MASTER THESIS

TITLE: Study of Multiplayer Features for Unity Applications

MASTER DEGREE: Master in Science in Telecommunication Engineering & Management

AUTHOR: David Andrés López Nuevo

DIRECTOR: Dolors Royo Vallés

DATE: December, 19th 2013
Resum

Durant els últims anys el creixement del sector del videojoc tan en el nostre país com a nivell internacional ha crescut exponencialment. Actualment els videojocs no son únicament una eina de diversió i esbarjo sinó que el seu ús ha evolucionat a àmbits com l’educació o la medicina. Els videojocs, per tant, es poden fer servir com a eina per a adquirir nous coneixements.

Una funcionalitat molt important dels videojocs és la anomenada multijugador, que més d’un jugador pugui jugar al mateix joc al mateix temps. Gràcies a Internet i a les xarxes socials aquest tipus de jocs ha tingut una demanda creixent (un 62% dels usuaris de videojocs juguen a jocs multijugador).

Una eina molt important en el desenvolupament de videojocs es Unity. Aquest entorn de treball permet desenvolupar aplicacions per a diferents plataformes (PC, MAC, Android, iOS, Windows Phone, PlayStation, XBOX, Wii, ...) i és utilitzada per més de dos milions de desenvolupadors, entre ells grans empreses com Microsoft o Ubisoft.

Aquest projecte pretén estudiar les diferents funcionalitats multijugador que ofereix Unity (RPC, State Synchronization Delta Compressed i Unreliable), explicant el seu funcionament i les seves limitacions. A més a més, per tal de superar aquestes limitacions, es proposen i es programen noves llibreries que ofereixen funcionalitats multijugador alternatives.
Overview

In the recent years, the growth of the video game industry has grown exponentially in our country as well as internationally. Today video games are not only fun and entertainment tool but their use has evolved in areas such as education or medicine. Therefore, video games can be used as a tool for acquiring new knowledge.

A very important feature of video games is the one called multiplayer: more than one player can play the same game at the same time. Thanks to the Internet and social networks these types of video games have increased its demand (62% of video games users play multiplayer games).

An important tool in game development is Unity. This working environment allows developing applications for different platforms (PC, Mac, Android, iOS, Windows Phone, PlayStation, XBOX, Wii ...) and it is used by over two million developers, including large companies such as Microsoft or Ubisoft.

This project aims to study the various multiplayer functionalities offered by Unity (RPC Compressed Delta, State Synchronization and Unreliable), explaining how they work and their limitations. Furthermore, in order to overcome these limitations, new libraries that offer alternative multiplayer functionalities are programmed.
INDEX

STRUCTURE OF THE DOCUMENT ................................................................. 1

CHAPTER 1. INTRODUCTION ........................................................................ 3

1.1. Context .................................................................................................. 3

1.2. Problem statement ............................................................................... 4

CHAPTER 2. UNITY ...................................................................................... 5

2.1. Introduction ......................................................................................... 5
   2.1.1. Platforms ..................................................................................... 5
   2.1.2. Licensing ................................................................................... 6

2.2. Editor .................................................................................................. 6
   2.2.1. Windows ..................................................................................... 6
   2.2.2. Scenes ......................................................................................... 8
   2.2.3. Game Objects ............................................................................. 8
   2.2.4. Components ............................................................................... 11
   2.2.5. Meshes ....................................................................................... 13
   2.2.6. Prefabs ....................................................................................... 13

CHAPTER 3. UNITY NETWORKING .......................................................... 15

3.1. Introduction ........................................................................................ 15

3.2. Networking architecture ................................................................... 15
   3.2.1. Authoritative server .................................................................. 15
   3.2.2. Non-Authoritative server .......................................................... 16

3.3. Networking Elements ........................................................................ 17
   3.3.1. Server ....................................................................................... 17
   3.3.2. Network Player ........................................................................ 17
   3.3.3. Network View ........................................................................... 17
   3.3.4. Remote Procedure Calls ............................................................ 18
   3.3.5. State Synchronization ................................................................. 19
   3.3.6. Master Server ........................................................................... 20

3.4. Unity Networking limitations ............................................................. 21

CHAPTER 4. UNITY NETWORKING APPLICATIONS ............................. 23

4.1. Introduction ........................................................................................ 23

4.2. Remote Procedure Call (RPC) ........................................................... 25
   4.2.1. Introduction ............................................................................... 25
   4.2.2. Server ........................................................................................ 25
   4.2.3. Client ......................................................................................... 27
   4.2.4. Remarkable points ................................................................... 28

4.3. State Synchronization ........................................................................ 30
   4.3.1. Introduction ............................................................................... 30
4.3.2. Server .................................................................................................................. 30
4.3.3. Client .................................................................................................................... 32
4.3.4. Remarkable points ............................................................................................... 33
4.3.5. Differences between Delta Compressed and Unreliable ...................................... 34

4.4. RPC and State Synchronization hybrid ........................................................................................................... 34
  4.4.1. Introduction .............................................................................................................. 34
  4.4.2. Server ..................................................................................................................... 35
  4.4.3. Client ....................................................................................................................... 36

4.5. Custom Networking ..................................................................................................................... 37
  4.5.1. Introduction .............................................................................................................. 37
  4.5.2. Server ..................................................................................................................... 39
  4.5.3. Client ....................................................................................................................... 40
  4.5.4. Remarkable points ............................................................................................... 41

CHAPTER 5. DELAY EVALUATION ......................................................................................... 43
  5.1. Introduction .............................................................................................................. 43
    5.1.1. Delay measured ................................................................................................. 43
  5.2. Results ..................................................................................................................... 44
  5.3. Conclusions ............................................................................................................ 46

CHAPTER 6. GANTT CHART ................................................................................................. 47

CHAPTER 7. CONCLUSIONS ............................................................................................... 49
  7.1. Future Work.............................................................................................................. 49

CHAPTER 8. BIBLIOGRAPHY ............................................................................................... 51
STRUCTURE OF THE DOCUMENT

The main objective of this document is to explain in detail the different alternatives that Unity offer to develop multiplayer game functionalities.

In first chapter a general introduction of the project is explained, along with the main objectives.

In second chapter an introduction of the Unity engine is detailed as well as its main features and how the default built-in multiplayer functionalities of Unity work. Furthermore, limitations of those functionalities are highlighted.

In third chapter a solution for the previous limitations of Unity multiplayer is proposed. A new custom networking functionality is introduced.

Different test scenarios are described in chapter four. Each one of these text scenarios is related to a different multiplayer technique (included the custom one).

In chapter five each one of the previous scenarios are analyzed. The results of the scenarios are studied in chapter five.

In chapter six the planned scheduler for the most important tasks of this thesis is shown by means of a Gantt chart.

Conclusions are exposed in chapter seven. In addition, a list of possible future tasks is shown.

Finally, the consulted documents are listed in the bibliography.
CHAPTER 1. INTRODUCTION

This chapter contains the introduction whose aim is to describe the project and highlight the main contributions of this thesis.

1.1. Context

In the recent years, the sales growth of video games had been increased exponentially; during the year 2012 14.8 billions of dollars were expended by consumers in gaming content in the U.S.A. (see [1]). Despite of this quantity is approximately a 10% less than in 2011, making 2012 the second consecutive year that has a decrease respect the previous years, is still a very significant number. It highlights the importance of this market that can be positioned on the same level as the film and music industries. The revenue in Spain during 2012 was of 822 million of euros (see [2]).

Nowadays, the video game industry represents a wide market composed by users that go from teenagers to seniors with an average age around 30 years old. On the other hand, the market is split by gender with a distribution of 47% and 53% of female and male users respectively (see [3]). Because of this market growth, new requirements to reach a higher variety of users appeared: the need of new types of games, development tools and technologies to fulfill the increasing demand of video games.

Two triggered factors of this market growth are the influence of smart phones and social networks. More than 420 million of smart phones, mostly iOS and Android devices, had been sold in 2011 (see [4]), and hundreds of thousands of applications appeared for those devices (see [5] and [6]). In the case of Apple, about a 20% of the total applications of the App Store are games (see [7]). A good example of a company that develops games for iOS is Rovio and its game Angry Birds that has achieved 1 billion downloads (see [8]).

Another important factor is the popularity of the social networks, which has a big amount of users interconnected among themselves. Regarding the video games, it has become an interesting issue for social network providers to offer games as a new service. An example of this is Zynga and its games for the Facebook social network, whose revenue was of $329 million for the 1st quarter ending (see [9]).

A lot of different companies that develop video games emerged to fill this gap in the market, like the ones previously mentioned. One of these companies is Unity Technologies, which has developed a tool called Unity (see [10]).

Unity is a development tool to design games that has over 2 million registered developers (see [11]). Furthermore, this tool is cross-platform, which allows the developers to create games for different platforms. Although Unity is defined like a game engine, other applications can be developed by Unity as
architectural walkthroughs, instructional interactive demonstrations, training simulations or product visualizations.

1.2. Problem statement

The main objective of this thesis is to perform an evaluation of the multiplayer networking options that Unity provides in its standard distribution, which are addressed to the creation of multiplayer games. A multiplayer game is a type of game in which more than one user is playing simultaneously. Adding multiplayer functionalities in a game is rather important because 62% of its audience plays multiplayer games (see [3]), and that corresponds to a huge portion of the market pie.

Unity networking components will be explained as well as an analysis about how they work. The functionalities provided by Unity to achieve networking will be studied.

Unity networking has some limitations, and because of them a new library able to perform network functionalities will be programmed in order to provide developers with an easy and intuitive way to use multiplayer capabilities in their games. From now on, these methods will be referred as custom networking.

Unity networking functionalities and custom networking functionalities will be evaluated in terms of delay.
CHAPTER 2. UNITY

In this chapter, the Unity game development tool is explained.

2.1. Introduction

Unity is a powerful development tool to design games, which has been published by Unity Technologies and has over 1.7 million registered developers; some of them are very important in the video game sector like Microsoft, Ubisoft or Electronic Arts.

Unity is composed by two main blocks: editor and engine.

- **Unity integrated editor**: It is aided to develop and design the application. Besides, it can run the application by emulating the target platform; hence it shows exactly what users will see in the released version for their platform.
- **Unity engine**: It includes rendering, lighting, terrains, substances, physics, path finding, audio, programming and networking functions.

2.1.1. Platforms

Unity is a cross-platform tool, which is one of its most interesting features because developers can design games for different platforms easily. Some of these platforms are highlighted below:

- PC
- MAC
- Web Player
- Flash
- iOS
- Android
- Nintendo Wii
- Sony PS3
- Microsoft XBOX 360

It is important to realize that some minor changes to migrate a project from one platform to other platform could be required, as for instance the module used to process the inputs inserted by the user; it must be adapted according to the platform. In the case of a desktop game, the inputs correspond to the keys pressed from the keyboard, whereas in a mobile game the inputs are performed by touching the screen Developers should take into account the type of device in which the Unity application is going to run, because technical specifications like use of CPU or amount of memory application can use are important limitations. Once the project is adapted to the platform it is just really easy get the corresponding application.
2.1.2. Licensing

There are eight different types of licenses for Unity: Unity, Unity Pro, Flash, Flash Pro, Android, Android Pro, iOS and iOS Pro (see [12]).

In order to develop applications for any platform that is not included in the previous licenses, a third party license from the appropriate company must be purchased.

The development of this thesis and all of the running tests are performed under the standard Unity license. This license has the advantage that is free and it includes all the multiplayer functionalities of Unity.

2.2. Editor

An integrated editor is the tool that allows developing and designing the game. The figure 2.1 shows the look of a project that has one cube in the scene.

![Unity editor](image)

Fig. 2.1 *Unity editor*

2.2.1. Windows

The editor is divided in five important sub-windows: scene, game, hierarchy, project and inspector. Developers can change the position and size of these windows, customizing the interface of the editor.
2.2.1.1. Scene window

This window view represents a 3D environment where all the game objects in the scene are placed. To navigate through this 3D world, it is possible to use the four first buttons of the toolbar (figure 2.2) or their shortcuts.

- Hand tool (Q key): while this tool is selected it is possible to click into the scene window and drag the mouse to move the current view of the scene.
- Move (W key): Move the selected game object/s by dragging the mouse over the scene window.
- Rotate (E key): Rotate the selected game object/s by dragging the mouse over the scene window.
- Scale (R key): Scale the selected game object/s by dragging the mouse over the scene window.

![Toolbar buttons with “Move” tool selected](image)

Fig. 2.2 Toolbar buttons with “Move” tool selected

2.2.1.2. Game window

Game window shows the running application when “Play mode” is activated. To activate “Play mode” the “Play” button in the editor toolbar must be clicked. During this mode, developers can test the game by playing it. In addition, a console will appear, showing information about compiling and runtime errors. Developers can write strings into this console by means of script “print” function.

2.2.1.3. Hierarchy window

This window displays a list of all the game objects that are in the current scene. Children game objects are shown with a hierarchal order from their parents. Disabled objects are shown with its name in light grey color.

2.2.1.4. Project window

Project window shows a list of all the resources that the project is using. These resources are the files inside the “Assets” folder of Unity directory.

By right clicking over the project window a box dialog appears, which allows developers to create new items such folders, scripts and prefabs or delete the existed ones. Also, import and export function will be shown in this dialog box. Another useful feature is the “Show in Explorer”, which will open a new file system explorer with the appropriate directory.
Every time an item is modified in the project window, it is automatically modified in the “Assets” folder of Unity directory.

2.2.1.5. Inspector window

All the information concerning a game object is offered by the inspector window. This information is listed below:

- Checkbox that indicates if game object is enabled or disabled.
- Name of the game object.
- Checkbox that indicates if game object is static.
- The tag of the selected game object (“untagged” by default).
- The layer of the selected game object (“Default” by default).

Furthermore this functionality contains a small lock icon that has two states: open (by default) or closed. If closed, the inspector will keep displaying the selected game object even if developer selects another one.

In addition to the previous information, the inspector window lists all the components that the selected game object contains. The values of some attributes of the components can be changed manually from this window, changing the behavior of the game object.

2.2.2. Scenes

Scenes contain all the game objects of one level of the game. Each level can have a different environment and objects. Scenes are useful because they allow building the game in different stages. It is a good way to keep the game organized.

By default, when a Unity project is opened, no scene is loaded. Therefore, be sure to open the scene you want to work with.

2.2.3. Game Objects

A game object is a container that has different functionalities attached to it. Each one of these functionalities is given by a component; hence a game object is an object that has some components that define the behavior of the object. Every object that exists inside a scene is a game object. In addition to components, each game object in the scene has a tag, a layer and a name associated to it, as well as an indication if it is static and if it is enabled or disabled.
A tag is a key word associated with the game object. It is used to identify different types of game objects.

A layer is an identifier associated to a game object that a camera will use to filter which objects will be rendered.

Static attribute of the object is related to occlusion culling. If an object is static, other objects those are behind the static one are not rendered because user cannot see them.

Unity has some default game objects with some components attached that developers can use immediately in their games. The most used types of game objects are: cameras, texts, lights, particle systems, planes and some geometric figures like cubes or spheres.

2.2.3.1. **Empty game object**

This is the most basic type of game object. An empty game object does not have any component attached to it apart from the “Transform” component, which all game objects must have.

Some scripts can be attached to it. These scripts can contain code that will be executed independently from the game object behavior.

2.2.3.2. **Cameras**

Camera game object captures a portion of the 3D world of the game. This is what user will see when playing the game. When a camera is selected in the hierarchy window of the editor, an extra window, named “Camera Preview” appears, showing to the developer the image of the world that is captured at that instant of time. In fact, in the game window of the editor, what is shown is what the cameras are capturing.

Camera game object most important component is called “Camera”. This component is what really makes a camera object acts as a camera. By means of this component is possible to define some attributes like the background, field of view of the camera, what layers are going to be rendered, etc.

There are no limitations on the maximum number of cameras an scene can have, but in order to use more than one camera some considerations must be taken into account:

- Only one “Audio Listener” per scene. An “Audio Listener” is one of the components that cameras have by default. It acts like a microphone that captures the sounds produced in the scene and reproduces them using the user speakers. “Audio Listener” components from extra cameras must be removed.
“Camera” component has an attribute called “Normalized view port rect”, which defines the portion of the screen, in normalized units, in which the camera is going to draw what it sees. It is necessary to distribute correctly the view port rectangles of the cameras in order to not have overlapping.

An interesting option that makes easier to focus the camera is “Align with view”. When camera object is selected and this option is pressed, the camera automatically will focus in its camera view the portion of the scene that the developer is seeing in the scene window. This option can be found under “Game Object” tab in the tool menu of Unity editor.

2.2.3.3. Geometrical figures

Unity provides default geometrical game objects that can be added to the scene. These objects are cubes, cylinders, spheres, capsules and planes. Each one of these figures has three components: Mesh Renderer, Mesh Filter and Collider.

One important thing to take into account when using planes is that the Unity plane has 200 triangles. This is not good, because more triangles mean that more graphical resources will be consumed. A good option when using planes is to use some tool like 3DS max to model a plane of just two triangles and import it to Unity.

2.2.3.4. Texts

Texts are game objects that contain graphical representations of a text string. There are two types of texts: “GUI Text” and “3D Text”.

“GUI Text” displays a text in the screen. This text will be always perpendicular to screen. It has attached the “GUIText” component that allows, among other options, to write the text that will be displayed, select the font size and select the font type.

On the other hand, “3D Text” displays a 3D text that can be rotated and scaled. This object has two components: “Text Mesh” and “Mesh Renderer”. The first of them is the component that generates the text. This component defines the string to be rendered and the font to be used. “Mesh Renderer” defines the material of the font. So a “3D text” can have transparency, can use a texture in the characters, and other visual options.

“GUI Text” is visually poor, and transform functions like rotate and scale are not enabled to use with it. That is why the most commonly used in final products is “3D Text”.
2.2.3.5. Lights

Lights are game objects that illuminate the objects in the scene. There are four types of lights: directional light, point light, spot light and area light that requires Unity Pro license; thus, it is out of the scope of this project.

Directional light is commonly used to illuminate outdoor scenarios, because it simulates the sun very well. A direction light is a light that comes from the infinity and illuminates the entire scene through a certain angle.

Point light is an omnidirectional light that is generated in one point and have a certain radius of action. They are used to simulate objects like light bulbs.

Spot lights are a directional light generated in a certain point and the halo of light is cone shaped. They can be used, for example, to simulate a frontal car light or a lantern.

2.2.3.6. Particle systems

Particle system is a game object that generates particles. Particles are planes with a texture that are shown always perpendicular to screen. A particle system can simulate effects like a spark, rain, smoke, etc. which add amazing visuals to the game.

“Particle System” component defines the number of particles and the lifetime of the particles that will be generated, size, color, the material of the particle, the shape that the particle system will throw the particles (cone, sphere, linear shape ...), and movement speed, among others. Some of these properties like color and size can be changed over time.

2.2.4. Components

Components define the behavior of the game objects, and are associated to them. A game object can have attached more than one component. Components have properties that can be changed by programming or by the inspector window. When any of these properties is changed, the behavior of the game object changes too. Some of the most used components are listed below.

2.2.4.1. Transform

The Transform component is one of the most important components in Unity. This component is mandatory in all game objects, thus all the objects must have
it attached. It defines the position, rotation and scale of the game object. *Unity* uses two types of special data to represent positions and rotations. *Vector3* represents a position in the space (a point) defined by three coordinates (x, y and z) or a direction. For example, the *Vector3* \((0, 0, 1)\) is a vector its direction is the same as the z axis. *Quaternions* represent the rotation an object has, *i.e.* how many degrees it is rotated from z, x and y axis.

### 2.2.4.2. Rigid Body

When a *Rigid Body* component is attached to a game object, the game object will follow the physics laws, making it responsive to different forces like gravity.

For example, suppose a scene that has a plane located at origin and a cube whose start position is 1 meter over the plane. If the cube game object does not have a Rigid Body” component attached, the cube will float at 1 m. On the other hand, if the cube has a “Rigid Body”, the gravity force will affect to the cube, making it to fall.

This component can have activated the option of “kinematic”. When this option is activated the object will be a kinematic rigid body. This kind of rigid body is not affected by any force like gravity or collisions. Some examples of the use of kinematic option can be the following: an object its physics are controlled by a script but in certain situations it must follow the physic laws. This behavior can be achieved just by enabling and disabling the kinematic property. Another example of use of kinematic object can be an object that can push other objects.

### 2.2.4.3. Mesh Filter

This component indicates to the *Mesh Renderer* what mesh is going to be rendered. The mesh must be located somewhere in the *Unity* Asset folder.

A cube object has a *Mesh Filter* component with a variable to define that the object is going to render a *Cube Mesh*, which is a mesh that graphically represents the eight vertex and the six faces of a cube.

### 2.2.4.4. Mesh Renderer

This component takes the mesh from the *Mesh Filter* component and renders it in the scene. This component also indicates if the object will cast shadows when a light is projected into it, if other shadows that are projected into this mesh are rendered and also have a list of materials that will render the model.
2.2.4.5. **Colliders**

This component allows game objects to detect collisions with other game objects. A collider is like an invisible structure that a game object has around it. A collision occurs when a collider touches another collider. This structure can have different forms, like a sphere or a cube (Sphere Collider, Cube Collider, etc.). There are colliders that will fit better to a game object depending on its shape. For example, a cube object has a *Box Collider*, because it is the collider that best fit its shape. In addition, it is possible to combine different colliders to surround a unique game object.

2.2.4.6. **Scripts**

This component associates an existent script with a game object. A script is a piece of code that provides certain behavior to the game object. This code can be written in *JavaScript*, *C#* or *Boo* languages, and different scripts written in different languages can be used by the same object. *Unity* provides online documentation of scripting through its webpage (see [13]).

It is mandatory that every script attached to a game object extends *MonoBehaviour* class. This class provides some special *Unity* methods like *Start* and *Update*. *Start* is called when the *GameObject* is created in the scene and *Update* is called on every frame during game execution.

*Unity* uses *Mono* framework as the scripting framework. *Mono* framework is defined by the *Mono* team as “an open source, cross-platform, implementation of *C#* and the *CLR* that is binary compatible with *Microsoft.NET*” (see [14]). This is a very interesting feature because this framework is integrated into *Unity* so developers can use almost all the *.NET* classes in their projects. In addition, *Microsoft* provides online documentation about all *.NET* classes (see [15]).

2.2.5. **Meshes**

A mesh is a 3D object formed by triangles that has a material associated to it. A material defines which texture and which color is used for rendering.

It is important to note that *Unity* do not have any tool to create meshes. In order to use them in a scene, meshes must be created with other software (like *Autodesk 3Ds MAX*) and imported into *Unity*.

2.2.6. **Prefabs**

Prefabs are special game objects that can be reused along the application, in different scenes. Prefabs can be instantiated in order to create new game
objects in the scene dynamically. All the instantiated prefabs will be exactly the same game objects at the moment of creation, but it is possible to modify the characteristic of every one at a time. So it is possible to get two different game objects that come from the same prefab.
CHAPTER 3. Unity Networking

In this chapter, the architecture, elements and operation scheme of the Unity networking are explained. An application that uses Unity networking is explained in detail.

3.1. Introduction

Unity provides developers with networking function that can be used to develop applications. Networking allows that different devices can communicate between them through the network. By means of this communication users can synchronize their game objects in order other players can see or interact with them.

3.2. Networking architecture

Unity uses client-server architecture in order to achieve network communication between clients (machine running the game). Clients does not know each other, thus, a well-known server is needed. A server will be the common node between all the clients. If client A want to send a message to client B, client A will send a message to the server and the server will redirect it to client B. One advantage of this is that the clients still remain unknown for each other; they did not know IP address of other clients, so they have certain privacy.

Once the clients establish a connection with the server, they can send information across the network. To be able to establish the connection, the IP address of the server must be public and the client must know it. A public IP address is an address that can be reached through internet.

Server can be a dedicated server, i.e. a dedicated machine known by all the clients or a player’s machine that acts at the same time as a client and a server. The last approach is useful in LAN environments.

Unity can use two different types of servers: authoritative server or non-authoritative server. It is important to select a designing approach in advance, because depending of the type of server selected, the programming of the game will change significantly.

3.2.1. Authoritative server

Authoritative servers are the ones that simulate all the world complexity; that means that the server manages all the world objects and actions of the players.
Even the movement of the players as well as all the actions associated to the player are managed by the server. Although the server has the information of the world state it is not necessary that the server renders all the world objects. Server only needs to know the rules of the game and the state of the world, it is not important that server has the visual components of the objects, such meshes or textures.

In this approach, clients just send to the server the user's input. In addition, clients render the entire world in its current state. When the user generates a new input, the client will send a message to the server with the information of that input and will wait a server response. Then, the server will process that input, analyzing if it is valid or not. If it is valid, the server will simulate the user action and will change the world state. The resulted world state will be sent to client, so the client can update its own world state.

Authoritative servers have much more complexity than a non-authoritative one, but have the advantage that cheating is far more difficult to achieve for malicious users. A user can manipulate their own client application, making that the client sends a malicious message to the server. This malicious message will not spread through other players because the server will process it and it will see that this action made by the user is not allowed. Only valid actions are the ones allowed by the server. For example, a client instead of saying to the server that has killed an enemy, the client will send a message that contains information about what type of weapon the user has and the position where he is aiming to. Then the server will process this message and make the decision if the player hits an enemy or not.

One of the important disadvantages of this type of server is that clients must wait for the server response in order to know the result of the user input. This wait time depends mostly on the delay introduced by the network, so it is a parameter that cannot be controlled by Unity application. If this delay is high enough, it is possible that the user has a bad user experience. For example, if the user presses a direction key in order to move some character, is expected that the character moves, but instead that instant movement, the client will send a message to server, so the character will remain motionless until a response arrives. This delay between the user press a key and the character is starting to move can be really annoying for the user (even the user can decide to stop playing). There are some techniques that try to solve this network latency issue, the two more important of them are known as client-side prediction and lag compensation techniques.

### 3.2.2. Non-Authoritative server

In this approach, each client has its own world locally. Hence, the client controls all the logic of the game, since all the game objects are created by it. Every time the user performs an action or the state of the world change, the client sends a message to the server, notifying the updated state of the world. Then, the server will send the new state to all clients that are connected to it.
This type of servers is easier to implement than the authoritative ones, because these servers just receive messages from a certain client and distribute them across the network to the other connected clients. Server does not need to know what action the client does nor do any kind of extra processing.

The main problem of this type of servers is that the server must trust what the client says the user does. Users can try to cheat the server by sending him malicious messages, indicating some kind of actions that will benefit the user. The server does not have any way to distinguish malicious messages, so it will resend the malicious message to all the clients.

3.3. Networking Elements

In order to achieve networking Unity provides a Network class, which includes the scripting functions of the Unity’s network implementation.

3.3.1. Server

Regardless of the type of server chosen, a server is necessary in order to achieve a multiplayer game. By calling Network.InitializeServer the application will listen for messages in a developer’s defined port. Once the server is listening, clients can connect to it by means of calling Network.Connect function with the proper IP and port number.

3.3.2. Network Player

A Network Player data structure represents each one of the players that are connected to the game server. Network Player is identified by an IP address, a port and an identifier. Developers can access to this data by calling Network.NetworkPlayer.

3.3.3. Network View

Network Views are components that allow game objects to communicate with other game objects across the network.

Each Network View has an identifier called NetworkViewID. This ID is unique among all the network clients that are connected to the server. When a client connects to the server, the server will assign a list of NetworkViewID that the client can use. This ID is necessary to know who will be the receiver of the messages.
A shared network object can be created by using `Network.Instantiate` function. This function will create an instance of a prefab object, which must have attached a `Network View` component, in all the clients connected to the server. So when a client creates an object in his scene, the rest of the users see that object in their respective scenes. The `NetworkViewID` of the instantiated object is assigned by its creator, so other clients that want to interact with this object can know the ID, and send a message to its owner. The range of those IDs for each client is assigned by the server. Only clients that instantiate an object are the only one able to control the object. Other clients can see the object but they cannot change its position directly. The `Network.Instantiate` acts like a buffered RPC.

`Network Views` has two configurable properties: `State Synchronization` and `Observed`. The first one indicates the way this `Network View` can communicate with others objects across the net, and can have three different values:

- Off: this value indicates that the `Network View` only way to communicate with others is by means of RPCs.
- Reliable Delta Compressed or Unreliable: indicates what type of State Synchronization is used by the `Network View`.

The `Observed` property indicates the element that is going to be synchronized across all the clients of the network.

`Network Views` implement two different types of communication, `Remote Procedure Calls` and `State Synchronization`.

### 3.3.4. Remote Procedure Calls

A `Remote Procedure Call (RPC)` is a mechanism that allows applications to call special methods that are defined on a remote machine. A remote machine does not necessary mean a server, but any machine (a server, a client, a group of clients, etc.). These methods are very similar to standard methods; they can have as many parameters as developer specified but it must have a special parameter that defines the receivers of this call and the name of the remote method to execute. Unity assures clients will receive all this calls.

Typically these calls are used to trigger an event on remote machines. For example, if in a multiplayer game a user opens a door, it will send a RPC to the server. The server will process the RPC and will make a decision to open the door. If the door is finally opened, the server will send a RPC to all clients saying that they must open the door. RPC contains the name of the remote method executed in the machine, so the server will send an RPC with the following parameters. The first parameter of the RPC indicates the name of the remote method “OpenDoor”. The second one is who will receive this message (the server) and the third one is a parameter that identifies what door must be opened, for instance “interior door 1”.
A *RPC* parameter must be one of the following types:

- *Int*
- *Float*
- *String*
- *NetworkPlayer*
- *NetworkViewID*
- *Vector3*
- *Quaternion*

When calling a remote method there is a parameter that is always sent of type *NetworkMessageInfo*. This parameter has information about who is the *NetworkPlayer* that calls this remote method.

Each time a *RPC* is sent, the sender must decide if it is going to be buffered or not. Buffered *RPCs* are stored into a buffer, so when a new client establishes a new connection with the server, the server will send all the *RPC* in the buffer to the client. These *RPC* are sorted, thus the client will receive first the initial stored *RPC*. Therefore the new client is able to change its state and update it to the actual state of the server and all the other connected clients.

If developers want to share data between machines that is changing continuously in each frame, a way to achieve that is by means of state synchronization.

### 3.3.5. State Synchronization

*State Synchronization* is the method by which a Game Object that has a *NetworkView* component attached is able to share its state. Synchronize an object state means to share among the other users one of the following components: *Transform*, *Animation*, *Rigidbody* and scripts that extend from *MonoBehaviour*.

- The *Transform* component synchronizes the position, the rotation and the scale of the game object (does not serialize or send any parenting information). Every time any of these attributes of the game object are changed, *Unity* automatically serializes and sends the component with the new values to the server.

- The *Transform* component synchronizes the animation information (time, weight and speed).

- The *Rigidbody* component synchronizes the position, rotation, velocity and angular velocity of the game object.

- If a *MonoBehaviour* script is the synchronized component, this script should implement a *Unity* method called *OnSerializedNetworkView*. This method is the responsible of manage the variables of the script that are
going to be synchronized across the network. Each time the value of that variable is changed, this method is called.

The `NetworkView` component has a property called `Observed`. This property should have assigned the component that is synchronized.

**State Synchronization** has two operating modes, **Reliable Delta Compressed** and **Unreliable**. **Reliable Delta Compressed** mode sends the information with the new values of the component across the network every time the component values change. Only the new values are sent, for example, if the observed component is the `Transform` one, and the position of that game object changes but the rotation and scale remain the same, only the position is sent. Thereby the communication efficiency increases because Unity does not send non useful information (that the server already has). In addition, Unity assures that the new data will be received by the client and it is order (if data is lost, it will be sent again). Unity manages this communication in application level; the network layer uses **UDP**. By the other hand, **Unreliable** mode does not guarantee that the datagrams arrive to its receiver, so clients do not know if the data arrive to server or not. That is why all the information of the observed component is sent every time. Unlike previous example, if the only thing that is changed in a game object is its position, now the position, rotation and scale are sent. **Unreliable** mode is frequently used when the game object is continuously changing, for example a game where the player is constantly moving (so its position change). In this case, it is not critical if datagrams drop because client will send a new one with the new position. The only problem here may be that user will experience a bad game experience, depending on the frequency of the player movements, if some datagram is dropped because the player will see non continuous movement.

### 3.3.6. Master Server

**Master Server** is a special server that manages information about the game servers and the clients that want to connect to them. This scenario is useful when clients want to act like a server, for example if a user wants to host a multiplayer game. Also, it is useful when is needed that a dynamic instance of a server is created.

In the first case, a problem is that usually user machines do not have a public static **IP**, so other users cannot connect to them directly. Something similar happens when an instance of server is created on a remote machine. It is needed a way to provide to the clients information about who is the server of the game.

The client that wants to host a game must be registered in a **Master Server**. **Master Server** is a well-known server (all clients know its **IP** address and its port) that give information to clients about the game server they must connect. When a client hosts a game, it sends information to the **Master Server** about its
IP and port, and other information relative to the game like a name that identifies the game or the user that created the game. Other users can ask to the Master Server for the games they can join, and the Master Server sends them a list with all the games that are created. The user then select one of them, and the Master Server sends to the client the necessary information to connect to that game (IP address and port).

Unity Technologies provides a Master Server for testing purposes that developers can use.

3.4. Unity Networking limitations

Unity networking uses UDP as a transport protocol. That means that this protocol is always used by Unity networking applications. Although UDP does not take any action against delay and drop of datagrams, Unity networking can analyze the sequence of the datagrams sent and receive in order to detect unsorted and lost datagrams. Once detected, the datagrams can be resend or sorted. But depending of the type of application maybe it is preferable to use another transport protocol, for example TCP, which implicitly takes actions against packet loss and unordered packets. Also, TCP provides mechanisms of congestion and flow control.

One of the limitations of Unity networking is the fact that the only way to send a RPC (Response Procedure Calls) is in the reliable way. Thus, functions that does not have a direct impact into the behavior of the game but does some type of visual effects are strongly dependent of the delay between client and server and probably is better never call to the method that call it with a high delay.

RPCs and state synchronization have one important limitation: the type of data that can be used. RPCs can only send int, float, string, NetworkPlayer, NetworkViewID, Vector3 and Quaternion data types. State synchronization can synchronize the following types only: Boolean, char, short, int, float, NetworkPlayer, NetworkViewID, Vector3 and Quaternion.

Another limitation of Unity networking is that it does not implement any mechanism to avoid the delay introduced by transmission and process time of the packets sent to the server by the client. This delay can affect game play in a very negative way, for example, it can make the user feel the controls non responsive if the user press a button to move forward, it is expected that the character moves immediately when the user press the button. In the case of multiplayer games with authoritative server, when the user press the button, a “move” message is sent to the server, the server computes the message and sends a response to the client. Then, the character moves. If the delay is large enough, user will not feel the controls responsive enough and it will stop playing that game.
CHAPTER 4. UNITY NETWORKING APPLICATIONS

In this chapter a detailed explanation about how to use Unity networking is illustrated through a game application. Three networking functionalities provided by Unity are explained: RPCs and State Synchronization in Reliable and Unreliable mode. In addition, a new networking technique (custom networking) is proposed in order to overcome the Unity networking limitations.

4.1. Introduction

The game application has been derived in four different implementations according to the different techniques offered by Unity: RPCs, Reliable State Synchronization, Unreliable State Synchronization and a hybrid one, which mixes RPCs and State Synchronization. Therefore, a fair comparison of the flow diagram of the different approaches can be achieved.

The application implemented is simple and focused in networking operations. It consists in a two dimension (x, y) square delimited space in which players are appearing as they are connecting to the server. All the players are “shared” objects, which means that any user can see all connected players moving in the space. Players are represented by colored cubes and each cube has a different color (see figure 4.1). Users can move its cube in the desired direction by pressing the arrow keys of their keyboards (allowed movements are up, down, right and left). A cube is always moved when the user presses a key whenever it is not blocked by a space delimiter or other player.

![Fig. 4.1 Game with four users](image)

For each networking implementation, a different server and client must be specified. Concerning to the server selection, an authoritative server has been chosen because it is the most common in multiplayer games.
All implementations have the following general system operation (see figure 4.2):

1. The Server starts and waits for some request from the client.
2. The client detects that a key is pressed by the user and sends to the server that the user wants to perform a movement of its player.
3. The server processes the information sent by the client and checks if the player can perform the movement.
4. If the server decides that the player can move to the desired position, the server moves the cube of the user (updates the global world state) and informs to all clients that they need to update the cube's position (update its local world state).
5. The client updates its local copy of the objects.

Fig. 4.2a Server Flow Diagram

Fig. 4.2b Client Flow Diagram
It is important to remark that the server must not have any graphical mesh. Only with the cube colliders the server is able to compute the operations that fulfill the conditions to know if a player can make the movement or not. By the other hand, the client does not have to calculate any logical behavior of the game objects, just need to send to the server the user's input and renders its local word. Hence, the computation complexity is only in the server and not in the client side.

4.2. Remote Procedure Call (RPC)

4.2.1. Introduction

In this implementation, the communication between the server and the clients is performed by means of Unity's remote methods calls (RPC). The calls to these remote methods are processed sequentially, if any of them is lost it will be resend automatically.

4.2.2. Server

The RPC Server main function is to create and maintain the world state. It is also the responsible to evaluate the actions that users want to performe and decide if they can be performed or not.

4.2.2.1. Flow diagram

The flow diagram of the RPC Server is shown in figure 4.3:

1. Server is initialized.
2. The server instantiates in the network a shared game object called Communicator. It allows to send and receive the RPCs. This game object has assigned a Network View component with its State Synchronization property set to Off, therefore its Observed property is set to None. This is because this server is only using RPCs.
3. At this point the server is ready to receive RPCs from the clients.
4. When a new client is connected, the server locally instantiates a cube object (that means that the server has its own player objects) and assigns an id to it. In addition, it sends a RPC through the Communicator to all the connected clients telling them that a new player has joined to the game (also sends the id assigned to the new player).
5. When a new client is disconnected, the server destroys the cube object related to the disconnected client. Besides, it sends a RPC to all the remaining clients telling them that a specific player has been disconnected from the game (also sends the id assigned to the disconnected player).
6. When the server receives a **RPC** from a client, it checks if the cube player can perform the movement by casting a ray from the player current position and in the direction the user wants to move; thus, it looks if the ray collides with some obstacle (delimiter or player). If a collision is detected, it means that the movement cannot be performed, hence the server sends to the client who sent the **RPC** a response indicating that the cube is not moved. On the other hand, if no collision is detected, the server will send a **RPC** to all the connected clients with the new position of that cube.

![RPC server diagram](image)

**Fig. 4.3 RPC server**
4.2.3. Client

Each client application represents a user that is connected with the server.

4.2.3.1. Flow Diagram

The flow diagram of the RPC Client (see figure 4.4) is explained below:
1. Client application is started.
2. Client establishes a connection with the server.
3. Waits for incoming RPC. All RPCs are buffered; it means that always the first RPC that a client will receive will be the one related with the instantiation of the Communicator object in the scene. This RPC is managed internally by Unity and it creates the Communicator object in the client. From now on, the client can send and receive RPCs by means of the Communicator object (that was instantiated by the server).
4. The second RPC that a client will receive is always a RPC indicating that a new player has joined the game. When a client receives this type of RPC, it locally instantiates a new Cube in the scene and then checks the cube id, if it is his own cube or it belongs to another player. If it is his/her own cube, it instantiates a cube object and waits for user input. User can now try to move that cube by pressing the directional keys. Otherwise, it instantiates a new player and assigns its corresponding id.
5. A client also receives a RPC when player is disconnected. When this occurs the client looks for that player cube (by checking its id) and destroys it.
6. Another type of RPC that a client can receive is the one telling him that a cube has been moved. When the client receives that one, it looks for the cube to which the RPC refers to, and moves it to the indicated position (updates its local world state).
4.2.4. Remarkable points

The following sections are important specific cases in a Unity’s RPC related scenario that developers should consider in order to avoid system malfunctions and errors.

4.2.4.1. Declaration of remote methods

The Communicator object instantiated in the network by the server must contain a script with the declaration of all the remote methods that are going to be called in the client as well as in the server side. The Communicator prefab must be included in both the server and the client applications. The difference between these prefabs is that the server only has the implementations of the methods called by the client, whereas the client contains the implementations of the methods called by the server.
4.2.4.2. Include all prefabs in the executable file

The prefabs that are not instantiated directly from the code in a Unity project must be assigned in a dummy variable in a script. When using RPC technique, Communicator client’s prefab (that is only instantiated by the server and not by the client) is contained in an array list of one dummy script attached to an empty scene game object. This is because Unity thinks that the prefab is not going to be used (because no script in this project is instantiate it) and therefore, when the "exe" file is created, Unity will not add the prefab file into the executable file; therefore, when it runs and the server tries to instantiate that prefab an error will occur. In order to solve this problem, the prefab must be used by some script of the Unity project. The addition of the prefabs to the dummy list must be performed by the Unity inspector.

4.2.4.3. Shared spot problem

When the server checks if a cube can be moved, it casts a ray from the cube in a certain direction and looks if the ray collides with some obstacle (delimiter or player). This requires that when the ray is casted all the players must be rendered in the correct place. It may happen that two players want to move to one spot (see fig 4.5). First, the blue player sends a RPC to the server indicating that he/she wants to move to the spot, then the frame ends and the system renders the world state. Next, the black player sends a RPC to the server indicating that wants to move in the same spot of the blue player. The ray will collide and the move will not be done.

![Fig. 4.5 Two players want to move to the same spot in different frames](image)

On the other hand, the RPC of the black player may be processed before the world state is drawn again, i.e., it is processed in the same frame (see figure 4.6). The ray will not detect that the spot is already taken by the blue player resulting in an erroneous situation where both players are located in the same spot. Therefore, it is necessary to store all player current positions and check them programmatically in addition to use the ray.
4.2.4.4. **Sleep Screen**

When in Editor Mode (*Windows*), if server is initialized and screen goes to sleep, *Unity* application will enter in “non responding” state.

### 4.3. **State Synchronization**

#### 4.3.1. **Introduction**

In this implementation, the communication between the server and the clients is performed by means of *Unity’s State Synchronization*.

#### 4.3.2. **Server**

As well as with the *RPC Server*, the *State Synchronized* main function is to create and maintain the world state. It is also the responsible of evaluate the actions that users want to perform and decide if they can be executed or not.

#### 4.3.2.1. **Flow diagram**

The flow diagram of the *State Synchronization* Server is shown in figure 4.7:

1. Server is initialized.
2. At this point the server is ready to manage new client connections.
3. Every time a new client is connected, the server instantiates in the network a shared game object called *Cube*. It allows writing and sending synchronized information to that particular client. This game object has assigned a *Network View* component with its *State Synchronization* property set to the proper mode of operation (i.e. *Delta Compressed* or *Unreliable*) and therefore its *Observed* property is set to the *Cube*’s script. By observing the *Cube*’s script it is possible to share and
synchronized some of its script variables. In this case these variables are the position of the player.

4. When a client is connected, the server also loads into the scene a Communicator game object created in the network by the client. Unlike the RPC scenario, in this case is necessary to create more than one synchronized game object. This is because the member that instantiates the game object in the network is the one able to change some synchronized properties of it, while the other members are only able to read that data. Therefore, if the server needs to send information to the clients is necessary that the server creates a shared game object; and if a single client wants to send information to the server, client must instantiate in the network a game object. So for each client, a Communicator object will be created.

5. When a new client is disconnected, the server destroys the shared Cube object that represents that client. Internally also sends a message to all connected clients telling them that one client is disconnected and its objects must be deleted.

6. When the server detects that some synchronized variable of the script attached to the Communicator object changes, it identifies to which player the Communicator objects belongs. Then it tries to move the cube associated to that player to the position sent by means of the Communicator object. If the cube can be moved, then the server updates the synchronized position of the Cube object. If not, the serves does not do anything.
4.3.3. Client

As well as in the RPC case, each client application represents a user that is connected with the server.

4.3.3.1. Flow Diagram

The flow diagram of the RPC Client (see figure 4.8) is explained below:

1. Client application is started.
2. Client establishes a connection with the server.
3. Waits for incoming RPC. All RPCs are buffered, it means that always the first RPC that a client will receive will be the one related with the instantiation of the Communicator object in the scene. This RPC is managed internally by Unity and it creates the Communicator object in the client. From now on, the client can send and receive RPCs by means of the Communicator object (that was instantiated by the server).
4. The second *RPC* that a client will receive is always a *RPC* indicating that a new player has joined the game. When a client receives this type of *RPC*, it locally instantiates a new Cube in the scene and then it checks if it is his own cube or is another player cube by comparing its *id*. If it is his/her own cube, it instantiates a cube object and waits for user input. User can now try to move that cube by pressing the directional keys. If it is related to other players cube, just instantiates a new player and assigns its corresponding *id*.

5. A client also receives a *RPC* when player is disconnected. When this occurs, the client looks for that player cube (by checking its *id*) and destroys it.

6. Another type of *RPC* that a client can receive is the one telling him that a cube has been moved. When the client receives that one, it looks for the cube to which the *RPC* refers to, and moves it to the indicated position (updates its local world state).

4.3.4. **Remarkable points**

This section describes important points that should take into account by developers in order to use the *State Synchronization* technique.
4.3.4.1. Communication between clients

A client is always connected directly with the server, never with other clients. Therefore, the data that clients sent to each other always goes through the server. Suppose client A instantiates an object in the network and synchronizes a script member field called hitpoints. Client A is able to write into hitpoints, for example, client A assigns the value of 10 to hitpoints. What is expected is that client B should be able to read it. But the real behavior is that when client B reads it, it assigns to hitpoints of client A the value of 0 instead of 10. The problem in this case is that server does not update automatically the hitpoints new value through the other clients and instead of write a 10, it writes a 0 (default value). To solve this problem, it is necessary that the server reads the hitpoints value written by client A and then it must write it again. Thereby, all the other clients will read the right value of hitpoints.

This problem only occurs when the synchronized element is a script (member field of the script). When synchronizing the position, for example, it is not necessary to do it, because the server automatically updates the state of the object in all clients.

4.3.5. Differences between Delta Compressed and Unreliable

State Synchronization has two different operation modes: Delta Compressed and Unreliable. The first one assures that all the messages are ordered and lossless (if the message is lost, an automatic retransmission it is performed by Unity). On the other hand, Unreliable mode does not sort messages nor takes any action against message loss. In addition, Delta Compressed mode only sends messages when the information has changed, unlike Unreliable, which sends the complete message although the information has not changed. The mode State Synchronization is going to use must be specified by the Network View component.

4.4. RPC and State Synchronization hybrid

4.4.1. Introduction

In this implementation, both methods RPC and state synchronization are used. Clients communicate with server by means of RPCs and servers communicate with clients using state synchronization.
4.4.2. Server

The server receives *RPCs* from the clients that are connected, then process them and, if necessary, updates the world state making changes in the appropriate client objects by means of *state synchronization*.

4.4.2.1. Flow diagram

Figure 4.9 shows the flow diagram of the hybrid server.

1. The server application is started.
2. The server instantiates a shared object in the network. This object is responsible of receive the *RPCs* sent by the clients.
3. Now, the server waits until an event occurs. There are three types of events that the server can detect: a client is disconnected, new client is connected or a *RPC* is received.
4. In the first one, the server destroys all objects related to the disconnected client (hence, all the clients destroyed those objects too).
5. When a new client is connected, the server instantiates in the network a new *cube* object that represents the new player (this new object is created in all clients).
6. If a *RPC* is received, it is processed. Basically the *RPC* has information about the client that sent it and the action the user wants to do (if he wants to move the cube up, down, right or left). The server checks if that movement can be made. If cannot move the cube (because it collides with a wall or with other cube), the server does nothing. On the other hand, if the cube can be moved, the server will move it to a new position and then it will synchronize the new position with all the connected clients (resulting in that all the clients will move that cube to the correct position).
4.4.3. Client

The client sends *RPCs* to the server in order to try to do some action (move the cube in certain direction). The client automatically updates the position of all the cubes in the scene by means of *state synchronization*.

This is the simplest client of all previous implementations.

4.4.3.1. Flow diagram

Client flow diagram is represented in figure 4.10.

1. Client connects with the server. Once connected, the client created in the scene the object that the server instantiated in the network. By means of this object the client now can send *RPCs* to the server.
2. Client waits for user input. When uses presses one of the movement keys (up, down, right and left arrows), client sends a *RPC* to the server indicating that the cube that belongs to this client wants to move in the chosen direction.
3. When the server decides to update some object position, it synchronizes the new position with all the clients, so the cubes will automatically be moved. Thereby, the world state is always shared with the clients.

4.5. Custom Networking

4.5.1. Introduction

A set of custom networking functionalities are developed to try to solve Unity networking limitations. Custom implementation does not use any of the network functionalities provided by Unity but it is achieved by programming sockets and sending bytes through them.

As well as in previous implementations, custom networking needs a server where all the clients are connected. Then, the clients can send messages to server, and vice versa. These messages can be broadcast or unicast (just one specific destination); can use TCP or UDP transport protocols; can be some default messages that are defined by the system (Create, Destroy, etc.) or can be custom messages defined by the developer. Custom messages call a remote method defined by the developer and can contain any type of data (integers, floats, strings, custom classes, etc.).
Developers just need to import the custom networking library into their project and then call the properly methods wherever they want. Also, they must program the remote methods they want to call.

4.5.1.1. Types of messages

The types of messages are the following:
- RequestPort
- SendId
- Create
- Destroy
- Ok
- Error
- DeleteUser
- ServerLost
- Custom

RequestPort message is first sent by the client to the server in order to ask for an *UDP* port. When the server receives this message, it sends a RequestPort response to the client containing his assigned *UDP* port. In addition, the server also sends a SendId message containing a numeric unique identifier for that specific client.

Create and Destroy messages can be sent either by the server or by the client. Create message makes a request to create a new game object (from an existing *prefab*) while Destroy message makes a request to destroy a specific existent game object in the scene.

Ok and Error messages are messages sent by the server and they return a result for a client request. Ok if the client request has been successfully processed and Error otherwise. Note that sending Error message does not have to mean that an error occurs, but for some reason the client request failed to perform. For example, if the user wants to move its character across a wall, the client will send a message to server and the server when process the message will determine that the client cannot move there, so an Error message will be sent to the client.

DeleteUser message are sent by the server when it detects a user has been disconnected. When a client receives this message, it will destroy all the objects and information related with that user.

ServerLost message is a message sent by a client to himself when it detects a failure in the socket (server is down, client disconnected, network unreachable, etc.).

Custom message is not a predefined system messages like the previous ones, but a customizable message by the developers. This message calls a remote method in the client or the server. This remote method must be previously
programmed by the developer, can be located in any object of the project and can accept any type and number of parameters.

4.5.2. **Server**

Server manages client connections and disconnections, receive messages from connected clients and processes that messages. Then, if needed, it changes the world state and sends messages to clients.

4.5.2.1. **Flow diagram**

The flow diagram of the custom server is shown in figure 4.11.

1. When the server starts, it listens in a **TCP** port for new client connections.
2. Then a client can be connected to the server and once the connection is successful, the server creates two output queues for that client: a **TCP** queue and an **UDP** queue. Server will check if those queues have messages, and every time they have, it will send the message through the **TCP** or **UDP** socket to the client.
3. On the other hand, clients can communicate with the server by sending messages through the socket (either **TCP** or **UDP** messages). When a new message arrives through the socket, the message is enqueued into a previously created input queue. Hence, the server have three threads and two sockets for each client: one thread that reads the bytes sent by the clients through a **TCP** and an **UDP** socket, another one that sends messages through the **TCP** socket and the third sends messages through the **UDP** socket. The queues are mutual exclusive queues, so the threads and the main thread of the Unity application can access any queue at any moment. If a thread tries to access a queue that another thread is using, the thread will wait until the queue is available (i.e. the other thread finish reading or writing into the queue).
4. Main Unity application thread checks if there are new messages into the input queue. If there is a message, it is dequeued and processed.
5. One type of special message is the **requestPort** message. It is always sent by the client in order to get an **UDP** port number to send **UDP** messages. When the server receives this message, the server sends to the client two **TCP** messages, one containing a **clientID** assigned to that client, and another one containing the **UDP** port that are listening for client **UDP** messages.
4.5.3. Client

Custom client connects to the server and sends custom messages by means of the socket. Each message represents a user action. Client also process server responses in order to know the result of each action.

4.5.3.1. Flow diagram

The flow diagram of the custom client is shown in figure 4.12.

1. When client is started it creates two input queues (one for TCP and another one for UDP messages) and one output queue (to send received messages from socket to main thread application).

2. Then the client connects with the server by means of a TCP socket and sends a requestPort message to the server. Once is connected, it starts listening for receiving TCP messages. When a requestPort response message is received, the client connects to the server by means of an UDP socket, using the port number contained in the requestPort response message. The client now listens by both the TCP and the UDP socket.
3. When the client receives a message from the server, it processes the message and if necessary changes its local world state. After that it waits until another message is received.
4. When the client detects that one of the output queues (TCP or UDP) have some element, it gets the message and sent it to the server by means of the properly socket.

![Fig. 4.12 Custom client](image)

### 4.5.4. Remarkable points

#### 4.5.4.1. Threads in Unity

One of the limitations of Unity is that it is not thread-safe. That means that only is possible to call Unity methods and components from the main application thread. The implementation of custom networking involves threads and those threads may need to change the world state (for example, by moving some scene game object). Because it is a thread it cannot do these changes directly. It needs to call some method in the main application thread that can interact with the scene objects. Hence, it is necessary to create a bridge between those two threads. The way this bridge is created is by means of queues. Both the main thread and other threads can request access to the mutual-exclusive queue that allows communication between the threads, more specifically to send messages between them.
CHAPTER 5. DELAY EVALUATION

In this chapter, a delay evaluation of the different techniques discussed in chapter 4 is explained. The evaluation is realized by measuring the total delay for each one of the previous techniques.

5.1. Introduction

Delay is a very important parameter in reference to multiplayer games. It is possible to classify multiplayer games in two big groups: first one are the games that require low delays (because they are in real time) whereas the other group defines the rest of games that are not critically affected by the delay, for example a turn based game.

For each technique (RPC, State Synchronization Reliable, State Synchronization Unreliable, RPC and State Synchronization hybrid, custom UDP, custom TCP, and custom TCP without Nagle algorithm) some delay measurements are realized, where the average delay and the standard deviation are also calculated.

Nagle algorithm is a property of TCP sockets. When it is enabled, data is not sent immediately yet the sender waits until its output buffer is full. This is important because TCP have a huge overhead and it is more efficient (header bytes versus data bytes relation) when a packet contains more useful data. When it is disabled, the socket sends a packet when it detects some bytes in the buffer (the packet is sent even if it is not filled, so it is possible to send a packet with more header bytes than data bytes).

5.1.1. Delay measured

The delay measured is defined as the interval of time elapsed since the user press a key until the response of the server is processed by the client. Hence, the delay is composed by five components: process delay in client, propagation delay (end-to-end), process delay in server, propagation delay (end-to-end), and process delay in client. It is important to note that this delay should be as low as possible in order to provide a good user experience to players for applications that are delay sensitive.

The delays are measured by each client and stored in a log file (plain text file). When the user wants to move the cube (player character), he/she presses the corresponding direction arrow key. Then the client assigns a sequence number to the action that is sent to the server and when the response associated to that sequence number is processed, the client computes the delay.
A set of test were proposed to calculate the medium delays of each one of the networking methods. The methodology for these tests is the following:

1. Server with the proper networking technique is started.
2. A single client is connected for 5 min.
3. Client is disconnected.
4. Client writes a log with all the measures taken in that interval of time.

All machines are connected to the same local area network. With these settings, propagation delay will be equivalent for each one of the tested techniques. The test took a five minutes because this time is an adequate interval to study the behavior of the different techniques (during this interval the behavior is quite constant).

The default behavior of the client applications has been changed to automatically move the player in a random direction every 0.2 seconds to automate the process and to assure a constant interval time between sent actions. The value of 0.2 seconds was selected because it is an average time a real user can perform actions on the keyboard.

One of the objectives of these tests is to compare the delay of the Unity networking techniques with the custom techniques. One of the conditions custom networking must fulfill in order to be used in commercial applications is that the latency obtained is similar to the Unity networking. This is required to assure that custom networking can be used in all types of networking applications; both delay tolerable and not delay tolerable.

### 5.2. Results

**RPC** technique is the Unity networking technique with the lowest delay. When analyzing the results of the **RPC** test, the first noticeable observation is that even with one client connected, the delays are around 0.016 s or around 0.033 s (see figure 5.1). The explanation of this behavior is because the application is running at a frame rate of 60 fps, hence the time between frames is $1 / 60$ (0.016 s). It is important to remark that the minimum delay corresponds with the inverse of the fps application value and all the delay measurements are near some multiple of this value. The delay is not measured until the server response is processed, and Unity seems to process the packet received at the start of each frame. That is why in the measurements there is an offset corresponding to the number of frames that have passed since the packet was sent.

Both **State Synchronization Delta Compressed** and **Unreliable** have higher delays than **RPC** technique (see figure 5.2). Also, they are not as stable as the **RPC** technique (the delays of the different requests vary more than in the **RPC** case). An important difference between **Delta Compressed** and **Unreliable** is that the first one does not calculate the delay of the 1500 requests because if the server concludes that the user should not move, it does not send a response to the client (this can happen when the user is blocked by the wall, its position will remain the same) and therefore the delay is not calculated.
Hybrid technique behavior is similar to State Synchronization ones. The delays of this technique are not as low the RPC one, but they are not as high as the ones of State Synchronization. This is expected because the hybrid technique uses both RPC and State Synchronization. As happened with Delta Compressed technique, some delays are not computed.

Custom TCP with Nagle algorithm has the higher delay between all techniques. This is expected because the requests wait until the output buffer is full. On the other hand, custom TCP can be equated to RPC technique in terms of delay and stability. Hence, it is possible to use Custom TCP in a commercial application.

Custom UDP technique is similar to the State Synchronization ones, but there are some requests that have higher delay. This is because when using UDP it is possible to lost packets and they can arrive unsorted, hence when using UDP it is necessary to control loss and disorder of packets at application level.
Regarding Unity techniques, RPC technique is the one with the lowest delay. Also, it is the most stable one.

Regarding custom networking, it is possible to conclude that the custom networking function developed in this project are equivalent in terms of delay to the Unity functions; they can be used in a multiplayer environment without affecting the user experience. Even if the custom UDP technique cannot match the values of the RPC technique, the TCP is an interesting option that does not need to be discarded and deserves to be studied.

It is important to remark that this test was performed using a very simple application (just some cubes that are moved randomly) and in a laboratory environment. It would be interesting to perform more tests in different environments (Internet) with more complex games.

Fig. 5.2 Delays of networking techniques

5.3. Conclusions
CHAPTER 6. GANTT CHART

In this chapter, a Gantt chart of the performed tasks in this project is shown (see figure 6.1a and 6.1b). The Gantt chart identifies the most important stages of this project and shows its start and finish dates of each one of them.
CHAPTER 7. CONCLUSIONS

The main objective of this thesis is to perform an evaluation of the multiplayer networking features that Unity provides in its standard distribution, which are addressed to the creation of multiplayer games.

However, the study of Unity networking reveals some important limitations like the mandatory use of UDP, or not being able to use certain type of data as parameters in RPCs or State Synchronization. In order to try to solve some of these problems, a set of custom classes have been programmed.

Several applications, each one of them use a different networking technique (RPC, State Synchronization Delta Compressed, State Synchronization Unreliable, Hybrid RPC-State Synchronization, UDP, TCP, TCP without Nagle algorithm), had been developed to compare its delays.

The delay times of the use of these custom classes and the ones provided by the default distribution of Unity have been evaluated in terms of latency. The results are similar, thus the user experience will not be affected by using one or another. It is important to highlight that the custom networking has some extra features that try to overcome previous commented Unity networking limitations, like the possibility of using TCP instead of UDP and the possibility to send any type of data as an argument of a remote method.

Custom networking pretend to be an extra option for developers whose needs are not solved by the networking functionalities of Unity. Also, it is a tool for developers who want to understand how networking works.

7.1. Future Work

This project offers continuity in a sense that could be upgraded by adding some new networking functionalities. In addition, it could be evaluated using different metrics and environments. Some examples of future work are listed next:

- Evaluate the delay of networking techniques in a real environment between different machines located in different worldwide points and connected by means of Internet. In this environment, delay introduced by the network must be taken into account.

- Evaluate the networking techniques in a complex commercial game.

- An interesting functionality that could be added to custom networking could be to develop some simple client prediction modules using a machine learning algorithm as the proposed by the WEKA engine for C# [16] and add it to the custom networking functionalities programmed in this project. For
example, the prediction algorithm may be as simple as read the last five user actions and try to predict the next action before the user made it. Then, evaluate the system and check if the delays achieved are lower than the ones obtained without the use of the prediction module. In principle, if the prediction algorithm has sufficient data and it is well designed, then the delays should be lower, because the client will perform an action without waiting for the server response. In addition, this prediction module would have to incorporate some systems to rectify erroneous predictions.

- Add secure communications by means of using encryption and evaluate if the time added by the encryption process affects to user experience.
CHAPTER 8. BIBLIOGRAPHY

[2] Balance económico de la industria del videojuego 2012 - aDeSe