Design and implementation of online game models

Marc Palenzuela Reyes

Abstract - Make good or even a playable on-line game is not easy. In this paper we explain what are the difficulties of on-line games designing and implementation and propose solutions to those difficulties. We also have made a new transport protocol suitable for those models and three proof of concept games to check the effectiveness of our new taxonomy. After reading this paper you will be able to sort out and design the different on-line models and have a slight idea of a transport protocol design.

Index terms- RUDP, on-line game models, video games, transport protocol.

Resumen - Crear un buen juego on-line o incluso jugable no es fácil. En este artículo explicamos cuales son las dificultades a la hora de diseñar e implementar juegos on-line, y proponemos soluciones a esas dificultades. Además hemos creado un nuevo protocolo de transporte adecuado para estos modelos y tres juegos on-line como pruebas de concepto para comprobar la efectividad de nuestra nueva taxonomia. Tras leer este artículo el lector será capaz de clasificar y diseñar los diferentes modelos de juego on-line y tener una leve idea del diseño de un protocolo de transporte.

Palabras clave- RUDP, modelos de juego on-line, video juegos, protocolo de transporte.

1 Introduction

It would hardly be an exaggeration to say that the majority of video games coming to market nowadays have an implicit on-line mode being sometimes the only mode available. On-line games have revolutionized the gaming market adding innovative game-play, new funding ways and a totally different gaming experience. It may be asserted, however, that the programming and designing paradigms of video games are no longer a simple matter. High latencies and desynchronization are two new difficulties for game programmers and designers, highly influencing the game experience. As a consequence, there are good video games and really bad video games to the extent that they are nearly unplayable. Some of the issues you may encounter in those video games are high latencies that prevent you from foresee the behaviour of your character and desynchronizations with multiple effects, leading from funny glitches in the game-play to random disconnections.

In this paper, we propose a taxonomy consisting on three variables: the connection paradigm, the latency perception reduction algorithm and the communication protocol used by a game. The combination of these variables give us the concept that we decided to name on-line game model. The concept of on-line game model offers us a simple way to sort out the different on-line video games and their features in addition to a simple way to generate the base design of an on-line game.

We have characterized the different connection paradigms we can encounter and designed and implemented a transport protocol halfway between UDP[1] and TCP[2], having the best of both transport protocols for on-line game designing: speed and reliability.

To demonstrate the veracity of our arguments we implemented three proofs of concept games. These are simple games that are not meant to be actually played, but used to validate our proposal. Those three games represent three of the key on-line game models, and they work as a base design for those models.

1.1 Objectives and methodology

Our main objectives were to sort out the different on-line game models and make a base design for the most representative of them. To achieve this, we have defined three sub-objectives. Analyse the available transport protocols, analyse the available latency perception reduction algorithms and to make a taxonomy of the different communication paradigm we could use. From these objectives, another one appeared. Analysing the communication protocols we realised that we needed to design a new one. All the objectives were achieved. In this paper we have the taxonomy of on-line game models thanks to the analysis of transport protocols, latency perception reduction algorithms and communication paradigms we identified, besides the new transport protocol we designed.

Contact mail: marc.palenzuela@e-campus.uab.cat
Information technology student
Tutored by Sergi Robles
2014/2015

2 State of the art

Every time a new technology appears, the community tries to apply that technology to the maximum number of possible fields. Nutrition, medicine, research... and games are not an exception. Since the time when the first personal computers appeared, there have been more and more games, evolving with the new hardware. With the arrival of the Internet, a new era for computer gaming made its coming. The first commercial on-line games appeared, but they were difficult to play via modem[3]. A single, but striking, example of this situation is the first *Age of Empires*. Like most of the contemporaneous games of *Age of Empires*, this game was designed over a peer to peer paradigm. Without any kind of supplementary protocol for data exchanging, peer to peer was rather bad to play with, like we will see in the section 3.2. The first commercial game which used a client-server paradigm was Doom. After the failure using the peer to peer paradigm with an FPS (First Person Shooter)[4], they decided to centralize all connections on a server. The first players complained about the latency. Despite using a client-server paradigm, you didn't see any change when you pressed a key, but you did see it when the data arrived to you after being processed by the server. With the releasing of *QuakeWorld*, a new algorithm to reduce the latency perception appeared: the movement prediction algorithm. We will see more about this algorithm in the section 3.3. One year later, the first commercial MMORPG[5], *Ultima Online*, using a client-server paradigm with multiple servers to allow playing to the thousand of players that player simultaneously this game.

Nowadays, those games are old legends. Games like *World of Warcraft* or *Call of Duty* use more and better paradigms, algorithms and transport protocols than their ancestors, and most of the high budget video games gives to the user a good on-line game experience. But there are still some games which doesn't make use of the good practices of those high budget games. In this paper, we will see which are the good practices and which are not. However, there is no standard for the on-line game models taxonomy nor a standard for on-line gaming communication protocol.

3 Analysis

In this section we will analyse the three key characteristics of an on-line game model.

An on-line game model comprises a Transport protocol, a Communication paradigm and a Latency perception reduction algorithm. The transport protocols we will analyse are TCP[2] and UDP[1], and decide which one is better for our purpose. There are not a definite list of known communication paradigms. We will try to define most of them and describe their utility. We will see too two latency perception reduction algorithms: the movement prediction algorithm and the delta-time algorithm.

3.1 Transport protocol

In this subsection we will analyse the advantages and disadvantages of UDP and TCP. This decision is very important. While UDP is faster, TCP is more reliable. But which one we need to make a good on-line video game?

3.1.1 UDP

Like we said earlier, UDP is faster than TCP. That is because the UDP protocols only includes a very small header and none of the TCP reliability methods. That means UDP is not going to check if an user datagram was received or not.

What if we send an important datagram and it is not received? This situation could make a player lose or win, and this is not what a player wants.

Otherwise, there are games where a lost user datagram or two are not so important. But since those game are only a little portion of the on-line games, we are going to discard UDP for the time being.

3.1.2 TCP

TCP may not be as fast as UDP, but it is more reliable. Thanks to its flow control, congestion avoiding methods, the sliding window and the dynamic time-outs, no data will be lost. It is true that TCP is very reliable, but all those reliability entails an slower performance. Also, most of the TCP functionalities are of no use for our purpose.

Some games are played at an slow pace. For those games we will want to use TCP, because a faster performance is not required. But for instance, for FPS games we will need the fastest performance. Using TCP on these games may suppose higher latencies and, as a consequence, having a bad gaming experience.

3.1.3 UDP + TCP

We need the speed of UDP and the reliability of TCP. So for most of the games, we can not use either UDP or TCP.

The best solution to this problem is a custom transport protocol, having the best of those two well-known transport protocols. That means that we need a fast protocols capable of check if a datagram was received or not. With the minimum functionalities we need, the new protocol could be fast and give us all we need to make a good on-line game.

We will talk more about this new transport protocol in the section 4.1.

3.2 Communication paradigms

In this section we are going to study the different communication paradigms that an on-line game could be using.

The communication paradigm refers to the structure of the network used to play the game. For example, client-server is a communication paradigm. We have defined a total of seven different communication

paradigms. We will see the common ones, the Peer to Peer paradigm and finally the hybrid paradigms resulting from the combination of other paradigms.

3.2.1 Multiple Client / Single Server

This communication paradigm is the common client-server paradigm. There is one single server and an indefinite number of connected clients. It is a simple and effective paradigm to implement games on, due to the easy design and the ease of administration.

In the other hand, there is no possible load balance in this paradigm. As a consequence, scaling a server with this paradigm is very difficult. Also, if the server does not work, there is no backup server.

3.2.2 Multiple Client / Multiple Server

This communication paradigm is very similar to the one seen in the section 3.2.1. There are an indefinite number of servers. Every server have an indefinite number of connected clients. This paradigm is very simple too. It covers some of the problems of the Multiple Client / Single Server paradigm, like the load balance. This paradigm is often used when we want more than one instance of a game being played at once, like in FPS games.

The main issue about this paradigm is the cost of the hardware, that scales with the number of servers.

3.2.3 Peer to Peer

This is a common and early communication paradigm for on-line games, as seen in section 2. While is a very cheap paradigm to implement the games with, it could be the worst one if not designed properly.

Without a good design, the game may result in a time-locked on-line game[1]. This is the fastest way to make a game nearly unplayable, so we want to avoid it.

The biggest issue of this paradigm is the lack of information about every single network the players are going to play in. Every network is different, and designing a generic peer to peer paradigm to play games on is very difficult.

Another way of looking to this question is using the peer to peer paradigm in the server side, like we will see in the sections 3.2.6 and 3.2.7.

However, peer to peer can provide an interesting on-line mode, as we can see at the *Huntercoin*[6] game, a p2p on-line game based on *Bitcoin*[7].

3.2.4 Multiple Client / Single Server + Multiple Client / Multiple Server

This paradigm is an hybrid version of the paradigms seen in the sections 3.2.1 and 3.2.2. We can solve a lot of problems with this paradigm and add extra functionality. We can see an example of this paradigm at Figure 1.

[1] An on-line game where you depend on the biggest latency between all the players.

For example we can reduce the load of the main server, or make the main server a list of the servers that contain an instance of the game, so the player can choose the one he or she wants.

With this paradigm we can obtain a transparent load balance[8] for the players in addition to all the advantages that the paradigms shown at the sections 3.2.1 and 3.2.2. However, this paradigm is more expensive than the Multiple Client / Multiple Server one.

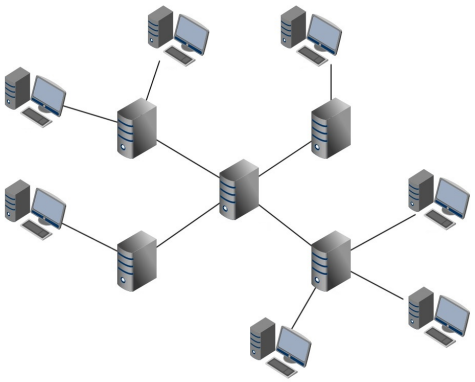

Figure 1: Example of Multiple Client / Single Server + Multiple Client / Multiple Server paradigm

3.2.5 Ad Hoc + DTN

This paradigm is not very used, but speaking personally I think this paradigm has a bright future. This paradigm is specially useful for mobile connections. With Ad Hoc we can create a temporary network between mobile phones, and send the results of a game to a master server using DTN, when we have available Internet access. This means that we can play in any location, with or without Internet access and send the result to the server when we find a Wi-Fi access.

We can see an example network for this paradigm at Figure 2.

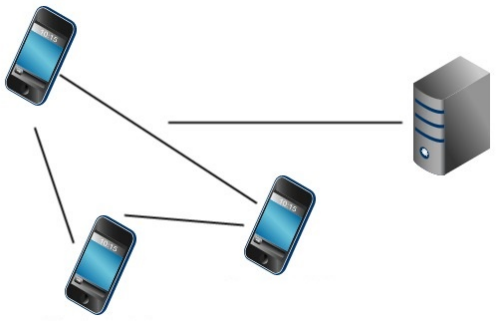

Figure 2: Example of Ad hoc + DTN paradigm

3.2.6 Multiple Client / Single Server + Peer to Peer

This paradigm has two different functionalities. The first one is that we can change between a Client-Server paradigm and a Peer to Peer one, as we have seen at the

sections 3.2.1 and 3.2.3 respectively. For example, one can play an MMORPG in a Client-Server paradigm. When we enter in a dungeon with our team members, we automatically change to a Peer to Peer connection with the team members.

The other functionality is that we can have a Client-Server paradigm with the support of the Peer to Peer connection. If one of the clients have limited connection with the server, another peer would help the first one and be a bridge between the first peer and the server.

We can see an example of this connection paradigm at Figure 3.

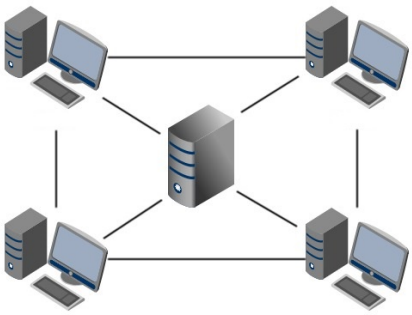

Figure 3: Example of Multiple Client / Single Server + Peer to Peer paradigm

3.2.7 Multiple Client / Multiple Server + Peer to Peer

This connection paradigm have multiple servers interconnected by a Peer to Peer connection. Thanks to this connection, there are multiple and different physical servers working as if there would be one.

A client connecting to this Peer to Peer servers network would choose automatically the best server for the client. Every server would share all the data, so all of the clients could connect with any server. We can replicate data too and make that every server have all the data.

This communication paradigm is an on-line game oriented distributed system[9]. A network example for this communication paradigm is the one we can see at Figure 4.

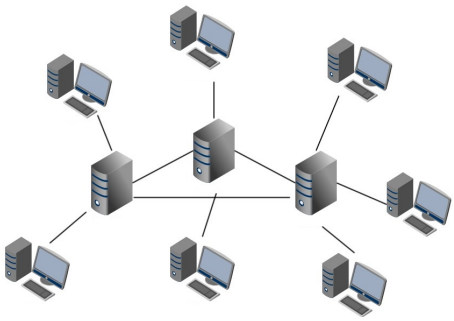

Figure 4: Example of Multiple Client / Multiple Server + Peer to Peer paradigm

3.2.8 Uses of the communication paradigms

In this section we have seen the seven different paradigms we have identified. Some of them are good for a game genre, some of them for other game genres. The Peer to Peer paradigm is not what we want for a video game. It implies a lot of latency. The fact of the matter is surely that designing a Peer to Peer communication between two or more peers with heterogeneous networks is a very difficult task. All this may well be true enough, but what if we have knowledge about the network? This is achievable for the server side. We know our servers and our network, so we can design a controlled Peer to Peer connection between them. With a server-side Peer to Peer paradigm we can solve most of the problems the other paradigms have. Besides there may be other communication paradigms, but they are of little interest for on-line purposes.

3.3 Latency perception reduction algorithms

In this section we are going to see two different algorithms that change the perception of the latency in an on-line game. That means that the latency is going to be exactly the same, but the user will see a game without or with a very little latency. Those algorithms are implemented in the client side and are totally independent from the server.

3.3.1 Movement prediction algorithm

This is a very used algorithm that allows the user to input game commands like movement and instantly see an in-game effect. With a Client-Server paradigm, the user would not see any effect until the data arrives the server, the server processes it and returns to the client. This algorithm simulates the effect the client will show and synchronizes with the result of the server.

We can divide this algorithm in two parts. The first one is simple: the client simulates the player action and show an effect. When the effect arrives from the server, the client corrects its first effect and update it to the server effect. We can see an example code for the first part at Figure 5.

```
action = input()
server.send(action)
update(action)
```

Figure 5: Movement prediction algorithm first part example

The second part becomes complicated. With the first part as base, we need to save a history of the client actions and rewind from the player actions every time we receive the effect from the server. Doing this, in every step of the process we will simulate again some of the player inputs for a better correction of the client effect and prevent a visual desynchronization[3][10]. We can see an example code for the second part of the algorithm at Figure 6.

```
actions = inputs()
update(actions)
history.append(actions)
server.send(actions)
server.recv(gameState)
updateTo(gameState)
for actions in history:
  update(actions)
history.remove(0)
```

Figure 6: Movement prediction algorithm second part example

3.3.2 Delta time algorithm

This algorithm tries to simulate the effects of some characters of the game other than the player character. We call this algorithm *Delta time algorithm* because of the basic formula:

$$x = V_0 * \Delta t$$

This is not an specific algorithm. We can use the same movement formula the server uses to predict the characters movement or a simplified one, and apply the algorithm with the same technique of the movement prediction algorithm, as seen in the section 3.3.1. Without this algorithm, we will only see movement for every update of the server. For high latencies, that could be very troublesome.

Those two algorithms are really indispensable for most of the possible on-line games. Without them, our games are going to be time-locked, what is going to worse the gaming experience.

3.4 On-line game models

Like we said earlier, a game model is the union between a transport protocol, a communication paradigm and a latency perception reduction algorithm, as we have seen at sections 3.1, 3.2 and 3.3 respectively. We can obtain the different on-line game models by making combinations of these three characteristics. Next we are going to see the following game models:

- MMORPG model with and without targets
- Action model
- RTS model
- Turn based model

3.4.1 MMORPG model (with targets)

First we are going to explain what means MMORPG with targets. An MMORPG with targets implies that if you select an enemy, all your attacks and skills that are not ground targeted are going to have as target the selected enemy. This may seem a game-play feature and nothing more. The truth is that this feature changes completely the way a game is designed.

Talking about the communication paradigm, there are some good paradigms for an MMORPG to use. Most of the Client-Server ones are good choices, but Multiple Client / Multiple Server + Peer to Peer as seen at the section 3.2.7 seems to be the best. It is not the cheaper nor the easiest to design and implement, but is undoubtedly the best one.

About the transport protocol we have two good choices. TCP is easy to use and is already designed and implemented, and a modified version of UDP would be faster. It is the designer duty to chose the best option for every game.

When it refers to the latency perception reduction algorithm, we can easily see that the delta-time algorithm is a need. We can say de same about the movement prediction algorithm. We want to see our character moving at the moment we make an input. However, the complete algorithm is not totally needed, but recommended.

3.4.2 MMORPG model (without target)

We now know what is an MMORPG with target, so we know what is an MMORPG without target too. That implies that latency is a more important issue in this model.

We can use the same communication paradigm as in the section 3.4.1. Talking about the communication paradigm there is no difference between an MMORPG with or whitout target.

For the transport protocol we mandatorily need the use of a modified version of UDP, as TCP is slow for our purpose and plain UDP is not reliable enough.

Talking about the latency perception reduction algorithm, we can say that is the same as in a MMORPG with target. Our needs about the perception of the game characters are the same. The MMORPG model without target is nearly equal to the model with target. The big difference is that in the model without target, the targeting for every attack or skill is more important, and it needs more speed than in the model with target. So the difference is in the transport protocol, being a must a modified version of UDP.

3.4.3 Action model

This model includes all first and third person shooter video games, the driving and sports ones and all the video games that need lots of inputs and outputs to give the user the best game experience.

The communication paradigm we need is not very definite. We can use Multiple Client / Single Server + Multiple Client / Multiple Server as seen in the section 3.2.4 so we can have a listing server to naturally distribute all the players between instances of the game.

Since we need the fastest game-play, we will want to use a custom UDP. TCP is not an option in this model, but plain UDP is.

Also, we will need the use of both latency perception reduction algorithms we have seen in section 3.3, having this time the complete version of the prediction movement algorithm.

3.4.4 RTS model

An RTS(Real Time Strategy) game is very similar to an MMORPG without target, as seen in the section 3.4.2,

with the difference of the quantity of players that are going to play in an instance of the game.

For the communication paradigm we can use most of the Client-Server ones. Two often used paradigms are Multiple Client / Multiple Server and Multiple Client / Single Server + Multiple Client / Multiple Server, as seen at the sections 3.2.2 and 3.2.4 respectively.

The transport protocol used in this model can be the same used in the MMORPG without target. So a custom UDP could be good.

Since an RTS game allows the player to click to a point in the map and the character goes following a route, latency perception reduction algorithms are not as important as in other models. The delta-time algorithm is not necessary, but recommended, and only the first part of the movement prediction algorithm is really needed.

3.4.5 Turn based model

A turn based on-line game offers the simplest design of the on-line models we have seen. The only need for this kind of game is to send and receive correctly the data turn by turn.

The communication paradigm could be any Client-Server paradigm. Like at the previous section, we could use Multiple Client / Multiple Server or Multiple Client / Single Server + Multiple Client / Multiple Server.

Since we don't need speed but only reliability, we are going to use TCP. We don't need to design and implement a new protocol if TCP is good enough for this purpose.

We don't really need any of the two algorithms we have seen. When we receive a message of a finished turn with the new game state we can process the movements client-side to draw at the screen how the new game state is seen.

We have seen some on-line game models with similar characteristics between ones, and very different for others. What makes different every model is the combination of every one of the three main characteristics, based on the needs of every game genre.

4 Design

In this section we are going to define the characteristics of a custom transport protocol and three proof of concept games. Those three proof of concept games represents three on-line game models: *Turns model*, *RTS model* and *Action model*. For further information about those models see the sections 3.4.5, 3.4.4 and 3.4.3.

4.1 RUDP

RUDP is the name we gave to our custom UDP based transport protocol and means Reliable User Datagram Protocol. This protocol offers the speed of UDP and reliability of TCP, having an acknowledgement system with time-outs. RUDP is message based, differing from TCP, that is byte based.

There are some important variables we are going to use in this protocol.

Talking about the timeout, if we want a fast transport protocol, we need to keep it simple. A fixed timeout is then a good option. For a default value, we have chosen 500 milliseconds, a value that makes almost any game unplayable.

This is the number of times a message can be sent before we decide to discard a connection is the number of retries. If we keep this variable high, it will only be of use for checking disconnections. Instead if we keep this variable low, it will check high latencies. The default value is 3.

As in TCP, RUDP uses SEQ and ACK numbers. TCP uses a maximum sequence number of $2^{32} - 1$, which is our default value. We need to keep this number big to avoid the prediction of the sequence number. This prediction is used to perform man in the middle attacks.

The window in RUDP is very similar to the one used in TCP. It limits the quantity of messages that can be sent before receiving an their acknowledgement. With our default value of $2^{31} - 1$, we can send have a list of sent and not acknowledged of $2^{31} - 1$ messages.

We need to sort the received messages to control the flow of messages. Affecting only the buffered messages, we have three different sorting methods: *first in first out*, which doesn't order the messages; *force order*, which order the messages ascendantly using the sequence number; remove obsoletes, which remove a message if an there has been a higher acknowledgement number.

We need to put the following data in every RUDP message: the sequence number, the acknowledgement number, a boolean that tells if we are acknowledging a message and a boolean that tells if we are opening a connection. Lastly, we need to put the application data. Using ; as delimiter, the format is as follows:

SEQ;ACK;ACKBOOL;SYNBOOL;DATA

With this concepts in mind and the diagram shown at the Figure 7 we are ready to implement RUDP.

4.2 Proof of concept: Turns model

In this section we will see what we need to implement our turn based on-line game.

Communication paradigm A client-server paradigm would be good for this proof of concept. We need a discrete amount of players, being two the minimum.

Transport protocol Like we said earlier in the section 3.4.5, TCP is okay with a turn based on-line game. We do not need communication speed.

Latency perception reduction algorithms Since all the visible movement of the game objects is client based we will not need any latency perception reduction algorithm.

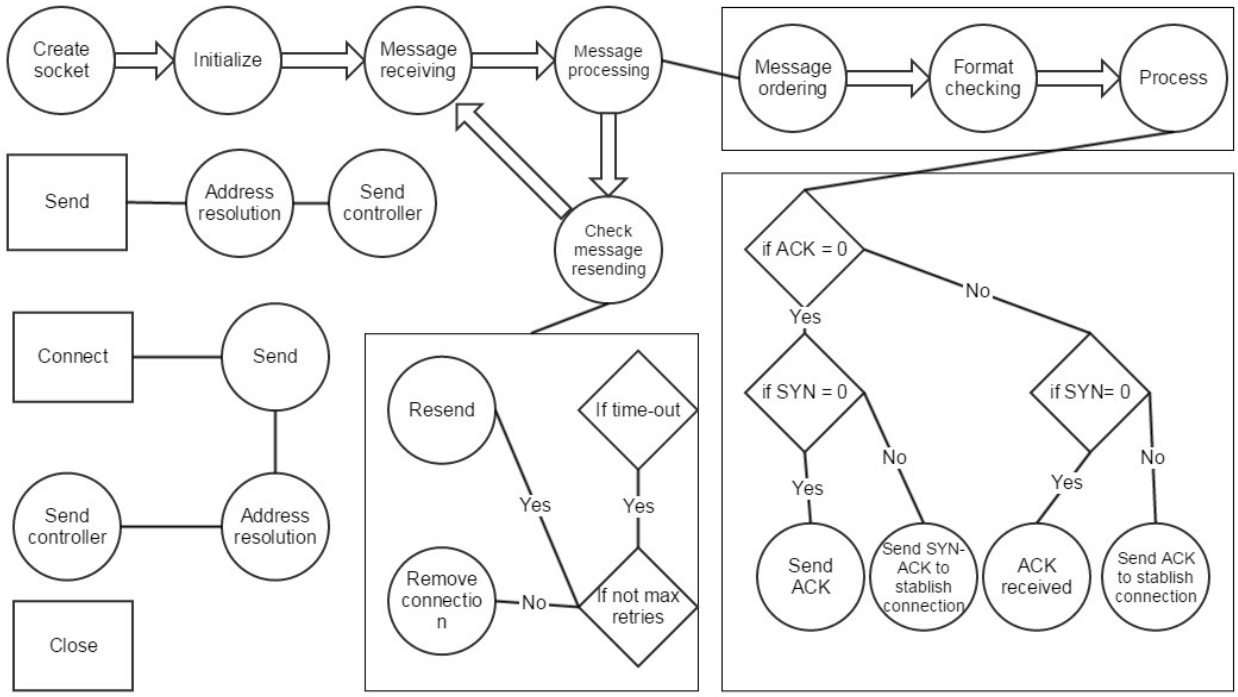

Figure 7: RUDP flow design

Server code design We can see an example of a server code design for a turn-based game at the Figure 8.

```
msg = server.recv()
if (msg == MY_TOURN_MESSAGE):
    action = input()
    server.send(action)
else:
    update(msg)
    render(gameState)
```

Figure 8: Turn model proof of concept server design

Client code design We can see an example of a client code design for a turn-based game at the Figure 9.

```
if (select(players, players, players)[0]):
nextTurn()
action = players[tourn].recv()
update(action)
for player in players:
  player.send(gameState)
```

Figure 9: Turn model proof of concept client design

4.3 Proof of concept: RTS model

In this section we will see what we need to implement our RTS on-line game.

Communication paradigm A client-server paradigm would be good for this proof of concept. We need a discrete amount of players, being two the minimum.

Transport protocol Like we said earlier in the section 3.4.4, our custom UDP, or RUDP, should be good for this proof of concept.

Latency perception reduction algorithms For this model we will need at least the first part of the movement prediction algorithm. We won't need the delta-time algorithm neither the complete movement prediction algorithm.

Server code design We can see an example of a server code design for an RTS game at the Figure 10.

```
if (select(players, players, players)[0]):
      for player in select(players, players,
    players)[0]
    actions.append(player.recv())
update(actions)
for player in players:
  player.send(gameState)
```

Figure 10: RTS model proof of concept server design

Client code design We can see an example of a client code design for an RTS game at the Figure 11.

```
actions = inputs()
update(actions)
server.send(actions)
draw(gameState)
if (select(server, server, server)[0]):
  gameState = server.recv()
  draw(gameState)
```

Figure 11: RTS model proof of concept client design

4.4 Proof of concept: Action model

In this section we will see what we need to implement our Action on-line game.

Communication paradigm A client-server paradigm would be good for this proof of concept. We need a discrete amount of players, being two the minimum.

Transport protocol Like we said earlier in the section 3.4.3, our custom UDP, or RUDP, should be good for this proof of concept.

Latency perception reduction algorithms For this model we will need at least the complete movement prediction algorithm. In this proof of concept we will not design or implement the delta-time algorithm, but it would be very recommended to consider.

Server code design The server code design is the same as in the section 4.3.

Client code design We can see an example of a client code design for an action game at the Figure 12.

```
actions = inputs()
update(actions)
history.append(actions)
server.send(actions)
if (select(server, server, server)[0]):
  gameState = server.recv()
  updateTo(gameState)
  for actions in history:
    update(actions)
  history.remove(0)
}
draw(gameState)
```

Figure 12: Action model proof of concept client design

5 Results

In this section we are going to see the implementation of RUDP and the proof of concept games. We are only going to see the most relevant aspects of every implementation, and the full implementation can not be found in this paper.

5.1 RUDP Implementation

RUDP has been implemented to be as easy to use as a standard TCP or UDP socket. As users we can create a socket, connect to another RUDP socket, send data to another RUDP socket and retrieve the received data from other RUDP socket.

The reception of data is done in another thread, beside the message processing and the resending checking. We can do another things at the time RUDP is receiving data, and retrieve it from the designed buffer at any time we want.

There is some configuration we can change as users. Next, there is a list of parameters of RUDP one can change as a user.

Timeout Changing the timeout implies that we will resend more or less messages, depending of the value we chose for the timeout. For low values, there will be a lot of resends. For high values, there will be few resends.

Maximum sequence number We can change the maximum sequence number of RUDP. A higher value offers us security because it would be more difficult to predict our current sequence number. A lower value is easier to manipulate.

Maximum number of retries This is the number of resendings of a message before we consider that we need to close a connection. A high number is more flexible but invalidates the purpose of RUDP.

Buffer length The default value of 4096 is considered a good value for the buffer lenght, but it is as easy to modify as the other parameters.

Sorting type We can choose three different sorting methods for the RUDP internal buffer. These sorting methods are *First-in-First-out*, *force order* and *remove obsolete*.

Debug level We can change the debug level to a number from 0 to 5, being the first the non-debug mode and the last the full-debug mode.

Simulation mode We can change the simulation mode between true and false. The simulation mode allows the user to simulate latencies and message loss without using low level tools like ip tables.

Latency The latency in seconds we want to simulate with RUDP.

Jitter The variation in seconds we want to give to our latency. A jitter of 1 second means that the latency can variate from -500ms to +500ms.

Message loss probability The probability of lossing any message. A value of 0.5 means that half of the messages will be lost.

At the Figure 13 we can see a server-side example of RUDP. There we can see how a user can configure a RUDP socket and make it listen to a port.

```
import rudp

rudpsock = rudp.RUDPSocket()
rudpsock.setDebug(5)
rudpsock.setSim(1)
rudpsock.setLatency(0.2)
rudpsock.setJitter(0.05)
rudpsock.setLoseProb(0.1)
rudpsock.listen(5000)
```

Figure 13: RUDP server-side example

At the Figure 14 we can see a client-side example. There we can see how a user can connect to another socket, send data and retrieve received data.

As we can see, RUDP is as easy to use as TCP or UDP, and it offers the reliability we chose with the configuration and the speed (almost) of UDP.

```
import rudp

myBuffer = []
rudpsock = rudp.RUDPSocket(myBuffer)
addr = ("192.168.1.5", 5000)
rudpsock.connect(addr)
data = "Hey, I'm testing RUDP!"
rudpsock.send(data, addr)
print str(myBuffer)
```

Figure 14: RUDP client-side example

5.2 Proof of concept games

The three proof of concept games have been implemented using Pygame[11]. With a very simple engine we developed, the graphics were not an issue. The first game, "The goose game", is a proof of concept game for the turn-based on-line game model. There are server and client files. This proof of concept game have no latency issues since it is turn-based.

For the user, the game shows as seen at the Figure 15.

Figure 15: Client view of the first proof of concept game

The second and third proof of concept games are very similar. The only difference is in the movement prediction algorithm, as seen in the section 3.3. In the second proof of concept game there only is the first part of the algorithm. In the third proof of concept game there is the full algorithm.

In a test play, the movement correction of the movement prediction algorithm is clearly visible for both of those proof of concept games.

For the user, the second and third proof of concept games are seen as at the Figure 16.

Also, thanks to the three implementations of the proof of concept games and RUDP we arrived to some conclusions that we will see in the next section.

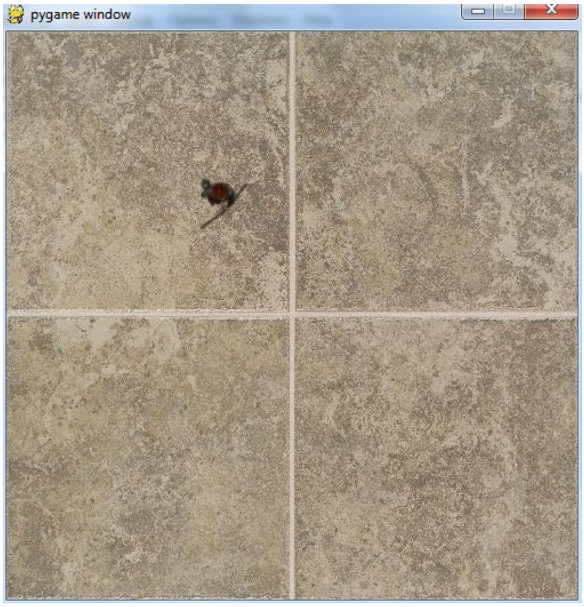

Figure 16: Client view of the second and third proof of concept game

6 Conclusions

In this paper we have seen how can we design an on-line game, focusing in the on-line part of the on-line game.

Surely the most important lesson to be learned from the section 3 is that there are more options than TCP and UDP. In fact, there are a lot if we consider every transport protocol we can design between UDP and TCP, RUDP being only one of these.

Moreover in the same section we can see a lot of communication paradigms. We can't be sure if those are all of the possible communication paradigms, but there are a lot of them. Also, we have concluded that the peer to peer paradigms are really good for the server side, and they are not as good for the client-side. This is due to that we need to know well the features of the network we are designing for.

Regarding latency reduction perception algorithms, we can say that they are really important for non turn-based on-line games, to the point that without them some games can be unplayable. They are relatively easy to implement and are only needed in the client, so it is good to invest in them.

Lastly and most important, we have seen that designing and implementing a new transport protocol or a good on-line mode in a game is not an easy task. It is a good investment, but it is a hard task and costs a lot of resources and time, thus not all games can afford it.

But, as seen in the paper and every on-line player could tell, a good on-line game mode with a low latency is really appreciated by users. The on-line game quality is one of the most important things to consider in designing time, and only with a good implementation and a good design could be done.

7 Future lines

We achieved a lot with this project, but there is still a lot of things to improve from this work. First, our RUDP implementation is far from perfect. The best we can do about RUDP is programming a C module of RUDP, to use it when programming with C, C++ and C#. We also could make it for Java. These are better platforms to make video games on than Python. Regarding RUDP too, a nice extra we can implement is a cryptography module. It would give the user options to encrypt the RUDP messages or sign them digitally. It is not very difficult and is a gain in security, because with encryption nobody but the server will know your inputs, and with digital signatures nobody will replace your messages.

More future lines could be designing in depth the Multiple Client / Multiple Server + Peer to Peer seen in the section 3.2.7. This design implies designing a full distributed system, and the communication would be only a little bit of the work to design a network to use with this communication paradigm.

Moreover, we could implement a full game using the knowledge obtained from this project. Using the taxonomy of on-line game models, we have already the base design for any game we want to make.

These are only three of the future lines we can think of. There is for sure a lot of work regarding the subject of on-line game designing.

8 Acknowledgements

I would like to thank Sergi por his patience and for how he motivated me. I would like to thank Álvaro, Juan Antonio and David too, for listening always and giving me ideas and their support.

References

[1] Postel, Jon. "RFC 768 - User Datagram Protocol". IETF Tools. IETF, 28 Aug. 1980.

[2] Information Sciences Institute. "RFC 793 - Transmission Control Protocol". IETF Tools. IETF, Sep. 1981.

[3] Fiedler, Glenn. "What every programmer needs to know about game networking". Gafferongames. Gafferongames, 25 Jan. 2010. (Web. 25 Jan. 2015)

[4] "First Person Shooter". wikipedia.org. (Web. 6 Feb. 2015)

[5] "Massively multiplayer online role-playing game". wikipedia.org. (Web. 6 Feb. 2015)

[6] Chronokings. "Huntercoin - A Human Mineable Crypto Currency". huntercoin. Chronokings, 2014. (Web. 30 Jan. 2015)

[7] Nakamoto, Satoshi. "Bitcoin: A Peer-to-Peer Electronic Cash System", Satoshi Nakamoto. 1 Nov 2008.

[8] Beskow, P.B.; Erikstad, G.A.; Halvorsen, P.; Griwodz, C., "Evaluating ginnungagap: a middleware for migration of partial game-state utilizing core-selection for latency reduction," Network and Systems Support for Games (NetGames), 2009 8th Annual Workshop on , vol., no., pp.1,6, 23-24 Nov. 2009 doi: 10.1109/NETGAMES.2009.5446220

[9] Ta Nguyen Binh Duong; Suiping Zhou, "A dynamic load sharing algorithm for massively multiplayer online games," Networks, 2003. ICON2003. The 11th IEEE International Conference on , vol., no., pp.131,136, 28 Sept.-1 Oct. 2003 doi: 10.1109/ICON.2003.1266179

[10] Bernier, Yahn. "Latency Compensating Methods in Client/Server In-game Protocol Design and Optimization". Valvesoftware. Valve, 2001. (Web. 26 Jan. 2015)

[11] "Pygame", pygame.org. Pygame. 2000. (Web. 30 Jan. 2015)