INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO
MESTRADO EM ENGENHARIA INFORMÁTICA




Framework 3D OpenWorld Maker

CARLOS ALBERTO SILVA CARVALHO PACHECO
Outubro de 2016

POLITÉCNICO
DO PORTO

Framework 3D OpenWorld Maker

Carlos Alberto Silva Carvalho Pacheco

Dissertação para obtenção do Grau de Mestre em Engenharia Informática, Área de Especialização em Sistemas Computacionais

Orientador: Prof. Doutor António Vieira de Castro

Porto, outubro 2016

Aos meus pais e irmãos por tudo o que me proporcionaram ao longo da vida

Resumo

Com a evolução das tecnologias de informação e das tendências no processo de desenvolvimento de *software*, torna-se necessário procurar novas soluções para aumentar a produtividade e diminuir custos na produção de vídeos jogos, mesmo utilizando *games engines* com uma experiência vasta sobre o mercado de vídeos jogos. Sendo assim, tornou-se necessário o desenvolvimento de uma *framework* escalável, onde seja possível a reutilização das suas funcionalidades de um jogo para outro jogo, assim como um servidor de autenticação/*master server*.

Esta *framework* para a criação de jogos *online* utiliza uma *framework Dapper* responsável por mapear as classes de negócio com as tabelas da base de dados, ou seja, um micro *Object Relationship Mapper* (ORM) que é usado no servidor de *login* para guardar os dados dos utilizadores. A *framework Unreal engine* é usada para a criação de vídeos jogos com gráficos realistas e com funcionalidades únicas para a rápida produção dos mesmos. Além disso, a seleção do *game engine*, passou por uma extensiva análise de *game engines* com capacidade de criar jogos *online* (como por exemplo, *Unity* 3D, *CryEngine*, *Unreal Engine* e Amazon Lumberyard).

Esta dissertação tem como propósito elaborar uma *framework* denominada 3D OpenWorld Maker para a criação de vídeo jogos *online* de uma forma rápida, simples e com baixos custos de produção, devido a ser possível reutilizar algumas funcionalidades entre jogos do mesmo género.

Palavras-Chave: Dapper, Sql Server, Unreal Engine, RUDP, Game Engine, Network.

Abstract

Considering the evolution of information technology and the trends in the process of software development, it has become necessary to look for new solutions to increase productivity and lower the costs of videogame production even in those game engines broadly used in the videogame scene. That said, it has become necessary to develop a scalable framework, capable of comprising an authentication server/master server and use it to reutilize a game's functionalities in another game.

This framework- created for the making of online games- uses a Dapper framework responsible for mapping the business classes with the database tables. In other words, it uses a micro Object Relationship Mapper (ORM) that is used in the login server to save the users' data. The framework is created with the intent of creating videogames with realistic graphics and having unique functionalities for the quick production of those games. Part of the process of selecting a game engine was an extensive analysis of the different game engines with the functionality to create online games (e.g.: Unity 3D, CryEngine, Unreal Engine and Amazon Lumberyard).

The objective of this dissertation is to formulate a framework named 3D OpenWorld Maker designed for the fast, low budget and simple creation of online videogames through its ability to reutilize some functionalities between games of the same genre.

Palavras-Chave: Dapper, Sql Server, Unreal Engine, RUDP, Game Engine, Network.

Agradecimentos

Gostaria de agradecer, em primeiro lugar, ao meu orientador do Instituto Superior de Engenharia do Porto (ISEP), Prof. Doutor António Vieira de Castro por aceitar ajudar-me nesta última etapa do curso, mas também por todo o tempo disponibilizado na realização da minha dissertação.

Um especial obrigado ao ISEP, em concreto ao departamento de informática por todos os momentos e por me ajudar a crescer a nível profissional, foi sem duvidas uma boa experiência e tem orgulho, ao longo destes setes anos, em fazer parte da família do ISEP.

Por fim, agradeço aos meus irmãos por todo o apoio que deram ao longo do curso e aos meus pais por todos os sacríficos que fizeram.

A todas as pessoas que sempre me apoiaram e confiaram em mim, deixo aqui o meu obrigado.

Índice

Lista de Figuras

Lista de Tabelas

Acrónimos e Símbolos

Lista de Acrónimos

GILT-ISEP	*Games, Interaction and Learning Technologies*
Open World	*Open world* ou Mundo aberto, onde um jogador pode movimentar-se livremente no mundo virtual.
MMO	*Massive Multiplayer Online*
Game Engine	*Game engine* ou motor de jogo para criação de jogos 2d e 3d.
Open Source	Em português Código Aberto, refere-se ao *software* que respeita as quatro normas definidas pela *Free Software Foundation*, sendo este de livre uso.
SQL	*Structured Query Language* – Linguagem de Consulta Estruturada
TCP	*Transmission Control Protocol* – protocolo de transporte de pacotes na rede
UDP	*User Datagram Protocol* – protocolo de transporte de pacotes na rede
AWS	*Amazon Web Services* – Serviços de computação em nuvem
ORM	*Object Relationship Mapper*

1 Introdução

Este capítulo inicia-se com a contextualização desta dissertação. De seguida, é abordado o problema proposto e referida a análise de valor que a solução proposta tem para os seus clientes-alvo. Além disso, será feita uma breve referência à abordagem preconizada, assim como aos resultados atingidos. Por último, é apresentada a estrutura que este documento segue.

1.1 Contextualização

Este documento descreve o processo de desenvolvimento da tese/dissertação (TMDEI) do ramo de mestrado de Sistemas Computacionais, do último ano do Mestrado de Engenharia Informática (MEI) do Instituto Superior de Engenharia do Porto (ISEP) do ano letivo 2015/2016.

Este projeto foi realizado na organização GILT-ISEP[1], e tem como objetivo implementar uma *framework* denominada "*3D OpenWorld Maker*" para possibilitar a criação de jogos *Massive Multiplayer Online* (MMO) de uma forma rápida e com baixo custo de produção.

1.2 Problema

O desenvolvimento de jogos *multiplayer*, em Portugal, é bastante baixo ou quase inexistente, não havendo qualquer motivação ou incentivo para que os jovens programadores comecem a ganhar gosto pelo desenvolvimento de competências próprias para criação de jogos. Este é um dos motivos pelo qual Portugal se encontra muito atrasado em comparação com o resto do mundo, fazendo com que a indústria dos jogos seja ainda inexplorada. Existem algumas ferramentas livres no mercado como o *Unreal Engine* 4, *CryEngine* e *Unity 3d* para projetos não comerciais, mas verifica-se alguma falta de conhecimentos nesta matéria. Outras ferramentas livres que não são complexas são muito limitadas podendo criar pouca

[1] Site GILT-ISEP: http://gilt.isep.ipp.pt/

motivação pelos jovens programadores. As ferramentas como o *Unreal Engine* 4, *CryEngine* e *Unity 3d* são mais complexas e difíceis de aprender, mas não existe limitações e os utilizadores podem levar a sua imaginação ao limite. Estas exigem conhecimentos de programação avançada e possuem uma curva de aprendizagem complexa. A solução passa por implementar uma *framework* dentro do *game engine* que seja mais simples, básico e com uma curva de aprendizagem rápida de utilizar. O principal objetivo é contribuir para o crescimento da produção deste tipo de jogos.

1.3 Análise de valor

Esta *framework* será integrada no game engine - *Unreal Engine* 4, que atualmente tem vindo a crescer em todos os níveis: desde de utilizadores pessoais a grandes empresas (exemplo o remake do *final fantasy* 7) [1], começando assim a ganhar mercado sobre o *Unity* 3D, por causa de ser gratuito. Sendo assim os clientes-alvo vão desde jovens programadores a pequenas e até grandes empresas. Esta *framework* deverá possibilitar a criação de jogos *Massive Multiplayer Online* (MMO) de uma forma básica, rápida, mais simples, com baixo custo de produção (aproveitar os recursos da *framework*), e com uma curva de aprendizagem rápida de utilizar. Pode assim contribuir para o crescimento da produção deste tipo de jogos.

Os clientes desta *framework* vão poder criar jogos *Online,* que são atualmente das maiores apostas a nível de jogos. Estes tendo vindo a crescer de uma forma esmagadora, e a taxa de sucesso é muito alta. Sobre os gastos com *Unreal engine,* este apenas pede 5% dos lucros por trimestre por produto quando este é lançado. Sendo assim, não existe custo com o *Unreal engine* na fase de desenvolvimento [2].

1.4 Abordagem preconizada

Pretende-se desenvolver uma *framework* denominada "*3D OpenWorld Maker*" para produzir jogos *Multiplayer*/*Open World/Sandbox*, que será dividida em vários componentes:

- Funcionalidades Básicas (Menus, ações, etc),
- *Network* (*Raknet*) ou a *Network* do próprio *engine*.

Com estes componentes pretende-se que seja possível criar vários tipos de jogos *online* e até jogos sérios, podendo no futuro vir a utilizar *Oculus Rift* conforme o tipo de jogo (opcional). Esta *framework* será implementada dentro de um *Game Engine* - *Unreal Engine* 4, ou seja, a *framework* deverá possibilitar a criação de jogos *Massive Multiplayer Online* (*MMO*) de uma forma rápida e com baixo custo de produção já que pode-se aproveitar até 80% dos recursos da *framework* para produzir outro jogo. Cada jogo será resultado de uma customização da *framework* (recursos base) adicionando as funcionalidades específicas do jogo a produzir.

Este *game engine* exige conhecimentos de programação avançada e com uma curva de aprendizagem complexa. Com isto será realizado uma análise ao estado da arte dos *games engines*, *Networks* e funcionalidades (Menus, Ações, entre outras). Pretende-se ainda analisar a construção e implementação da *framework* dentro do *game engine*, por exemplo, se o *game engine* permite a criação de componentes/menus customizado dentro da sua interface. Com esta análise, será possível escolher o *game engine* que vem ao encontro das principais necessidades para resolver o problema. Por fim, com o desenvolvimento da *framework* será necessário testá-la com um conjunto de jovens, para avaliar se a solução atingiu os teus objetivos inicialmente definidos.

1.5 Resultados atingidos

O principal objetivo desta dissertação foi concluído com sucesso, ou seja, o desenvolvimento da *framework* denominada "*3D OpenWorld Maker*" para o desenvolvimento de jogos *Multiplayer*/*Open World/Sandbox* foi implementado com sucesso e todos os objetivos inicialmente definidos foram atingidos.

1.6 Estrutura do documento

Este documento está organizado em sete capítulos do seguinte modo:

No capítulo 1 – Introdução, será apresentado a contextualização do projeto, o resumo do problema e da análise de valor. Contém também a abordagem preconizada.

No capítulo 2 – Contexto, serão abordados com pormenor o contexto e problema do projeto, análise de valor, como também o estado da arte dos *game engines* que possam ser utilizadas para auxiliar o desenvolvimento *framework*, entre outros.

No capítulo 3 – Avaliação de soluções, serão analisadas as possíveis soluções para o problema em análise.

No capítulo 4 – Design da solução, serão analisadas e descritas as funcionalidades funcionais e não funcionais do projeto, assim como a sua modelação e *design* da solução.

No capítulo 5 – Construção da solução, será descrito o desenvolvimento do projeto. Sendo discutido e apresentado como será executada a implementação da *framework* dentro do *game engine*, sendo também indicados os componentes necessários, para tornar a aplicação mais prática e fácil de utilizar.

No capítulo 6 – Avaliação da solução, serão documentados os testes, validações, grandezas, hipóteses e metodologia de avaliação realizadas ao longo do projeto, de modo a garantir que foram atingidos os objetos para o problema.

Finalmente o capítulo 7 – Conclusão, serão apresentadas as conclusões finais e dificuldades sentidas ao longo do projeto desenvolvido. Serão também abordadas melhorias futuras e a apreciação do projeto desenvolvido.

2 Contexto

2.1 Contexto e Problema

O grande crescimento das tecnologias ao longo dos anos tem feito aumentar o número de pessoas com dispositivos móveis e computadores. Atualmente existem dois telemóveis por habitante no mundo.

O mundo dos jogos existe há muitos anos e movimenta milhões por ano, o que implica muito investimento envolvido neste setor digital. Pode-se mesmo dizer que a produção de um jogo é comparada com a produção de um filme. Por exemplo, *Uncharted*, depois do sucesso deste título que rendeu milhões, a empresa *Naugty Dog* lançou um modo *multiplayer* que atualmente tem 9 mil milhões de utilizadores. Este sucesso deve-se à fama, e porque jogo *online* é *Play for Free* [3].

A taxa de sucesso dos jogos *online* é muito maior que um jogo *offline,* porque os jogos *offline*, após chegarem ao fim da sua história, deixam de ser úteis. Por outro lado, os jogos *online* nunca acabam e o entretenimento com outras pessoas faz com que exista uma maior motivação e interesse em jogar. O mercado dos jogos tem vindo a crescer cada vez mais e temos por exemplo os *streamings* de jogos que rendem milhões a alguns *youtubers* e *twich.tv*. Mas na área *online*, onde existem muitos jogos é preciso ter atenção entre um jogo pago ou um *play for free* com micro transações dentro do jogo. Cada vez mais o sucesso de um jogo *online* passa por não ser um jogo pago, mas sim por ser gratuito com micro transações, isto leva a existir muito jogadores que podem ser possíveis clientes.

Atualmente, o desenvolvimento de jogos *multiplayer* em Portugal é considerado baixo ou quase inexistente não havendo qualquer motivação ou incentivo para que os jovens programadores comecem a ganhar gosto pelo desenvolvimento de competências próprias para criação de jogos. Este é um dos motivos pelo qual Portugal se encontra muito atrasado em comparação com o resto do mundo, fazendo com que a indústria dos jogos seja uma indústria inexplorada. Existem algumas ferramentas livres no mercado como o *Unreal Engine* 4 para projetos não comerciais, mas verifica-se alguma falta de conhecimentos nesta matéria. Outras ferramentas livres que não se consubstanciam como complexas são muito limitadas podendo criar pouca motivação para os jovens programadores, as ferramentas como o *Unreal Engine* 4 são mais complexas e difíceis de aprender, mas não existem limitações e os utilizadores podem levar a sua imaginação ao seu limite. Estas ferramentas exigem conhecimentos de programação avançada e com uma curva de aprendizagem complexa. O que propomos é uma *framework* dentro do *Unreal Engine* 4 que seja mais simples, básico e com uma curva de aprendizagem rápida de utilizar. É assim objetivo contribuir para o crescimento da produção deste tipo de jogos

Pretende-se implementar uma *framework* denominada "*3D OpenWorld Maker*" para a produzir jogos *Multiplayer/Open World/Sandbox*, que será dividida em vários componentes: Funcionalidades Básicas (Menus, ações, etc), *Network* (*Raknet*) ou a *Network* do próprio *engine*. Com estes componentes pretendemos que seja possível criar vários tipos de jogos *online* e até jogos sérios, podendo no futuro vir a utilizar *Oculus Rift* conforme o tipo de jogo (opcional). Esta *framework* será implementada dentro de

um *game engine* (*Unreal Engine* 4), ou seja, a *framework* deverá possibilitar a criação de jogos *Massive Multiplayer Online (MMO)* de uma forma rápida e com baixo custo de produção já que se pode aproveitar até 80% dos recursos da *framework* para produzir outro jogo. Cada jogo será resultado de uma customização da *framework* (recursos base), adicionando as funcionalidades específicas do jogo a produzir.

Em relação às restrições para o desenvolvimento da *framework*, destaca-se apenas que a sua implementação será feita na linguagem de programação C++, sendo esse a língua usada pelo Unreal *Engine* 4.

2.2 Análise de valor

Em todos produtos existe sempre uma proposta de valor para o seu sucesso. Este valor é um conceito difícil de conseguir, entender e modelar, por isso a criação de valor é uma troca entre benefícios e sacrifícios percebidos pelos entre clientes durante a oferta do fornecedor. Sendo assim, a proposta de valor deve ser bem definida e sempre em atenção na relação entre cliente e fornecedor, só assim será possível conhecer o mercado e criar valor para os clientes [4].

Esta *framework* será integrada no *game engine* - *Unreal Engine* 4, tendo como clientes-alvo jovens programadores e pequenas e grandes empresas. A *framework* "*3D OpenWorld Maker*" pretende aumentar a produtividade e simplicidade e diminuir a curva de aprendizagem, uma vez que que o principal objetivo desta *framework* é motivar os jovens programadores a entrarem nesta área dos vídeos jogos. Sendo assim, os clientes podem não só usar esta *framework* para criarem os seus jogos, mas também para aplicar novas técnicas no desenvolvimento dos jogos, devido a ser possível reutilizar muitos componentes da *framework*. Como já foi falado anteriormente, cada novo jogo é uma customização tendo por base a *framework* (baixos custos de produção), com isto existe mais tempo para as equipas de desenvolvimento focarem-se no conteúdo do jogo. No caso dos jovens, isto pode ser o ponto de partida para ganharem motivação para seguirem programação e aprenderem mais sobre ela de uma forma mais atrativa para eles, porque os videojogos estão sempre presentes em qualquer infância e adolescência.

Esta *framework* deverá possibilitar a criação de jogos *Massive Multiplayer Online* (*MMO*) de uma forma básica, rápida, mais simples, com baixo custo de produção (aproveitar os recursos da *framework*) e com uma curva de aprendizagem rápida de utilizar. Assim contribuir para o crescimento da produção deste tipo de jogos.
Os clientes desta *framework* vão, assim, poder criar jogos Online, que atualmente são das maiores apostas ao nível de jogos, tendo vindo a crescer de uma forma esmagadora e a taxa de sucesso é muito alta. Sobre os gastos com *Unreal Engine* 4, este apenas pede 5% dos lucros por trimestre por produto quando este é lançado, sendo assim não existe custo com o *unreal engine* na fase de desenvolvimento.

Numa perspetiva de longitudinal de valor está dividida em quatro fases:

- Antes de efetuar a compra,
- No momento da compra ou experiência,

- Depois da compra,
- Depois o uso e experiência.

Como foi referido no ponto do problema, 1.2 , o desenvolvimento de jogos *multiplayer* em Portugal é bastante baixo ou quase inexistente não havendo qualquer motivação ou incentivo para que os jovens programadores comecem a ganhar gosto pelo desenvolvimento de competências próprias para criação de jogos e este é um dos motivos pelo qual Portugal se encontra muito atrasado comparado com o resto do mundo, fazendo da indústria dos jogos uma indústria inexplorada. Sendo assim, os jovens programadores não têm motivação ou incentivo para começarem a explorar o desenvolvimento de jogos.

No momento da compra ou experiência, o cliente tem acesso a uma *framework* dentro no *Unreal Engine* 4, que vai permitir criar jogos *Massive Multiplayer Online* (*MMO*) de uma forma básica, rápida, mais simples, com baixo custo de produção (aproveitar os recursos da *framework*) e com uma curva de aprendizagem rápida de utilizar. O cliente vai ter a acesso a uma *framework* que disponibiliza um conjunto de funcionalidades que auxiliam na produção de jogos *MMO*.

Depois da compra, o cliente terá acesso a um menu onde pode escolher as funcionalidades que deseja usar/reutilizar no seu jogo. Depois da tal seleção, a *framework* irá criar a base destas funcionalidades de forma automática.

Depois de usar e experiência da *framework*, o cliente pode verificar o ganho que teve comparando o tempo e custos do produto usando a *framework*.

Em relação aos possíveis cenários de negócio, o modelo de negócio usa uma negociação integrativa, ou seja, para haver um ganho entre ambas as partes é preciso garantir o máximo possível de benefícios para ambas. Neste tipo de negociações ganha-ganha é necessário não só ter em atenção a identificação dos pontos indiscutíveis, mas também definir as expectativas e objetivos na negociação. Contudo, é necessário prever contraofertas e dominar todos os assuntos envolvidos, só assim será é possível conhecer melhor o que a outra parte pretende. Por fim, nestas negociações é preciso definir o valor máximo e mínimo e estar preparado para explicar esses valores a receber ou dar [5].

Foi utilizado o modelo de negócio de *canvas* para descrever de forma clara a ideia de negócio, esse modelo pode ser consultado no Anexo 9.1. O segmento de clientes vai desde de jovens programadores a pequenas e grandes empresas de videojogos. Sobre a proposição de valor do projeto, pretende-se que este projeto se torne a produção de jogos *Massive Multiplayer Online* (MMO) mais rápida, com baixo custo de produção, mais simples, básico e com uma curva de aprendizagem rápida de utilizar. Deve assim contribuir para o crescimento da produção deste tipo de videojogos. A nível dos canais, serão utilizados os meios onde existem mais pessoas da área dos videojogos, ou seja, *youtube* e *twitch.tv*. Também vão ser usadas as redes sociais e a loja de venda de produtos do *Unreal Engine*. Em termos do relacionamento com cliente, este será feito através das redes sociais e pelo fórum e comunidade *Unreal Engine*. Sobre as fontes de receita, esta é feita através da compra do produto dentro da loja do *Unreal Engine*, sendo no futuro o preço mais caro conforme se vai aumentado as funcionalidades do produto. A nível dos parceiros-chave são *game engines* e empresas de videojogos, sendo a atividade-chave a produção de

videojogos. Por fim a estrutura de custos, temos a equipa de desenvolvimento e o material, tanto *hardware* e software, necessários para o desenvolvimento do produto [5].

A criação de valor pode ser analisada através do Modelo Conceptual para Decomposição do Valor para o Cliente, é preciso identificar quatro pontos.

- Os exógenos
- Os endógenos.
- Os benefícios e sacrifícios de ambos as partes (fornecedor e cliente)
- Os benefícios percebidos e sacrifícios de ambos as partes (fornecedor e cliente)

Com esta análise é possível saber a criação de valor, sendo assim, estes quatros pontos são usados para avaliar de que forma o produto pode evoluir para dar valor ao cliente e à empresa que fornece o produto [6].

2.3 Estado da arte

Neste subcapítulo, é realizada o estado da arte das tecnologias do projeto que vão ser descritas em seguida.

2.3.1 Game Engine

O nome *game engine* nasceu no meio da década de 1990, especialmente em conexão com jogos 3D. O primeiro motor 3D a ser usado para a criação de jogos de computador foi o *Freescape Engine* [7], desenvolvido pela Incentive Software em 1986, usado para criar jogos do estilo FPS em 1987. Este *game engine* tem atualmente um preço que pode variar do gratuito, para jogos amadores, como pode chegar a custar entre 19€/mês ou 57€/mês no caso dos *game engines* AAA, ou até mais caros. O processo de desenvolvimento de jogos é frequentemente agilizado, quando um mesmo *game engine* é usado para criar diferentes jogos.

Game Engine é um *software* desenhado para a criação e desenvolvimento de jogos de vídeo, é para jogos de consola, jogos para dispositivos móveis e computadores pessoais.

Estes *Game Engine* tipicamente têm um motor gráfico para fazer o render em 2D ou 3D e um motor de física que simula a física nos objetos ou deteta colisões. Estes *Game Engine* por norma possuem uma interface gráfica intuitiva e também fornecem animações, sons, inteligência artificial, *networking*, gestão de memória, gestão de arquivos, gestão de modelos 3d e suporte de uma linguagem de programação.

2.3.2 As Game Networking

A descrição básica de uma *Game Networking* é a conexão de dois ou vários computadores onde um deles é o servidor e os restantes os clientes, ou mesmo onde o servidor é em simultâneo um cliente também, por exemplo caso das LAN.

Com isto, existem muitas maneiras de criar um jogo em rede, e como tal é necessário conhecer as diferenças de cada uma. Nesse sentido, apresenta-se de seguida cada uma delas.

2.3.2.1 Servidor dedicado (não corre num cliente)

Este tipo de servidor é usado normalmente em jogos *multiplayer,* onde os jogadores não têm a opção de criar um servidor próprio, mas apenas de se conectarem a um servidor já existente. Em alguns casos existe a opção para criar uma instância do jogo no servidor dedicado, sendo assim não existe um cliente a correr na mesma máquina e com isto não existem problemas de performance, devido a correr ao mesmo tempo um cliente e um servidor do jogo. Com isto, aumenta a possibilidade de suportar mais clientes ligados ao servidor e também pode reduzir o *lag* entre servidor cliente.

2.3.2.2 Servidor não dedicado (corre num cliente)

Permite que os jogadores criem o servidor do jogo o eles mesmos também são um cliente, ou seja, o jogador cria o servidor para os outros se ligarem a ele, e ele também é um cliente na mesma máquina. Um dos benefícios é o jogador poder criar o seu servidor sem necessitar de um outro computador ou de configurações completas na rede. Em contrapartida, este tipo de servidores tem que receber os pedidos de rede de todos os clientes, pode também afetar o jogo para todos os clientes caso o cliente que é também servidor tenha algum problema com performance ou caso os recursos do computador estejam a ser usados para servidor e cliente.

Depois das informações sobre o que é um servidor dedicado e um servidor não dedicado, segue-se os tipos de servidores de jogo que podem existir.

2.3.2.3 Não Autoritária (*Peer to Peer*)

Neste tipo de servidor de jogo, todo o esforço/processamento é realizado nos clientes e os pacotes são apenas enviados de um cliente para os outros clientes. Isto proporciona uma melhor performance das ações/eventos ao serem executadas localmente no instante que o jogador faz algo e também é atualizado para outros clientes de forma assíncrona.

No entanto, é possível haver *hacking/cheating* e também pode acontecer resultados inexplicáveis devido ao *lag* ou a outras questões. Exemplo: um jogador explode, mas no cliente remota nada causa a explosão devido ao *lag* ou a perda do pacote na rede. Ao mesmo tempo, num cliente diferente, uma bomba

explode e mata o jogador e, num terceiro cliente, o jogador estava longe da bomba, mas morre pouco tempo depois sem razão aparente, quando o estado da sessão é replicada.

Atualmente, com dispositivos controlados tais como *smartphones*, *iPads* ou consolas, não é necessária a preocupação com os problemas de *hacking* como num computador [8].

2.3.2.4 Autoritária

O servidor de jogo autoritário é justamente o oposto do servidor de jogo de retransmissão. Neste modelo o cliente envia todos os comandos para o servidor de jogo e este decide se o comando é aceitável, executando assim a ação apropriada (por exemplo, andar para a frente) e em seguida atualiza todos os clientes com a nova posição e orientação de todos os objetos. Isso é comum nos MMORPG, onde a ação é mais baseada em turnos, mas menos ideal para jogos em tempo real como um FPS que tem ações rápidas e espera por respostas quase em tempo real. Todos os clientes estão sempre dependentes das ações/eventos uns dos outros, mas com o *lag* do servidor de jogo, pode haver lentidão e o fluxo do jogo não será tão natural para o jogador. Alguns jogos usam uma abordagem autoritária, mas acrescentam uma previsão local para que seja precisa e natural. Neste caso, um jogador envia um pedido para se mover, ou seja, o pedido é enviado para o servidor, mas ao mesmo tempo assume-se que o servidor irá aprovar esse pedido então inicia-se o movimento local. Desta forma, o jogador tem um fluxo natural e sem atrasos ou *lag*, mas o servidor de jogo ainda vai avaliar o pedido e, caso não aprove o pedido, o jogador volta para onde estava anteriormente [8].

2.3.2.5 Transmission Control Protocol (*TCP*) *User Datagram Protocol* (*UDP*)

TCP e UDP são protocolos de redes. O mais popular e usado é o protocolo *TCP*, que devido às suas funcionalidades para detetar erros garante a entrega dos pacotes com um mecanismo que deteta quando um pacote é perdido na rede e reenvia o mesmo outra vez. Este protocolo também divide os dados em pequenos pacotes e envia-os, garantido sempre a ordem de chegada dos pacotes ao destinatário. Estas funcionalidades não existem no protocolo UDP, neste protocolo quando os pacotes são enviados não existem mecanismos para detetar quando um pacote é perdido na rede. Também não garante a ordem de envio e podem haver casos de pacotes duplicados. Com isto, podemos afirmar que o *TCP* é o melhor protocolo para jogos, mas fá-lo-íamos erradamente. Uma vez que, no mundo dos jogos, cada segundo conta para o bom sucesso do jogo, o protoloco com as funcionalidades primeiramente indicadas faz com que exista uma maior demora no envio dos pacotes pela rede, fazendo assim com que a *UDP*, que não possui estes mecanismos, seja mais rápida. Por este motivo é que em *stream* de vídeo é usado o protocolo *UDP*. No entanto, o protocolo *UDP* não serve para alguns tipos de jogos e a solução passa por um protocolo chamado *RUDP*, que é nada mais que o protocolo *UDP* com as funcionalidades do controlo dos pacotes, sendo assim fica mais rápido do que o *TCP*, porque são apenas implementadas as funcionalidades necessárias e não de uma forma geral [9].

Atualmente existem inúmeros *game engines* no mercado que, como o projeto, têm como um dos objetivos não limitar as funcionalidades que os utilizadores vão ter acesso, então procedeu-se a um estudo dos *games engines* do tipo *AAA* para o desenvolvimento do projeto. Apresenta-se, de seguida, uma análise mais detalhada de cada um dos *game engines*.

2.3.3 Game engines

Nesta secção será feita extensiva análise mais detalhada de cada um dos *game engines* disponíveis no mercado para a criação de vídeos jogos *online*.

2.3.3.1 Unity 3d

A plataforma *Unity (*Figura 1*)* apresenta um desenvolvimento bastante flexível, eficiente e oferece vários serviços integrados sendo muito usada para experiências interativas 3D e 2D em multiplataforma e para criação de diferentes jogos. É um ecossistema único para montar um negócio de conteúdo avançado e integração com os leais clientes e jogadores.

Figura 1 – Unity logotipo

O *Unity* possui um desenvolvimento extremamente refinado para uma multiplataforma e proporciona confiança em relação às plataformas modernas e às futuristas onde otimiza o desempenho com ferramentas avançadas entre plataformas, totalmente personalizável e com um desenvolvimento rápida através da sua interface intuitiva, também é possível adicionar ferramentas próprias ao *Unity* para tirar o máximo de produtividade.

Esta plataforma dispõe de uma gama de milhares de recursos variadas tais como: extensões do Editor, *plug-ins*, ambientes, modelos, entre outras. Estes recursos podem ser acedidos facilmente de forma gratuita ou paga através do editor do *Unity* ou do navegador *web*. Além de todas as ferramentas de desenvolvimento que este *game engine* oferece, também integra uma série de serviços que ajudam o utilizador a gerir o público do seu jogo, como por exemplo, as suas ferramentas de monetização de conteúdo [10].

2.3.3.2 Unreal engine

O *Unreal Engine* 4 (Figura 2) é um conjunto de ferramentas de desenvolvimento feito por criadores de jogos para criadores de jogos que pode fazer desde jogos 2D para telemóveis até jogos de realidade virtual com impressionantes efeitos visuais *high-end*. Este *game engine* permite criar jogos para um diverso leque de plataformas, nomeadamente para consolas *next-gen*, computador, *iOS*, *Android* e realidade virtual, mostrando-se assim uma ferramenta poderosa em termos de variedade de plataformas.

No *Unreal Engine* não há limite para o que se pretende conseguir com a tecnologia nomeadamente ao fazer-se um aplicativo de um simples puzzle ou mesmo de um jogo de ação em mundo aberto sendo bastante usado por estudantes e pequenas ou grandes equipas.

Figura 2 – Unreal Engine logotipo

Este *game engine* foi desenvolvido sem nenhum descuido para a área de desenvolvimento de jogos *mobile*, por isso é possível criar desde jogos 2D simples a jogos com gráficos impressionantes e depois exportar os mesmos para dispositivos *iOS* ou *Android* sem qualquer dificuldade. Também é uma plataforma em constante desenvolvimento, uma vez que é usada por estudantes, criadores de jogos *indie* e equipas grandes.

O *Unreal Engine* até recentemente custava ao utilizador, mas atualmente *Epic games* alterou e passou a ser quando um jogo criado vendesse mais de 3000€ a cada três meses, o criador do jogo teria que pagar 5% das *royalties*. De momento apenas se pagam as *royalties*, a mensalidade deixou de existir, tornando o *Unreal Engine* um *software* mais acessível [2].

2.3.3.3 Cryengine

O *CryEngine* (Figura 3) é um motor de jogo desenvolvido pelos criadores alemães *Crytek* e é considerado a primeira solução de desenvolvimento *all-in-one* com gráficos multi-premiados de uma computação verdadeiramente oscilante, *state-of-the-art lighting*, com uma física realista incrível, *scripting* visual bastante intuitivo bem como o áudio e muito mais. Tem sido usado em todos os jogos da *Crytek* e a sua versão inicial foi usada em *Far Cry*. Foi atualizado para que possa suportar novas consolas e *hardware* para os novos jogos da empresa e ainda é usado, sobre regime de licenciamento da *Crytek*, por muitos jogos de terceiros nomeadamente *Ghost Warrior 2* e *Snow*.

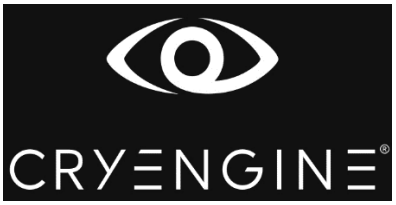


Figura 3 – CryEngine logotipo

A *Ubisoft* tem também uma versão própria do *CryEngine* original usado no *Far Cry*, apesar de ter sido fortemente modificada, designada por *Dunia Engine* que foi usada para as posteriores iterações das séries *Far Cry* depois da *Ubisoft* se ter apoderado do desenvolvimento da série.

Em abril de 2015, o *CryEngine* for comprado pela *Amazon* e, consequentemente em fevereiro de 2016, foi lançado o *Lumberyard CryEngine* que é uma versão do *CryEngine* substancialmente modificada e mais extensa.

A *Crytek* lançou a versão 1.2 que foi desenvolvida graças a placas de vídeo com suporte a sombreamento de pixeis e vértices e que foi usada de modo a desenvolver melhores gráficos. Seguidamente foi lançada a *CryEngine* 1.3, com um suporte adicional a iluminação HDR. Posteriormente foi associada à *NCSoft* para a criação do seu jogo MMORPG *Aion: The Tower of Eternity.*

CryEngine é usado por *AAA* e criadores independentes para construir os jogos mais ambiciosos e desejados pelos jogadores e permitir criar o motor de jogo mais poderoso da indústria a nível das diversas ferramentas que criam experiências de jogo de classe mundial. Até à versão 3.5.8 o programa era completamente gratuito, *royalty-free* e sem mensalidade. Agora, embora ainda não seja necessário pagar *royalties*, é necessário pagar 9.90€/mês para usar esta plataforma, podendo fazer esta subscrição no *Steam* [11].

2.3.3.4 Amazon Lumberyardis

O *Amazon Lumberyard (*Figura 4*)* é um mecanismo gratuito de jogos 3D que possui uma plataforma cruzada que permite criar jogos de alta qualidade, conectá-los a recursos de computação e armazenamento da nuvem, *Amazon Web Services (AWS),* e envolver os fãs no *Twitch*.

O *Lumberyard* permite a criação de projetos de jogos investindo o tempo na criação e desenvolvimento de uma ótima jogabilidade e assim promover comunidades de fãs, evitando certas execuções de tarefas mais pesadas e o gerenciamento da infraestrutura dos servidores. Através do seu editor completo, o desempenho de código nativo, o visual fantástico e várias outras centenas de recursos do *Amazon Lumberyard* permitiram aos desenvolvedores profissionais elaborarem mais facilmente os seus jogos de nível global através das ferramentas e da tecnologia avançada que este mecanismo oferece.

Figura 4 – Lumberyard logotipo

Este *game engine* fornece uma ferramenta de script visual que permite, a pessoas com pouca experiência em *back-ends*, acrescentar rapidamente aos jogos recursos ligados à nuvem como por exemplo o *feed* de notícias da comunidade, usando apenas scripts visuais do tipo *drag-and-drop.*

O *Amazon Lumberyard* contém o código fonte completo, o que permite, a utilizadores mais avançados, a sua completa personalização com base nas suas necessidades. Tem também a vantagem de ser a custo zero, apenas se pagam os serviços da *AWS* que se utilizarem.

O uso do *Amazon GameLift*, um novo serviço da *AWS* para implantação, operação e escalabilidade de jogos multijogador com base em sessão, permite aumentar e reduzir rapidamente a escala de servidores de jogos de alto desempenho de acordo com a demanda dos jogadores, sem necessidade de atividades de engenharia e sem custos antecipados [12].

3 Avaliação de soluções

Com base no estado da arte sobre os *game engines AAA* descritas na Tabela 1 para o desenvolvimento de vídeos jogos e, sendo a escolha do *game engine* baseado em critérios rigorosos, tendo em conta as suas limitações e complexidades, procedeu-se à seleção da *game engine* [13].

Tabela 1 – Comparação dos *game engines*

Funcionalidades	**Lumberyardis [12]**	**Unity 3d [10]**	**CryEngine [11]**	**Unreal Engine [2]**
Loja (Assets)	Sem informação	Grande	Pequena	Normal (crescer)
Preço	Grátis com os serviços da *Amazon Web Service* (AWS)	A partir de 57€ mês (sem *iOS* e *Android*)	9,90€ Mês	Grátis, 5% dos lucros por produto e por trimestre depois dos primeiros 3000€
Gráficos	5/5	4/5	5/5	5/5
Interface amigável	Difícil	Fácil	Difícil	Normal
Aprendizagem	Difícil (Ainda não existe)	Fácil (Grande comunidade)	Difícil (Pouca comunidade)	Normal (comunidade em crescimento)
Plataformas	Consolas, computadores, mobile	Web, consolas, computadores, mobile	Consolas, computadores, mobile	Web, consolas, computadores, mobile
Network	Nova *Gamelift*, pouca informação sobre ela	Não dedicada, é preciso usar uma externa *RakNet*	Dedicada	Dedicada
Linguagens programação	C++	C#, Unity script, Boo	C#, C++	C++, Blueprints linguagem de programação Visual

Sendo assim, foi selecionada *Unreal engine* para *game engine* tendo por base a existência de uma network completa que nos permite assim criar qualquer tipo de jogos sem recorrer a um *middleware* para a network, por exemplo, o *RakNet*, que passou a ser gratuito, esta *network* é utilizada pelo *Unity 3d*, sendo muito possível que no futuro a *network* do *Unity 3d* possa vir a ser dedicada. O *Unreal engine* está em crescimento desde que *Epic games* colocou o seu *game engine* de forma gratuita, sendo assim uma excelente aposta, não só porque é um *game engine* completo, sem ser necessário andar a comprar componentes extras (como é o caso do *Unity* 3d), mas também pela sua qualidade gráfica. Sendo assim,

a *framework* vai trazer aos utilizadores uma nova maneira de produzir vídeos jogos com uma curva de aprendizagem curta e a possibilidade de gestão de cada funcionalidade, fora outros pontos que podem trazer vantagens no desenvolvimento de vídeo jogos.

4 Design da solução

Neste capítulo, serão analisadas e descritas as funcionalidades funcionais e não funcionais do projeto, assim como a sua modelação e *design* da solução

4.1 Análise de Requisitos

Depois de realizada o estado da arte, a secção de análise de requisitos é uma das mais importantes fases no desenvolvimento de *software*, onde é realizado o levantamento dos requisitos que o software deverá possuir.

Nesta fase, é realizada análise sobre as funcionalidades que o projeto necessita para poder concretizar os objetivos definidos. Pelo facto de este projeto ser utilizado por vários utilizadores em simultâneo, é necessário ter cuidado com os padrões de qualidade do *software*.

4.1.1 Requisitos Funcionais

Os requisitos funcionais são requisitos que expressam funções ou serviços que um *software* deve ou pode ser capaz de executar ou fornecer. As funções ou serviços são, em geral, processos que utilizam entradas para produzir saídas.

4.1.1.1 Adicionar *Login*

Permite ao utilizador indicar se deseja adicionar *login* ao seu jogo para os jogadores efetuarem o *login* para entrar no jogo.

4.1.1.2 Adicionar Registo

Permite ao utilizador indicar se pretende adicionar a opção de registar uma nova conta para os jogadores efetuarem o *login* no jogo.

4.1.1.3 Adicionar *Chat*

Permite ao utilizador selecionar se deseja ter um *chat* público dentro do jogo para comunicar com os outros jogadores.

4.1.1.4 Mostrar nome do jogador

Permite que seja possível mostrar o nome do jogador aos outros jogadores no jogo.

4.1.1.5 Definir *spawn* do jogador

Definir os *spawn* do jogador pelo mundo virtual do jogo.

4.1.2 Requisitos Não Funcionais

Os requisitos não funcionais são requisitos que declaram uma restrição, ou atributos de qualidade para um *software* e/ou para o processo de desenvolvimento deste projeto.

4.1.2.1 Desempenho

Como o objetivo é a comunicação entre clientes e servidor, os utilizadores necessitam que os seus pedidos efetuados ao servidor sejam processados de uma forma rápida (quase em tempo real), evitando atrasos da resposta e sincronização entre o cliente e o servidor.

4.1.2.2 Confiabilidade

Devido a este serviço permitir a troca de informação entre o clientes e servidor em tempo real, é necessário garantir que o serviço esteja operacional, tendo assim a menor probabilidade de indisponibilidade ou ocorrência de falhas.

4.1.2.3 Segurança

Cada utilizador, ao utilizar o jogo, só deverá ter acesso a este depois de efetuar o registo e criar as sua credenciais de *login*.

Como se trata de um jogo *online* os pedidos são efetuados sobre um protoloco, sendo assim garantido a segurança dos dados entre cliente e servidor. As contas dos utilizadores podem ser desativadas em vários casos, por exemplo, quando um jogador utiliza programas ilegais (*cheats*) para ganhar vantagens sobre os outros jogadores, pode ser banido permanentemente ou temporariamente. A *framework* deve estar protegida contra *SQL Injection* pela importância de proteger os dados existentes nas bases de dados.

4.1.2.4 Usabilidade

Sendo o projeto uma *framework*, existe a necessidade de ter em conta que a interface deverá ser simples, permitido assim auxiliar de uma forma rápida a gestão das funcionalidades aos seus utilizadores.

4.1.2.5 Escalabilidade

Sendo o projeto uma *framework* para o desenvolvimento de vídeos jogos, é preciso ter em atenção a sua escalabilidade, visto que qualquer jogo *online* pode crescer rapidamente e o sistema tem que estar preparado para tal, existindo assim a divisão dos utilizadores por vários servidores de jogo.

4.1.2.6 Logs

Como na maioria do *software* e as boas práticas de engenharia ditam, deverá existir um controlo sobre erros e ações. Para tal existe um sistema de *logs* onde serão registadas as tentativas de login, os erros internos da *framework* e também todas as ações/eventos dos utilizadores.

4.2 Análise e Modelação

Neste subcapítulo, é realizada a documentação relativa aos casos de uso, acompanhados por tabelas com a sua descrição. A descrição do modelo de dados e a análise da solução para a arquitetura.

4.2.1 Casos de Uso

Depois da análise de requisitos tornou-se necessário a especificação das funcionalidades do projeto, para tal, recorreu-se à elaboração dos casos de uso. No sentido de tornar a nossa *framework* eficaz, funcional e eficiente consideramos abordar boas práticas não só no que respeita ao modelo de negócio, mas ao nível técnico-científico. Consideramos necessário implementar as seguintes funcionalidades.

De seguida serão descritos os casos de usos na Figura 5.

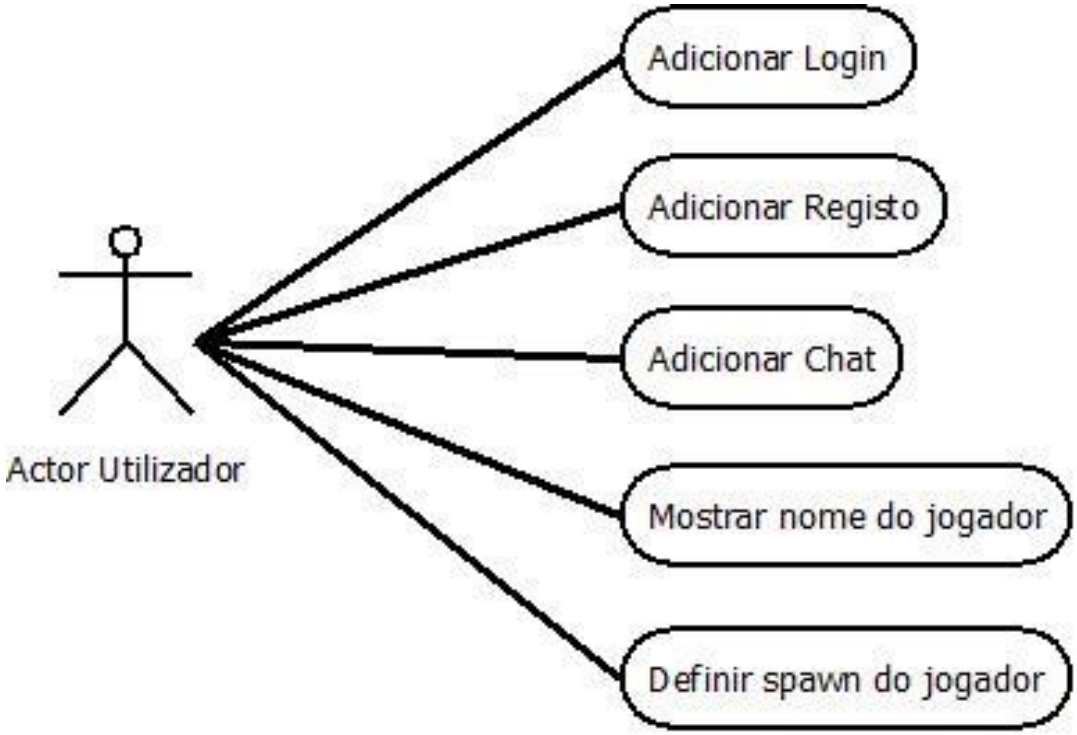


Figura 5 – Diagrama de caso de usos

A Figura 5 representa o diagrama de caso de usos, onde o actor Utilizador tem os seguintes casos de uso: Efetuar *login*, efetuar registo, criar *Chat*, mostra nome do jogador e definir *spawn* do jogador. Estes casos de uso são descritos na Tabela 2.

Tabela 2 – Breve descrição dos casos de uso

ID Caso de Uso	Nome	Descrição
1	Adicionar *login*	Permite ao utilizar indicar se deseja adicionar *login* ao seu jogo para os jogadores efetuarem o *login* para entrar no jogo.
2	Adicionar registo	Permite ao utilizador indicar se pretende adicionar a opção de registar uma nova conta para os jogadores efetuarem o login no jogo.
3	Adicionar *chat*	Permite ao utilizador selecionar se deseja ter um *chat* público dentro do jogo para comunicar com os outros jogadores.
4	Mostrar nome do jogador	Permite que seja possível mostrar o nome do jogador aos outros jogadores no jogo.
5	Definir *spawn* do jogador	Definir os *spawn* do jogador pelo mundo virtual do jogo.

4.2.2 Arquitetura do Sistema

Neste subcapítulo, será apresentada a arquitetura da solução, sendo uma das partes mais importantes no desenvolvimento de *software*, é preciso ter atenção como o sistema, neste caso o jogo, vai ficar a curto e a longo prazo. Na Figura 6 pode ser consultada a arquitetura do sistema.

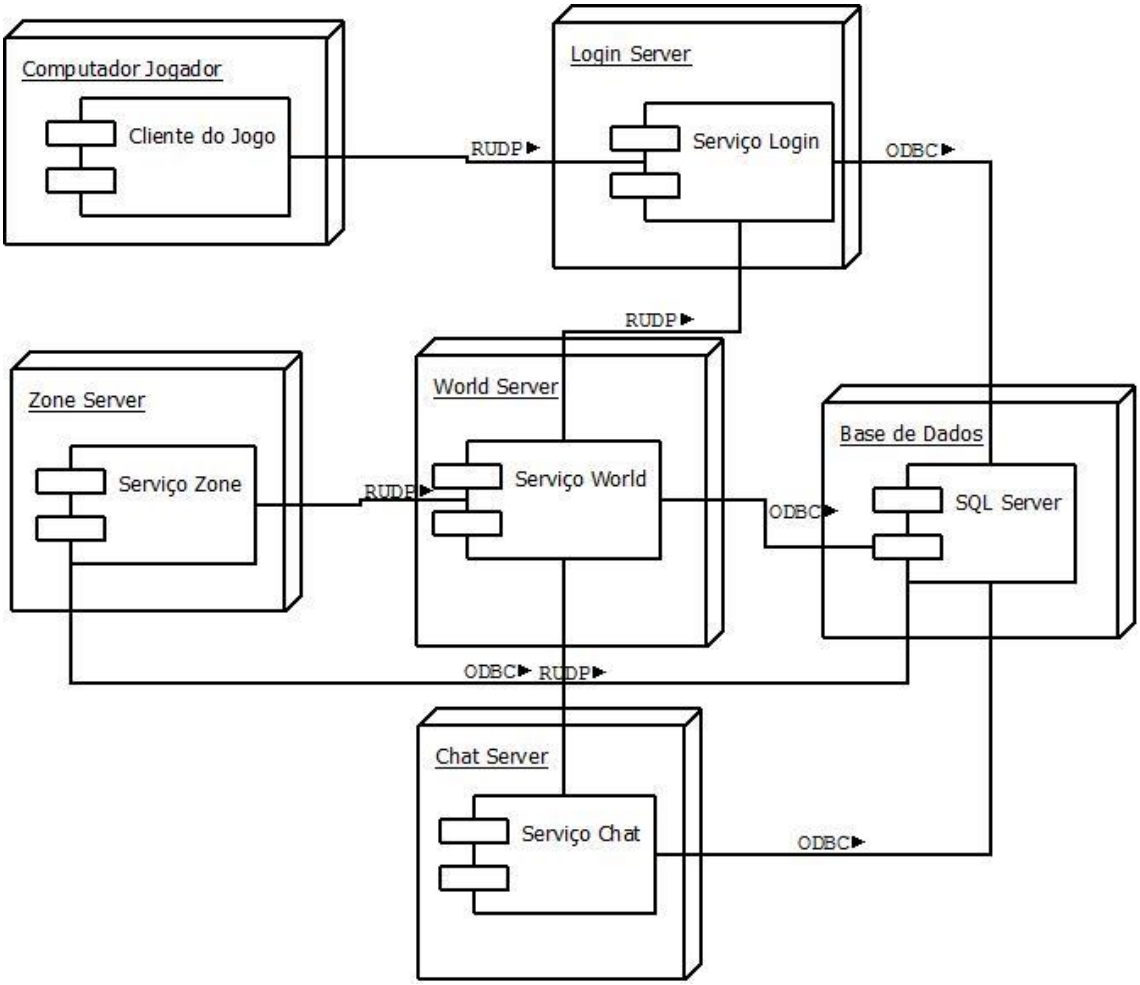


Figura 6 – Arquitetura do sistema

Na arquitetura do sistema apresentada na Figura 6, o utilizador, através de um computador, acede via *RUDP* ao *Login Server*, caso o utilizador tenha acesso ao jogo, o *Login Server* vai comunicar com o *World Server* que existe um utilizador a querer estabelecer uma conexão com ele. Depois de validar o utilizador, o *Login Server* pede ao *World Server* para adicionar o utilizador à sua lista de utilizadores. Neste caso, *World Server* vai carregar os dados da base de dados relativos ao utilizador em questão. Depois disto, o *World Server* vai reencaminhar este utilizador para o *Zone Server* (existe um ou vários servidores do tipo *Zone Server*), onde o utilizador se encontra (ultimo registo guardado na base de dados depois do *logoff* do utilizador). O *World Server* que recebe um pedido do tipo *Chat* envia estes pedidos para o *Chat Server* e quando recebe algum pedido do jogo envia para o *Zone Server*. Por fim, o *World Server*, *Zone Server* e *Chat Server* efetuam comunicação com o servidor que tem a base de dados, ou seja, onde é feita persistência de dados na base de dados usando *SQL Server*.

5 Construção da solução

Neste capítulo serão apresentadas as principais partes do desenvolvimento da *framework* para a criação de vídeos jogos online. Será apresentada como foi executada a implementação do modelo de dados para o servidor de login que utilizava o micro ORM *Dapper* que foi criado pelo *stackoverflow* e que é utilizado por eles, e ainda será feita analise de apresentação dos diagramas de classes.

Também serão apresentadas as funcionalidades desenvolvidas na *framework*, com a sua descrição passo a passo e como são usadas e colocadas em prática num caso real, neste caso foi escolhido o projeto de exemplo que o *unreal engine* fornece, o *Shooter Game*.

Por fim, a parte da configuração do servidor do *Sql Server* que é usado pelo o serviço de *login*. Também é mostrada a configuração do *Unreal Engine* deste, da cópia efetuada do repositório da *Epic Games*, que é vital para a criação do servidor dedicado do jogo até a instalação do projeto de exemplo, *Shooter Game*.

5.1 Modelos de dados

Sobre o modelo de dados implementado, foi elaborado para permitir armazenar a informação relevante do serviço de *login*, como os dados de *login* de cada utilizador e as suas configurações principais para um jogo online.

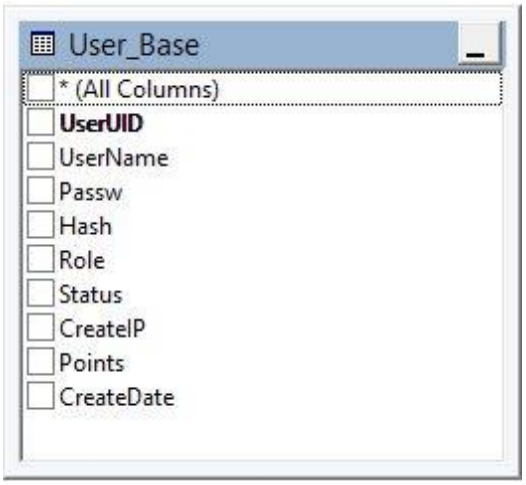


Figura 7 – Modelo de dados do serviço *login*

Na Figura 7 podem ser visualizados os campos necessários para um utilizador poder ser registado no sistema e posteriormente ser feito o *login* com as credenciais, depois de serem validadas todas as condições e caso o utilizador não tenha sido banido ou bloqueado temporariamente.

Na Tabela 3,Tabela 1 a descrição geral das tabelas do modelo de dados.

Tabela 3 – Documentação da tabela User_Base

Nome da Tabela	Descrição	
User_Base	Tabela dos utilizadores	
Nome da Coluna	**Tipo de dados**	**Descrição**
UserUId	PK, Serial, Not Null	Identificação do utilizador.
UserName	Varchar (18), Not Null	*Username* do utilizador.
Passw	Varchar (18), Not Null	Senha do utilizador, encriptada com o *sha512*.
Hash	Varchar (32), Not Null	*Hash* único do utilizador, para ser utilizado ao gerir as senhas, encriptado com o *sha512*.
Role	Char (1), Not Null	Premissoes do utilizador, ou seja, N Normal ou A Admistrador
Status	Tinyint, Not Null	Estado do utilizador, indica se o utilizador está bloqueado, banido ou normal.
CreateIP	Varchar (15), Null	Número de tentativas feitas ao efetuar *login*.
Points	Int, Not Null	Pontos no jogo para usar na compra de itens (pontos pagos com dinheiro real).
CreateDate	Smalldatatime, Not Null	Data de criação do utilizador

5.2 Diagrama de classes

O diagrama de classes da *framework* de comunicação foi dividido em duas partes: numa é apresentado o diagrama de classes relativo ao *Login Server* e, na outra, é apresentado o diagrama de classes relativo ao *ShootGame* que dá origem a comunicação entre o serviço de autenticação e o começo do jogo.

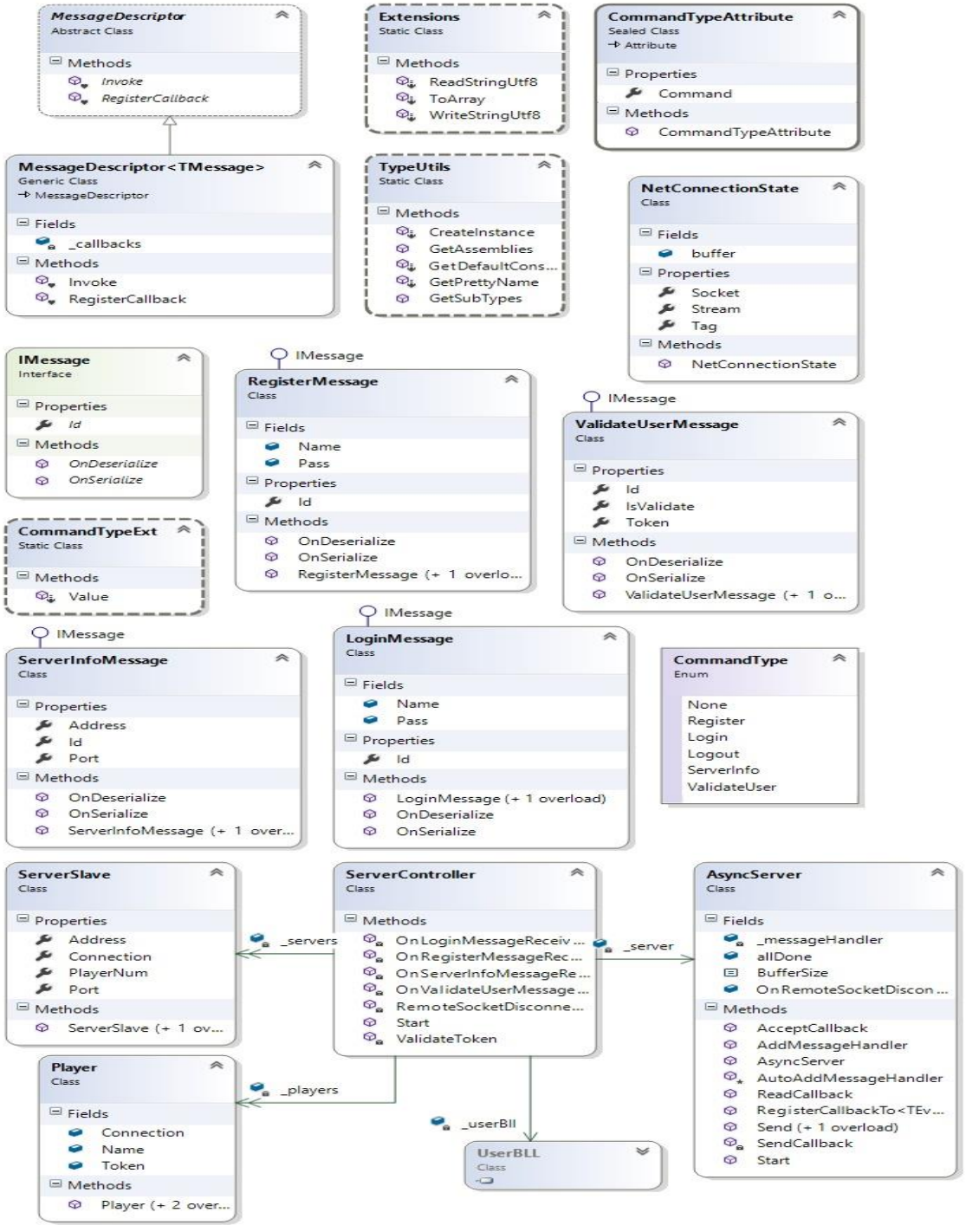


Figura 8 – Diagrama de classes *Login Server*

Na Figura 8, é representado o diagrama de classes do *Login Server*, onde a classe `ServerController` tem a responsabilidade de adicionar as funções de *callback* para cada determinado tipo de mensagem a ser trocada entre o servidor de *login*, clientes(jogo) e servidor dedicado (servidor do jogo), é onde são registados os dados de cada servidor dedicado que, posteriormente, vão ser enviados aos clientes depois de ser validado o *login* dos mesmos. Também tem a responsabilidade de arrancar o serviço de *login*, ou seja, criar a classe responsável pela comunicação via *socket*.

A classe `AsyncServer` é a classe mais importante do *Login Server*, é aqui que são feitas as trocas de mensagens por *socket*, como também é guardado um dicionário de *callback* por cada tipo de mensagem: cada mensagem que se pretende enviar, de um ponto para outro, necessita ter algumas precondições para depois adicionar um *callback* a esse tipo de mensagem, por exemplo, para criar uma nova mensagem é preciso criar uma nova classe que implemente a interface `IMessage` e depois que tenha o atributo do tipo `CommandTypeAttribute`. Que recebe como parâmetro um `Enum` chamado de `CommandType` e cada um dos *enums* tem um valor inteiro que é o seu indentificador da mensagem.

```
[CommandType(CommandType.Register)]
    public class RegisterMessage : IMessage
```

Neste exemplo de código pode-se ver como o objeto do tipo `RegisterMessage` é feito para poder ser enviado e recebido de uma forma abstrata da camada de comunicação via *socket*, depois de ter o objeto mensagem criado é só necessário adicionar o *callback* para ele, como é mostrado no código seguinte:

```
_server.RegisterCallbackTo<RegisterMessage>(OnRegisterMessageReceived);
```

Relembrando que o objeto `_server` é do tipo `AsyncServer.`Quando é recebido um novo conjunto de *bytes* é verificado qual é o tipo da mensagem atrás do primeiro *byte* recebido e, assim, por reflexão, é criado o objeto do `Type` da mensagem recebida.

A classe `TypeUtils` possui métodos do tipo *extesion* para o tipo `Type` para facilitar e centralizar métodos recorrentes, neste caso para buscar o nome, criar objetos de um certo tipo, como buscar todos os *types* que fazem parte de um determinado objeto.

Ao nível da base de dados, existe uma classe ORM (*dapper*) que tem como função fazer o mapeamento entre os dados das classes e da base de dados, sendo esta exclusiva apenas da tecnologia *sql server*.

Sendo assim independente de qualquer tipo de base de dados. Na camada *DAL*/*BLL* existem *gateway* genéricas, uma *factory* responsável por criar os objetos de *gateway*, para permitir uma maior flexibilidade para alterar o tipo de base de dados sem impacto para as classes já existentes, possuindo também mecanismos de segurança (*SQL Injection)*.

As classes `RegisterMessage`, `ValidateUserMessage`, `ServerInfoMessage`, `LoginMessage` são classes de comunicação, ou seja, implementam a classe `IMessage` e são usadas para enviar dados via *socket* entre o servidor e os clientes. Sobre a classe `Extesions`, esta tem alguns métodos para converter uma `string` em bytes e converter bytes em uma `string`, isto com encode utf-8, devido aos métodos básicos não usarem 4 bytes para tamanho da `string`.

A Figura 8 pode ser consultada com mais detalhe no Anexo 9.2.

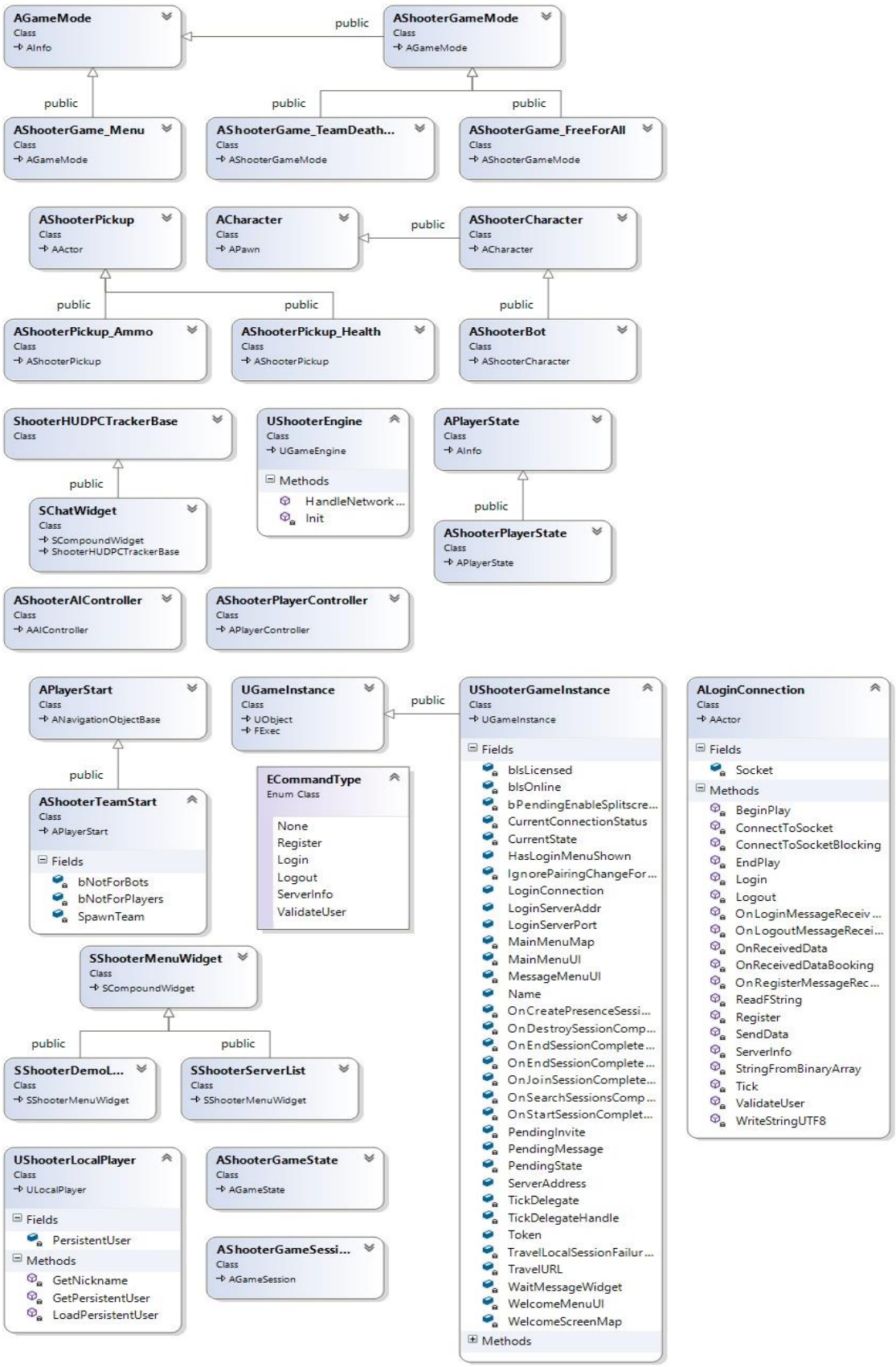


Figura 9 – Diagrama de classes ShooterGame

Na Figura 9 – Diagrama de classes ShooterGame, é representado o diagrama de classes do *ShooterGame*, onde apenas tem as principais classes para o funcionamento do exemplo utilizando a *framework*, onde a classe `ALoginConnection` é responsável pela comunicação entre o jogo e o *Login server*. É aqui que são recebidas e envidas as mensagens, como o seu tratamento para serem chamados os métodos para cada tipo de mensagem, neste caso esses métodos são chamados antes dos eventos em *blueprints* do *unreal engine*.

A classe `AGameMode` define o jogo a ser jogado e aplica as regras do jogo. Algumas das funcionalidades padrão no `AGameMode` são quaisquer novas funções ou variáveis que definem as regras do jogo e devem ser adicionados em uma subclasse da classe `AGameMode`. Qualquer coisa, desde o inventário de itens com que um jogador começa ou quantas vidas estão disponíveis para um determinado tempo e pontuação necessária para terminar o jogo pertencem ao *GameMode*. A subclasse da classe `AGameMode` pode ser criada para cada tipo de jogo para esse jogo ter o comportamento desejado deve ser incluído esse *GameMode*. Um jogo pode ter qualquer número de tipos de jogos, e assim subclasses da classe `AGameMode`; no entanto, apenas um tipo de jogo pode estar a ser utilizado em um determinado momento. Um Ator *GameMode* é instanciado cada vez que um nível é inicializado para o jogo através da função `UGameEngine::LoadMap()`. O tipo de jogo define este Ator que será usado em toda a duração do nível, ou seja, até o nível terminar. Este tipo de classes só é instanciado no servidor e nunca vai existir no cliente. Também é nesta classe que podemos rejeitar uma conexão ao jogo através dos métodos de `login`, `prelogin`, `postlogin`.

A classe `AGameState` é replicada e é válida em servidores e clientes, é uma extensão do `AGameMode`, ou seja, uma parte para colocar algumas regras não globais, é também responsável por monitorar o estado do jogo no cliente. Conceitualmente, devemos pensar na *gamestate* como o estado do jogo. Ele pode manter o controlo de propriedades como scores, a lista de jogadores conectados, número de caixas em um jogo, onde as peças estão em um jogo de xadrez, quais as missões que são precisas para poder concluir um jogo de mundo aberto, etc. Em geral, o *gamestate* deve controlar e ter propriedades que mudam durante o jogo. O *gamestate* existe no servidor e todos os clientes e pode ser replicado para manter todas as máquinas atualizadas.

A classe `APlayerState` é criada para cada jogador no servidor (ou em um jogo *standalone*). *PlayerStates* são replicadas para todos os clientes, esta classe contem informação relevante sobre o jogador num jogo online (*network game*), como, por exemplo: nome do jogador, pontuação, etc. Estes dados que são replicados por toda a rede do jogo não podem ser alterados pelos clientes de forma direta (é uma cópia local apenas), ou seja, como queremos manter que o jogo seja livre de *cheaters* e *hackers*, esta informação só existe no servidor e é apenas replicada para os clientes, caso seja necessário alterar esta informação é necessário enviar um *RPC* para o servidor a solicitar as alterações e assim o servidor volta a replicar esses dados para os outros clientes.

A classe `APlayerStart` indica um local onde um jogador pode fazer *spawn* (aparecer no jogo) quando o jogo começa.

A classe `UGameInstance` é a classe uma das classes mais importantes no *unreal engine*, `UGameInstance` é um objeto de gestão de alto nível para uma instância do jogo em execução. É instanciado na criação do jogo e não é destruído até instância jogo ser terminada. Executando como um jogo independente, haverá

sempre um destes objetos em execução. No caso de correr em PIE (*play-in-editor*) irá ser criado um objeto por cada instância PIE. Esta classe também é onde se persistem os estados entre níveis num jogo, enquanto as classes `AGameMode` ou `APlayerController` são sempre reiniciadas e todos os dados armazenados nelas são apagados e voltam a ser iniciados com os valores de padrão, no `UGameInstance` os dados nunca são apagados enquanto o jogo não terminar, isto é, enquanto o executável do jogo estiver a correr os dados são mantidos em memória. Todos os dados que sejam para ser mantidos entre níveis dentro do jogo e que são importantes guardar para serem usados noutros níveis devem ser guardados nesta classe.

A Figura 9 pode ser consultada com mais detalhe no Anexo 9.3.

5.3 Funcionalidades

Neste subcapítulo são apresentadas as funcionalidades desenvolvidas ao longo deste projeto, tais como: adicionar *login*, registar, *chat*, mostrar nome do jogador e *spawn* do jogador.

5.3.1.1 Adicionar *Login*

Ao efetuar login, os dados são enviados através de uma função em C++ que converte os dados em bytes e envia estes por *socket* ao *Login Server*. Esta função pode ser vista na Figura 10.

```
bool ALoginConnection::Login(const FString& Name, const FString& Password)
{
    return SendData((uint8)ECommandType::Login, Name, Password);
}
```

Figura 10 – Método para efetuar o *login*, C++

```cpp
bool ALoginConnection::SendData(uint8 Command, const FString& Name, const FString& Message)
{
    if (!Socket) return false;

    int32 sent = 0;

    FString myname = Name;
    FString mymessage = Message;

    FTCHARToUTF8 nameUTF8(*myname);
    int namelength = nameUTF8.Length();

    FTCHARToUTF8 messageUTF8(*mymessage);
    int messagelength = messageUTF8.Length();

    // 1 byte Comand, 4 bytes to int namelength and other 4 bytes to int messageLength plus the 2 strings
    int32 size = 1 + 4 + 4 + namelength + messagelength;
    uint8* withprefixes = (uint8*)malloc(size);

    FArrayWriter Writer;
    Writer << Command;
    //Writer << namelength;
    Writer << myname;
    //Writer << messagelength;
    Writer << mymessage;

    //add command:
    withprefixes[0] = Command;

    int index = 1;
    WriteStringUTF8(withprefixes, index, nameUTF8);
    WriteStringUTF8(withprefixes, index, messageUTF8);

    return Socket->Send(withprefixes, size, sent);
}
```

Figura 11 – Método SendData c++

Depois de ser chamada a função para efetuar o *login* é chamada uma função centralizada que tem o nome `SendData` como é mostrada na Figura 11, nesta função os dados são convertidos para um `uint8` que trabalha como um *array* de *bytes*, mas não é um *array* de *bytes*, o que acontece é que são reservados os *bytes* necessários nessa variável e depois são deslocados os *bytes* conforme o seu tamanho.

Na Figura 12 podemos ver ao pormenor como é feito o deslocamento dos *bytes* para uma *string* em c++. Neste caso sabemos que o tamanho da *string* vai ocupar quatro bytes e por isso temos que fazer o deslocamento de oito *bits* por cada *byte*, no caso do valor da *string* como já foi convertida em `uint8` é feito o deslocamento como se fosse um *array* normal.

```cpp
void ALoginConnection::WriteStringUTF8(uint8 *BinaryArray, int& index, FTCHARToUTF8& value)
{

    uint8* val = (uint8*)value.Get();
    int length = value.Length();

    //add name length:
    for (int i = 0; i < 4; i++)
    {
        BinaryArray[index] = length >> i * 8;
        index++;
    }

    // add name:
    for (int i = 0; i < length; i++)
    {
        BinaryArray[index] = val[i];
        index++;
    }
}
```

Figura 12 – Método WriteStringUTF8 c++

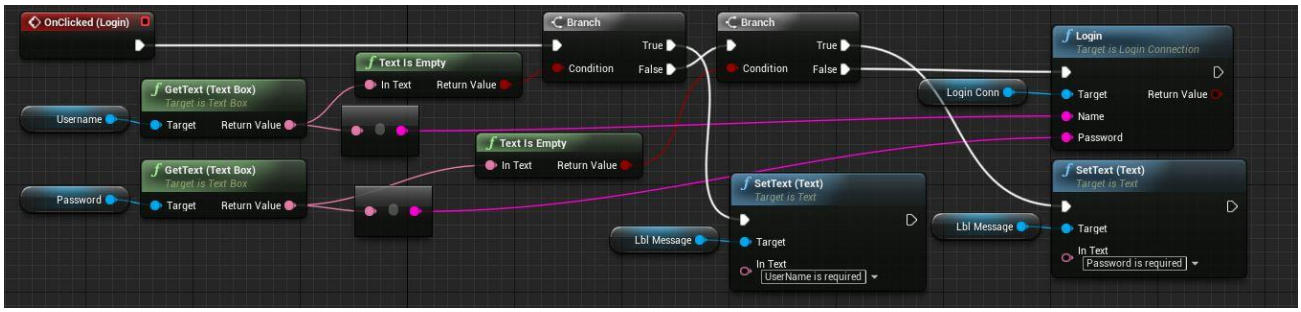


Figura 13 – Efetuar o Login Blueprint

O que está representado acima na Figura 13 é executado quando o utilizador clica no botão *login*, neste *blueprint* são validados os dados de entrada, ou seja, *username* e *password* e se tiver todo preenchido são enviados para a classe `LoginConnection` que vai executar o método `Login` no C++ como foi possível ver na Figura 10.

Este *blueprint* pode ser consultado com mais detalhe no 9.4.

```
1 reference
private void OnLoginMessageReceived(LoginMessage message, NetConnectionState netState)
{
    Socket socket = netState.Socket;

    Console.WriteLine("logged in: " + message.Name);

    UserBase user =_userBll.GetUserByUserName(message.Name);
    if(user == null)
    {
        BinaryWriter binWriter = new BinaryWriter(new MemoryStream());
        binWriter.Write((byte)CommandType.Login);
        binWriter.Write(false);//fail login
        binWriter.WriteStringUtf8(message.Name);
        binWriter.WriteStringUtf8(string.Empty);//token
        binWriter.WriteStringUtf8(string.Empty);

        _server.Send(netState, binWriter);
        return;
    }
```

Figura 14 – Método OnLoginMessageReceived parte 1

Na Figura 14 é apresentado o fluxo de código no *Login Server* quando é recebida uma mensagem do tipo `LoginMessage`, vinda do cliente de jogo. Neste método começa-se por validar e buscar os dados guardados na base de dados com os dados recebidos do utilizador, se estes estiverem inválidos é enviada de volta uma mensagem do tipo *login* para o cliente que se encontra à espera da resposta.

Figura 15 - Método OnLoginMessageReceived parte 2

```
    //TODO: dummy token http://stackoverflow.com/questions/14643735/how-to-generate-a-unique-token-which-expires-after-24-hours
    Player newPlayer = new Player(socket, message.Name, Convert.ToBase64String(Guid.NewGuid().ToByteArray())); // "s723dEqQ0UeLJiIJcGxErA"//

    if (!_players.Contains(newPlayer))
    {
        netState.Tag = newPlayer;

        lock (_players)
        {
            _players.Add(newPlayer);
        }

        //create the response message
        BinaryWriter binWriter = new BinaryWriter(new MemoryStream());
        binWriter.Write((byte)CommandType.Login);
        binWriter.Write(true);//success login
        binWriter.WriteStringUtf8(message.Name);
        binWriter.WriteStringUtf8(newPlayer.Token);//token

        //TODO: the unreal socket only read 1 server address, need to be change to read a list of address
        foreach (ServerSlave svrSlave in _servers)
        {
            binWriter.WriteStringUtf8(svrSlave.Address);
        }
        //binWriter.WriteStringUtf8("127.0.0.1");

        _server.Send(netState, binWriter);
        Console.WriteLine("added a player");
    }
} « void OnLoginMessageReceived(LoginMessage, NetConnectionState)
```

Quando os dados recebidos (Figura 15) estão de acordo com os que foram guardados na base de dados (feito com o registo de utilizador), então o *Login Server* cria um novo registo em memória para esse jogador e um *token*, enviando de volta ao utilizador o nome do mesmo e o *token* de login, também junto é enviada a lista de servidores dedicados. Sobre o funcionamento do registo dos servidores dedicados, será explicado depois da descrição do efetuar *login*.

```
switch (Command)
{
case ECommandType::Login:
    index++;
    Success = ReceivedData[index] == 1;
    ReceivedName = ReadFString(ReceivedData, index);
    ReceivedToken = ReadFString(ReceivedData, index);
    ServerAddress = ReadFString(ReceivedData, index);

    OnLoginMessageReceived(Success, *ReceivedName, *ReceivedToken, *ServerAddress);
    break;
```

Figura 16 – Método OnReceivedData

Depois de receber os dados é invocado o método `OnLoginMessageReceived` na classe *blueprint* que herda da classe `LoginConnection` em C++ como é mostrado na Figura 16.

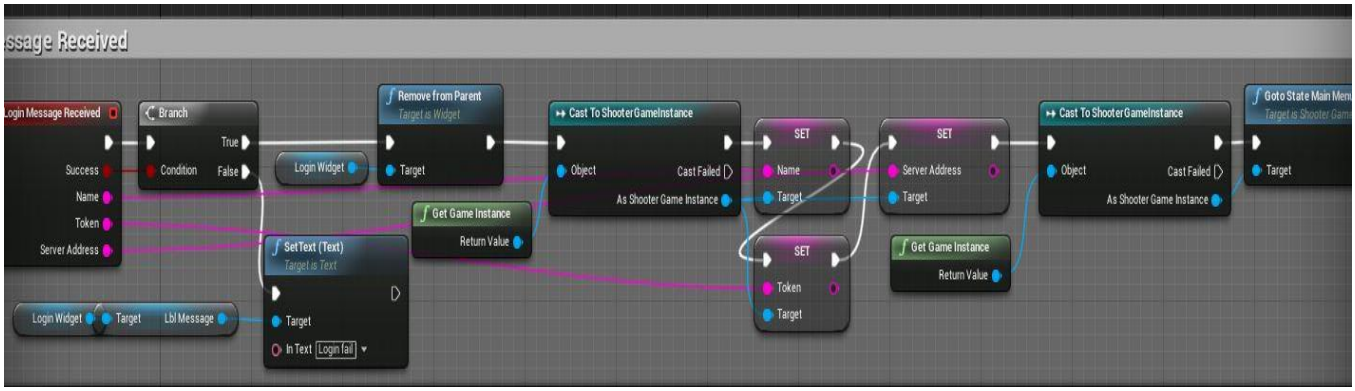


Figura 17 – Método OnLoginMessageReceived blueprint

No lado do cliente a função com a responsabilidade de processar os dados recebidos é a `OnLoginMessageReiceved` que foi feita em *blueprint*, é aqui que são guardados os dados relativos ao *token* e a lista de servidores dedicados, como se pode ver na Figura 17, que possuem instâncias do jogo com vários mapas a serem executados neles.

Este blueprint pode ser consultado com mais detalhe no 9.4.

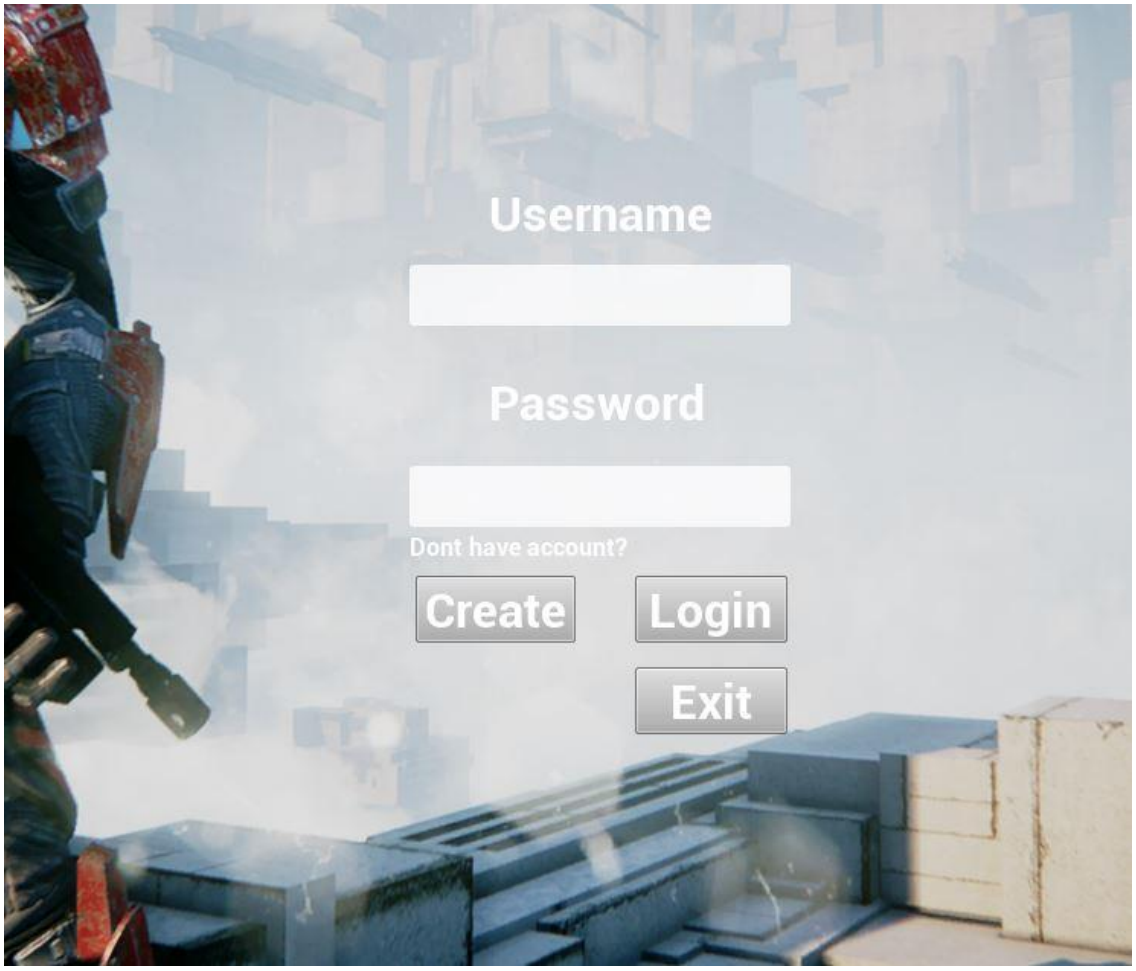


Figura 18 – Interface do login

Como resultado final, na Figura 18, é representada a *interface* de *login*, onde o utilizador introduz o nome do utilizador (*username*) e a sua senha (*password*), depois disto o utilizador clica no botão *"Login"* e é executado o fluxo que foi explicado anteriormente nesta funcionalidade. Também é aqui que o utilizador pode começar o processo para criar um novo registo no *Login Server* (botão "*Create"*), isto será explicado com mais detalhe na funcionalidade adicionar registo.

Um ponto importante neste processo de *login* é como os servidores dedicados comunicam com o *Login Server* e o que acontece depois de ser feito o login com sucesso.

Do lado de cada servidor dedicado é criado um objeto do tipo `ALoginConnection`, como é representado na Figura 19, na classe `UShooterGameInstance`, assim este objeto nunca é destruído.

```cpp
UShooterGameInstance::UShooterGameInstance(const FObjectInitializer& ObjectInitializer)
    : Super(ObjectInitializer)
    , bIsOnline(true) // Default to online
    , bIsLicensed(true) // Default to licensed (should have been checked by OS on boot)
{
    CurrentState = ShooterGameInstanceState::None;
    if (IsRunningDedicatedServer())
    {
        LoginConnection = ObjectInitializer.CreateDefaultSubobject<ALoginConnection>(this, TEXT("LoginConnection"));
    }
}
```

Figura 19 – Construtor da classe UShooterGameInstance

Quando o objeto `ALoginConnection` está criado e a classe `UShooterGameInstance` também, é chamada a função `Init`, na Figura 20, pode ser visualizado apenas no servidor dedicado que executa as funções para enviar os dados dele para o *Login Server*, neste caso é buscado o endereço de ip da máquina onde o processo do servidor dedicado está a executar, mas este método não é o melhor, futuramente deve-se colocar o endereço de IP público ou interno da rede no ficheiro de configurações do `GameInstance`, ou seja, no ficheiro DefaultGame.ini. Neste ficheiro já é guardado o endereço e a porta para o *Login Server*, que são as variáveis `LoginServerAddr` e `LoginServerPort`, depois de obter o endereço do servidor enviamos esses dados para o *Login Server*, utilizando a função `ServerInfo` (Figura 22).

```cpp
void UShooterGameInstance::Init()
{
    Super::Init();
    HasLoginMenuShown = false;

    //only on dedicated server
    if (IsRunningDedicatedServer())
    {
        UE_LOG(LogNet, Log, TEXT("create login ConnectToSocketBlocking"));
        FString HostName;
        /*if (!ISocketSubsystem::Get()->GetHostName(HostName))
        {*/
            // could not get hostname, use address
            bool bCanBindAll;
            TSharedPtr<class FInternetAddr> Addr = ISocketSubsystem::Get()->GetLocalHostAddr(*GLog, bCanBindAll);
            if (Addr->IsValid())
            {
                HostName = Addr->ToString(false);
            }
        //}

        LoginConnection->ConnectToSocketBlocking(LoginServerAddr, LoginServerPort);
        LoginConnection->ServerInfo(GetWorld()->GetAddressURL(), HostName);
        UE_LOG(LogNet, Log, TEXT("create login connection done"));
    }
```

Figura 20 – Método Init da classe UShooterGameInstance

```cpp
bool ALoginConnection::ConnectToSocketBlocking(const FString& Address, int32 port)
{
    Socket = ISocketSubsystem::Get(PLATFORM_SOCKETSUBSYSTEM)->CreateSocket(NAME_Stream, TEXT("default"), false);

    FIPv4Address ip;
    FIPv4Address::Parse(Address, ip);

    TSharedRef<FInternetAddr> addr = ISocketSubsystem::Get(PLATFORM_SOCKETSUBSYSTEM)->CreateInternetAddr();
    addr->SetIp(ip.Value);
    addr->SetPort(port);

    return Socket->Connect(*addr);
}
```

Figura 21 – Método ConnectToSocketBlocking

Como foi possível ver na Figura 20, para o envio da informação do servidor dedicado para o *Login Server* existem duas funções que são precisas para tal, primeiro é preciso criar o objeto `ALoginConnection` de uma forma diferente do normal, ou seja, sem o timer que ia ser chamado de 0.01 segundos numa *thread* separada da *thread* principal, como se trata do servidor dedicado é preciso ficar à espera da resposta do outro servidor para se poder continuar, mais à frente será possível entender a escolha desta solução, quando for preciso validar o *token* de login que é recebido do cliente.

Na Figura 21, é possível ver que o *socket* é criado apenas e é feita a sua conexão sem utilizar um timer para verificar de x em x tempo se existe *bytes* no *socket* para serem lidos.

```
bool ALoginConnection::ServerInfo(const FString& Address, const FString& Port)
{
    if (SendData((uint8)ECommandType::ServerInfo, Address, Port))
    {
        return true;
    }
    return false;
}
```

Figura 22 – Método ServerInfo

A segunda função é `ServerInfo`, como se pode ver na Figura 22, aqui são enviados os dados do servidor dedicado ao *Login Server*, neste caso o seu endereço. Como não só se quer enviar informação não é necessário ficar à espera de qualquer resposta do *Login Server*, sendo assim uma mensagem unidirecional.

```
private void OnServerInfoMessageReceived(ServerInfoMessage message, NetConnectionState netState)
{
    string address = string.Empty;
    string port = string.Empty;
    //process the received ips
    try
    {
        string[] spliaddress = message.Address?.Split(':');
        address = spliaddress[0];
        if(string.IsNullOrWhiteSpace(address))
        {
            //TODO: change this, see where get allways the ip and port from unreal engine
            //TODO: on the localhost the unreal return allways empty ip
            address = message.Port;//yes the name is stupid
        }

        port = spliaddress[1];

        ServerSlave srvSlave = new ServerSlave(netState.Socket, address, port);
        netState.Tag = srvSlave;

        lock (_servers)
        {
            _servers.Add(srvSlave);
        }
    }
    catch(Exception ex)
    {
        Console.WriteLine($"{ex.Message}{Environment.NewLine}{ex.StackTrace}");
    }
} « void OnServerInfoMessageReceived(ServerInfoMessage, NetConnectionState)
```

Figura 23 – Método OnServerInforMessageReceived do Login Server

Do lado do *Login Server* é recebida a mensagem do servidor dedicado e os dados são guardados em memória numa lista com todos os servidores dedicados. O *Login Server* também pode ser visto como um servidor mestre que tem vários servidores escravos, que neste caso são os servidores dedicados a correr um mapa do jogo, como podemos ver na Figura 23.

Figura 24 – Menu principal

Na Figura 24, é representado o menu principal do demo *ShooterGame*, este menu aprece depois de ser efetuado o *login* com sucesso, ou seja, o estado `ShooterGameInstanceState::LoginMenu` passa para o estado `ShooterGameInstanceState::MainMenu`, neste momento a conexão ao *Login Server* é fechada e o próximo passo é o utilizador criar o seu próprio mapa, e assim, torna-se um servidor dedicado/cliente ou pode pesquisar por um servidor dedicado usando o botão *Join* do menu principal.

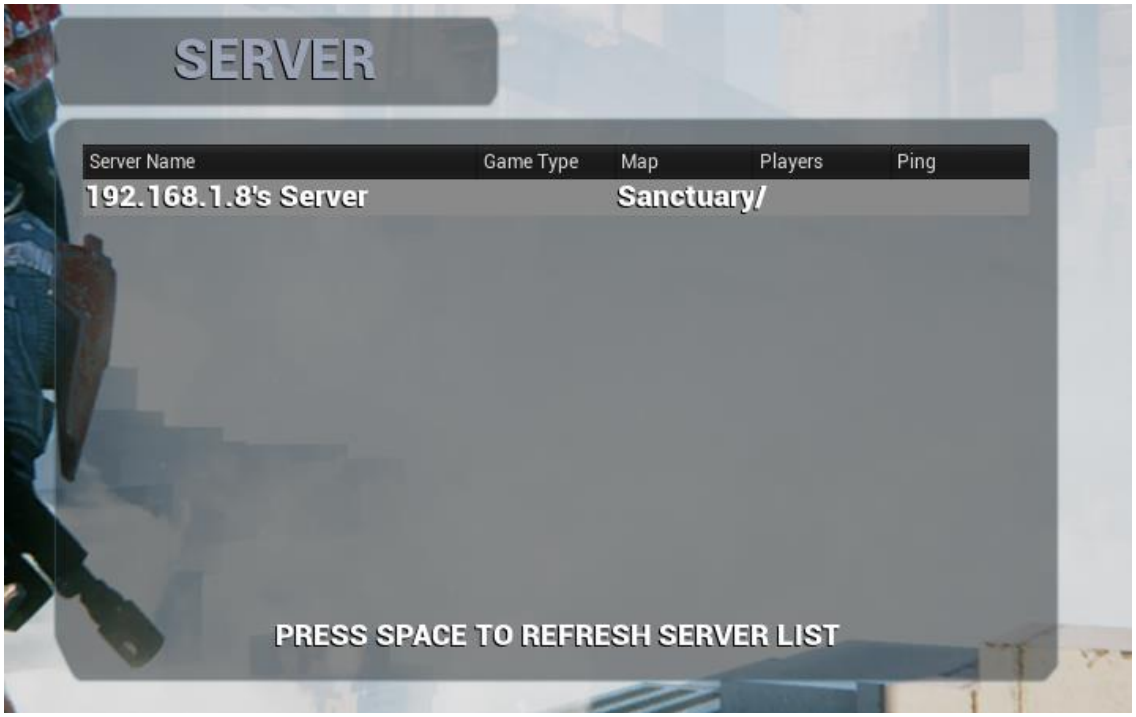
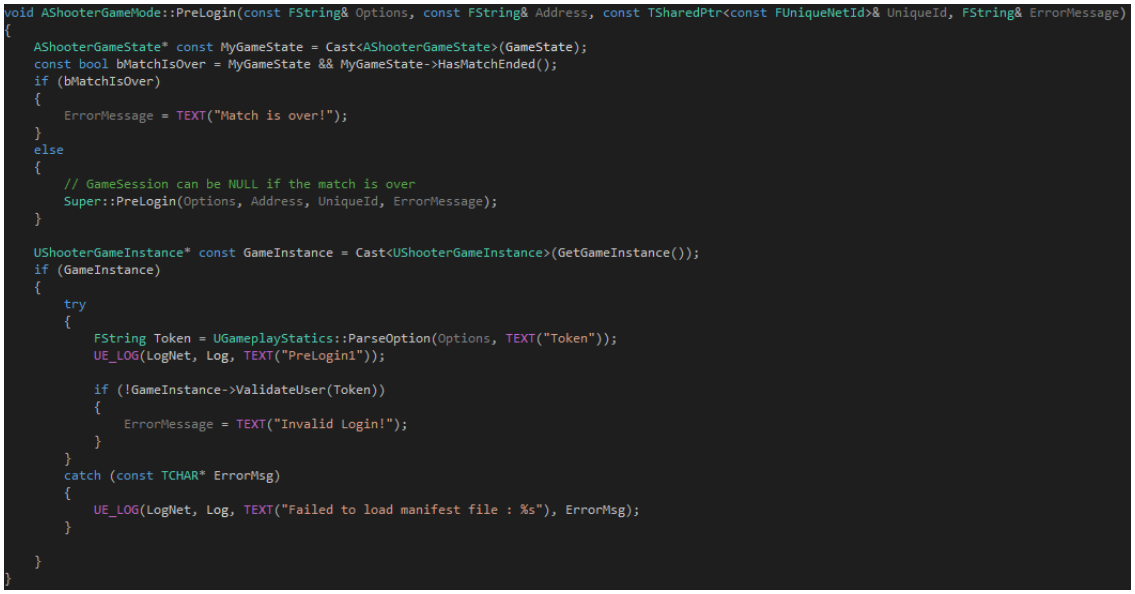


Figura 25 – Lista de servidores ativos

Como é possível ver na Figura 25, é representada a lista de servidores dedicados que o utilizador pode escolher para conectar-se, nesta lista vão sempre aparecer os endereços de IP que foram recebidos pelo cliente quando este fez o *login*, ou seja, a lista de servidores dedicados enviada pelo *Login Server*.



```
void AShooterGameMode::PreLogin(const FString& Options, const FString& Address, const TSharedPtr<const FUniqueNetId>& UniqueId, FString& ErrorMessage)
{
    AShooterGameState* const MyGameState = Cast<AShooterGameState>(GameState);
    const bool bMatchIsOver = MyGameState && MyGameState->HasMatchEnded();
    if (bMatchIsOver)
    {
        ErrorMessage = TEXT("Match is over!");
    }
    else
    {
        // GameSession can be NULL if the match is over
        Super::PreLogin(Options, Address, UniqueId, ErrorMessage);
    }

    UShooterGameInstance* const GameInstance = Cast<UShooterGameInstance>(GetGameInstance());
    if (GameInstance)
    {
        try
        {
            FString Token = UGameplayStatics::ParseOption(Options, TEXT("Token"));
            UE_LOG(LogNet, Log, TEXT("PreLogin1"));

            if (!GameInstance->ValidateUser(Token))
            {
                ErrorMessage = TEXT("Invalid Login!");
            }
        }
        catch (const TCHAR* ErrorMsg)
        {
            UE_LOG(LogNet, Log, TEXT("Failed to load manifest file : %s"), ErrorMsg);
        }

    }
}
```

Figura 26 – Método PreLogin da classe AShooterGameMode

O método `PreLogin` é sempre invocado depois do método `Login`, é responsável por aceitar ou rejeitar um cliente que esteja a tentar conectar-se ao servidor dedicado, ou seja, o *Unreal* fez assim ser possível controlar quem entra no servidor dedicado, sendo aqui o sitio para rejeitar a nova conexão.

Também é de salientar que ainda não foi criado qualquer `APlayerController`, para rejeitar a nova conexão é necessário que a variável `ErrorMessage` seja diferente de vazio, como é apresentado na Figura 26, primeiro é preciso buscar um dos parâmetros que são enviados pelo cliente que está a tentar ligar-se ao servidor, para tal é preciso ir a string `Options`, sendo nesta *string* onde são enviados os dados que queremos passar do cliente para o servidor. Neste caso o parâmetro que se precisa é o *token*, este que foi criado no *Login Server* e recebido depois de efetuar o *Login* com sucesso.

Com o valor do *token* obtido dos parâmetros é necessário processar a sua validação, para tal, envia-se o *token* para o serviço responsável para validar os *token* que é o Login Server, sendo assim invoca-se o método `ValidarUser` (Figura 27) dá classe `UShooterGameInstance`.

```cpp
bool ALoginConnection::ValidateUser(const FString& Token)
{
    int32 sent = 0;
    FString myToken = Token;

    FTCHARToUTF8 tokenUTF8(*myToken);
    int tokenlength = tokenUTF8.Length();

    //alloc the memory
    int32 size = 1 + 4 + tokenlength;
    uint8* withprefixes = (uint8*)malloc(size);

    //add command:
    withprefixes[0] = (uint8)ECommandType::ValidateUser;

    int index = 1;
    WriteStringUTF8(withprefixes, index, tokenUTF8);

    if (Socket->Send(withprefixes, size, sent))
    {
        return OnReceivedDataBooking();
    }
    return false;
}
```

Figura 27 – Método ValidarUser da classe ALoginConnection

Aqui, na Figura 28, temos a representação da lógica por detrás da validação do *token* do utilizador, nesta função é enviada a *string* que representa o *token* guardado anteriormente e este é enviado para o *Login Server* para ser validado, como foi falado anteriormente, no servidor dedicado os pedidos podem ser bloqueados até terem uma resposta do outro lado, neste caso do *Login Server* com a indicação se o *token* é valido ou não, o método `OnReceivedDataBlocking` (será explicado com mais detalhe em seguida).

```cpp
private void OnValidateUserMessageReceived(ValidateUserMessage message, NetConnectionState netState)
{
    if (!(netState.Tag is ServerSlave)) return;

    Socket socket = netState.Socket;
    ServerSlave server = (ServerSlave)netState.Tag;
    message.IsValidate = ValidateToken(message.Token);
    if (message.IsValidate)
    {
        lock (server)
        {
            server.PlayerNum++;
        }
    }
    _server.Send(netState, message);
} « void OnValidateUserMessageReceived(ValidateUserMessage, NetConnectionState)
```

Figura 28 – Método OnValidateUserMessageReceived do Login Server

```
private bool ValidateToken(string token)
{
    foreach (Player p in _players)
    {
        if (p.Token == token)
        {
            return true;
        }
    }
    return false;
} « bool ValidateToken(string)
```

Figura 29 – Método ValidateToken do Login Server

Na Figura 28 e Figura 29, é representado o fluxo de validação do *token* no *Login Server*, de uma forma simples é verificado se o *token* recebido existe na lista de utilizadores, ou seja, se o *token* é válido e se foi mesmo criado pelo *Login Server*, depois de validado é então atualizado o número de jogadores que existem naquele servidor dedicado, assim é possível ter um controlo mais assertivo sobre quais os servidores que se encontram cheios ou vazios.

```
bool ALoginConnection::OnReceivedDataBooking()
{
    if (!Socket) return false;

    TArray<uint8> ReceivedData;
    //FArrayReaderPtr Reader = MakeShareable(new FArrayReader(true));

    //blocking the thread from 3 seconds
    if (Socket->Wait(ESocketWaitConditions::WaitForRead, FTimespan::FromSeconds(3)))
    {
        uint32 Size;
        while (Socket->HasPendingData(Size))
        {
            //Reader->SetNumUninitialized(FMath::Min(Size, 65507u));
            ReceivedData.SetNumUninitialized(FMath::Min(Size, 1024u));

            int32 Read = 0;
            Socket->Recv(ReceivedData.GetData(), ReceivedData.Num(), Read);
        }

        //check if we receive any data
        if (ReceivedData.Num() <= 0) return false;

        //convert the data
        int index = 0;
        ECommandType Command = (ECommandType)ReceivedData[index];

        switch (Command)
        {
        case ECommandType::ServerInfo:
            break;
        case ECommandType::ValidateUser:
            //the [1] is the byte for the validate result
            return ReceivedData[1] == 1;
        case ECommandType::None:
            return false;
        }
    }

    return false;
}
```

Figura 30 – Método OnReceivedDataBlocking em C++

Como já tinha sido mencionado anteriormente, o método `OnReceivedDataBlocking`, Figura 30, permite bloquear a *thread* até existir alguma coisa para ser lido do *socket* ou se passar do tempo limite de espera, que foi definido para apenas esperar no limite três segundo, claro que este tempo pode ser alterado para dez segundos, dando assim o tempo necessário para o outro lado (neste caso, *Login Server*) poder processar o pedido e responder, mas como se trata de uma funcionalidade simples que é validar um *token*, três segundos são suficientes para receber a resposta do *Login Server*.

Quando o *socket* tem bytes para serem lidos, é então verificado se existe mesmo alguma coisa para ser lida, porque podemos estar no caso de ter passado o tempo limite de espera, com isso validamos se o *array* contem alguma informação, caso tenha alguns dados nele verifica-se o primeiro *byte* para saber qual o tipo do comando (mensagem) e caso seja do tipo `ValidateUser` só é preciso comparar o *byte* na posição um se é igual a um (`true`).

Figura 31 – Gameplay do demo ShooterGame

Por fim, Figura 31, temos a representação do jogo em si, depois de validar o login, selecionar o servidor dedicado e validar o *token*, o servidor dedicado autoriza o cliente a entrar dentro do mapa que esta a correr, como foi utilizado um o demo *ShooterGame* é possível jogar com vários jogadores um jogo igual a qualquer jogo comercial com o mesmo grau de experiencia.

Pode-se concluir que não é complicado adicionar o *login* a um jogo e como foi adicionado a um jogo *demo* que tem tudo para ser completo fica assim mostrada a facilidade de adicionar um *login* a um jogo, deste tipo ou doutro tipo, claro que isso vai obrigar a alguns ajustes, mas nunca começar algo do zero.

5.3.1.2 Adicionar Registo

Ao escolher a opção de registar e preencher os dados, esses dados vão ser enviados através de uma função em C++, `Register` e `SendData`, que converte os dados em *bytes*, enviando estes por *socket* ao *Login Server*. Esta função pode ser vista na Figura 32.

```
bool ALoginConnection::Register(const FString& Name, const FString& Password)
{
    return SendData((uint8)ECommandType::Register, Name, Password);
}
```

Figura 32 – Método Register em c++

Sobre a função `SendData`, Figura 11, é a mesma que foi explicada anteriormente.

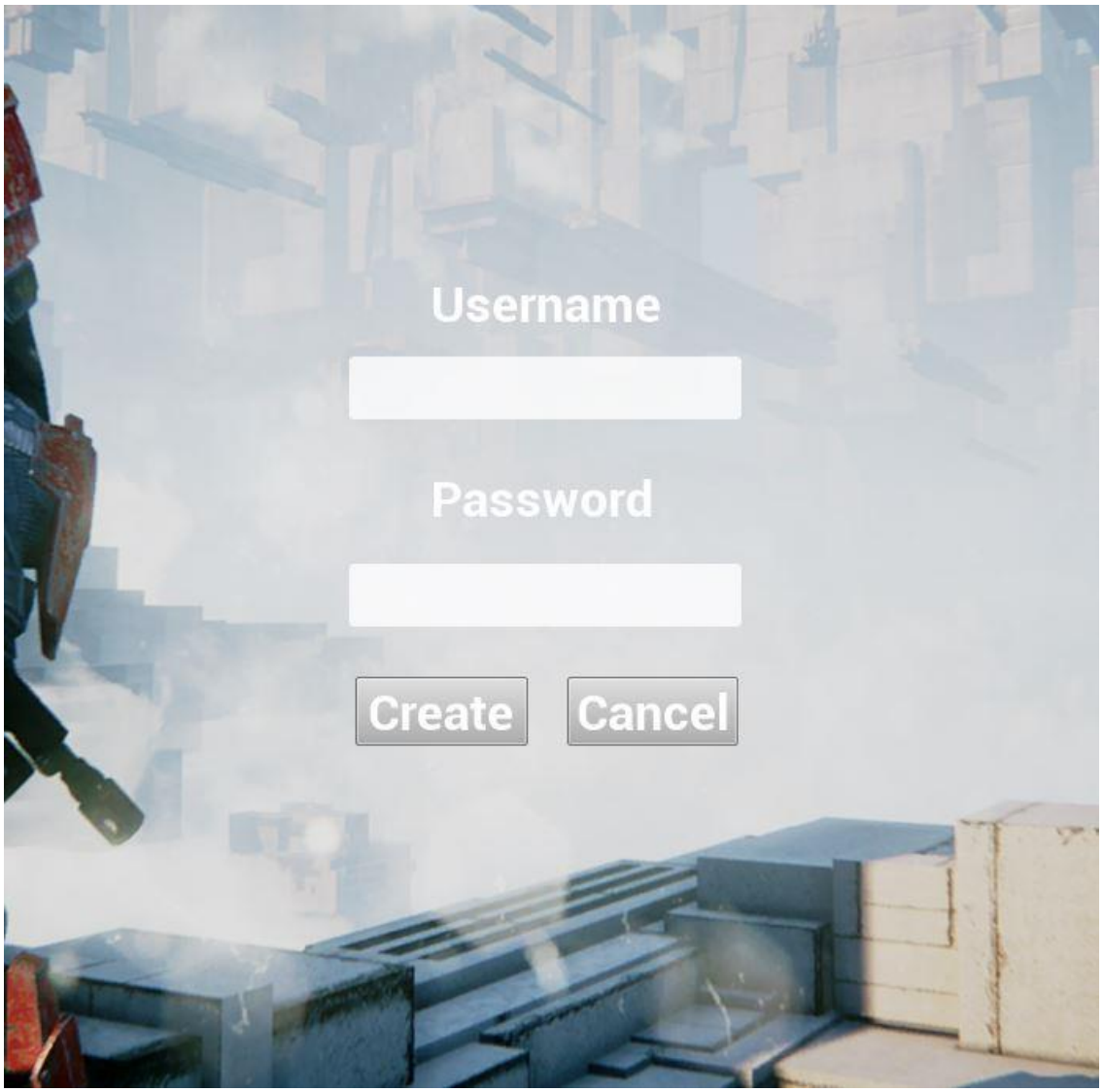


Figura 33 – Interface de registar um novo utilizador

A opção de registar um utilizador, Figura 33, permite criar um novo utilizador, para tal, é necessário introduzir o nome do utilizador (*username*) e a senha (*password*), depois de ter os dados preenchidos ao clicar no botão *Create* os dados vão ser enviados para o *Login Server*.

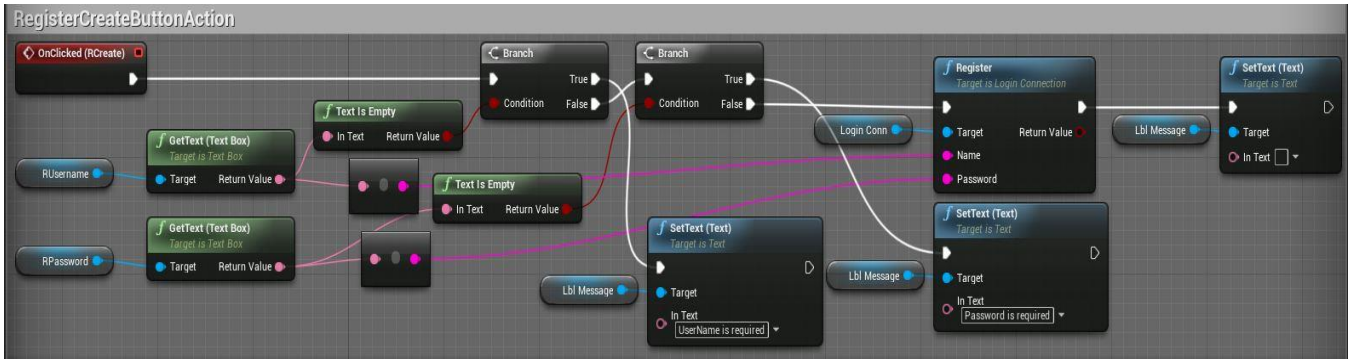


Figura 34 – Efetuar o registo *Blueprint*

Mas antes disso, Figura 34 , quando o utilizador clica no botão *Create* é executado no lado do cliente um *script* de *blueprint* onde são validados os dados que são preenchidos e só depois disso que se processa o envio deste para o *Login Server*, através da classe `LoginConnection`, que vai executar o método `Register` no C++ como foi possível ver na Figura 32.

Este blueprint pode ser consultado com mais detalhe no 9.5.

```
private void OnRegisterMessageReceived(RegisterMessage message, NetConnectionState netState)
{
    Socket socket = netState.Socket;
    //LoginMessage msg = new LoginMessage(stream);

    Console.WriteLine(message.Pass);
    //TODO: change the Role to enum
    UserBase user = new UserBase(message.Name, message.Pass, 'N');

    //create the response message
    BinaryWriter binWriter = new BinaryWriter(new MemoryStream());
    binWriter.Write((byte)CommandType.Register);
    binWriter.Write(_userBll.InsertUser(user));//success register
    binWriter.WriteStringUtf8(message.Name);

    _server.Send(netState, binWriter);
} « void OnRegisterMessageReceived(RegisterMessage, NetConnectionState)
```

Figura 35 – Método OnRegistserMessageReceived no *Login Server*

A Figura 35 representa o processo de registar um novo utilizador no *Login Server*, nesta função são recebidas todas as mensagens do tipo `RegisterMessage`, vindas de um cliente.

Nesta função é criado o objeto que representa o utilizador do tipo `UserBase`, será guardado na base de dados utilizando o *Dapper* (*ORM*) os dados base do utilizador, ou seja, nome, senha, hash[2] e com a `Role`

[2] *Hash* único do utilizador, para ser utilizado ao gerir as senhas, encriptado com o *sha512*.

'N' que significa 'Normal', isto quer dizer que o utilizador a ser criado é do tipo normal não sendo um utilizador especial como, por exemplo, administrador. A senha é encriptada com sha512[3].

Depois da criação do novo utilizador na base de dados é criada a mensagem de reposta a ser enviada para o cliente, esta mensagem tem apenas três dados, o identificador da mensagem, se o utilizador foi criado com sucesso e o nome do utilizador.

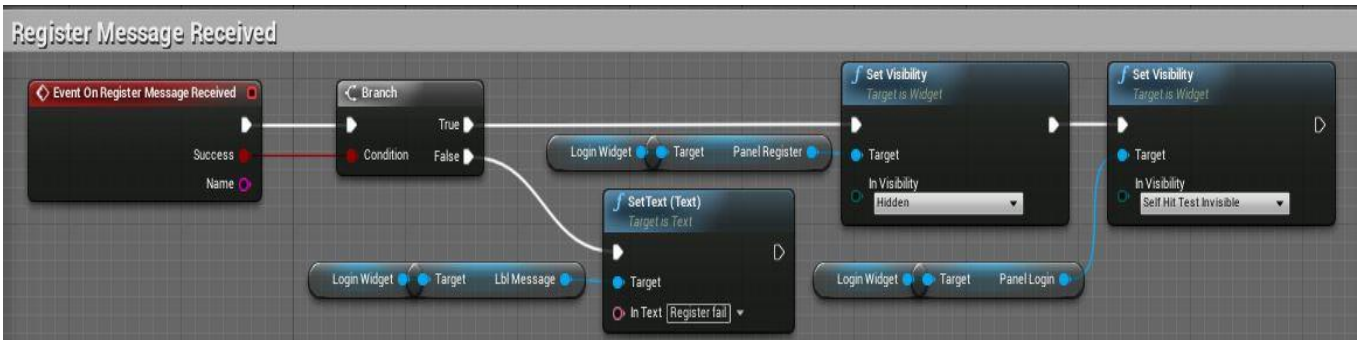


Figura 36 – Método OnRegisterMessageReiceved em blueprint

O método `OnRegisterMessageReiceved`, Figura 36, é responsável por tratar dos dados que são recebidos vindos do *Login Server* quando a sua mensagem é referente ao tipo registo, quando os dados são recebidos no *socket* este vai verificar qual o tipo de mensagem e depois chama o método responsável pela o tratamento dessa mensagem, neste caso é de registo.

Este *blueprint* tem uma funcionalidade muito simples, que se resume apenas a verificar se o utilizador foi criado com sucesso ou não, no caso afirmativo é escondido o painel do registo e é mostrado de volta o painel de *login*, sendo assim possível ao utilizador poder efetuar o login com o seu nome e senha que escolheu para criar a sua conta, caso exista um erro é exibida uma mensagem de erro: `Register Fail`.

Este *blueprint* pode ser consultado com mais detalhe no 9.5

[3] Sha512 é um algoritmo de encriptação de 512 bits para realizar a encriptação.

5.3.1.3 Adicionar *Chat*

Figura 37 – Interface chat

Ao clicar na tecla "*c*" é aberta uma janela de chat, Figura 37, onde o utilizador pode introduzir o texto que deseja que seja enviado para os elementos da sua equipa.

```
void SChatWidget::OnChatTextCommitted(const FText& InText, ETextCommit::Type InCommitInfo)
{
    if (InCommitInfo == ETextCommit::OnEnter)
    {
        if (GetPlayerController().IsValid() && !InText.IsEmpty())
        {
            // broadcast chat to other players
            GetPlayerController()->Say(InText.ToString());

            if(ChatEditBox.IsValid())
            {
                // Add the string so we see it too (we will ignore our own strings in the receive function)
                AddChatLine( InText, true );

                // Clear the text
                ChatEditBox->SetText(FText());

                // Audible indication we sent a message
                FSlateApplication::Get().PlaySound(ChatStyle->TxMessgeSound);
            }
        }
    }

    // If we want to dismiss chat after say, and we are not always visible, hide it now.
    if ((bAlwaysVisible == false) && ( bDismissAfterSay == true ) )
    {
        SetEntryVisibility(EVisibility::Hidden);
    }
}
```

Figura 38 – Método OnChatTextCommitted em C++

Quando o utilizador escreve o texto a ser enviado e clica na tecla "*Enter*" vai ser executada a função `OnChatTextCommitted`, Figura 38 , nesta função é onde o texto é enviado por *broadcast* para os outros jogadores, isto enviado primeiro para o servidor dedicado e este envia essa mensagem para os outros jogadores.

Também é inserida logo na caixa de mensagem do *chat* o que o utilizador escreveu porque vai ser ignorado no método que recebe as mensagens do *chat*.

```
void AShooterPlayerController::Say( const FString& Msg )
{
    ServerSay(Msg.Left(128));
}

bool AShooterPlayerController::ServerSay_Validate( const FString& Msg )
{
    return true;
}

void AShooterPlayerController::ServerSay_Implementation( const FString& Msg )
{
    GetWorld()->GetAuthGameMode()->Broadcast(this, Msg, ServerSayString);
}
```

Figura 39 – Método Say e RPC ServerSay em C++

O método `Say`, Figura 39, é invocado no lado do cliente e por sua vez executa um *Remote Procedure Call* (*RPC*) com o nome `ServerSay`, este método é executado apenas no servidor dedicado, sendo assim necessário verificar se o código está a correr no servidor dedicado, dái o que se vê na Figura 39, a chamada ao método `GetAuthGameMode`.

```
/* Overriden Message implementation. */
virtual void ClientTeamMessage_Implementation( APlayerState* SenderPlayerState, const FString& S, FName Type, float MsgLifeTime ) override;

/* Tell the HUD to toggle the chat window. */
void ToggleChatWindow();

/** Local function say a string */
UFUNCTION(exec)
virtual void Say(const FString& Msg);

/** RPC for clients to talk to server */
UFUNCTION(unreliable, server, WithValidation)
void ServerSay(const FString& Msg);
```

Figura 40 – Header AShooterPlayerController

```
/** @todo document */
UFUNCTION(Reliable, Client)
void ClientTeamMessage(class APlayerState* SenderPlayerState, const FString& S, FName Type, float MsgLifeTime = 0);
```

Figura 41 – Header APlayerController

Existe algo importante a saber sobre os *RPC*, para declarar uma função como *RPC* é preciso dizer ao compilador do *Unreal* que a função é um RPC para tal, é preciso colocar se a mensagem é `Reliable` ou `Unreliable` e onde esta vai ser executada; `Server`, `Client` ou `NetMulticast`.

Como se pode verificar na Figura 40 e Figura 41, o método `ServerSay` é do tipo `Unreliable`, ou seja, se a mensagem se perder não vai ser enviada outra vez, tem validação `WithValidation` neste caso retorna-se verdadeiro e é executada apenas no servidor, Server.

A Figura 41 serve como enquadramento para saber que o método `ClientTeamMessage` é definido na classe pai como `Reliable` e a executar no cliente apenas, é este método que recebe a mensagem do servidor dedicado.

```
void AGameMode::Broadcast( AActor* Sender, const FString& Msg, FName Type )
{
    APlayerState* SenderPlayerState = NULL;
    if ( Cast<APawn>(Sender) != NULL )
    {
        SenderPlayerState = Cast<APawn>(Sender)->PlayerState;
    }
    else if ( Cast<AController>(Sender) != NULL )
    {
        SenderPlayerState = Cast<AController>(Sender)->PlayerState;
    }

    for( FConstPlayerControllerIterator Iterator = GetWorld()->GetPlayerControllerIterator(); Iterator; ++Iterator )
    {
        (*Iterator)->ClientTeamMessage( SenderPlayerState, Msg, Type );
    }
}
```

Figura 42 – Método Broadcast em C++

Quando o `ServerSay` executa o método `broadcast` do `AGameMode`, Figura 42, permite assim que a mensagem recebida pelo servidor seja enviada para todos os clientes ligados ao servidor dedicado.

```
void AShooterPlayerController::ClientTeamMessage_Implementation( APlayerState* SenderPlayerState, const FString& S, FName Type, float MsgLifeTime  )
{
    AShooterHUD* ShooterHUD = Cast<AShooterHUD>(GetHUD());
    if (ShooterHUD)
    {
        if( Type == ServerSayString )
        {
            if( SenderPlayerState != PlayerState  )
            {
                ShooterHUD->AddChatLine(FText::FromString(S), false);
            }
        }
    }
}
```

Figura 43 – Método ClientTeamMessage

O método `ClientTeamMessage`, Figura 43, é executado em cada cliente quando o servidor dedicado envia um *broadcast* para os clientes com o texto para ser introduzido no chat de cada cliente ligado ao servidor.

Este é um fluxo de envio de mensagens de texto pelo chat dentro do jogo, que já existia no demo *ShooterGame*, como se pode ver, é algo muito simples de ser implementado usando o *Unreal Engine*.

5.3.1.4 Mostrar nome do jogador

Figura 44 – Nome do jogador

Na Figura 44 é possível visualizar o nome do jogador, *Bot 1*, por cima da sua personagem no jogo, neste caso é um NPC ou por outras palavras um jogador controlado pelo computador. Para adicionar o nome do jogador num jogo foi necessário alterar a classe principal que controla as personagens dentro do jogo, ou seja, a classe AShooterCharacter, que é tanto usada para o *blueprint* de um jogador humano (*PlayerPawn*) como para um jogador do computador (*BotPawn*).

Figura 45 – Construtor da classe AShooterCharacter

```
FloatingTextHeight = 10.f;

static ConstructorHelpers::FClassFinder<UUserWidget> PlayerNameWidget(TEXT("/Game/Blueprints/UI/PlayerName"));
PlayerNameWidgetClass = PlayerNameWidget.Class;
FloatingPlayerName = ObjectInitializer.CreateDefaultSubobject<UWidgetComponent>(this, TEXT("FloatingPlayerName"));
FloatingPlayerName->SetupAttachment(RootComponent);
FloatingPlayerName->SetWidgetClass(PlayerNameWidgetClass);
FloatingPlayerName->SetWidgetSpace(EWidgetSpace::World);
FloatingPlayerName->SetWorldLocation(FVector(0, 0, GetCapsuleComponent()->GetCollisionShape().GetCapsuleHalfHeight() + FloatingTextHeight));
FloatingPlayerName->SetCollisionEnabled(ECollisionEnabled::NoCollision);
FloatingPlayerName->SetMaxInteractionDistance(250.f);
FloatingPlayerName->SetPivot(FVector2D(0.5f, 0.5f));
FloatingPlayerName->SetDrawSize(FVector2D(350, 80));
FloatingPlayerName->bOnlyOwnerSee = false;
FloatingPlayerName->bOwnerNoSee = true;
FloatingPlayerName->SetVisibility(true);
```

Para permitir que cada jogador tenha o nome dele por cima da sua personagem é preciso criar um *Widge.* Neste caso, a solução passou por ser criada um *widget blueprint* que herda da classe UUserWidget.

Depois disso é preciso criar um componente, UWidgetComponent, que ainda se encontra num estado beta, este componente permite criar Widget 3D dentro do jogo de uma forma muito simples, basta então indicar o *Widget* que o componente tem que fazer *render* e outros dados básicos para o seu funcionamento conforme o comportamento que se quer, mas só com a indicação do que o *widget* fica pronto para ser usado.

Este componente depois de configurado é adicionado à árvore de componente, sendo assim possível visualizar o componente tanto no jogo como no *blueprint*. No caso do *blueprint* é possível ver isso na Figura 46 (`FloatingPlayerName`) e no caso do jogo, na Figura 44.

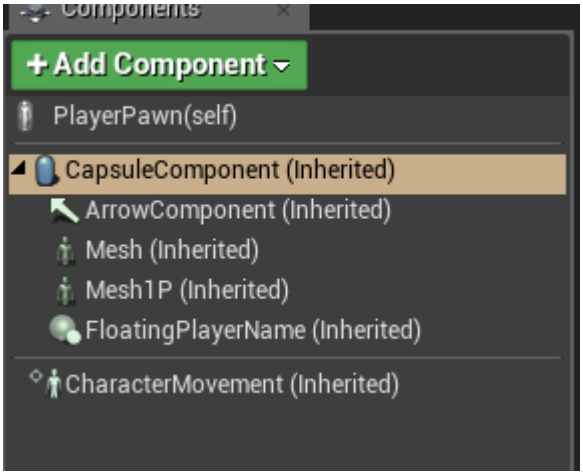


Figura 46 – Árvore de components do *blueprint* PlayerPawn

Na implementação em C++ houve alguns problemas devido a não se conseguir aceder ao `APlayerState` do *blueprint* porque, ao tentar aceder ao `APlayerState` estava-se a aceder a outro objeto errado, uma vez que estamos a aceder dentro de uma classe que ainda não tem acesso ao objeto. Com este problema foi necessário criar duas funções no evento `Tick` uma em C++ e outra em *blueprint*.

Figura 47 – Método UpdateFloatingText em C++

```
void AShooterCharacter::UpdateFloatingText(class AShooterPlayerController* MyPC)
{
    //update the floating text rotation to look at player
    ACharacter* myCharacter = UGameplayStatics::GetPlayerCharacter(GetWorld(), 0);
    if (myCharacter && FloatingPlayerName)
    {
        if (!FloatingPlayerName->IsVisible())
        {
            FloatingPlayerName->SetVisibility(true);
        }

        FRotator newRotator = UKismetMathLibrary::FindLookAtRotation(FloatingPlayerName->GetComponentLocation(), myCharacter->GetActorLocation());
        FloatingPlayerName->SetWorldRotation(FRotator(UKismetMathLibrary::Clamp(newRotator.Pitch, 0, 90), newRotator.Yaw, 0.f).Clamp());
    }
    else
    {
        FloatingPlayerName->SetVisibility(false);
    }
}
```

A primeira função, Figura 47, passa por verificar se o objeto com o texto nome não é nulo e se o `ACharacter` do jogador local (o jogador que está a jogar) também não é nulo. Caso seja nulo, então são escondidos os nomes dos jogadores, devido a ficar com um comportamento estático e sem efetuar o olhar para a câmara. Caso não seja nulo, são mostrados os nomes dos jogadores e o nome vai ser rodado conforme a posição do jogador local, como a câmara se encontra junto a personagem, então o nome vai rodar e vai ficar a olhar para a câmara. Também foi preciso limitar a rotação máximo no eixo do Y, porque senão quando um jogador salta perto de outro jogador o nome ia rodar demais ficando com o comportamento incorreto, ou seja, ia ficar como se fosse uma linha no ecrã do outro jogador que estava a saltar.

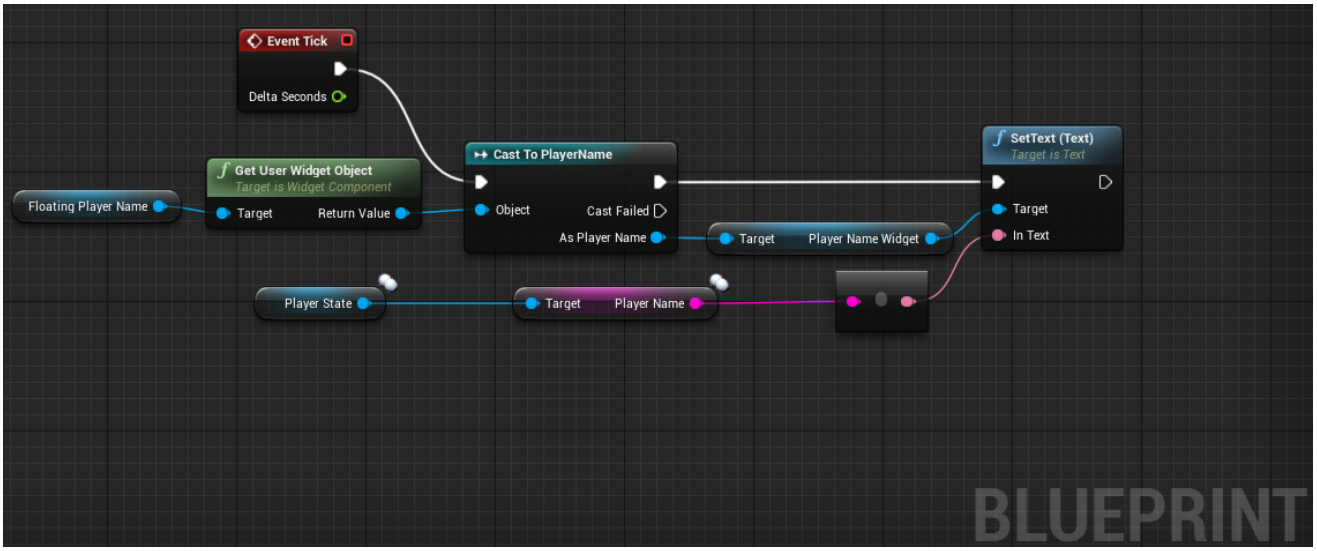


Figura 48 – Evento Tick do blueprint PlayerPawn ou BotPawn

A segunda função, Figura 48, foi implementada em *blueprint* devido ao problema do `AplayerState`. Existem alguns objetos que só são criados no *blueprint*, isto quando se utiliza classes *bleuprint* porque se utilizar todo com classes em C++ já se evitam alguns destes problemas. Este evento é executado por cada `Tick` do jogo e o objetivo dela é alterar o nome do jogador, para isso é acedido o componente com `Widget` e depois é preciso buscar o `PlayerName` ao `APlayerState` fazendo assim o `SetText` do `Widget`.

Este *blueprint* pode ser consultado com mais detalhe no 9.6.

5.3.1.5 Definir *spawn* do jogador

```
class AShooterTeamStart : public APlayerStart
{
    GENERATED_UCLASS_BODY()

    /** Which team can start at this point */
    UPROPERTY(EditInstanceOnly, Category=Team)
    int32 SpawnTeam;

    /** Whether players can start at this point */
    UPROPERTY(EditInstanceOnly, Category=Team)
    uint32 bNotForPlayers:1;

    /** Whether bots can start at this point */
    UPROPERTY(EditInstanceOnly, Category=Team)
    uint32 bNotForBots:1;
};
```

Figura 49 – Classe AShooterTeamStart

Para definir os *spawn* do jogador pelo mundo virtual do jogo basta criar uma classe que herda da classe `APlayerStart`, Figura 49, esta classe fica assim com as funcionalidades base que o próprio *Unreal Engine* já fez. O Unreal Engine já tem métodos para escolher um ponto de *spawn* à sorte (`ChoosePlayerStart`), ou seja, ele escolhe aleatoriamente um objeto na cena do jogo que seja do tipo `APlayerStart`.

Como o projeto está a utilizar o demo *ShooterGame,* existem as próprias classes para *spwan* e também o `override` do método `ChoosePlayerStart`, Figura 50 e Figura 51.

```
AActor* AShooterGameMode::ChoosePlayerStart_Implementation(AController* Player)
{
    TArray<APlayerStart*> PreferredSpawns;
    TArray<APlayerStart*> FallbackSpawns;

    APlayerStart* BestStart = NULL;
    for (TActorIterator<APlayerStart> It(GetWorld()); It; ++It)
    {
        APlayerStart* TestSpawn = *It;
        if (TestSpawn->IsA<APlayerStartPIE>())
        {
            // Always prefer the first "Play from Here" PlayerStart, if we find one while in PIE mode
            BestStart = TestSpawn;
            break;
        }
        else
        {
            if (IsSpawnpointAllowed(TestSpawn, Player))
            {
                if (IsSpawnpointPreferred(TestSpawn, Player))
                {
                    PreferredSpawns.Add(TestSpawn);
                }
                else
                {
                    FallbackSpawns.Add(TestSpawn);
                }
            }
        }
    }
```

Figura 50 – Método ChoosePlayerStart em c++ parte 1

```
    if (BestStart == NULL)
    {
        if (PreferredSpawns.Num() > 0)
        {
            BestStart = PreferredSpawns[FMath::RandHelper(PreferredSpawns.Num())];
        }
        else if (FallbackSpawns.Num() > 0)
        {
            BestStart = FallbackSpawns[FMath::RandHelper(FallbackSpawns.Num())];
        }
    }

    return BestStart ? BestStart : Super::ChoosePlayerStart_Implementation(Player);
}
```

Figura 51 - Método ChoosePlayerStart em c++ parte 2

Na Figura 50 e Figura 51 é representado o método `ChoosePlayerStart` da classe `AShooterGameMode`, isto significa que este método faz `override` ao método por defeito do *unreal engine*.

Neste método é alterado o comportamento da escolha do *spawn* para um jogador nascer, o critério é escolher os *spawn* que sejam mais preferidos e depois escolher um *spawn* aleatório entre eles.

Caso não existam *spawn* preferidos ou de reserva, é utilizado o método por defeito do *Unreal Engine*.

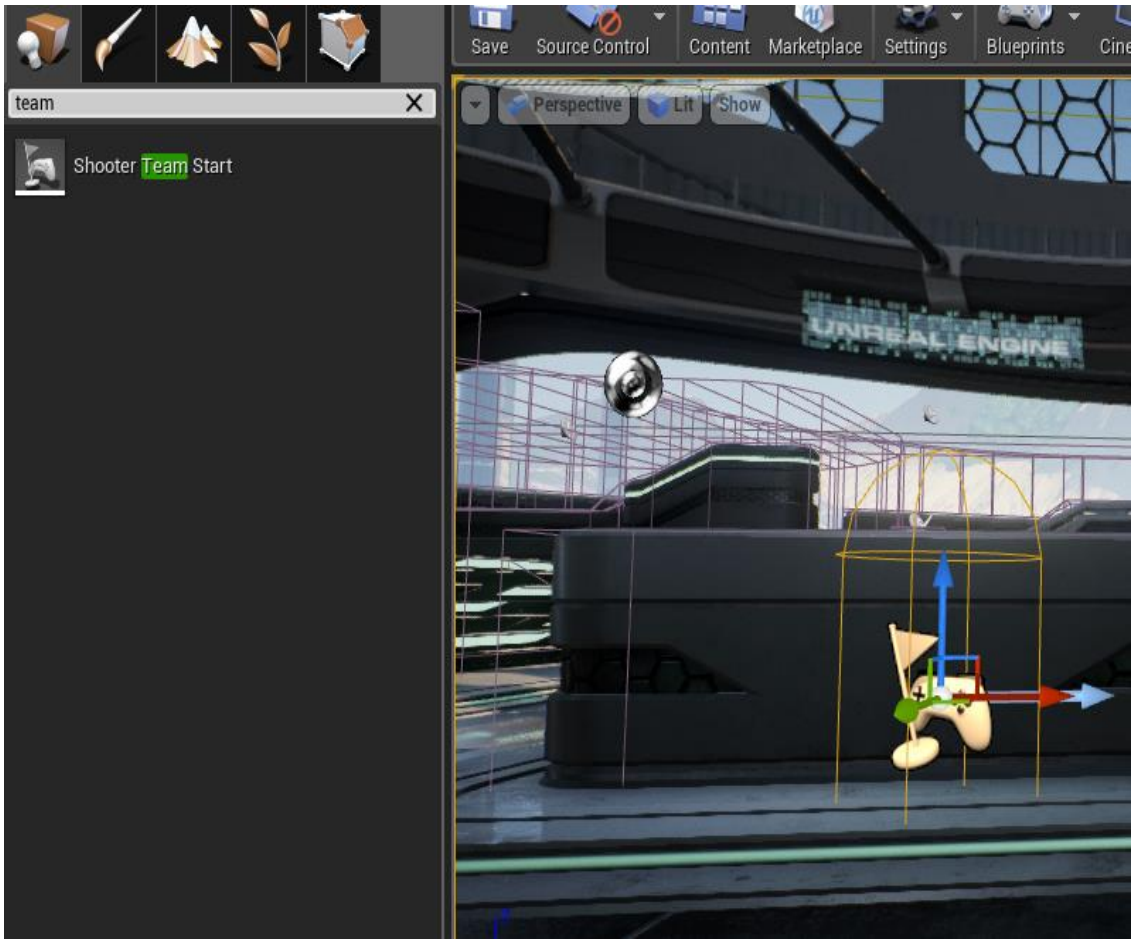


Figura 52 – Interface do editor, pesquisar pelo Shooter Team Start

Para adicionar um ponto a cena de jogo, é preciso pesquisar o `Actor` dentro do editor na Figura 52 é possível ver como é feito isso, depois disso basta arrastar o `Actor` para a cena do jogo e colocar onde se deseja que fique posicionado o novo *spawn*. De uma maneira muito simples é adicionado um novo *spawn* graças ao *Unreal Engine* ter já implementado todas as funcionalidades base para simplificar esta tarefa.

5.4 Configurações

Neste subcapítulo são apresentados os passos necessários para a configuração do servidor do *Sql Server*, que é usado pelo serviço de *login*. Também é mostrada a configuração do *Unreal Engine* deste e da cópia efetuada do repositório da *Epic Games* que é vital para a criação do servidor dedicado do jogo até à instalação do projeto de exemplo, *Shooter Game*.

5.4.1 Sql Server

Para a utilização da *framework* é necessário configurar o servidor de *Sql Server* 2014,

Primeiro passo é fazer o download do *sql server* 2014 e deve-se escolher a versão com o *sql management studio*, ou seja, a que diz *expresss and tools*.

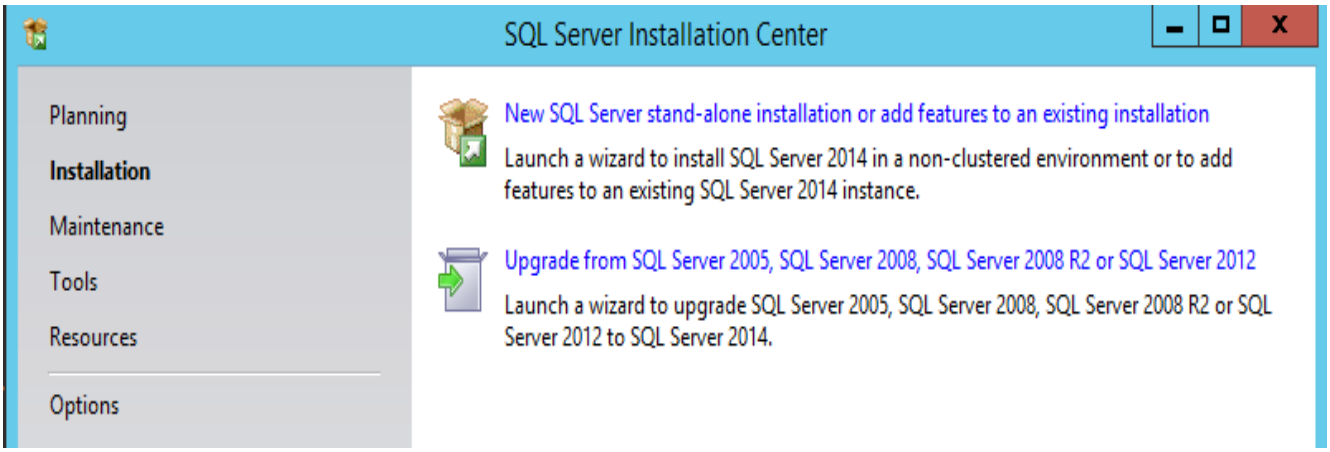


Figura 53 – Intalação do Sql Server

Para realizar a instalação do *sql server*, Figura 53, é necessário escolher a opção de instalar uma nova instância do *sql server*, ou seja, a primeira opção como é mostrado na Figura 53.

Ao longo dos passos da instalação é quase sempre indicado escolher o próximo passo no instalador, não existem muitas coisas para escolher por agora. Dependendo do caso pode ser necessário instalar mais alguma funcionalidade extra, mas neste caso são só necessárias as funcionalidades básicas.

Um dos passos que é necessário ter em atenção é nos serviços, quando se estão a configurar as opções no instalador é preciso verificar se estes serviços estão a correr como serviços de redes, porque senão vai ser impossível usar eles para comunicar entre a rede local ou fora.

Outro passo no instalador é configurar o utilizador administrador para ligar-se à instância *do sql server*, Figura 54, é representado esse passo e precisa-se de colocar a segunda opção no Modo de Autenticação.

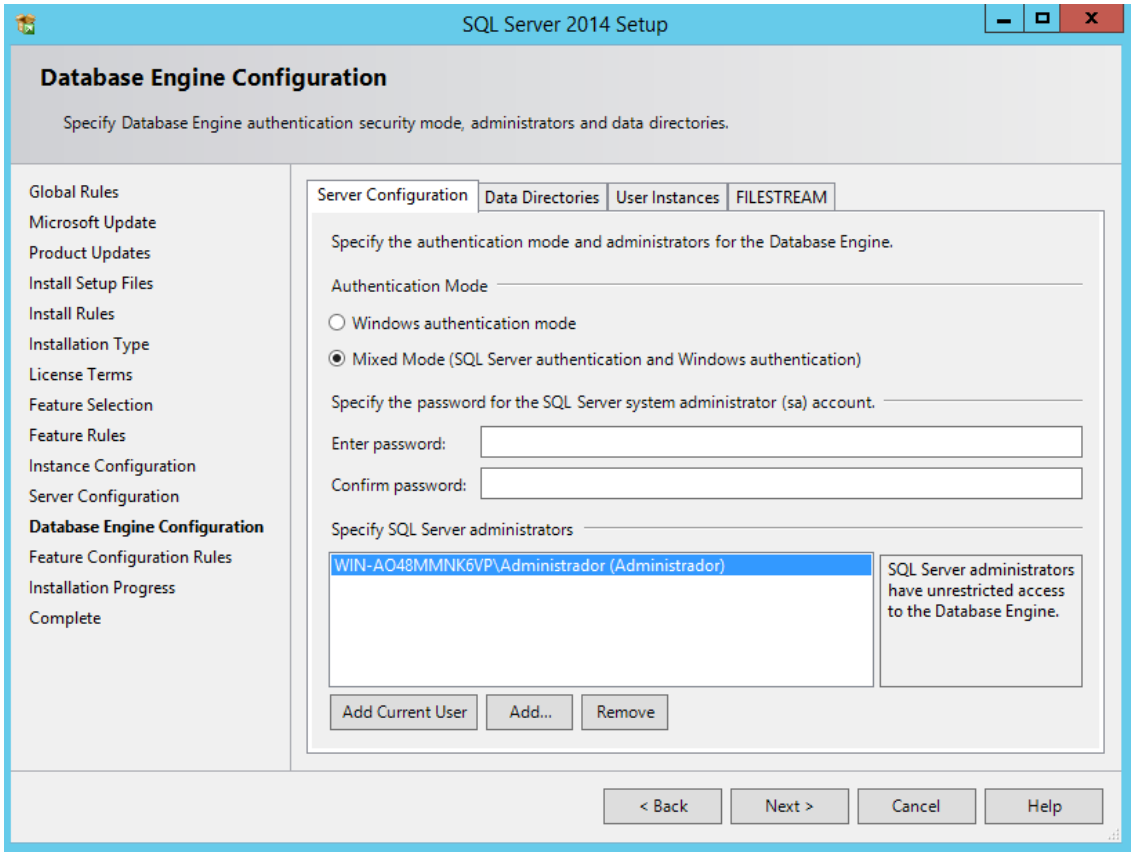


Figura 54 – Instalação *sql server database engine configuration*

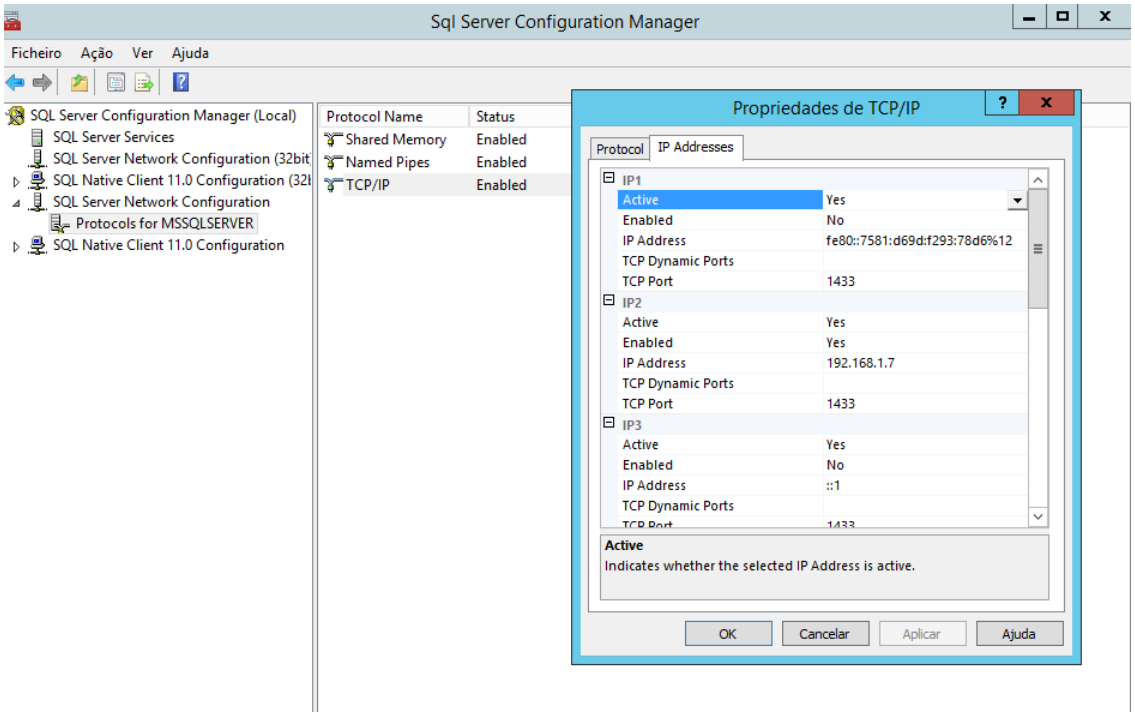


Figura 55 – *Sql Server Configuration Manager*

Na Figura 55 é apresentado o *Sql Server Configuration Manager*. Aqui é possível alterar a porta e os endereços de acesso a uma instância do *sql server* e, como tal, é necessário ativar o protoloco de TCP/IP e os endereços local (127.0.0.1) e de rede (192.168.1.x).

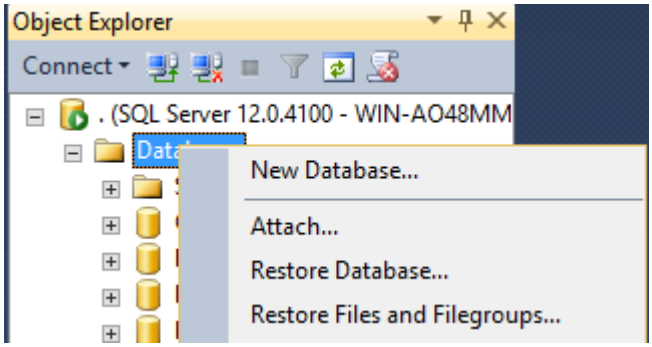


Figura 56 – *Restore Database*

Depois de abrir o *Sql Management* e entrar com as credenciais, é preciso fazer o restauro da base de dados, para isso clica-se no botão direito do rato nas bases de dados e escolhe-se a terceira opção "*Restore Database"*, Figura 56, escolhe-se o *device* onde está o ficheiro e clica-se em Ok.

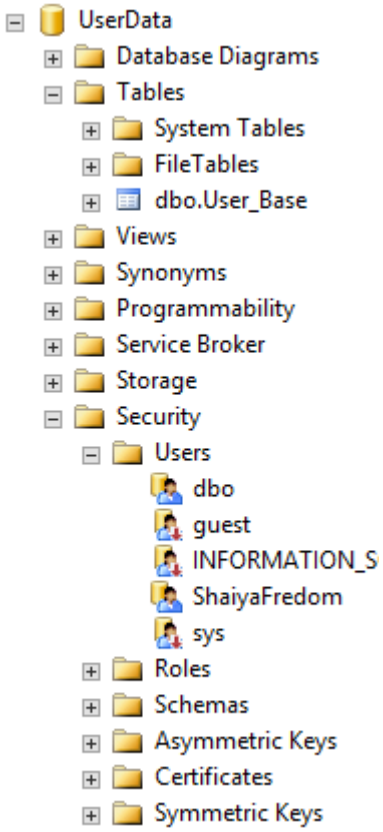


Figura 57 – Base de dados UserData

Depois do Restauro vai aparecer uma base de dados *UserData* como é possível ver na Figura 57, com uma única tabela User*_Base*.

5.4.2 Unreal Engine 4

Sobre a configuração do *Unreal Engine,* é preciso criar uma conta no *GitHub* porque é necessário ter o código fonte do *Unreal Engine* para se poder compilar o servidor dedicado.

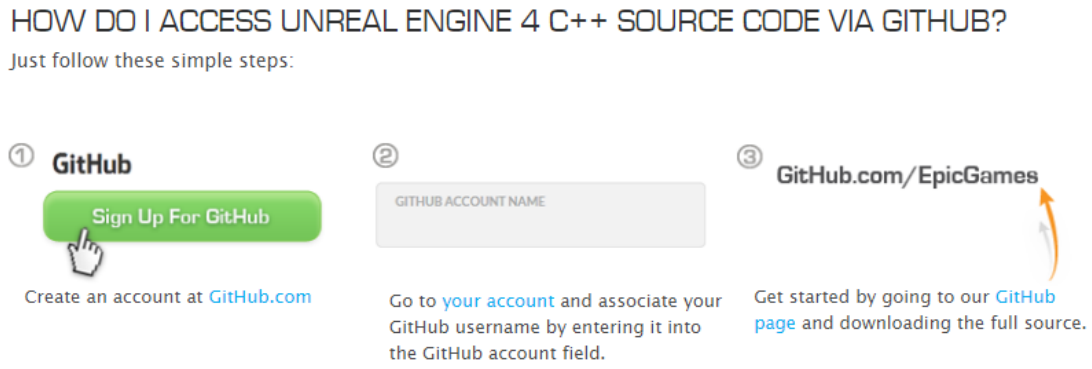


Figura 58 – Associar GitHub a conta EpicGames

Primeiro é preciso criar uma conta no *Unreal engine* e também no *GitHub*[4]. Depois disto, é preciso ativar na conta do *Unreal Engine* o acesso ao código fonte do *Github*, ou seja, associar a conta *Unreal Engine* com o *username* da conta do *GitHub*. Quando isto estiver completo, é apenas necessário fazer o *download* do código fonte.

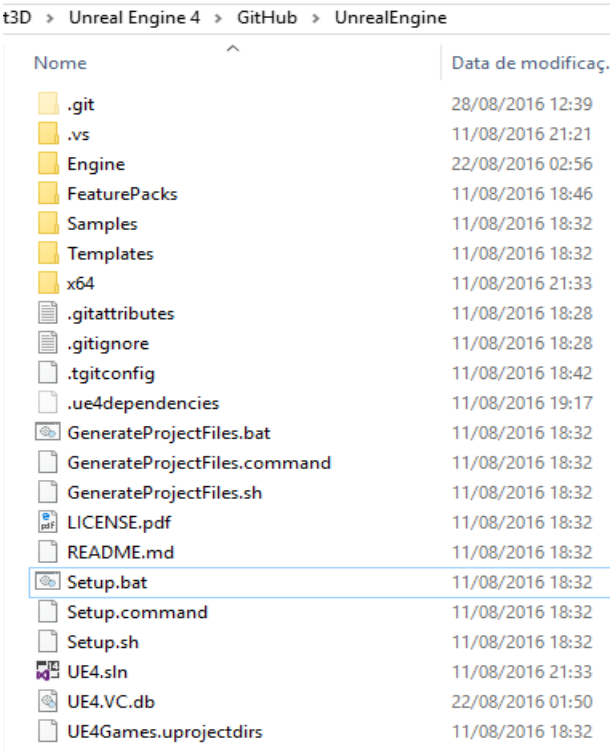


Figura 59 – Código fonte do *Unreal Engine* do *GitHub*

[4]Site do repositório de código GitHub: https://github.com/

Para o segundo passo, Figura 59, é necessário compilar o código fonte do *Unreal Engine*. Para isso é preciso executar o ficheiro *Setup.bat*. Este processo vai demorar algum tempo porque estão a ser criados os ficheiros principais do *Unreal Engine*. Depois de compilados os ficheiros, é preciso criar os ficheiros de projeto para o *Visual Studio*, ou seja, executar o outro ficheiro que se chama *GenerateProjectFiles.bat*, este exemplo aplica-se a um computador com o sistema operativo Windows.

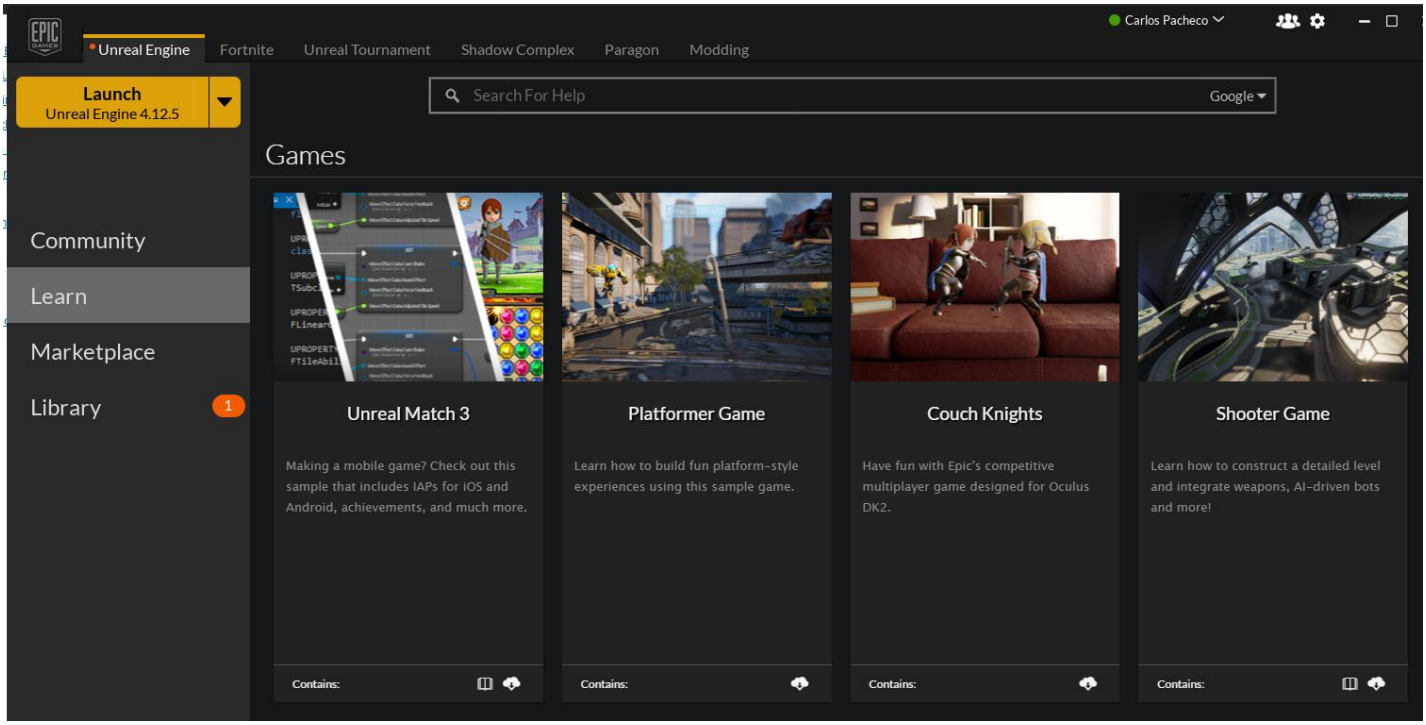


Figura 60 – Instalar o *demo Shooter Game*

A terceira etapa na configuração do *Unreal Engine* passa por instalar o *demo* que vai ser utilizado como base neste projeto com a *framework* desenvolvida.

O demo encontra-se no submenu *Learn* do programa da *Epic Games*, depois de fazer *scroll* o projeto vai estar mais ou menos nos últimos exemplos na categoria *Games*, como é mostrado na Figura 60. Clica-se onde diz *Shooter Game* e coloca-se a fazer *download*, depois cria-se um projeto com o exemplo *Shooter Game*, basta para isso clicar no botão por baixo do exemplo.

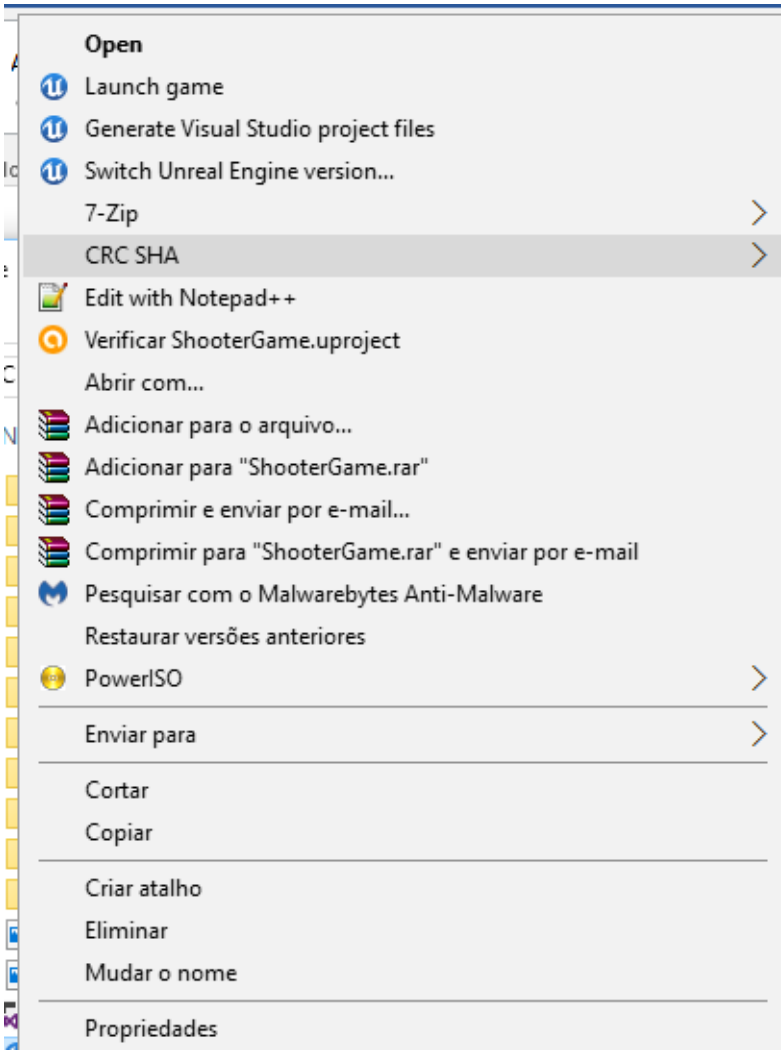


Figura 61 – Menu do ficheiro ShooterGame.uproject

O quarto passo é aplicado depois da criação do projeto com o *demo ShooterGame*, só com o projeto criado é possível fazer este passo. Na pasta escolhida para criar o projeto existe um ficheiro com o nome *ShooterGame.uproject*, clica-se nele com o botão direito do rato e selecciona-se a opção *"Switch Unreal Engine version…"*, Figura 61.

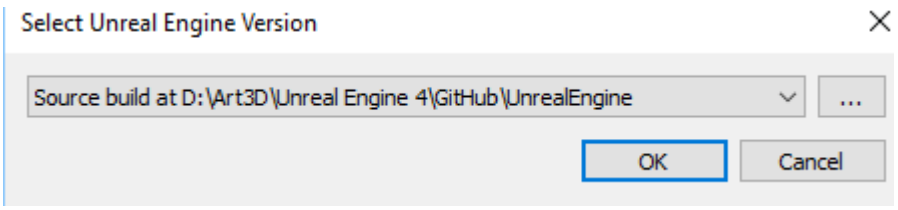


Figura 62 – Opções da versão do *Unreal Engine* para um projeto

Na Figura 62 é possível ver a lista com as varias versões existentes no computador, isto depois de configurar o código fonte, vai ser possível ver a opção para escolher a versão do *Unreal Engine* do *GitHub*.

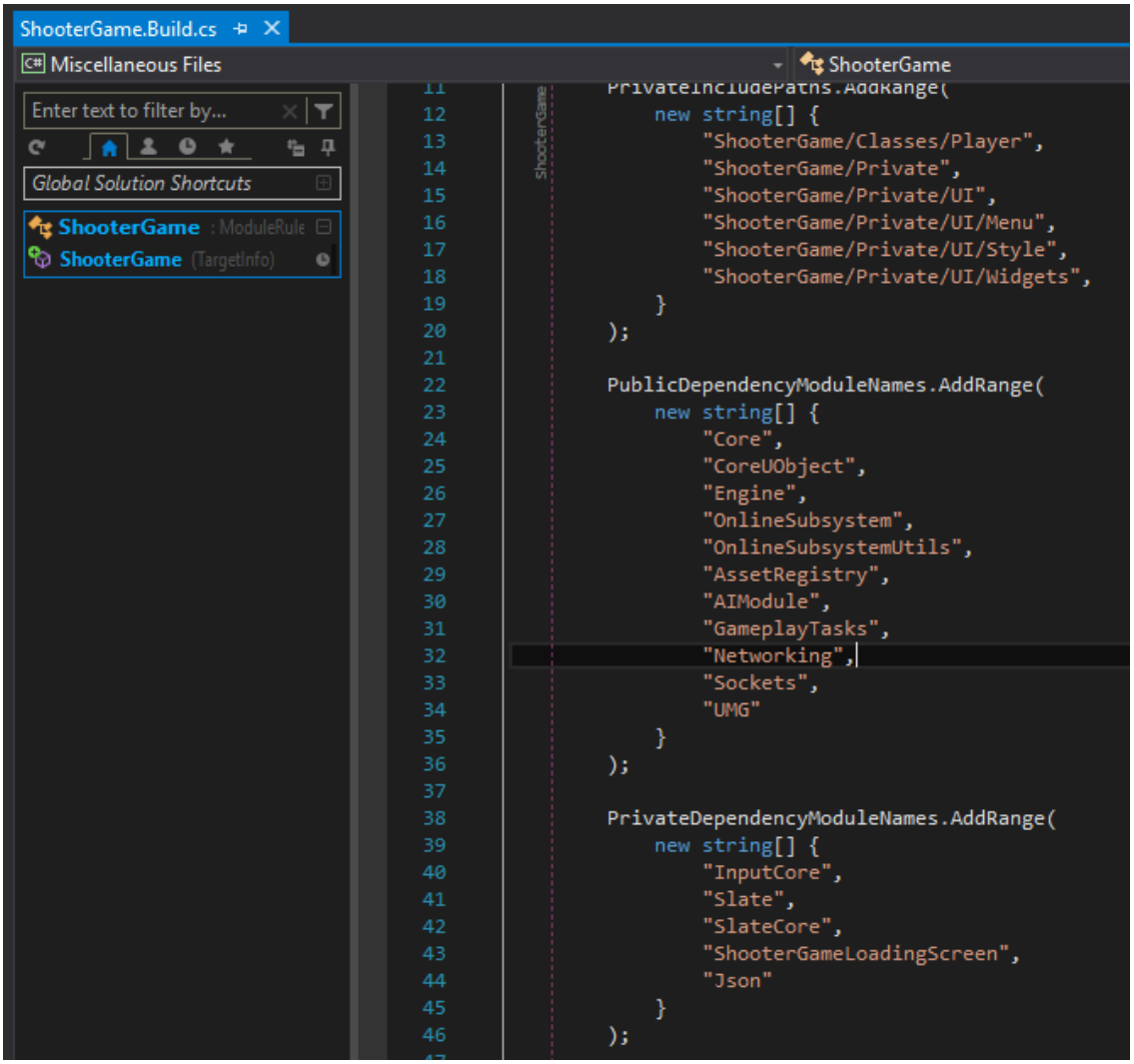


Figura 63 – Ficheiro de configuração ShooterGame.Build

Por fim, Figura 63, existe um ficheiro de configuração que foi criado junto com o demo *ShooterGame*. Este ficheiro permite adicionar as dependências a outros módulos criados pela comunidade ou pela equipa do *Unreal Engine*.

Para a *framework* trabalhar é necessário adicionar alguns módulos, que são:

- *Networking*
- *Sockets*
- *UMG*

Os módulos *networking* e *socket* servem para que seja possível usar as classes base do *unreal engine* para criar conexões ao *Login Server*.

O módulo *UMG* tem como funcionalidade criar os *widgets* no *Visual Studio,* que é o caso do nome do jogador.

6 Avaliação da solução

Neste capítulo será descrita a avaliação da solução. Com isto vão ser apresentadas as grandezas, hipóteses e metodologias usadas para avaliar a solução. Por fim, serão apresentados os resultados feitos, a solução e com isto concluir se a solução apresentada resolveu o problema inicial.

6.1 Grandezas e hipóteses

Sobre a avaliação da solução implementada serão necessárias utilizar duas grandezas. Em primeiro lugar, depois dos utilizadores utilizarem esta *framework* é preciso saber se os utilizadores gostaram da maneira como a criação de jogos foi feita e qual é a sua motivação depois de a usar. Segunda grandeza é qual foi o conhecimento que os utilizadores ganharam ao criar os jogos usando a *framework*, ou seja, conhecimento sobre a programação de vídeos jogos.

Neste aspeto existem duas hipóteses para avaliar estas grandezas:

- Os utilizadores conseguiram criar jogo *multiplayer* de forma rápida e simples?
- Os utilizadores tiveram mais conhecimentos depois de criarem um jogo?

6.2 Metodologias de avaliação

Pretende-se que, no final da implementação, a *framework* seja testada por um grupo de jovens onde o objetivo é testarem e criarem um jogo *multiplayer* (recorrendo a um guia) onde seja possível comunicarem entre si, isto de uma forma rápida e simples, sendo assim desnecessário ter conhecimentos avançados em programação.

Depois da utilização da *framework* os utilizadores vão preencher alguns questionários para avaliar as grandezas acima descritas, ou seja, a motivação atual dos utilizadores e os conhecimentos adquiridos.

6.3 Resultados

Para avaliação das grandezas inicialmente definidas foram realizados dois questionários, um antes do uso da *framework* (apenas com *Unreal Engine*) e um depois do uso da *framework*.

Nesses questionários realizados ao grupo de teste, existem duas perguntas que vão ser o ponto de partida para a avaliação, que são:

- Qual a sua motivação para desenvolver vídeos jogos?
- Considera que adquiriu novos conhecimentos na programação?

De seguida, serão explicados os dados obtidos nesses questionários por pergunta e comparados, para assim facilitar as conclusões obtidas.

Qual a sua motivação para desenvolver vídeos jogos?						
	Não		Talvez		Sim	
	P	IC	P	IC	P	IC
Com framework	20 %	7,60% - 32,39%	30%	15,79% - 44,20%	50%	34,50% - 65,49%
Sem framework	60%	43,02% - 76,97%	30%	15,79% - 44,20%	10%	0,70% - 19,29%

Tabela 4 – Pergunta: Qual a sua motivação para desenvolver vídeos jogos

Na Tabela 4, são representados os dados obtidos à pergunta "Qual a sua motivação para desenvolver vídeos jogos", por proporção em percentagem (P) e por intervalo de confiança (IC).

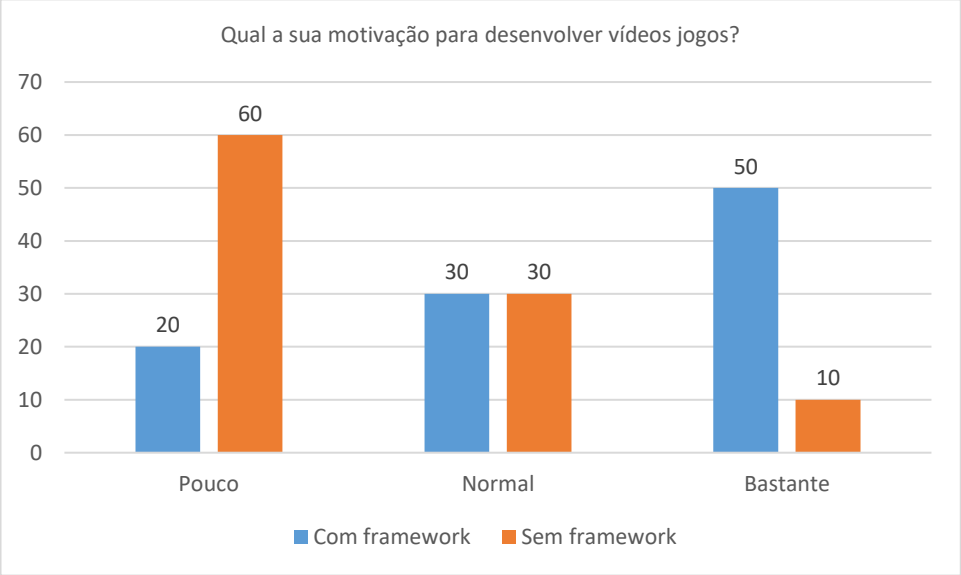


Figura 64 – Gráfico para pergunta: Qual a sua motivação para desenvolver vídeos jogos

Analisando o gráfico da Figura 64, pode-se concluir que existe vantagem em usar a *framework* com os dados obtidos. O número de utilizadores que responderam "Pouco" sem utilizar a *framework* desceu para um terço depois de usarem a *framework*. Enquanto que os utilizadores que responderam "Normal" se mantiveram tanto com a *framework* como sem. Por fim, e sem margem de dúvidas existiu um aumento significativo na motivação para o desenvolvimento de vídeos jogos de cinco vezes, para quem usou a *framework*, comparado com os utilizadores que não usaram a *framework*.

Conclui-se, assim, que 60% dos utilizadores que não usam a *framework* têm pouca motivação para desenvolver vídeos jogos, e que apenas, 20% de quem utiliza a *framework* tem pouca motivação. Existindo assim uma diferença de 80% com *framework* contra 40% sem *framework* de ter alguma motivação para desenvolver vídeos jogos.

Considera que adquiriu novos conhecimentos na programação?						
	Pouco		Normal		Bastante	
	P	IC	P	IC	P	IC
Com framework	0%	0%	20%	7,60% - 32,39%	80%	67,60% - 92,39%
Sem framework	30%	15,79% - 44,20%	40%	24,81% - 55,18%	30%	15,79% - 44,20%

Tabela 5 – Pergunta: Considera que adquiriu novo conhecimentos na programação

Na Tabela 5, são representados os dados obtidos à pergunta "Considera que adquiriu novos conhecimentos na programação?", por proporção em percentagem (P) e por intervalo de confiança (IC).

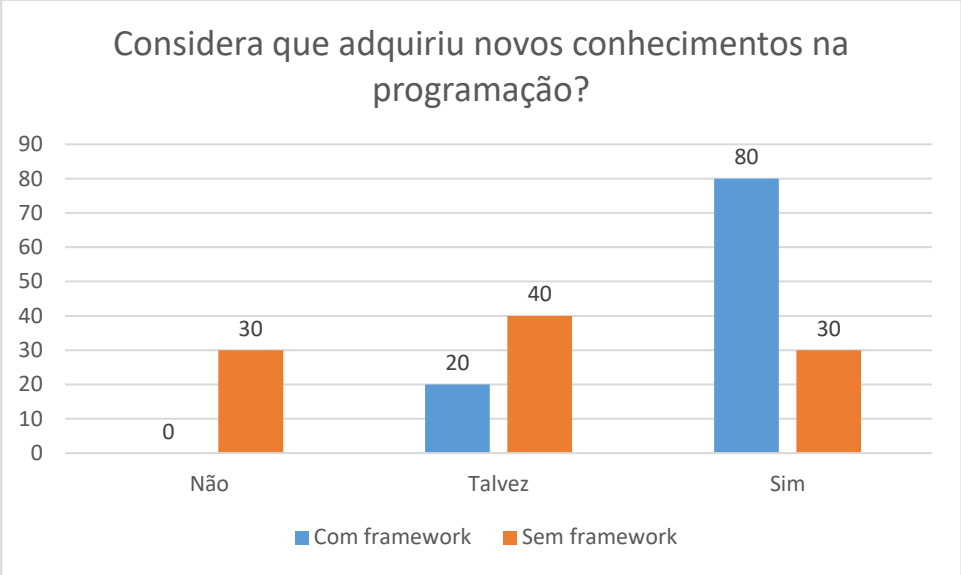


Figura 65 – Gráfico da pergunta: "Considera que adquiriu novos conhecimentos na programação?"

Os dados obtidos para a pergunta "Considera que adquiriu novos conhecimentos na programação", mostram que, Figura 65, os utilizadores quando não utilizam a *framework* aprendem muito menos dos que utilizam a *framework*. Também se verifica que existem zero utilizadores que não aprenderam nada ao utilizar a *framework*, e que os resultados de "Talvez" foi metade comparado com que não utiliza a *framework*, enquanto que a maior diferença é na resposta com "Sim" de 80% com *framework* contra 30% sem *framework*.

Pode-se assim concluir que, para quem utiliza a *framework,* existe 100% de probabilidade de aprender alguma coisa, ou seja, de responder "Talvez" ou "Sim" em contrapartida com que não usa a *framework* que é de 70%.

7 Conclusão

Neste último capítulo, serão descritas as conclusões finais do projeto, sendo identificados os objetivos realizados, limitações e possíveis trabalhos futuros.

Sobre a primeira parte, pode-se concluir que ficou em falta descrever as *frameworks* já existentes e a descrição do modelo de dados.

Tendo em conta que o objetivo desta dissertação era a implementação de uma *framework* denominada "3D OpenWorld Maker" para possibilitar a criação de jogos *Massive Multiplayer Online* (MMO) de uma forma rápida e com baixo custo de produção, foram criadas várias funcionalidades que podem ser reutilizadas em diversos projetos utilizando o *Unreal Engine* e algumas funcionalidades que tiveram que ser implementadas com outro serviço. Este serviço é o *Login Server* que funciona como um servidor de autenticação, mas também como servidor principal onde outros servidores como, por exemplo, os servidores dedicados no *Unreal Engine*.

Em relação a alguns pontos principais ao longo do desenvolvimento, análise de *games engines*, *Login Server* e depois a *framework* "*3D openWorld Maker*", pude tirar algumas conclusões que serão de seguida apresentadas.

O levantamento do estado da arte mostrou que não existem *frameworks* a facilitar a utilização das mesmas funcionalidades no desenvolvimento de jogos *Online*, sendo que as que existem são apenas *kits* com exemplos *demo* de algumas funcionalidades para jogos específicos sendo, assim, impossível utilizar em outros jogos com género diferente.

Também foi possível comparar os diversos *game engines* que existem no mercado e escolher o que melhor respeitava os critérios que se procuravam para implementar a *framework*. Também foi possível obter uma explicação das diferenças entre o tipo de protocolos nos jogos online, o tipo de arquitetura que eles podem ter quando existe um servidor dedicado ou quando um jogador é cliente e servidor ao mesmo tempo e, claro, os problemas que existem em cada arquitetura.

Com análise aos *games engines* foi concluído que o *Unreal Engine* tinha todas as ferramentas necessárias para se poder implementar a *framework*, desde as suas funcionalidades de origem até a sua longa história na produção de jogos e também do seu *game engine* com uma larga experiência no mercado. O Unreal Engine mostrou-se, assim, um *game engine* de confiança, maduro e em constante desenvolvimento e melhoria.

Sobre a implementação do *Login Server*, foram concluídos todos os seus requisitos. Com o *Login Server*, é possível assim gerir o acesso a qualquer jogo, através de um registo e seguida o *login*, com isto é possível então controlar quantas pessoas cada servidor dedicado possui e validar se cada cliente está mesmo autenticado e não é um *hacker* a tentar entrar num servidor dedicado sem acesso.

Como foi falado anteriormente, existe um mecanismo de segurança que passa por usar *token* que é criado no *Login Server* quando um cliente faz o seu *login* e, quando este tenta entrar num servidor

dedicado, o servidor dedicado ppr sua vez tem a responsabilidade de perguntar ao *Login Server* se o cliente que esta a tentar ligar-se a si é válido, passando assim o *token* para o *Login Server* o validar e responder ao servidor dedicado que vai ter a responsabilidade de aceitar ou rejeitar o pedido do cliente que quer conectar-se ao jogo que esta a correr no servidor dedicado. Visto isto, existe agora um serviço, *Login Server*, onde os dados dos utilizadores são persistidos numa base de dados, utilizando o *dapper ORM*, para futura utilização em um ou vários jogos, dependendo do objetivo da empresa ou jovem que utilizará a *framework*, o que permite uma grande facilidade de utilização.

Por fim, foram cumpridos todos os requisitos inicialmente definidos e a obteve-se a conclusão completa da *framework* "*3D openWorld Maker*", assim como também foram realizadas as grandezas, as hipóteses e as metodologias que permitiram avaliar se a solução desenvolvida atinge os objetivos definidos.

Sobre as limitações e trabalho futuro, a *framework* sendo construída sobre *Unreal Engine* tem a limitação de, em caso de haver necessidade de alterar o *game engine*, será necessário redefinir as funcionalidades da *framework*.

A nível do serviço de *Login Server*, tendo sido este desenvolvido de raiz, pode ser facilmente modificado com algumas alterações a nível do código.

Existem alguns aspetos a ser melhorados em algumas das funcionalidades, como passar algumas partes das funções em *blueprint* para C++, estudar outras soluções para buscar o objeto `APlayerState`. Também existem os problemas conhecidos como passar uma lista com a informação dos servidores dedicados em vez de apenas um servidor dedicado.

Outo problema, é sobre o endereço de rede que cada servidor dedicado tem que enviar, que deve ficar localizado num ficheiro de configuração com o endereço e porta pública. Como também o mecanismo na criação do *token* de autenticação, este pode passar por no futuro utilizar-se um sistema de autenticação mais robusto como por exemplo o *oauth2*[5].

[5] OAuth é um código fonte aberto para autorização

8 Referências

[1] UnrealEngine, “final-fantasy-vii-remake-built-with-ue4,” Janeiro 2016. [Online]. Available: https://www.unrealengine.com/blog/final-fantasy-vii-remake-built-with-ue4.

[2] UnrealEngine, “what-is-unreal-engine-4,” Janeiro 2016. [Online]. Available: https://www.unrealengine.com/what-is-unreal-engine-4.

[3] NaughtyDog, “uncharted3,” Janeiro 2016. [Online]. Available: http://www.naughtydog.com/stats/uncharted3/.

[4] S.Nicola, “Value Analytics,” Janeiro 2016. [Online]. Available: https://moodle.isep.ipp.pt/pluginfile.php/91647/mod_resource/content/2/An%C3%A1lise_Valor_Aula1.pdf.

[5] S.Nicola, “Analise de valor negocio,” Janeiro 2016. [Online]. Available: https://moodle.isep.ipp.pt/pluginfile.php/92054/mod_resource/content/3/An%C3%A1lise_Valor_Aula3.pdf.

[6] S.Nicola, “Value Network,” Janeiro 2016. [Online]. Available: https://moodle.isep.ipp.pt/pluginfile.php/91648/mod_resource/content/3/An%C3%A1lise_Valor_Aula2.pdf.

[7] wikipedia, “game engine,” Janeiro 2016. [Online]. Available: https://en.wikipedia.org/wiki/Game_engine.

[8] gamedevelopment, “networking,” Janeiro 2016. [Online]. Available: http://gamedevelopment.tutsplus.com/tutorials/building-a-peer-to-peer-multiplayer-networked-game--gamedev-10074.

[9] Gafferongames, “udp vs tcp,” Janeiro 2016. [Online]. Available: http://gafferongames.com/networking-for-game-programmers/udp-vs-tcp/.

[10] Unity, “unity,” Janeiro 2016. [Online]. Available: https://unity3d.com/pt.

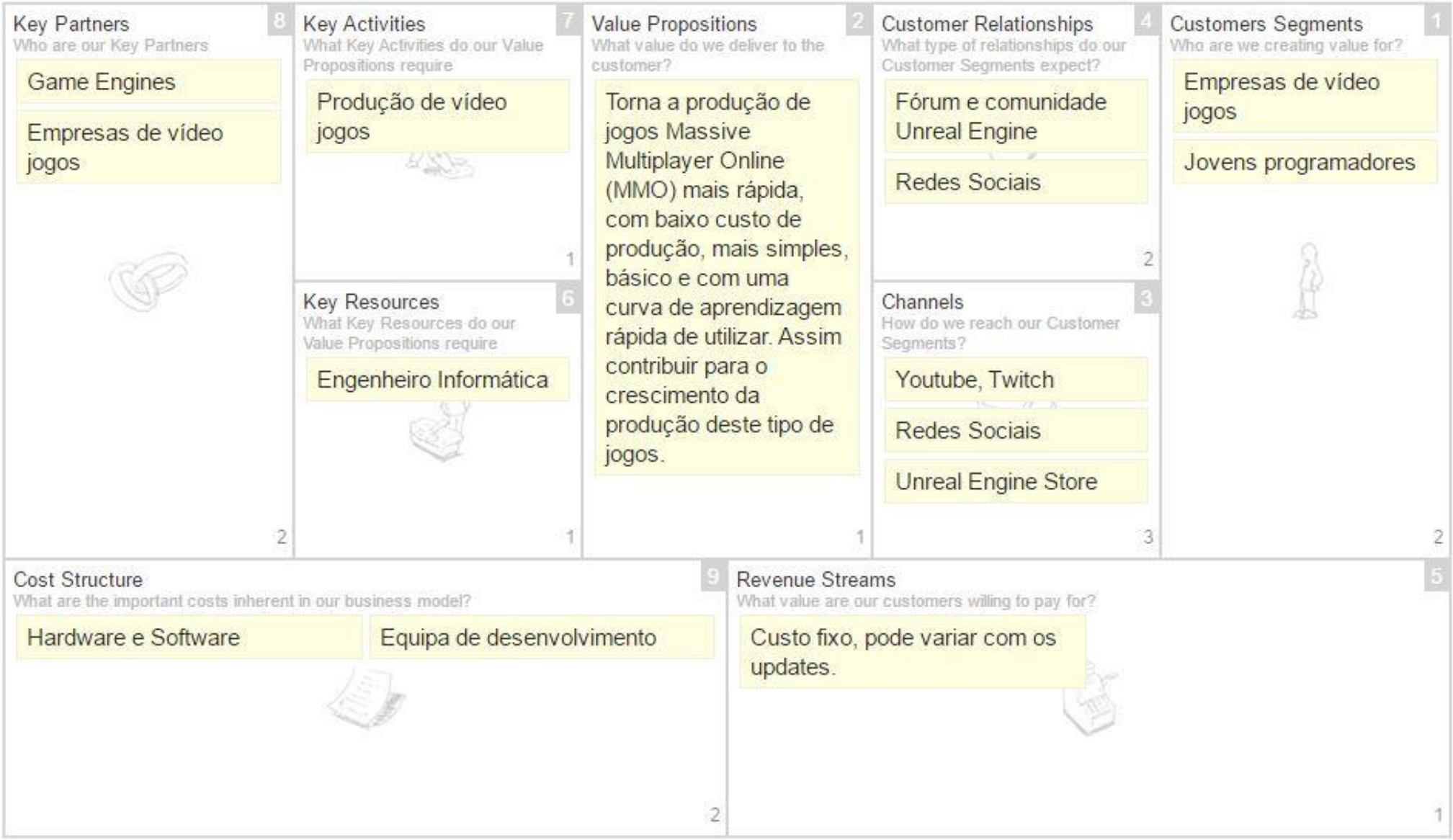
[11] cryengine, “cryengine,” Janeiro 2016. [Online]. Available: http://cryengine.com/.

[12] Amazon, “lumberyard,” Janeiro 2016. [Online]. Available: https://aws.amazon.com/pt/lumberyard/.

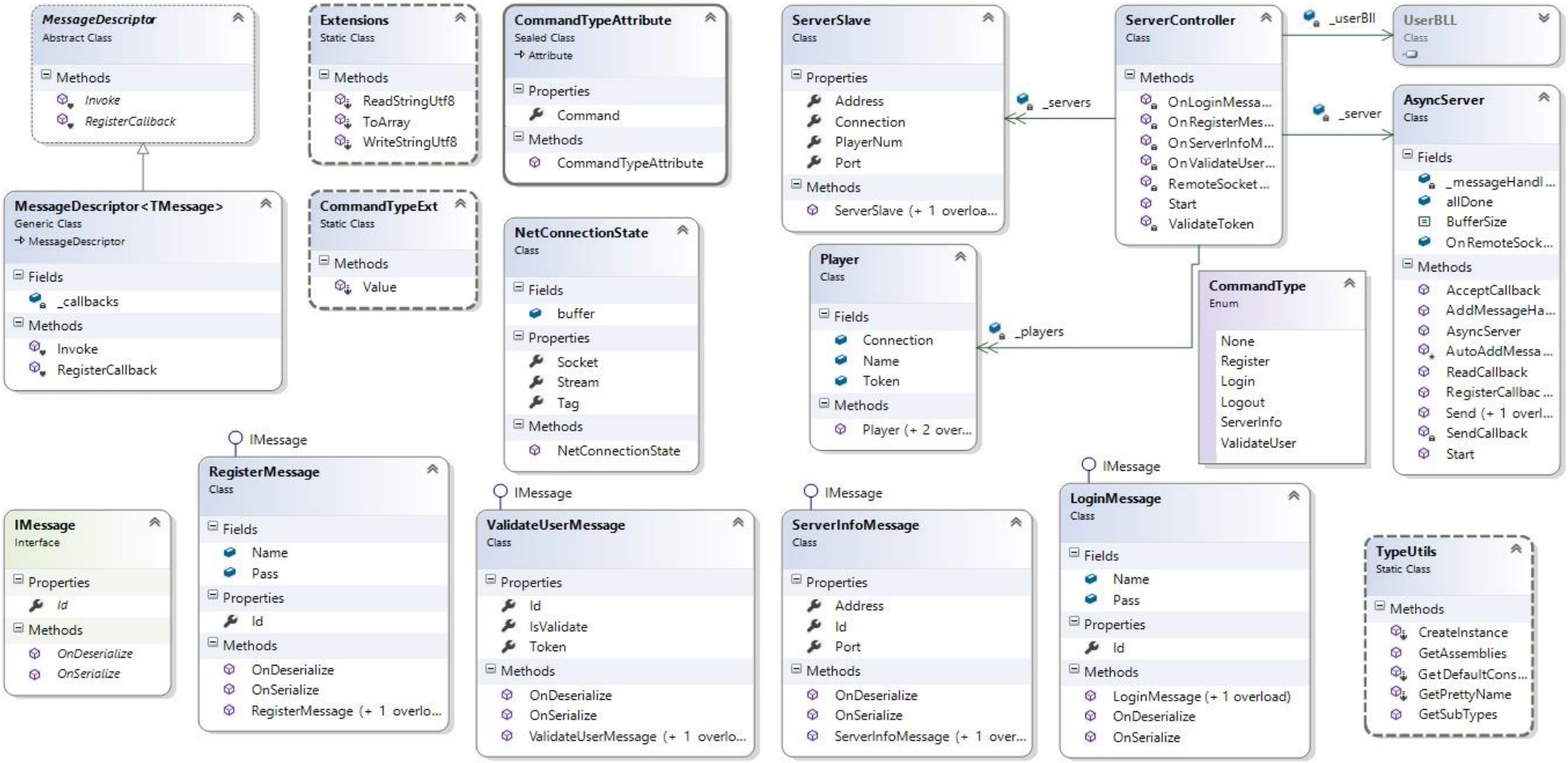
[13] digitaltutors, “unity-udk-cryengine-game-engine-choose,” Janeiro 2016. [Online]. Available: http://blog.digitaltutors.com/unity-udk-cryengine-game-engine-choose/.

9 Anexos

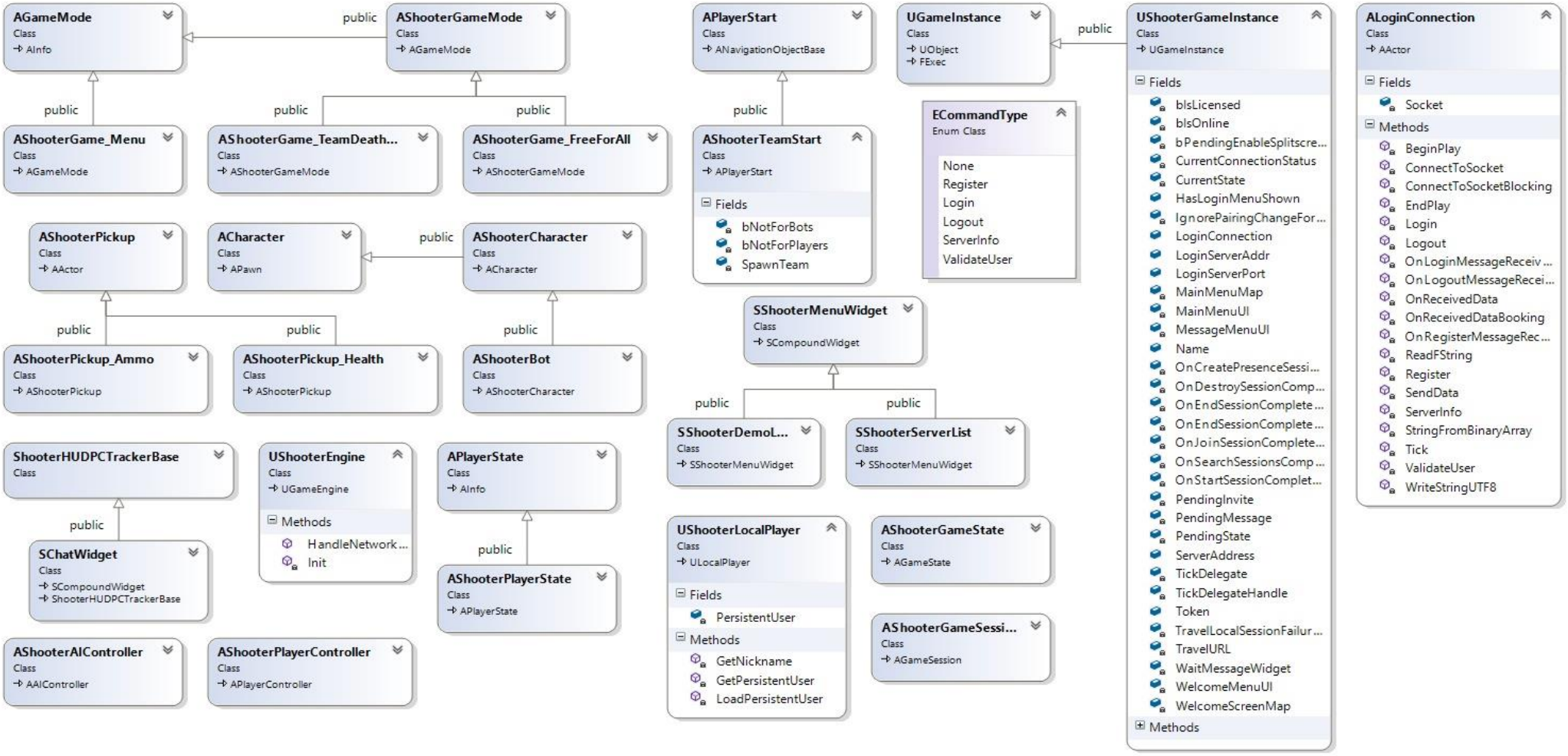
9.1 Modelo de negócio canvas



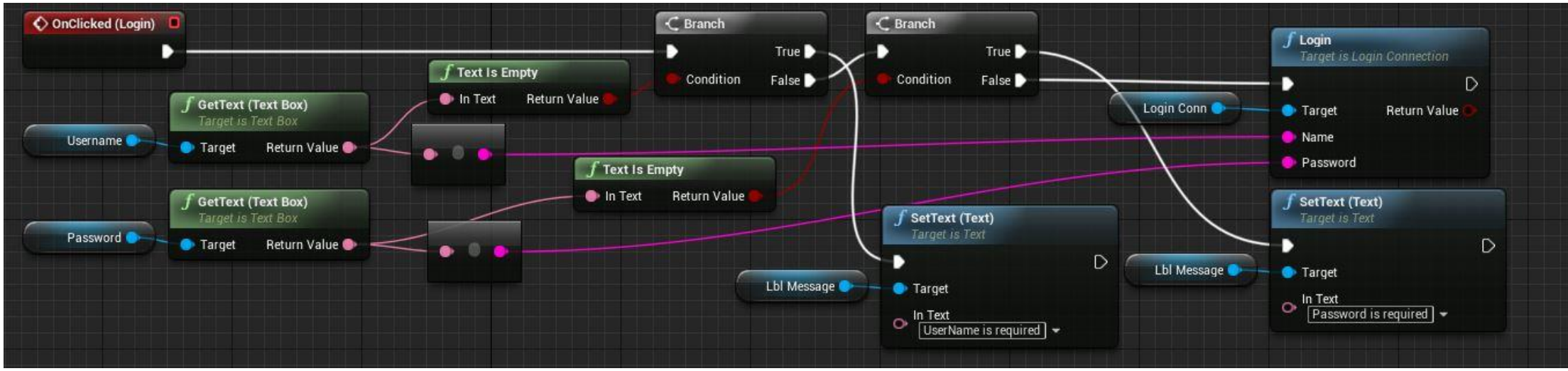
9.2 Diagrama de classes LoginServer

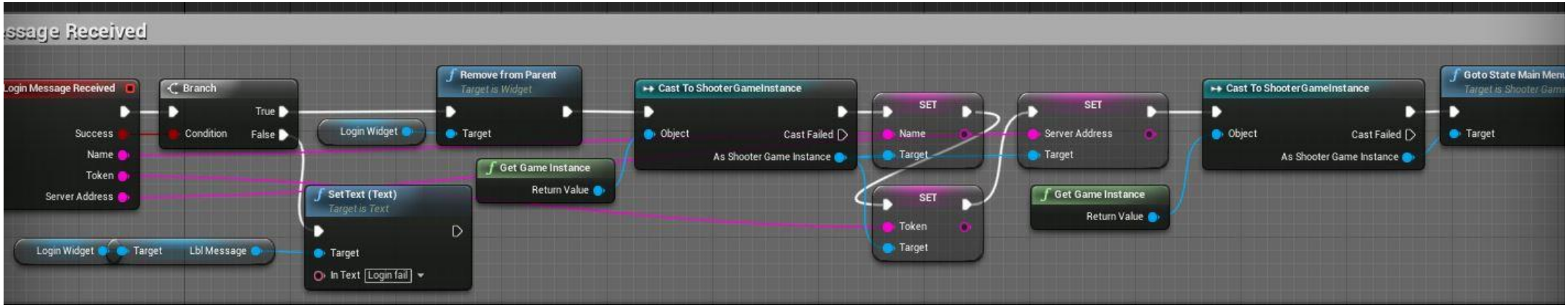


9.3 Diagrama de classes ShooterGame

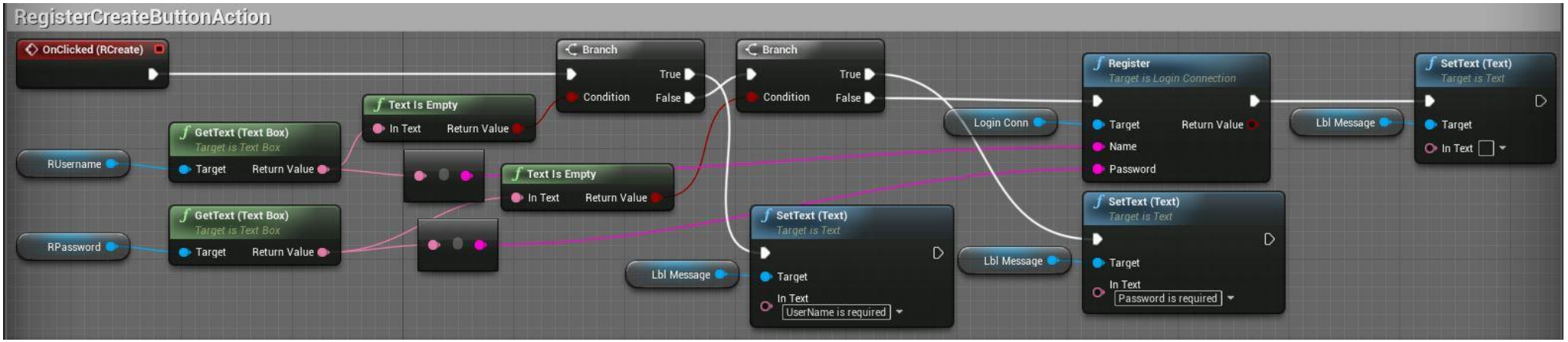


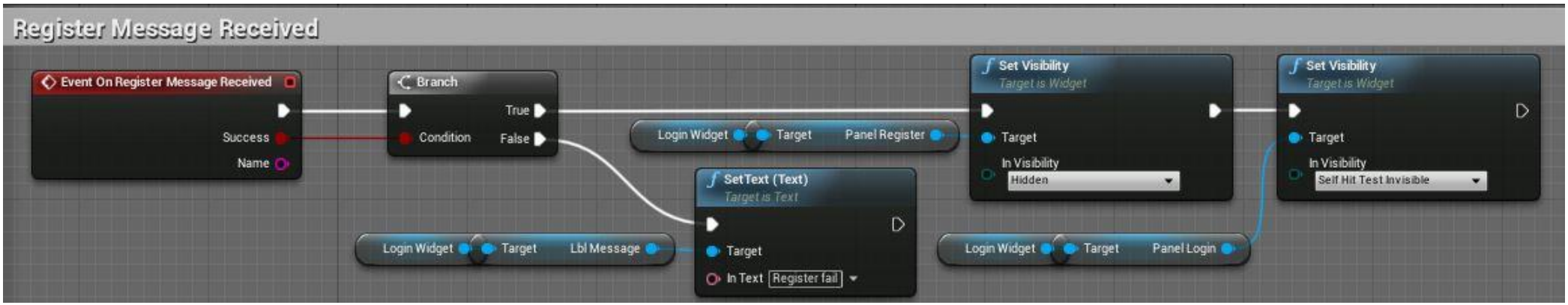
9.4 Login Blueprints





9.5 Register Blueprints





9.6 Evento Tick PlayerPawn ou BotPawn