But, of course, there are some difficulties that have to be solved. One point is the consistency of the game-world that has to be guaranteed. Another one is to minimize the outgoing traffic of the participating clients. And there is cheating. Without any authority, cheating will be much easier. In this paper, some solutions are presented to solve those arising problems.

## 1. Introduction

Massively Multiplayer Online Games (MMOG) are implemented to allow thousands of players to join a single virtual game-world. Each player can theoretically interact with every other player in this world.

More and more people participate in MMOGs. In the future, there will be a great demand on games that can handle millions of players.

The client-server architecture, which is mostly used, is not very flexible. There are some servers that are responsible for the consistency and the development of the game. Only a limited number of players can participate in a game, because of the limited bandwidth and calculation-power of the servers. Since the numbers of players of those MMOGs are increasing very fast (see figure 1), there is a great demand for techniques that are not that

limited to a bounded number of players. Of course, the advantages that the client-server technique offers should not get lost.

In chapter 2, a standard multiplayer game is shown. The characteristics are independent of the genre of the game.

Chapter 3 is about the general differences between the different approaches – with or without centralized servers or with decentralized servers.

There are approaches that do not have such big problems with handling a great number of players. They are more flexible. The advantages and disadvantages of these server-assisted peer-to-peer architectures are shown in chapter 4. Also the characteristics of them are presented.

Chapter 5 presents two methods to share the workload between the participating server more equally.

The problems with which the peer-to-peer approaches have to deal are presented in the chapters 6 and 7.

## 2. Setup of a Multiplayer Game

In nearly every Multiplayer-Game, like Role Playing Games (RPG) – examples are World of Warcraft [1], GuildWars [2] – or First Person Shooter (FPS) – examples are Neocron Evolution [3] or Battleground Europe [4] – the player controls an avatar. This is the representative of the player in the virtual game-world. The players can only see and hear things through their avatar. So the area they notice, is limited by the abilities of the avatar.

The settings of the games can be very different. It can be that the player acts in a fantasy world with dragons and magicians or there can be a world full of aliens and spaceships. There are also non-player characters like monsters, enemies or bots, mutable objects like food, ammo, weapons and mutable and immutable landscape information. Mutable landscape could be a window or a door. The state of these objects can be changed, but they do not affect the abilities or belongings of the player.

Normally, the virtual world is divided into regions or dimensions. The player can travel between them by using portals or doors or something like that. Each region or dimension may have other areas (see chapter 4).

## 2.1 Game states

Textures, some algorithms or something like that are normally included in the client-software. The immutable landscape information, the terrain, can be seen as a 2 or 3-dimensional vector. Mutable landscape information like windows or movable furniture or mutable items just have a position and a status, like “window is broken” or “bottle with water is 40% full”. The player's avatar has to contain more information like its current position, running direction, abilities, health or its possessions. Of course, there can be more things like these, but this depends on the specific game.

Interacting with objects or other players affects the state of every participating object. For example if two players fight against each other, the health of both will decrease. Or if a player drinks water from a bottle, the player is not thirsty anymore and the bottle contains less water.

Some states, like the landscape, will be transferred to the client at the beginning of the game. This will be kept statically until the player leaves the dimension, area or region. But most objects change their state frequently. The simplest and most updated state is the position of a player. But there are many other states like those of the various objects that change their state when they are moved, used or just collected.

## 2.2 State updates

Each object in the world is associated to a so called “think function”. This function updates the objects according to their characteristics and how their surrounding exerts influence on them. In serverbased Multiplayer-Games, the server frequently starts these think functions to update the state of every object in the virtual world. The updated states are send to those players, who are interested in the information. How this can be handled is described in chapter 4.

In FPS games, about 10 to 20 updates are made per second. A good result is archived at around 15 [5]. Slower games like RPGs do not need such a high rate. Here, the main-focus is on the strategic aspect and not on fast gameplay.

But the states are not necessarily stored on a central server to be distributed. There are different approaches of how to handle the data. More about that in the chapter below.

## 3. Different techniques to implement MMOGs

There are different possibilities to implement the network-structure of a game. Every approach has its own characteristics, and so, its own advantages and disadvantages. They differ in scalability, complexity of the implementation and some other points.

## 3.1 Pure client-server

At this time, the most used approach for Multiplayer Games, is the the client-server architecture. This model is used in most cases because it is very easy to implement and easy to be controlled by the company, which implemented the game. The servers are located in huge server-farms. Of course, this is not free of charge for the company, which runs that game. So, the players often have to pay for the use of the Multiplayer-Online-Game.

The servers are used to maintain the current state of the players and of the whole virtual world. This world is able to exist consistently. Players can leave the game and start again at the same point where they left. The current state, like the position of an avatar is sent to the server. Now, the states of the avatars or objects that are interesting to a player, are sent from the server to this player. So, one always knows, what is happening in the near surrounding. But there are other tasks, the centralized servers have to assume. They take care of the account-data of all the players. No unauthorized person should be allowed to use a player or change its abilities or belongings. They also can arrange a chat, where all the participating players can talk with each other. Since the servers are controlled by the producer of the game, it would be an advantage, if the battles between single players or groups are calculated and stored on them. Otherwise, there would be a possibility to cheat.

So, the players connect themselves to these centralized servers. To allow more players to participate in the game, more servers are needed. Another possibility is to divide the virtual world into many parallel worlds, which are not connected with each other. For example one world for Europe, one for Asia, etc.

A client-server based MMOG is not very scalable. The company which implemented the game can only have a limited number of servers. So, the maximum amount of players who can participate in the game is limited by the maximum workload of these servers. Since the number of players are increasing very fast (see figure 1), the maximum amount of players that servers can handle will be reached very soon.

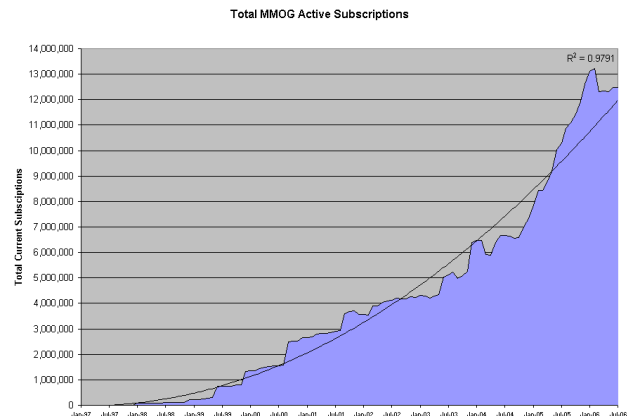


Figure 1: Total active subscribers in MMOGs from January 1997 to July 2006 [7]

The average number of players might not be the problem. If this average number is known, the number of servers or their maximum upload bandwidth can be adjusted to that. That can be done easily. But the peaks are a problem. Here, the servers are confronted with high loads, which could be far beyond the average load. So, the server-based MMOG contains a bottleneck. If more players want to join the game than the servers are able to handle, players have to be rejected, put on queue or have to live with very high response times. The games are often not playable anymore [6].

### 3.1.1 Valuation of suitability

If one considers that there are about 6.6 million people playing World of Warcraft in July 2006 [7], the client-server model is not that suitable. The costs for the servers will increase very fast. This is not a technique that is ready for the future. There will be other MMOGs with even more players participating. Scalability will be a fundamental characteristic of these games.

Of course, the client-server architecture has some characteristics that are not so easy to implement without the assistance of servers. These disadvantages of serverless architectures will be described in the following subchapters.

## 3.2 Pure P2P

The Pure Peer-to-Peer approach does not need any servers. So, there is no authority which sets up the rules of the world or which cares for keeping the status of the objects consistent. But since no servers are participating, these architectures are theoretically scalable to unlimited numbers of players.

One example is the open-source system *Solipsis* [8], which is a mathematical model for a massively multi-participant shared virtual world. The peer-to-peer network of *Solipsis* is modelled by a graph that consists of a set of nodes (also called entities or peers) and a set of connections between those nodes. Each one has a unique ID. The peers are responsible for their own state. They are able so sense a part of the virtual world, which is inhabited by other entities. Neighbours of a node should recognize on their own initiative that the status of the node has changed.

Each peer has a neighbour-list of those peers that are closed-by. This relation is bidirectional, this means that the nodes know one-another. If a peer changes its state, this change will be propagated only to those peers that are in the list of neighbours. So, a peer has to know all the entities that are in a certain region around it. This region is called *Awareness-Area* (see figure 2).

Another requirement, called *Global Connectivity*, for a peer is that it is somehow connected via other entities to every single peer. These rules ensure that a single entity or a group of entities does not get disconnected from the rest of the peers by walking in a region that is not populated.

Since the nodes can freely walk around in the virtual world, the neighbourhood of a node changes frequently. Thus, each node can ask its neighbours if they noticed another, foreign node that

is now located in their neighbourhood. But it is better to do this the other way round. A lot of traffic can be saved, if the nodes inform their neighbours, when a foreign node has entered its area. This information, which is propagated, contains the ID and the current position of the new node.

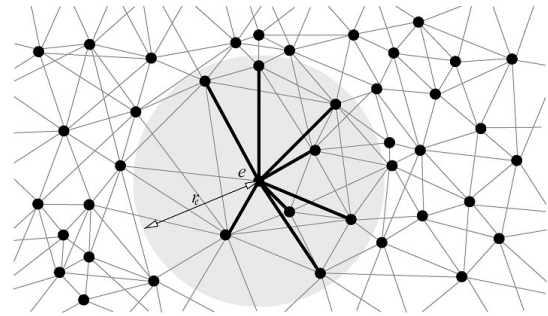


Figure 2: Awareness area in *Solipsis*. The variable  $r$  is the radius of this area [8]. This radius depends on the density of entities in this area and the physical capacity of  $e$ .

Due to the limited resources of a node, it has to drop connections to nodes from time to time. If a peer leaves the awareness-area of a node, this connection is cancelled, because the information, they can give each other, is not interesting anymore. If all the resources of a node are spent but there are although no connections that can be cancelled, other ways have to be found.

The simplest one is to reduce the radius of the awareness-area. Thus, all the connection to the nodes that are outside this area, can be dropped.

Another way is to decrease the number of ways, the node is connected globally. If the node has more than one possibility to theoretically reach every other node, some connections are useless, because others can do the same job equally or even better. But it is difficult to decide, which node(s) can be dropped. Of course, there should be no disadvantage. *Solipsis* [8] knows four algorithms that help to make this decision by ordering the relevant peers:

- The expected lifespan of a node and the amount of messages, the node sends. A node that sends out many messages is expected to life longer than nodes with fewer outgoing messages. Those node with a great expected lifespan are preferred to be kept.
- The size of the awareness-area. A node that knows a large part of the world is preferred to be kept as adjacent.
- The size of the boundary of the awareness-area of a node. A large boundary is advantageous for finding new nodes that enter the awareness-area.
- The size of the convex hull that the adjacent nodes form.

Players should be able to start somewhere in the virtual world. This will be, when the players have just connected to the game or when they appear again after they have been killed. But at the beginning, they do not know the nodes that are in the awareness-area around the desired position. Since this is required by Solipsis, they have to get to know these nodes. The following algorithm is used (see also figure 3):

1. At least one node ( $e_0$ ) has to be known. This is the starting-point.
2. Get the position and ID of that node out of all the nodes, the current peer knows that is the nearest one to the destination.
3. If there is no nearest node (this happens in figure 3 at node  $e_n$ ), choose one of the neighbours.
4. If this neighbour has a node that is nearer to the destination than the current node, just proceed with step 2 and search for other peers. If not, go back and try another neighbour of  $e_n$ .
5. If a node ( $e_n'$ ) is reached that would be a direct neighbour of the destination, just go counterclockwise around the destination from node to node and be aware of the positions and IDs of these nodes. They will be those nodes that form the first step of an awareness-area. Now, they can tell the new peer, which other nodes are inside this awareness-area with the radius, the new node defines.

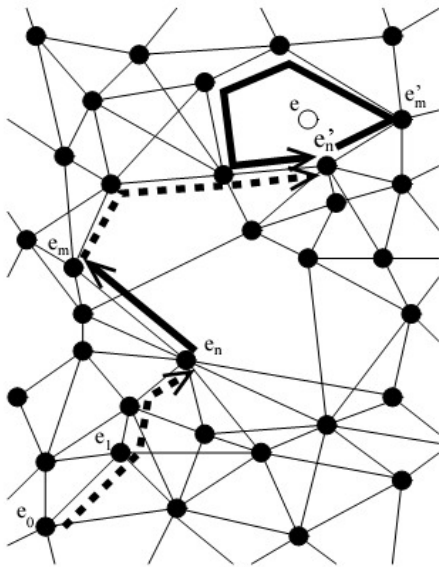


Figure 3: This is how a node  $e$  gets to know its neighbours. At the beginning, this node  $e$  only knows node  $e_0$  [8]

### 3.2.1 Current status of Solipsis

Currently, there is no real implementation of Solipsis. The existing protocol gives a node the ability to know, where it is located in the virtual world. The navigator of a node is able to lead the node to a certain position. And it can also retrieve information about the virtual surrounding of the node. This approach is more theoretical than practical. There is no real game implemented that is already playable. Currently, Solipsis is nothing more than a chat-client.

### 3.2.2 Valuation of suitability

I think that Solipsis is an interesting approach, but it is far away from being used in MMOGs. Theoretically, it has great characteristics, but there are many unsolved problems. For example, there is the cheating that is very difficult to be handled. Or there is no authority, where the data of the players can be stored. Thus, there is no real possibility for accounting. Another point is that there is no real implementation at the moment. The current state is that there is only a chat-client implemented. Thus, it is currently not ready to be used in games.

But there are other architectures that are already implemented in games like GuildWars [2] or Neocron Evolution [3]. These approaches are the Hybrid P2P architectures.

## 3.3 Hybrid P2P

The Hybrid P2P architecture forms the middle course between a client-server system and a pure P2P system. There are servers used in these architectures, but they do not have so many tasks to fulfill. They are mostly used to support the peer-to-peer structure of the nodes.

Which part of the whole peer-to-peer structure is handled by server depends on the approach itself. But mostly, the authorisation (if existing) is made by the servers. So, it can be ensured that only those players, who are authorised can access the game.

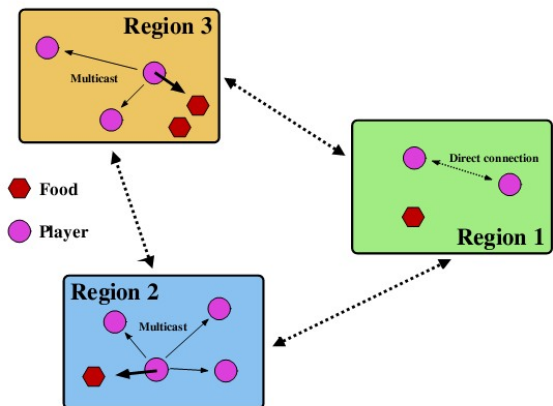


Figure 4: Partition of the world as it is done in [9]

There are approaches [9],[10] where the servers store the persistent player states, like account-information or the avatar's abilities and possessions. The rest of the game-state is completely handled by the peers themselves.

This is done by dividing the world into regions. The players in each region propagate their current state to the others (see figure 4 and chapter 4).

The players themselves take care of the objects (food, ammo, weapons,...) in the world. In these architectures, nothing of that is handled by central servers. Of course, this can be a disadvantage because it offers the possibility of cheating.

Since I think that these hybrid peer-to-peer approaches are very implementable – they combine the advantages of both extreme approaches with only little deduction - I will refer to them in the next chapter.

## 4. P2P Multiplayer Games

Since pure client-server approaches are not that scalable, they lack an important feature for the future. On the other side, the pure peer-to-peer approaches are scalable to a nearly unlimited amount of players. But they do have other disadvantages, respectively they have unsolved problems.

### 4.1 Advantages and Disadvantages

As written above, client-server approaches have a scaling problem. The outgoing traffic is not linear to the numbers of players. If each player is alone in his area-of-interest, it would be linear. But since more players result in more interaction between each other, the traffic increases more than linear [10]. So, it is easy to see that the maximum number of players cannot be increased endlessly. More powerful servers are always needed.

To improve this disadvantage, there are some peer-to-peer-approaches that try to take care of the scalability without losing the advantages of the pure client-server model.

These advantages are that cheating is relatively easy to control. Every action that takes place in the game, will be transferred to the servers. Thus, they can control and verify the changes. If there are irregularities, the centralized server can decide if this was due to cheating or not. But this advantage will be lost – if there will not be further development – in the pure peer-to-peer-approaches. Here, every player takes care of his own avatar and its abilities. The state of this avatar is stored on the clients themselves. It would be very easy to change this data to get better weapons, abilities or something like that. There will be no authority that checks, if the player's avatar has been changed manually.

If there are no central servers, there is also no authority, which checks the authorisation-data of the participants. So, everyone can join the network without being checked. This can only be of advantage in some special applications.

An advantage is that peer-to-peer MMOGs will be much cheaper for the developer because no such expensive server-clusters are needed to handle this amount of players. An advantage for

smaller companies. The players will accept that they have to participate in keeping the game alive by granting some upload resources. An evidence for that are the well known peer-to-peer-filesharing-programs or the fact that most of the game-server for FPS-games are run by private persons or groups. In times of broadband-internet, this necessity is more and more accepted.

### 4.2 Traffic improvement

Since there are only few servers and the possibility is given that some players only use modems, the outgoing traffic of the peers should be minimized. Two of these methods to reduce the traffic that is used to distribute the game state to the players are area-of-interest and delta-encoding.

When a server sends a certain state delta-encoded, it just sends the difference between the current state and the state at the last iteration. But not every update should be sent in this way. It could be that a message gets lost, so other peers would have different knowledge of the current position of an other player. Thus, the current state should be transmitted completely from time to time.

Since the players only know a limited area of the world, they are only interested in a small part of the map. This part is called area-of-interest. So, the player only has to send these information which are important to the other players that are in this area.

Since the position of a player is the most variable thing in a game, it has to be updated very frequently. But it is also possible that the position is only updated when it has been changed. This reduces the traffic but other problems may occur. If an update on the position gets lost, it may be that no other player gets to know about this loss.

Another thing is that the players do not need to know everything about the objects in their area. When they walk through their world, they can only see the appearance of an object, no further information. The content of a closed chest is only important if it is opened.

### 4.3 Area of Interest

The players have only limited moving-speed and they can only see a certain area around themselves. This fact is the reason that every player does not need to know every object in the virtual world. They only receive updates of these objects that are important for them. The other objects can be neglected.

The world is now divided into regions that form these areas-of-interest [9]. The players in these areas distribute their current position to each other. They form a multicast-group. When players move from one region to another, they also change the players to whom they distribute their current state. Of course, those at the border of a region have to know something about what is happening in the neighbour-region. So each player should also be able to listen to the data that the players in his neighbourhood are distributing.

But, of course, other ways to handle a certain area are here possible, too. A server can do the job of the coordinator of an

area. Now, not the players are responsible for distributing the data to each other, but the servers are. They now form the peer-to-peer network (see chapter 5).

## 4.4 Coordinator

A *Distributed Hash Table* (DHT) is a datastructure. Every node of this table has a unique ID (or hash-value). The messages for those nodes can be routed very efficient though the datastructure. It is also very scalable and there is no great problem to let nodes join or to handle failed and disconnected nodes.

There are many objects in the virtual world. For example, there are weapons, ammo, food, health packs or other things like that. These objects also have a status. Someone or something has to be responsible for their status. Since an object has a specific ID, it can be assigned to that player that has the numerically closest ID in the DHT. This player does not necessarily have to be in the same area as the object. Strictly speaking, it is an advantage if the player is not in the same region. This is due to the possibility of cheating (see chapter 7).

Coordinators are only contacted when the state of the object, they take care of, is needed. So, a failure of this coordinator will only attract attention, when it is required. Once a failure is discovered, the replica will take over the job of the old coordinator and will start to build new replica(s) of itself. The time between the failure and its detection depends on the number of players in that certain region. The more players are participating, the more often the coordinator will be asked about the current state of its object.

Not every kind of data can be or should be handled transient. The graphics for example can be stored on the client's computer. As shown above, authorisation data and the states of the players should be stored on a central server for security reason.

The coordinator of an area, no matter, if it is a server or a player, is root of the associated multicast tree.

But, just like it is written in the last subchapter, there are other ways to coordinate the objects. Again, servers can take over this job.

## 4.5 Replication

Since one can not rely on the stability of every node in the DHT, replicas of every object have to be made. If a node, which is coordinator of one or more objects fails or disconnects, the object will get lost forever. To avoid this, replicas are made. Node A for example coordinates an object B. The node C with the numerical second-closest ID will be the coordinator of the replica of object B. If node A fails or disconnects, there is only the replica of the object B left. To ensure that this object is kept consistent, a new replica is needed. The node C will now be the new coordinator of B, because it now is the one with the numerically closest ID. It now replicates B to the node with the second-numerically closest ID, and so on. Otherwise, if a node joins the network that has an ID, which is numerically closer to an object than the node that coordinates this object, it will become the new coordinator of this

object. Here, it does not matter if this object is a replica or an object itself.

Now, the new node will receive updates on the object or the replica. If a node that coordinated an object before it handed this task over to the current coordinator, receives updates on this object, it will transfer these updates to the correct node. But it will store these updates until the transfer of the data is finished. This is due to consistency, if the current coordinator does not exist anymore. If it fails, the old coordinator will just continue doing its old job.

If an object has to be updated, the updates are sent to the coordinator of this object. This coordinator now sends this updated data to the object's replicas. These replica-updates are similar to requests of a player on the current status of a given object. It is also possible that the replicas are updated, but this should not be the standard method. The nodes have to store the last couple of updates to keep the right sequence. This can happen, if a new node joined the network and got an ID that is numerically closer to a certain object.

Due to the fact that the ID of a node is independent from its geographic location, it is very unlikely that numerically close IDs fail at the same time. The reason why geographically close nodes can fail at the same time can be various. But this does not affect the consistency aspect because there is no direct relation between geographic neighbourhood and the neighbourhood in the DHT.

Thus, each node has a kind of store for its objects, no matter if these are original objects or only replicas of objects. A node also has a manager that keeps the replicas of the objects up to date.

Of course, the quantity of replicas can be greater than one. This increases the consistency of the objects, but more traffic is needed to keep the replicas up to date. But the number of replicas can be variable, depending on the current failure- or disconnect-rate.

## 5. Load-Balancing

Since one can not act on the assumption that the players are evenly distributed on the map. There always will be places that are fully crowded and on the other hand there are places, where not a single player is walking around. Also, if something interesting is happening, like a fight, a former empty place can suddenly be crowded. So, one can not rely on the map characteristics to divide the map into areas. The partition has to be adjusted to the current state of the game and to the current load of the coordinators of the areas. The coordinator of a crowded area will have to handle a bigger workload than a coordinator of an empty area. If there are too many players in an area, the coordinator might get problems with updating the states of the players. It could lead to the fact that the game is not playable anymore in this region.

So, the idea arises to split areas with many players and give some parts to coordinators that do not have so much work to do.

These coordinators are mostly servers. But if servers coordinate the areas, where is the peer-to-peer part? Is this not just like the standard client-server approach?

Servers have just more upload resources than the average client. But these resources are not always used completely in the client-server approach. Thus, the idea arises, to distribute the server over the whole game. Now, a server-cluster is not responsible for the whole world, but a single server is responsible for a small part of the world (see figure 5). An advantage of this approach is that servers can be added or removed without great problems. Also, these servers do not need to be located in a certain place. And maybe – if the game supports this – the servers can be run by private persons or groups. These servers now form the peer-to-peer network of the game.

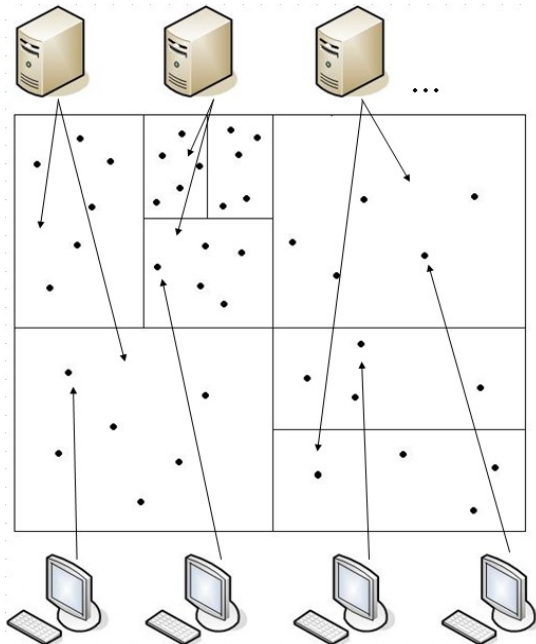


Figure 5: The distributed servers (top) are responsible for different regions. The players (bottom) can switch between these areas (similar to [11])

As written above, the players are not distributed equally. So, if the servers are responsible for evenly divided parts of the world, there would be no advantage of this approach.

## 5.1 Dividing the world

There are again many approaches how to divide given areas. In this paper, two of them will be described. The goal of both is to balance the load of all the coordinators of the areas. The map is divided into rectangular parts. Each area is coordinated by a server. On the other hand, a single server can take care of some areas. Each server knows those of the adjacent areas.

In the first one [11], areas that are too crowded, can be divided in different ways.

There could be a horizontal split through the center of the area. This method, called *SplitCenter*, does not take care of the players in that area.

The method *MaxDistToBorders* decides whether to split horizontally or vertically along the center. It maximizes the average distance of the players to the new border.

Another method, called *IntelliDistance*, also splits vertically or horizontally across the center. But this time, as few players as possible shall be able to take a look above the new border.

*EqualNumbers* again splits vertically or horizontally along the center. Here, the numbers of the players in both of the new areas shall be as equal as possible.

The last one is *VarAreas*. It works just like *EqualNumbers*, except that the split is not made across the center. It will be across the centroid of the players. So, there will be an equal amount of players in each region.

The new created area will now be coordinated by that server with the fewest workload. On the other hand, adjacent areas with very few players can be merged to one area. So, there is a server remaining that can take over another region, which is too crowded.

The other approach [12] does not divide the areas when they are already overpopulated. One world is divided into so called *microcells* from the beginning. So, a server coordinates a set of these cells. If the interaction in an area gets too busy, one or more microcells can be transferred to an adjacent area, and thus to another server (see figure 6).

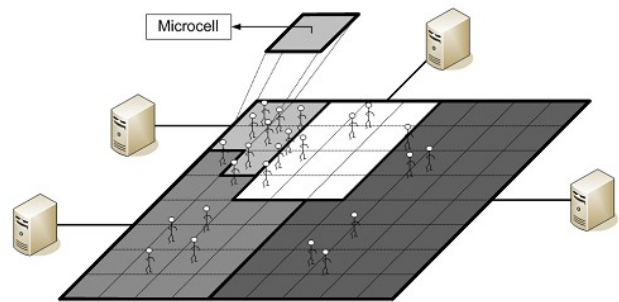


Figure 6: The area of authority of the servers is divided into microcells [12]

Of course, the division of the areas causes some extra communication between the cells. But there is an easy way to compensate this disadvantage. The microcells of one area are not treated as completely independent areas, because their data is stored in the same database. So, it is easy for a server to switch a player from one cell to another. And it is easy for a player to get to know, what is happening in the adjacent cells, because the server just knows, what is happening in these cells.

### 5.1.1 Comparison

Both techniques have advantages and disadvantages.

In the first one, the servers just have to control a certain area that is splitted in two areas, when too many players are located in that area. When there is no split, this approach is easier to be handled than the microcell approach. The microcells have to be combined in one database to reduce the communication overhead between them, but they somehow have to be autonomous, so they can be switched to another server. The switching of a microcell will not be that difficult. No further calculation has to be made. But in the first approach, the area has to be splitted before another server can do the job as the coordinator of the new area.

It just depends on the game itself, which approach is better. If there are many splits, the microcell-approach might be better. If not, the other one might be the better approach.

## 5.2 Server communication

What is happening, if players wants to take a look beyond the border of their server's area? Or what happens, if a server fails or gets disconnected?

The servers always have to know those of the adjacent areas. Each one now distributes the state of its area to all the neighbours. Because of this fact, the status of an area will always have a minimum of one replica. And just like the split of an area, the adjacent servers with the fewest workload can easily takeover the work of the failed one. This easily means that these servers already know everything about this area. Of course, the new areas can be splitted to reduce the workload of those servers.

And since every server has a copy of the state of every adjacent area, it can show its players what is happening in the those areas, they also can see.

## 6. Latency & Inconsistency

The state of the game has to be nearly consistent for the whole time. It must not happen that the virtual world appears different to player A than to player B. If this happens often, no real game can be played. For someone, a certain player can be dead, for someone else, he can still be alive. Or if a player wants to hit another with a rocket and shoots it in his direction, then the attacker and the player, who is attacked have to agree on their current positions and on the current position of the rocket.

### 6.1 Latency

In standard client-server architectures, the bad scalability is responsible that the servers can be easily overloaded. They try to answer every request from the players, but this ends in longer response-times. Some players will receive their updates of the world later than others. This is, where inconsistencies appear. Thus, latency and inconsistency are affiliated with each other.

Of course, delay is always there. The messages just take some time from the sender to the receiver. But it depends on the game itself, how much delay can be accepted and when this game is not playable anymore. FPSs do not accept such a high delay as the RPGs do. So, a game, which accepts only a small delay will have to deal with more inconsistencies than a game which allows for example 400ms of delay. There are measurements [10], which attest that there are about 4% missing objects when no delay is allowed. This can be decreased to about 1% when 400ms delay is accepted. This measurement was made with about 60 nodes. When about 100 nodes are participating, then missing objects of the no-delay-game are increased to about 8%, the 400ms-delay-game nearly keeps its rate of about 1%.

So, the latency problem is not such a great one in a peer-to-peer-based multiplayer-game. There is a simple way to compensate the latency. A player always has current position, a direction and a movement speed. It will be very easy to calculate the position of the player for the next few moments. The objects that the player will see or use in these moments, can be loaded before they are needed.

Of course, this prediction is not always correct. Objects that are loaded can be of no use, because the player may have changed the direction. The greater the prediction-time is, the more objects are unnecessarily loaded. This causes extra traffic. So, the prediction-time should not be too great. The shorter this time period is, the more frequently updates have to be made (see figure 7).

Since only about 13.6 % of the people in the OECD-countries have a broadband-internet-connection [13], the players with slower connection can not be neglected. They have a disadvantage compared to those players with a fast connection. On the other hand, they may slow down the whole game by just answering late on some object requests.

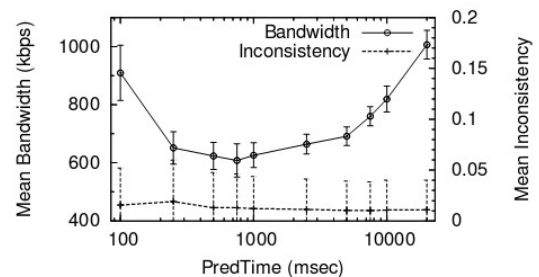


Figure 7: Impact of varying the prediction-time on the then mean bandwidth per node and inconsistencies [10]

### 6.2 Consistency

There are some kinds of consistency that have to be archived to keep the game playable.

The first one is per-object-consistency. This means that the state of the objects and their replicas in the virtual world should be the same all the time.

The other one is view-consistency. The players should always be able to see the current status of all the objects that are around them. Thus, the state of the objects have to be accessible all the time.

The number of replicas should not be too large but it either should not be too small. Many replicas ensure that the state of an object is secured. Here, the possibility that all the coordinators fail is less than when there are only few replicas. On the other hand, there is the traffic that is used to keep all the replicas up to date. Since the replicas are mostly coordinated by normal clients, the upload bandwidth is relatively small. And so, the traffic should be reduced to a minimum.

## 7. Cheating

Cheats are defined as any action of a player that gives him an unfair advantage over the other players [14]. They can easily be downloaded from miscellaneous sites in the internet. The players even do not need to understand anything about the background of the cheat, it just needs to be applied.

There are many ways of cheating. For example, there can be errors in the game itself which can be used to get an advantage. Or the client-software can be modified. But to create such cheats the availability of the source-code of the game is demanded and the ability to change it is needed. An example for this is the ability to walk through walls. The next possibility of cheating is to change the messages, which are sent from and received by the client.

The approaches, where servers are responsible for certain regions are not that vulnerable to cheating as those approaches, where some player form multicast-groups. The players can rely on the decisions, a server makes. When inconsistencies appear or packets are sent from the clients, which are not conform to the rules of the game, the server can decide, what has to be done to vanish the advantage of cheating. The serverless approaches – or serverless in a certain area – have more difficulties to handle cheating. A player can not trust every other player.

When the players are responsible for their own state – no replica is stored on a server - it should be no great problem to change the data. Of course, this can happen offline. No other player will get to know of the changes. There might be no method to prevent this kind of cheating when no replicas are made on servers. Other kinds of cheating can be prevented.

And this is the reason, why the coordinator of an object should not be in the same area as the object itself. If the players have no advantage by changing the state of their object – in fact, they only have disadvantages – they will not do it.

## 7.1 Packet-Cheating

These packet-cheating can be divided into some categories [14]:

- *Fixed Delay Cheat*: The outgoing packets will get a certain fixed delay. This allows the player to react on the action that is happening in the game, as fast as the other players do. But since the messages from the cheating player are delayed, the others can not react that quick on the changes.
- *Timestamp Cheat*: Due to consistency, every update to an object has a timestamp. A player, who receives a packet from another player can now send out a packet with a timestamp that is smaller than that of the received one. So, it appears that this packet with the smaller timestamp is delayed. With this cheat, a player can react on an action that already has happened before it happened.
- *Suppressed Update Cheat*: A player just stops sending messages of its current state to the other players. He seems to be invisible or not moving to them. But he still receives their updates. He just sends messages before he will be dropped out of the game.
- *Inconsistency Cheat*: A player can send different state updates to different players. Thus, the other players disagree with the current position of this player. The next updates should merge these different updates to a single, correct current position.
- *Collusion Cheat*: Two or more players can collude to gain an advantage. A player can, for example, inform other players about the position of an object or a player that the others can not see (because of their current position)

The approach New Event Ordering (NEO) [14] uses rounds. In every round, each player sends an encrypted update of his state to the other players. In the next round, the keys to the updates are submitted. The length of a round depends on the genre of the game. The players are allowed to send their update in  $2d$ , where  $d$  is the length of a round. This has the disadvantage that some players may not be able to send their updates in time. The length of a round can not be too long. Otherwise, the players have to deal with inconsistencies.

Due to latency and network characteristics, the players do not always get the updates of the others at the same time. In this approach, the updates are only accepted, when a majority of players have received this one. This assures consistency to those players that send their updates in time. Since the updates can not be decrypted before the next round, a player can do a kind of voting for the received updates. If the majority of the players voted positive – the update arrived in time – for this update, it is accepted in this area.

To allow that the delay of a message is only bounded by the length of a round without giving the other players – with a faster

connection – a disadvantage, the messages can be pipelined (see figure 8). So, if a message of a player was accepted or declined by the majority, he can send the next message, including the key to the last one. He does not have to wait until all the updates are verified, but only until all the other updates that are sent, are received by the others.

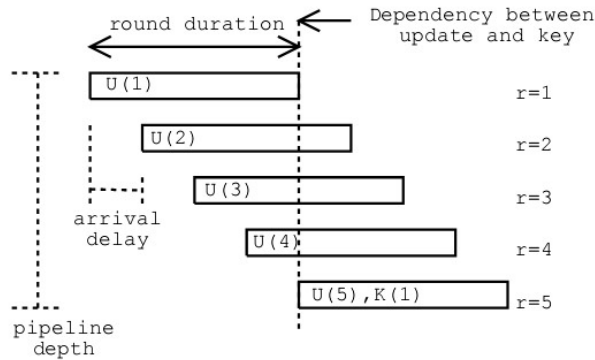


Figure 8: Pipelining in NEO [14]

The round now ends, when the the last update is received. Without pipelining, this round would last until the last update is verified.

## 7.2 Preventing Packet-Cheating

These are the ways to prevent packet-cheating:

- *Fixed Delay Cheat*: The rounds have a maximum length, so those updates that are not received in time, are simply ignored.
- *Timestamp Cheat*: When a round is over, no updates are allowed for this time. Especially none with a timestamp that lies in the past.
- *Suppressed Update Cheat*: If a player stops sending updates, the others will get to know this, because they vote for the updates of this player. If the majority voted negative for a player for a certain small number of rounds, he will get no updates from the others, because something might be wrong. Thus, the advantage is vanished.
- *Inconsistency Cheat*: This cheat can be eliminated through the voting-system. The voting should happen in

a kind of hash-value of the received update. Differences in updates can easily be identified.

- *Collusion Cheat*: The level, when an amount of positive votes for an update is considered as majority, can be adjusted to a sufficiently high value. Another way is to randomly select witnesses for a certain action. The more witnesses exist, the less is the possibility of collusion.

## 8. Conclusion

In this paper, the general differences between the currently most popular client-server model and various peer-to-peer models are described. Also, the big disadvantages of this client-server model are explained and how this can be solved in a way that there are no or only few new problems arising. Also, I explained, how the advantages of the client-server model can be adopted by the other models.

If the development of the number of active players increases further in this way, the pure client-server models will have great problems to provide each of them equally. Thus, the peer-to-peer models will experience a great demand. There are many good approaches, but they all have some disadvantages or unsolved problems.

Solipsis, for example, is currently not much more than a theoretical model. The models presented in chapter 3.3. are more practical. There are implementations for real games (Colyseus [10] for Quake II [15] or SimMud [9]). But since servers are maximally used here to secure the data of the players, it gets difficult to control cheating. The overhead of cheat-control increases if there is no trusted party that decides, what has to be done.

The currently best models are those, where the servers control a part of the world and balance their load. I think that this is a good mixture between scalability, playability and safety for the player's data.

## References

[1] Blizzard Entertainment: "<http://www.blizzard.com>"

[2] ArenaNet, Inc.: "<http://www.guildwars.com>"

- [3] 10Tacle Studios AG: "<http://ng.neocron.com>"
- [4] Playnet, Inc.: "<http://www.battlegroundeurope.com>"
- [5] M. Claypool, K. Claypool, F. Damaa: "The Effects of Frame Rate and Resolution on Users Playing First Person Shooter Games" in Multimedia Computing and Networking (MMCN) Conference, San Jose, CA, 2006
- [7] MMOGChart: "<http://www.mmogchart.com/>"
- [6] The Inquirer: "<http://www.theinquirer.net/default.aspx?article=30105>"
- [8] J. Keller, G. Simon: "Solipsis: A Massively Multi-Participant Virtual World" in Proc. of PDPTA, Las Vegas, NV, 2003
- [9] B. Knutsson, H. Lu, W. Xu, B. Hopkins: "Peer-to-Peer Support for Massively Multiplayer Games" in Proc. of IEEE INFOCOM, Hong Kong, China, 2004
- [10] A. Bharambe, J. Pang, S. Seshan: "Colyseus: A Distributed Architecture for Online Multiplayer Games" NSDI, 2006
- [11] S. Rieche, K. Wehrle, M. Fouquet, H. Niedermayer, L. Petrak, G. Carle: "Peer-to-Peer-based Infrastructure Support for Massively Multiplayer Online Games" in Proc. of IEEE CCNC, Las Vegas, NV, 2007
- [12] B. de Vleeschauwer, B. van den Bossche, T. Verdickt, F. de Turk, B. Dhoedt, P. Demeester: "Dynamic Microcell Assignment for Massively Multiplayer Online Gaming" InNetGames '05: Proc. of ACM SIGCOMM workshop on Network and system support for games, New York, NY, 2005
- [13] Wikipedia: Broadband Internet access worldwide: "[http://en.wikipedia.org/wiki/Broadband\\_Internet\\_access\\_worldwide](http://en.wikipedia.org/wiki/Broadband_Internet_access_worldwide)"
- [14] C. GauthierDickey, D. Zappala, V. Lo, J. Marr: "Low Latency and Cheat-proof Event Ordering for Peer-to-Peer Games" in Proc. of the 14th international workshop on Network and operating systems support for digital audio and video, 2004
- [15] Quake II: "<http://www.idsoftware.com/games/quake/quake2>"