# P2P MMOG with Pastry

Jacob Damgaard Jensen
20092138

Andreas Fyrstelin Kristiansen
20092027

Rasmus Kampmann Olsen
20093692

## Abstract

►*write abstract*◄

## 1. Introduction

Traditionally massively multiplayer online games are client / server based. This doesn't scale without large investments in server hardware. We have therefore investigated the possibilities to distribute the workload between the participating players, in order to allow scaling without the associated hardware costs.

Moving the enforcing of the game rules to the participating players, opens for cheating possibilities we also needs to prevent.

## 2. Related work

In this section we describe related work dealing with building P2P MMOGs, our primary focus is on architecture and cheat mitigation.

### 2.1. Design issues

Fan et. al[?] list a comprehensive set of 6 design issues to be addressed by P2P MMOGs in the following subsections, we describe these issues.

**2.1.1. Interest Management.** Since the player only have limited movement speed and sensing capabilities, and he does not need to know about things happening in the game which does not affect him, we can talk about an Area of Interest (AoI). The task of interest management is then to manage which game events is relevant for each player.

**2.1.2. Game event dissemination.** The approach used to distribute the game events to the interested parties. Depends on the underlying interest management, but both unicast and multicast could be used.

**2.1.3. NPC host allocation.** MMOGs does not only have human controlled player characters, AI controlled non-player characters (NPC) are also essential, but when you do not have a central server to host these, you need to distribute then to common game participants.

**2.1.4. Game state persistency.** Game state persistency is the task of making the game state persistent, so players keep their stats between login sessions, even though the

game continuously evolves, even when they are off-line.

**2.1.5. Cheating mitigation.** In a C/S architecture where the server has the entire state, it is easy to verify the request from clients are valid. Without this central authority, it becomes much harder to prevent cheating. We have a more detail decription of cheat mitigation approaches in the Cheat mitigation section on the following page

**2.1.6. Incentive Mechanisms.** When we move the work from the central server, and distribute it on the participating players, we rely on them to provide extra computing resources. Without some incentive mechanisms, we risk the players will limit the extra resources they provide, for instance by not hosting NPCs or participating in the cheat mitigation.

### 2.2. Architecture

When building a P2P MMOG, you need a more sophisticated network architecture than used in traditional Client-Server based implementations. Hampel et. al suggests a P2P architecture based on the Pastry DHT. They use Pastry for maintaining the network, and routing of messages, while using the extension PAST for object management, and SCRIBE for event based messaging.

In the article they present an approach where the game world is divided into several regions, each region has a region controller, that is a peer selected to keep the game state for that region consistent. In addition to the region controller they also use backup controllers, which both works as live backups in case the region controller is disconnected, but also monitors the region controller in order to detect if it has been compromised and tries to cheat[**?**].

### 2.3. Cheat Classification

Cheating is a wide spread problem in games, no matter if your are using a C/S or a P2P architecture, but mitigating cheating becomes more complex in a P2P architecture. Web and Soh[**?**] lists a classification of cheats into 4 levels. In this section we will shortly present these levels, and approaches to mitigate them.

**2.3.1. Game level Cheats.** Cheats happening within the game without modifications, either by exploiting bugs, or depending on game, getting unfair advantages by buying help/items from other players in exchange for real money transaction. The general solution for handling bug exploitations, is to release game patches, removing these bugs. Preventing real money transactions is far more difficult, since it happens outside the game, but analysis of log files could be a way to detect this behavior.

**2.3.2. Application level cheats.** Modifying the game client in order to expose informations, automate play/enhance reflexes or issue invalid commands are examples of application level cheats. Preventing information exposure involves not storing secrets in the game client, but store these on trusted third party, which is not easy in a P2P architecture. Requiring peers to run cheat detecting applications like PunkBuster is an approach well suited to make the life harder for cheaters. Issuing invalid commands not available to honest players, like improving the characters abilities, can be mitigated by using a referee to simulate and validate all commands.

**2.3.3. Protocol level cheats.** These are cheats involving interfering with the

packets sent to and from the game. For instance time-stamping actions in the past, after receiving opponents actions, or use spoofing or replay attacks to hurt opponent players. These attacks can for most cases be countered by using adequately signing of messages and enforcing a cheat prof protocol, or using trusted referees.

**2.3.4. Infrastructure level cheats.** Using network sniffers to expose information about opponents, is an example of a cheat happening on the infrastructure level, similar is the modification of the display driver to render walls transparent. Again using cheat detecting software like PunkBuster to detect modified executables is an approach to handle these cheats. Another type of infrastructure cheats, which cannot be detected by this approach, is using a proxy to modify the game actions, e.g. by auto aiming before passing on a fire command.

### 2.4. Cheat mitigation

Mitigation of cheats can be achieved using both proactive and reactive approaches. A proactive approach prevents cheats, that could be using the lockstep protocol to ensure peers reacts to the same state on every round in the game, preventing protocol level cheats. Reactive approaches on the other hand detects cheats, and handles accordingly, reducing the overhead involved in proactive approaches[**?**]. One way to detect protocol level cheats is to use controllers and backup controllers as proposed by Hampel et. al[**?**], another variation of this idea is proposed by Izaiku et. al[**?**]. They propose an architecture where the game world is divided into subareas, each subarea has a responsible node and some monitor nodes. All events from players, including the hash of the last received event list from the responsible node is then sent to both the responsible and the monitoring nodes. If the monitoring nodes detects inconsistent game state, either because of cheats or network delay, they will start a vote to decide which state is the current.

## 3. Game Design

We have designed a simple game for this project. It is fairly simple and contains few mechanics which, from a P2P point of view, is quite interesting.

First of all we need a world which can be devided into subspaces. As a typical dungeon crawler, we have chosen to create a simple world containing lots of rooms - each room have a ID on the form "(x,y)". To give some variation, rooms can be 5, 7 or 9 tiles high and wide. Each room can the be asigned to a peer, and the workload can be distributed.

Second, we need some characters which users can play. For simplicity, every user have the same character: a hunk. The users can the move between and inside a room and is free to move as far as he want in one move. The room a user is in, is his Area of Interest. In the AoI you should be aware of other players and they should somehow be displayed in the GUI.

Now a set of entities is needed for the game to make sense. A door (or something similar) is obviously need to move between room. Thus every room have four doors - one in each direction. Then there should be some challenges and items to boost you character: Chicks and Drinks. The goal is to make as many Chicks as posible and Drinks should help doing that. Here is some challenges to ensure that only one peer interacts with an entity at a time.

Some kind of stats should be introduced, such that you can improve your chances of making a chick: Charm and Moneys. When

you drink a drink you get Charm (with a high probability) and when you make chicks you receive money (because that the way it work in real life!).

Finally we need to roll dice, such that you cannot predict the outcome of drinking a Drink or hitting on a Chick. Everytime you interact with a Drink or a Chick, two dice is rolled. For drink the following rules applies:

- The sum of the eyes is 3, you loose 2 charm points
- The sum of the eyes is 5, you loose 1 charm point
- The sum of the eyes is 7, you gain 1 charm point and recieves 3 moneys
- else, you gain 1 charm point

For Chicks, the first roll is against you and the other againt her. The Chick is given some initial Charm when she have no more Charm, you have hooked up with her and gains sqrt initial Charm moneys. First, the Hunk loose the number of eyes on the first die. If the still have charm left, the chick looses the second dice times the squareroot of the Hunks moneys plus the squareroot of the Hunks charm. Thus charm is necessary to hit on chick (since you cannot hit on a Chick if you goth less than 1 Charm) and Moneys will make it easier to hook up with her, since it is multiplied with a die roll.

Now a simple game is designed, which have some quite interesting mechanics for a P2P game.

## 4. system design & implementation

In this section we discuss the architecture we have chosen for our project, and our implementation. Everything describe in this section are features that are implemented in our project unless specifically noted. When mentioning Pastry and Scribe, the underlying implementation used is FreePastry

### 4.1. Resource management

We decided to follow [REFERENCE] and split the world into multiple areas of interest - in our case a room in which players can navigate and interact in. Each room is assigned an ID in the Pastry ID space such that a player can be designated as the owner and holder of the room ressource. As hashes of the rooms position in our world are used to create this ID we can expect these IDs to be placed uniformly in the ID space of Pastry, thereby ensuring that we do not overload any single peer.

These rooms state, combined with player states, are our only required resource. Players each have their own state and these are updated accordingly as the game progress. Any room in which a player is located in also has this specific players essential state. We discuss later on in [... link to cheat mitigation?] the challenges of keeping the player state locally this present, and our intended solution. Monitors for our area of interest are used to keep track of the room state.

### 4.2. Monitors

Monitors are introduced in [..ref.?]. This is a mechanic in which multiple peers can keep track of, and correct, state updates from the owner of an area of interest. In our case each client is responsible for sending all relevant messages to x number of monitors as well. These monitors are chosen in a manner similar to the room owners, again making the distribution as uniform as possible. Each monitor keeps track of any changes to the rooms in which they are

designated a monitor. When a room owner pushes a new state to the relevant clients these monitors can then simply check if they agree with this new state or not. If they do not agree with the state from the room owner they are able to start an internal vote amongst themselves to do any required rollback - this is described later on in [ cheat migitation link?].

Due to this voting system we can confidently add new peers to the network, and give them the responsibility of either being a room owner or a monitor to a room - in the case of a new peer suddenly joining and assuming responsibility for a monitor ID this peer could receive state updates and actions for a room - however because this one monitor was there not there from the start it would initiate a vote among all the monitors as it did not know about the state of the room. As long as there are enough old monitors they will all agree on the correct state and everything will work as expected. This same principle applies if the new peer joining the network is put in charge of a room that is already populated or have had state changes as the monitors would then do any required rollback.

### 4.3. Pastry

We decided early on that Pastry should be used for important actions - such as informing the host of a room that we entered, or that a client wish to interact with a specific Entity. The reason for this was that we decided these actions had to be given to both monitors and room owners as fast as possible, and to make sure that any other peers in the same area of interest are not informed of these actions. This principle proved to be quite advantageous when doing our actual implementation as this defined both naming conventions and what happens where in our code and messaging.

### 4.4. Scribe

Scribe is used for all state updates and position updates (as we do not consider a peers position to be part of his state). We did this to limit the amount of traffic the owner of the area of interest needs to handle as much as possible. Whenever a peer enters a new area they subscribe to a relevant Scribe tree and immediately requests the room and position state for the room - this means that when a user enters a new room they only send a “enterRoom” message to the owner and monitor of the room, but the actual room state is provided via an Any-cast that anyone can reply too if they have the relevant information and have not just joined the room themselves as well. When a peer moves around the room they push their new position via a scribe multicast, again to limit the amount of traffic the owner has to handle. Both the room owner and all monitors are subscribed to Scribe as well - they do this the very first time they get a message indicating that a peer has entered the relevant room.

## 5. Cheat Mitigation

When moving from a Client/Server structure to a Peer to Peer structure, it is fairly easy to cheat, since there is no central component which could be considered fair.

### 5.1. Preconditions and Assumptions

In this report, we will not discuss vulnerabilities with Pastry or general issues related to Peer to Peer. Instead we will focus on issues related to Hunks and Chicks implemented in a Peer to Peer fashion. Furthermore we assume that every message is signed and messages will be forwarded as

planned.

### 5.2. Vulnerabilities

There is quite a few vulnerabilities in Hunks and Chicks:

- A peer can easy find a room he is responsible for, enter it and spawn a room that is favorable for him. In general a peer can create his room in a beneficial maner for him.

- In the same way, a peer which is responsible for a room, sometimes have to roll some dice. No one can validate that the peer actually rolled some dice instead of just choosing a beneficial outcome.

- When a peer is asked to create a room, he can simply create an empty room, since that will require less CPU.

- When entering a room a peer can tell the owner that he comes from a room that he controlls, and thus pass a superior state to the new room.

To protect against these, we will introduce the monitor concept ►reference◄.

### 5.3. General Cheat Protection

By introducing a number of monitors, more than one peer will be responsible for the state in a room - only one will multicast the state though. The monitors are chosen from the room ID concatinated with _mon-n, where n is the monitor number. Thus the monitors should be uniformly distributed on the pastry ring. If a monitor detect a corrupt state , it will start a vote and the vote result will be the new state. To cheat you have to find a room which you are coordinator for and own more than half of the monitors. The more peers and monitors, the harder it becomes to cheat. The probability of owning more than half of the monitors for a given room that you are responsible is:

$$p = \frac{1}{n^{\frac{m}{2}}}$$

where $n$ is the number of players and $m$ is the number of monitors pr room. Since Hunks and Chicks is considered a MMOG 100.000 players would not be unlikely and with a number of five monitors will give a probability of $10^{-15}$ which requires around $2^{50}$ calculation. This will take some CPU time - and it is no problem to increase the number of monitors to 7 or 9 increasing the number of calculations to $2^{66}$ and $2^{83}$ respectivly. And since there is only $2^{64}$ rooms, there is no guarantee such a room will exist.

## 6. Experiments/evaluation

## 7. Conclusion

## 8. Future work

There are many possibilities for expanding or modifying our current implementation of Chicks and Hunks. One feature that we did not have time to implement is some sort of room validation upon creation. As it is right now a peer is able to create any sort of room they desire - we discussed different strategies where a room could be validated. The obvious solution would be to use our already implemented monitor solution to either agree on a random dice to be used upon creation, or implement a point system where a room needs to conform to a set of rules of using a max of x number of points - for example adding a specific entity could use x number of points for a room. This would allow peers to set a strategy that they would prefer the rooms to be

created by while still maintaining game balance. The room could then again be validated by the monitors upon creation. A different solution would be to select a totally disinterested peer and allow him to create a room as he sees fit - because he is disinterested the assumption is that he would not simply create an empty room to avoid traffic. It would also be interesting to investigate if there are any alternatives to our current implementation of the monitor concept - for example by considering if all monitors should be allowed to start a vote or otherwise limit their voting strength. The idea behind this is that it could limit the amount of unnecessary votes and overhead traffic, but the difficulty is of course to ensure that we will achieve correct state at all times. Finally a comparison to a pure client/server implementation of our game would be very interesting - in this case it would be relevant to investigate how well we can scale and what the traffic requirements would be of a single server compared to our P2P architecture - not to mention what this means for any clients both in terms of bandwidth used and also for the overall experience of playing the game.

**Manual**

How to use our game ►**Write a install and run guide**◄