Distributed Servers Sub-System

Contents

I. INTRODUCTION

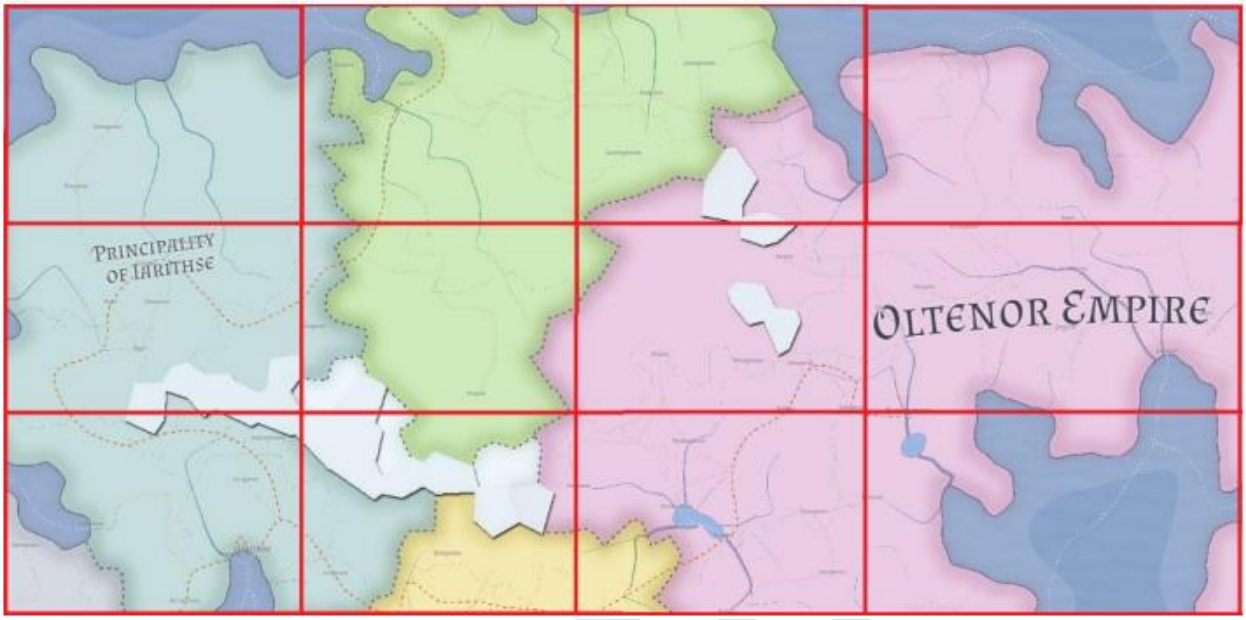


Figure 1:Each red box represents a zone that has many shards running on multiple zone hosts.

Distributed servers' sub-system is set of plugins that helps in scaling your game players to the order of thousands by providing horizontally scalable approach, benefiting from latest cloud technology. A set of containers/Virtual machines are dynamically instantiated based on the requirements and referred as zone hosts. Each zone host is hosting multiple shards of a zone, where zone is an Unreal engine map or part of a map and shard is an instance of that zone. The dynamically instantiated Zone hosts and shards are registered and can communicate between each other synchronously and asynchronously.

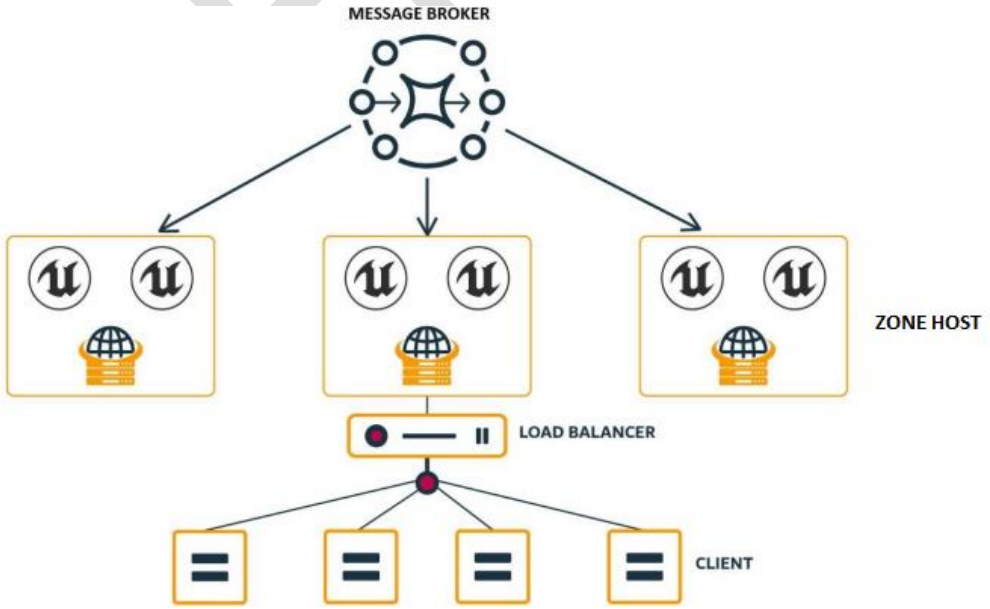


Figure 2:An Abstracted architecture.

II. DSS SUBSYSTEMS

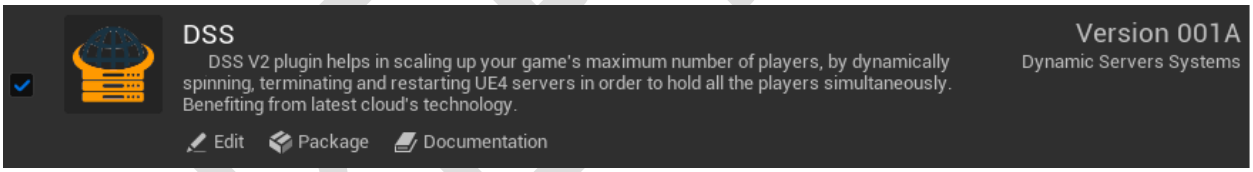
DSS is made up of seven subsystems in total, all the sub-systems work together to provide a complete scalable MMO architecture without compromising the player's experience. All the sub-systems are cross platform and supports **Linux x86_64**, **Windows x86_64** and **Linux ARM64** hosting.

a. DSS SignalR

In order for the client to communicate with server, DSS uses a well-known RPC framework called SignalR. This framework is based on HTTP1.1. DSS SignalR is the third-party SignalR wrapper for unreal engine, it is used by some of the sub-systems. Maintained independently to simplify the security patching.

b. Dynamic servers' sub-system

Dynamic servers' sub-system is the core of DSS, its responsible for scaling the whole infrastructure to dynamically adapt to the load. For optimal performance, DSS distributes players based on the host performance, where each shard is assigned a single core, thus providing a cost-efficient hosting with an optimal performance. Zone hosts are scaled in and out as per the need.



i. Unreal Engine Servers configuration

In order to register your unreal engine server, you have to configure the `Path` and `Name`. Path represents the absolute path of the unreal engine and name is the full name and extension.

```
{
  "Path": "Server Path",
  "LogsPath": "Logs Path",
  "Name": "UE Server Executable Name",
  "Port": 7770,
  "EnableLogs": true,
  "DynamicProcessorAffinity": true,
  "Levels": [
    {
      "Name": "Level Name and Path",
      "ServerLimit": 50,
      "MinimumInstances": 1,
      "Args": []
    }
  ],
  "Timeout": 20000,
  "DungeonTimeout": 10000
}
```

`LogsPath` Can be configured to reflect where your unreal engine saves logs in development builds, in order to fetch those logs through apis, without the need to access the actual zone host.

`Port` Can be used to configure from which port DSS will start spinning unreal engine servers. Priore to port assigning, DSS test if the port is available. Additionally, all the ports of previously scaled out unreal engine servers will be used if it is available.

`EnableLogs` If set to true, it will prompt unreal engine server to save logs, setting it to false will disable that feature.

`DynamicProcessorAffinity` If set to true, DSS will assign a single CPU thread for each unreal engine server depending on the thread's availability. In most cases, it will increase the performance if enabled due to the single thread architecture of unreal engine server. Setting it to false will leave the thread assignation to the OS task schedular which means your server might jump between multiple threads.

`Levels` It is a list of all the levels inside your game. Levels could be either dungeons or normal maps and can be assigned as public or private (to be discussed further). Each level has four main args. `Name` represents the name and full path of the level such as `"/Game/ThirdPersonCPP/Maps/Map1"`. `ServerLimit` is the maximum number of players per server instance after which the unreal engine server will no longer accept new players. `MinimumInstances` guarantee minimum number shards of a certain zone inside a zone host. `Args` is an array of string arguments that can be passed to the unreal engine server. It can be accessed from blueprint using `"Get Command Line and Parse Command line"` or through C++ `"FParse::Value(FCommandLine::Get(), TEXT("DSSPort"), DSSPort)"`.

`Timeout` is the cooldown time of normal maps in milli seconds. The server will be scaled out if it stayed empty for the given time period. Similarly, `DungeonTimeout` is for the dungeons.

ii. Connecting to DSS

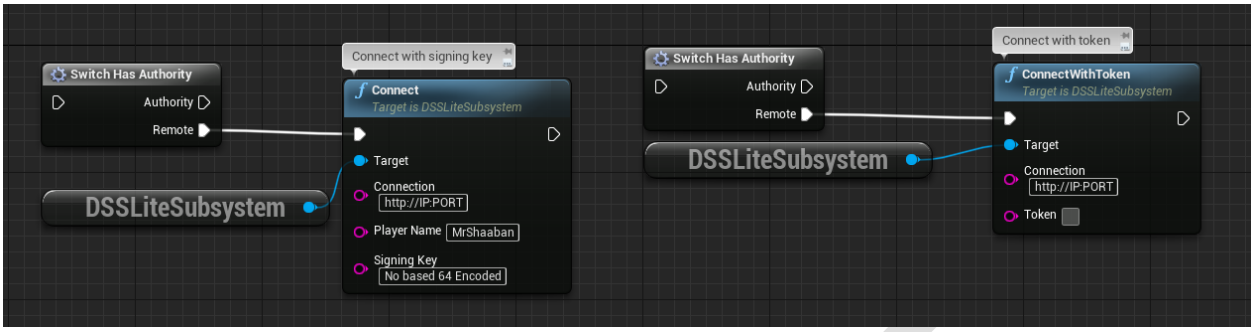


Figure 3: Connect to DSS from blueprint

There are two ways to connect to DSS from the client side. Both are supported from blueprint and C++. `Connect` will allow to connect using a signing key which means the token is going to be created locally. **Connection** represents the DSS Service IP and Port, and signing key should be the same as the one configured in DSS appsettings.json `"DSS:SigningKey"`. It is ideal for testing purposes only; Unauthorized access error might happen in case of the time zone difference between the client and the zone server. `ConnectWithToken` Will allow you to connect to the server using a signed JWT token. The signing key should be the same as the one configured in DSS appsettings.json `"DSS:SigningKey"`. JWT token should contain two claims, first is the `"name"` and it should contain the player's name, the other one is `"role"` and it should be always set to `"client"`. Additionally, you have to include the expiration time `"exp"` as a Unix timestamp. You can refer to this site in order to test your token https://jwt.io/. The token has to be signed using **HS256** algorithm. Provided a token example.

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ
9.eyJyb2xlIjoiQ2xpZW50IiwibmFtZSI6I
k1yU2hhYWJhbiIsImV4cCI6MTcyMzc5
NDY3Mn0.lTr4xFoVp4dnNvwrWvZGF
13BOeylyJ6JnA27_s9iOgs
```

iii. Traveling between servers

In order to travel between servers, DSS provide a handy API that can be called from C++ or Blueprint.

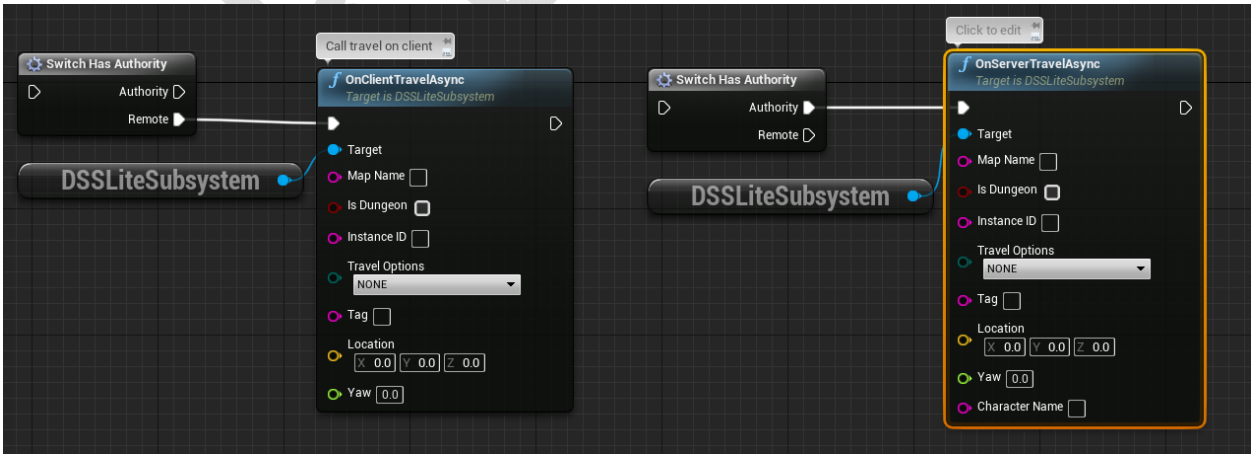


Figure 4: Traveling from blueprint

`OnServerTravelAsync` and `OnClientTravelAsync` can be similarly used from C++.

Both functions are asynchronous, which means you will get a callback with the response.

OnServerTravelAsync is callable from server only, and the server is going to get the callback then forward it to the player. On the other hand, OnClientTravelAsync can be called on client only and the client is going to get the callback directly from DSS Server.

- MapName: MapName and full path of where to travel
- IsDungeon: Self explanatory
- InstanceID: Dungeon InstanceID to connect to in case IsDungeon is true
- TravelOptions: where to spawn player (None,Coordinate,Tag)
- Tag: Player start tag if Travel Options set to Tag
- Location: where to spawn the player if Coordinates TravelOptions selected
- Yaw: Rotation around Z-Axis Coordinates TravelOptions selected

NB: Player is not bounded to one zone host, DSS will pick the best shard for the player across all hosted zones.

NB: Players connected to multiple zone host can still team up on the same dungeon if same dungeon ID is used.

iv. Portals

There are two types of portals in DSS. They act similarly in terms of performance, except one is visible and one is hidden. Additionally, teleport NPC is supported. The hidden portals are used to split one big world into multiple shards.

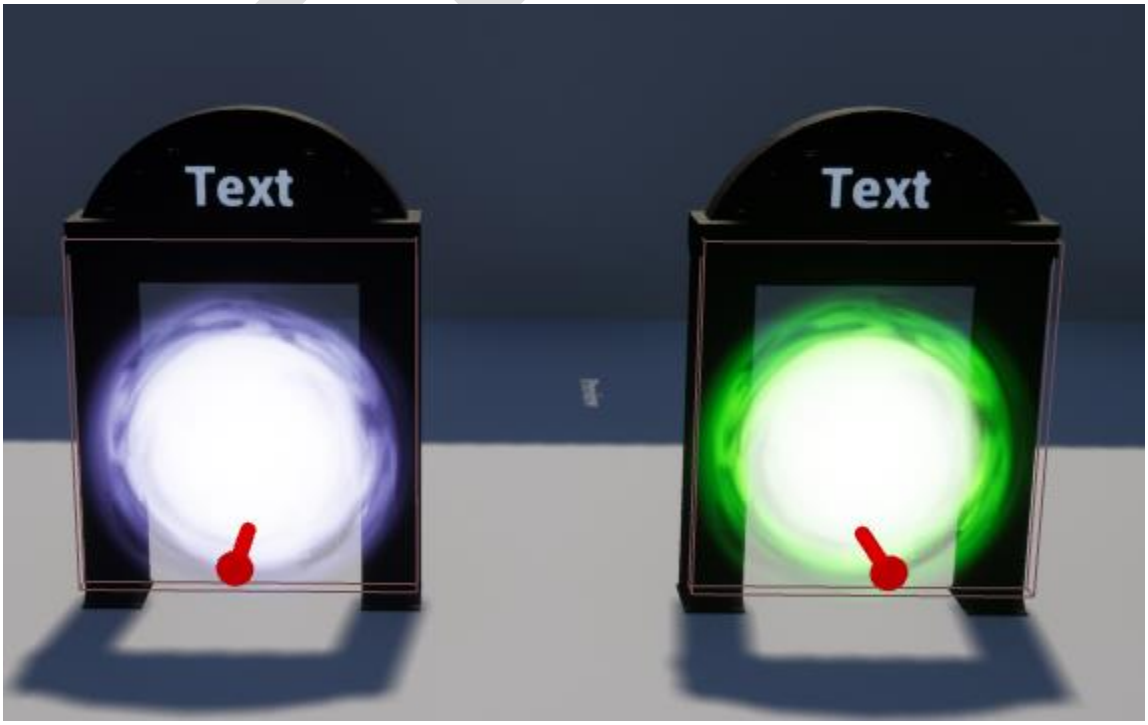


Figure 5: Visible portals

Figure 6:Invisible portal

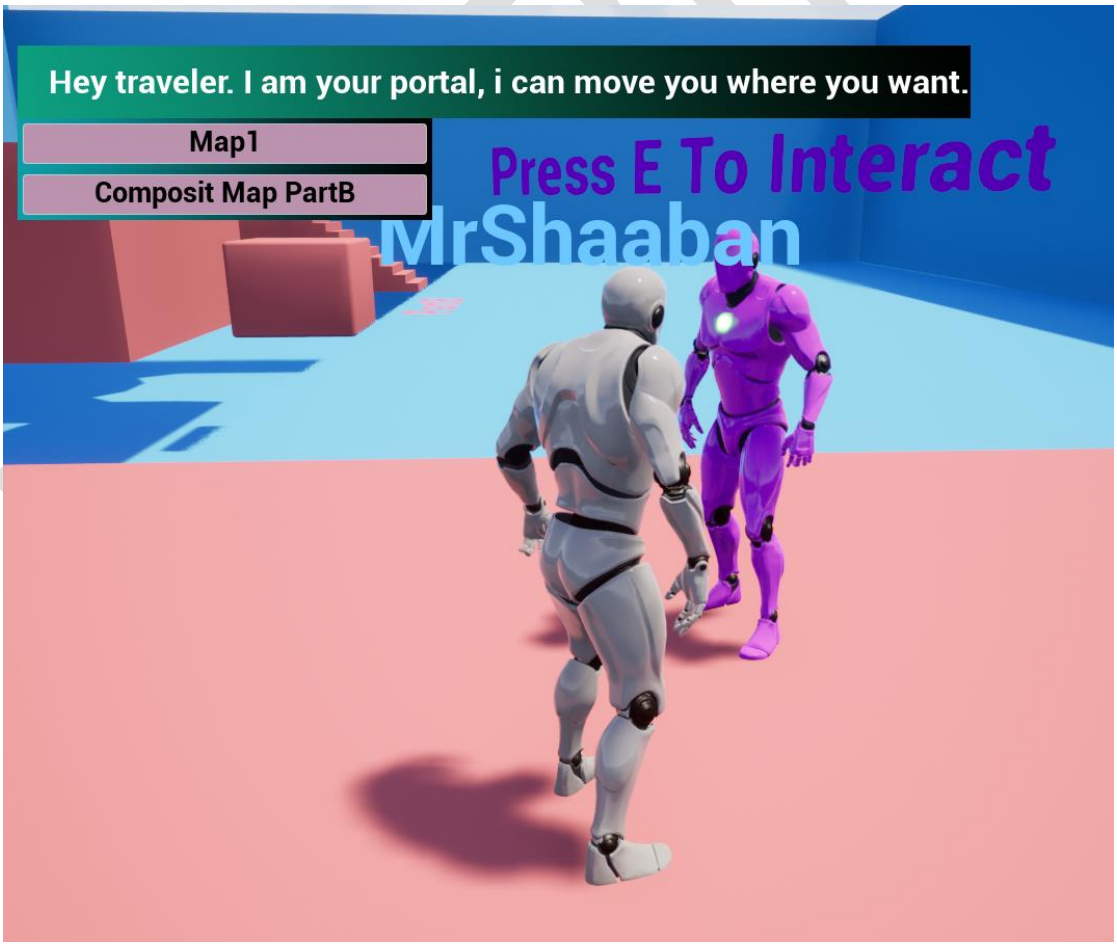


Figure 7: Teleport NPC

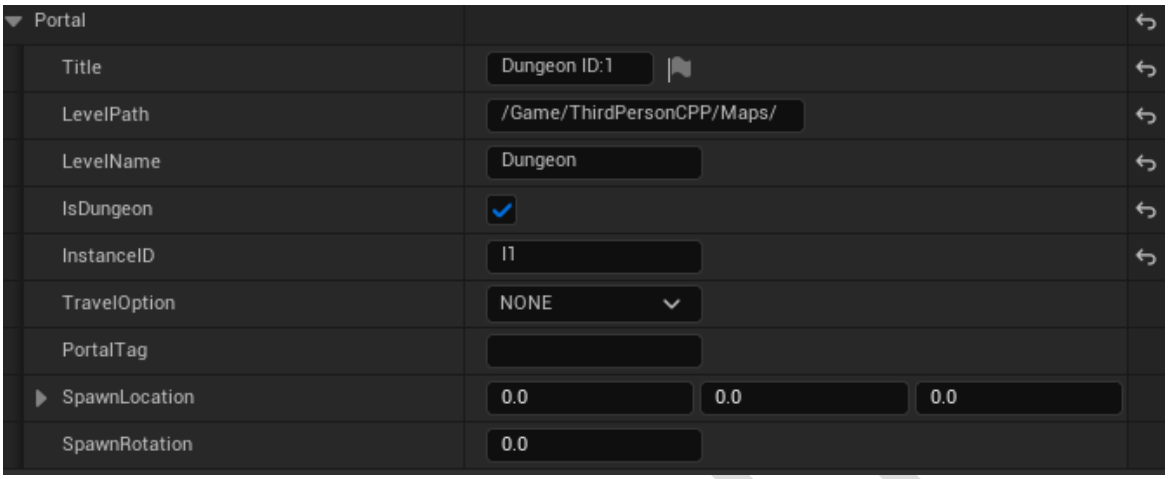


Figure 8: Portal's configuration

Both hidden and visible portals share the same configuration, and it is essentially a list of the arguments used by the `OnServerTravelAsync` and `OnClientTravelAsync.`

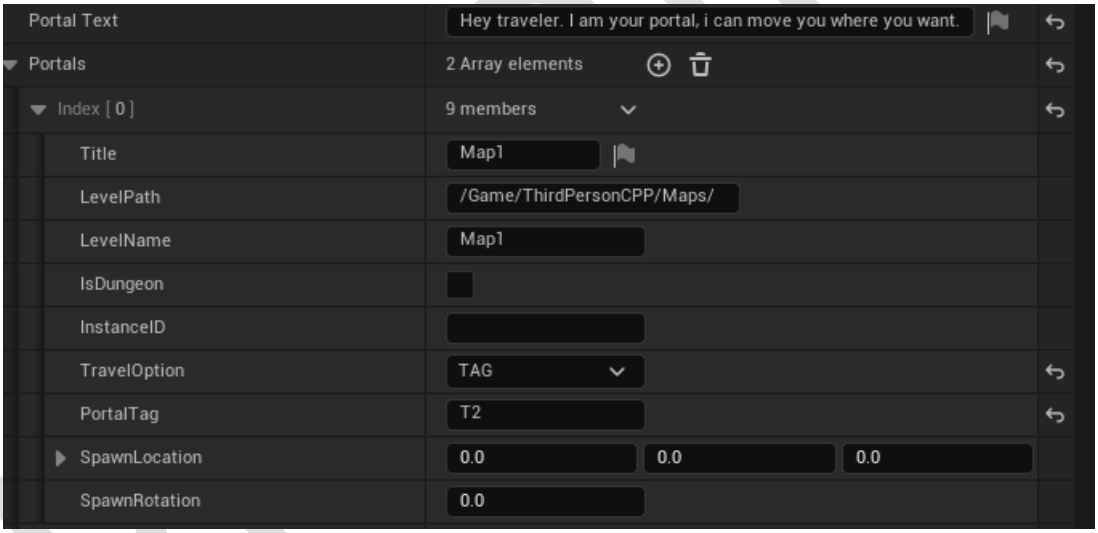


Figure 9: Teleport NPC Configuration

Teleport NPC has two additional fields compared to normal portal. **Portal Text,** essentially used to display some text when the player interacts with the NPC and **Title** which is basically the text displayed on the button.

v. Delegates

Since all the API calls are asynchronous, the response is going to be received through a delegate. DSS provides a set of delegates that help customize the game logic.

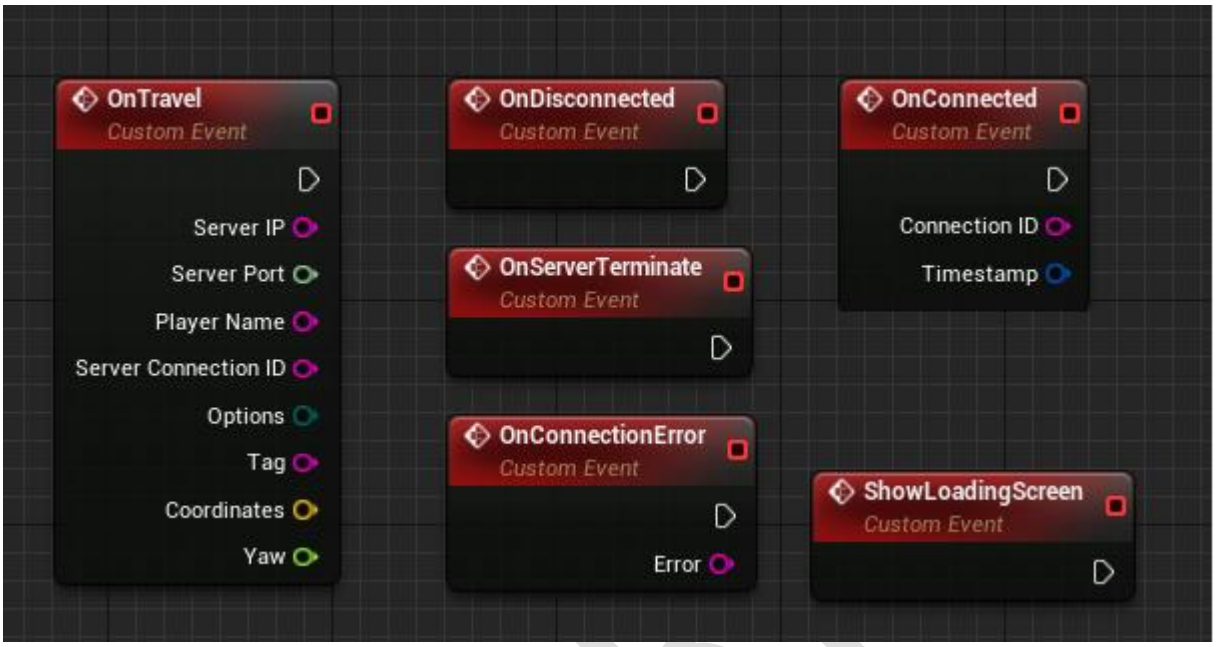


Figure 10: Delegates

- **OnTravel:** called after server respond to travel request. It will fire on server or on client if `OnServerTravelAsync` or `OnClientTravelAsync` are called respectively.

```
1. ServerIP: Virtual machine Ip where to connect
2. ServerPort: UE4 Server port
3. PlayerName: Character name that requested the travel
4. ServerConnectionID: Unique UE server ID could be used for servers' communication
5. Coordinates: corrisponds to the spwaning location
```

- **OnDisconnected**: Called after UE Server/Client disconnects from DSS Server
- **OnServerTerminate**: Called by DSS Server on UE Server to request server shutdown. Server will shutdown after 30 second of the request, but will stop receiving new connections immediately
- **OnConnectionError**: Called when connection attempt failed with Error message
- **OnConnected**: Called on the Client and UE Server after connecting to DSS Server

```
1. ConnectionID: Unique ID for UE servers and clients
2. Timestamp: Time synchronization across all UE Servers
```

- **ShowLoadingScreen:** Called on Travel to show loading screen

vi. Redis server

DSS services uses Redis server as a shared memory cache, in order to share important data between all the zone hosts. Data can be used by developer in order to query and understand what is happening. All the data in Redis server are prefixed with a namespace to be discussed further.

```
{
    "IsEnabled": true,
    "Host": "",
    "Port": 15517,
    "EnableSSl": false,
    "Password": ""
  }
```

`IsEnabled` if set to true, Redis server will be used

`Host` Server Ip or domain name

`Port` Server Port typically it is 15517

`EnableSSl` Should be set to true if TLS is enabled on server

`Password` Redis server password

SET	PvE:DSS:Clients:Players	No limit	264 B
STRING	PvE:DSS:DSSServer:Instance:93540b5e-52c3-4cf3-af37-a1db287128c5	11 s	312 B
SET	PvE:DSS:UEServers:Type:Dungeons	No limit	256 B
SET	PvE:DSS:DSSServers:Instances	No limit	304 B
STRING	PvE:DSS:DSSServer:Stat:93540b5e-52c3-4cf3-af37-a1db287128c5	28 s	200 B
SET	PvE:DSS:UEServers:DSSInstance:93540b5e-52c3-4cf3-af37-a1db287128c5	No limit	384 B
SORTED SET	PvE:DSS:UEServers:Level:Accepting:/Game/ThirdPersonCPP/Maps/Map1	No limit	205 B
SET	PvE:DSS:UEServers:Type:Maps	No limit	288 B
SET	PvE:DSS:UEServers:Level:/Game/ThirdPersonCPP/Maps/Dungeon	No limit	280 B
STRING	PvE:DSS:UEServer:I2	2 min	384 B
SET	PvE:DSS:UEServers:Level:/Game/ThirdPersonCPP/Maps/Map1	No limit	320 B
STRING	PvE:DSS:Client:Player:MrShaaban	No limit	272 B
STRING	PvE:DSS:UEServer:d1dfb248-f392-4a0e-a207-40b2049e2b3b	2 min	424 B
STRING	PvE:DSS:Global:Config:AcceptConnection	No limit	96 B
SORTED SET	PvE:DSS:DSSServer:Stat	No limit	136 B

Figure 11: Redis server cache

- **DSS:Clients:Players** : SET of all the online players names
- **DSS:UESServers:Type:Dungeon** : SET of all the dungeon shard IDs
- **DSS:UESServers:Type:Maps** : SET of all the normal shards IDs
- **DSS:DSSServers:Instances**: SET of all DSS servers instance IDs (zone host)
- **DSS:UEServers:DSSInstance:DSSInstanceId**: SET of all shards in a certain DSS zone host
- **DSS:UEServers:Level:LevelName:** SET of all shards of certain level name

DSS Retain all normal shards of each Level name in a sorted sets based on performance metrics to be discussed. **DSS:UEServers:Levels:Accepting:LevelName**

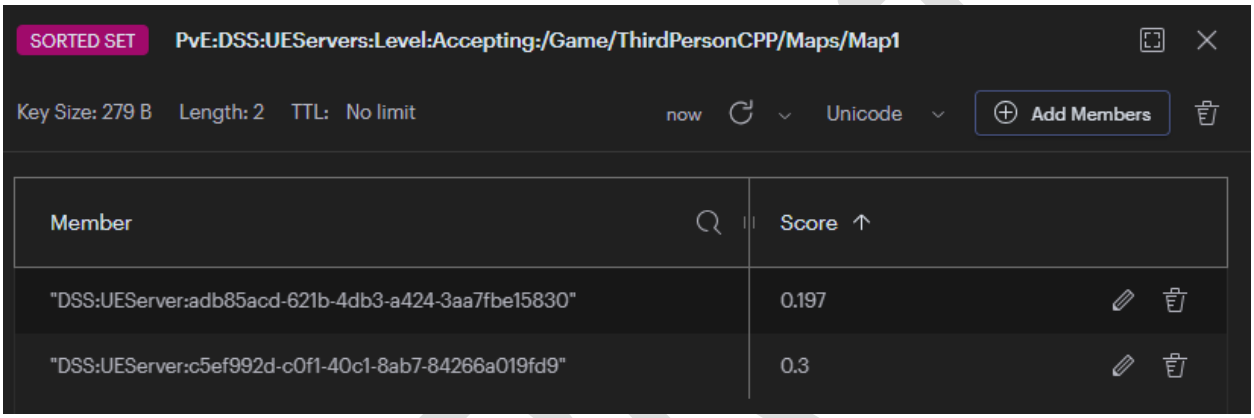


Figure 12: An example of shards ranking sorted set

Similarly, all DSS zone hosts are ranked inside a sorted set. **PvE:DSS:DSSServer:Stat**

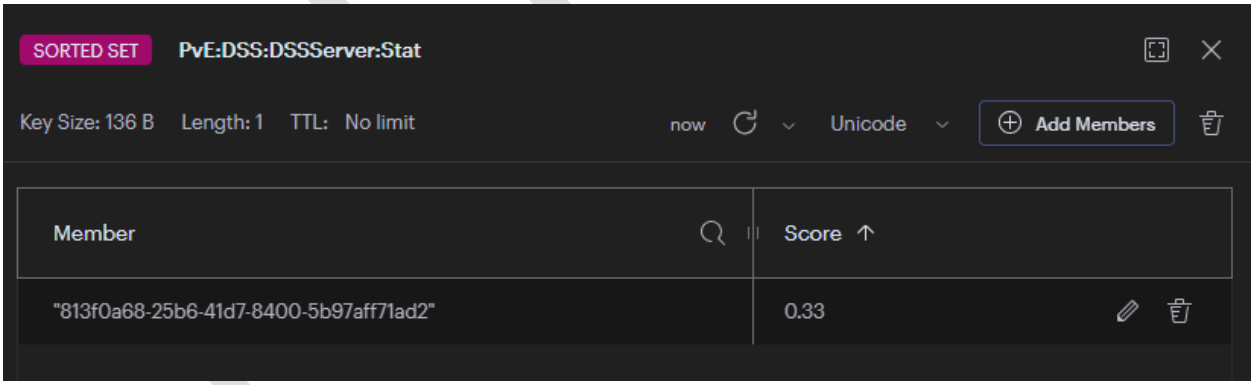


Figure 13:An example of zone hosts ranking in a sorted set

Each zone host has a key-value pair in Redis server, **DSS:DSSServer:Instance:DSSInstanceID.**

```
{
  "DSSInstanceID": "813f0a68-25...",
  "DSSServerPrivateIP": "127.0.0.1",
  "DSSServerPublicIP": "127.0.0.1",
  "ServerPort": 5000,
  "S2SPort": 5002,
  "IsAcceptingConnections": true
}
```

Each zone host has a key-value pair in Redis for server performance **PvE:DSS:DSSServer:Stat: DSSInstanceID.**

`ServersCount` Represents the number of shards in the zone host.

Both `CpuUsage` & `RamUsage` is in percentage.

```
{
  "PlayersCount": 0,
  "ServersCount": 3,
  "WorkerQueueCount": 0,
  "CpuUsage": 17,
  "RamUsage": 42
}
```

Each shard has a key-value pair in Redis **DSS:UEServer:ShardId**

```
{
  "InstanceID": "Shard Id",
  "DSSInstanceID": "Hosted Zone Id",
  "DSSServerPrivateIP": "127.0.0.1",
  "DSSServerPublicIP": "127.0.0.1",
  "LevelName": Level name and path",
  "ServerPort": 7772,
  "S2SPort": 5002,
  "IsDungeon": false,
  "IsPublic": true
}
```

vii. Shards ranking & cross server team-up

Based on a metric that can be defined by the client, all shards are kept tracked and ordered in Redis. This will allow DSS to teleport the player to the best performing shard in all the zone hosts. For instance, if player is connected to **zone host A** and attempt to travel to **Zone Z** it might be teleported to a **shard of Zone Z** running in **zone host B** if it is performing better than the other shards. In other words, player is going to be always teleported to the best shard of a certain zone.

```
{
    "IsEnabled": true,
    "CPUScore": 30,
    "RamScore": 70,
    "ClientsScore": 20,
    "ScoreMargin": 10
  }
```

The overall score of a shard is determined based on user configuration, and it is normalized to 100. `CPUScore` represent how much the CPU usage will affect the shard ranking, `RamScore` represent how much the RAM usage will affect the shard ranking and finally `ClientsScore` represent how much the total number of players in a shard divided by the max number of players will affect the shard ranking. `ScoreMargin` specify the margin in percentage of when to update the ranking of servers. For instance, in this case if shard score changed by more than 10% up or down, it will be updated in Redis server.

viii. Zone host ranking and load balancing

Based on a metric that can be defined by the client, all zone hosts are kept tracked and ordered in Redis. This will allow DSS to redirect the player to the best performing zone hosts on connect.

```
{
    "IsEnabled": true,
    "CPUScore": 30,
    "RamScore": 70,
    "MaxAttempts": 3,
    "ScoreMargin": 10,
    "MaxCpuUsage": 70,
    "MaxRamUsage": 70,
    "MaxPlayers": 1,
    "SingleShardPerCore":  true
  }
```

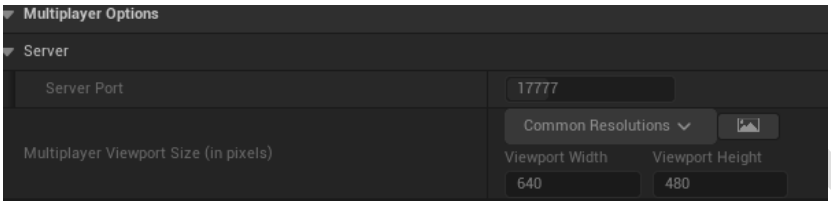
`MaxAttempts` represent how many time DSS can redirect to another host before definitely accepting the connection.

`CPUScore`, `RamScore`, and `ScoreMargin` are similar to the shards ranking.

`MaxCpuUsage`, `MaxRamUsage`, `MaxPlayers` and `SingleShardPerCore` will affect if the zone host will accept the player connection or attempt to redirect it to another zone host. In case of `MaxAttempts` reached, zone host will accept the connection anyway.

ix. Zone host configuration and Namespace

If `TestInEditor` is set to true, all the connection attempts will be redirected to the `EditorPort` that represents the UE server port that is launched by the editor and it can be determined from the editor preferences. Play Standalone Net mode should be selected to test in editor.




```
{
    "EditorPort": 17777,
    "Port": 5000,
    "SigningKey": "",
    "TestInEditor": false,
    "S2SEnabled": false,
    "S2SPort": 5002,
    "AcceptConnections": true,
    "Namespace": {
      "IsEnabled": true,
      "Value": "PvE"
    }
  }
```

`Namespace` will allow DSS to isolate the players using the same Redis server, same messaging broker and even same zone host. This feature is quite important if you are attempting to create a sandbox MMO. Where each player is going to be assigned to a certain namespace and will only meet players of the same namespace.

If `IsDebuggingLocally` is set true, DSS will redirect all the connections to localhost. Should be enabled if you are testing locally on your machine.

`StaticIP` is the fallback Ip that is going to be used by DSS in case first option is disabled

```
{
    "IsDebuggingLocally": true,
    "StaticIP": "",
    "IPForwardingServer": ""
  }
```

`IPForwardingServer` is the fallback if the first option is disabled and second option is kept empty. It should be used if you are hosting DSS behind a load balancer. Should be filled with a URL for a GET API that either return caller IP as a string or as a JSON e.g: {"ip":"79.79.94.229"}

Since DSS follows a distributed architecture, it uses message broker, and it can be integrated with `Redis, RabbitMQ or Kafka.`

```
{
    "Redis": {
      "IsEnabled": true
    },
    "RabbitMQ": {
      "IsEnabled": false
    },
    "Kafka": {
      "IsEnabled": false
    }
  }
```

x. Seamless travel and Predictive server travel -Beta

Aiming at solving the seamless travel bottleneck of Unreal engine in multiplayer mode, DSS provide an important feature that allow the player to travel from one shard to another wile retaining all players data without the need of any database or storage. Regardless of in which hosted zone the shards are, Grpc over HTTP2 is used to transfer serialized player data. This feature can be used if only OnServerTravel is used.

Additionally, if predictive server travel is enabled, DSS will reserve a seat in a shard for the player if it predicts that the player will travel before it attempt it.

xi. DSS Integration

DSS Support three types of Integrations:

1. Console application
2. Integration with systemd
3. Integration with Windows Services

In case of using the first option DSS will start as a normal console application, while using systemd and Windows Services will allow DSS to interact with the OS, so that the OS will guarantee the health of DSS and will manage its lifespan.

Windows service can be created using the following PowerShell script

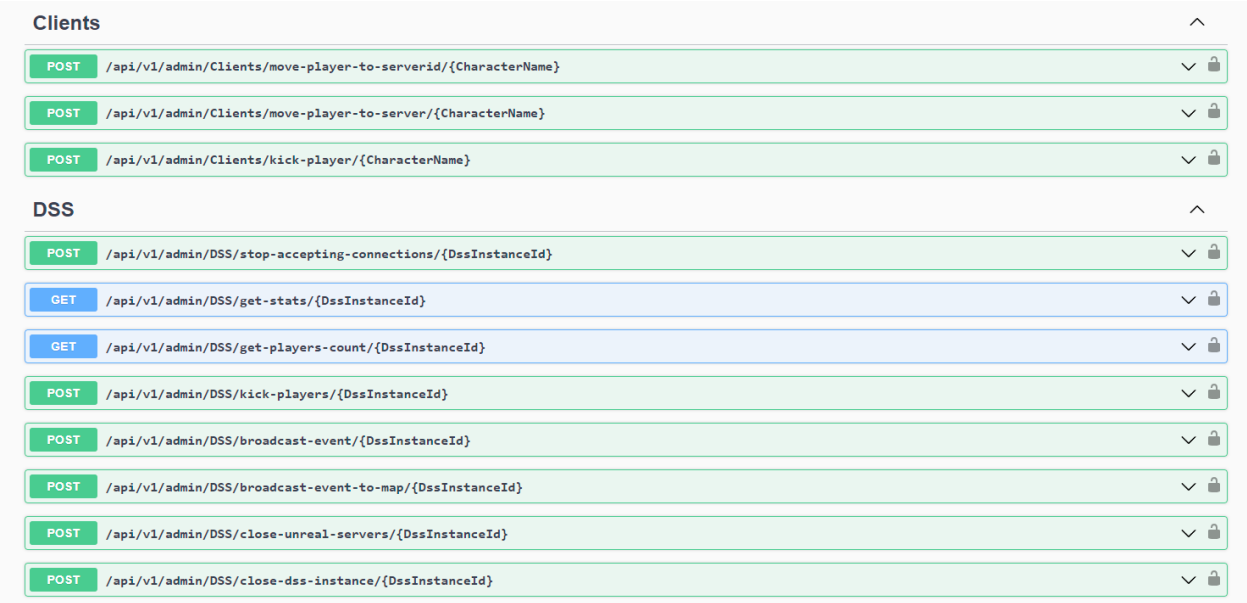
```
New-Service -Name "DSSService" -BinaryPathName "Path To the Binary" -DisplayName "DSS" -Description "This is a DSS service."
```

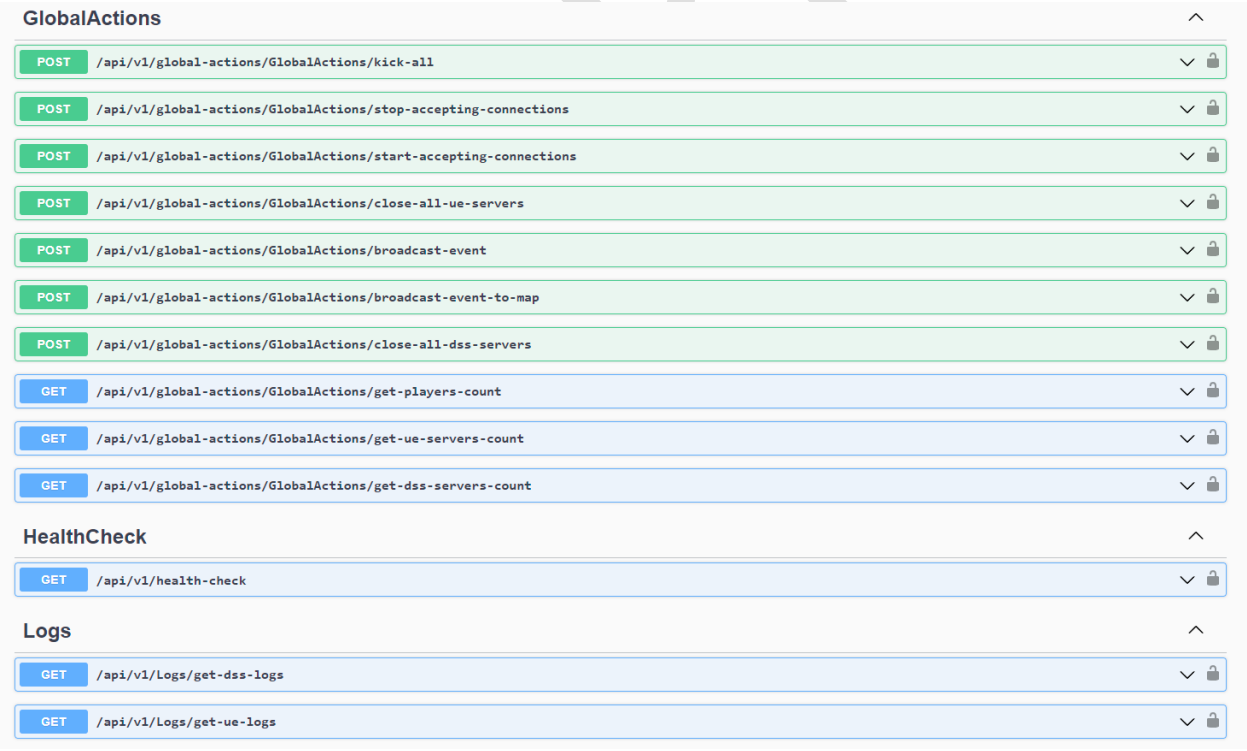
Systemd service might look like this

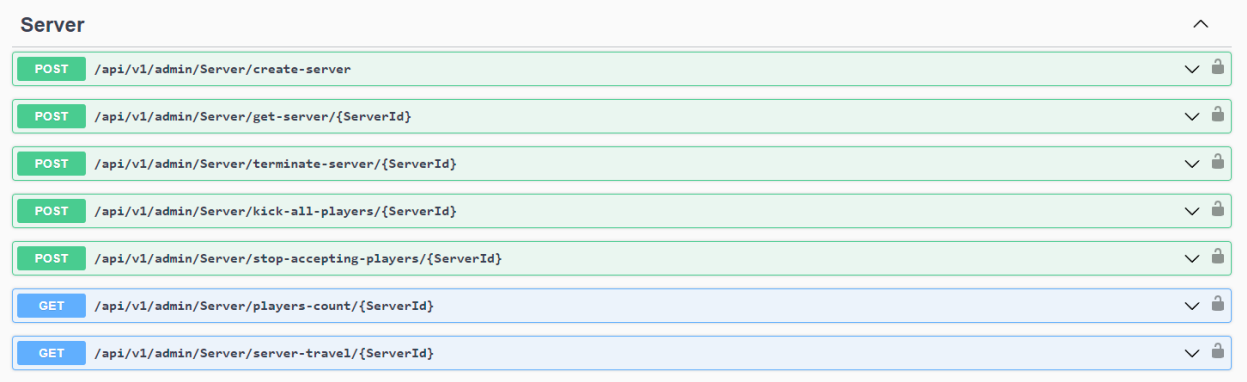
```
[Unit]
Description=Dynamic servers subsystem service
[Service]
Type=notify
ExecStart=DSSServerV2 --jsonconfig  appsettings.json --log
[Install]
WantedBy=multi-user.target
```

xii. DSS Apis

DSS has an HTTP1.1 API that allow to control everything from spinning servers to kicking players. Open Api3 and swagger files are available.







xiii. DSS Command Line

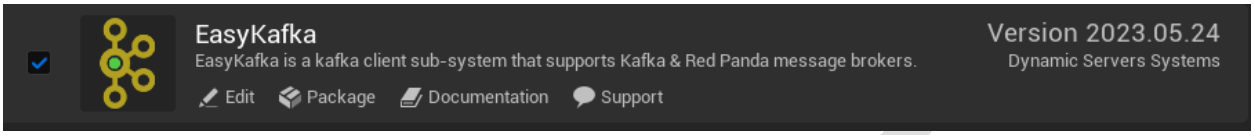
- `--log` Prompt DSS to log to text File, logs can be fetched through an API

```
[16:46:41 INF] Initializing Application.
[16:46:41 INF] Reading Configuration From Json File:appsettings.json
[16:46:41 INF] Parsing Configuration.
[16:46:41 INF] Testing Redis Server.
[16:46:42 INF] Testing Message broker.
[16:46:42 INF] Using Redis as a message broker.
[16:46:42 INF] Running as a Console App. Key listener enabled.
[16:46:48 INF] Application Intialized.
[16:46:48 INF] Working Directory:C:\Users\..\source\repos\DSSLiteServerV2
[16:46:48 INF] DSS Namespace:PvE
[16:46:48 INF] OS:Microsoft Windows 11 Home
[16:46:48 INF] OS Version:10.0.22621
[16:46:48 INF] OS Architecture:X64
[16:46:48 INF] Processor:11th Gen Intel(R) Core(TM) i7-11800H @ 2.30GHz
[16:46:48 INF] Processor Architecture:X64
[16:46:48 INF] Total Threads:16
[16:46:48 INF] Total Memory:65221MB.
[16:46:48 INF] DSS Server Version:001A
[16:47:02 INF] UEServers:3 Players:0 WorkerQueue:2 RAM:42% CPU:16%
[16:47:02 INF] Server utilization score 34%
```

- `--dssport` will override DSS port
- `--s2sport` will override S2S port
- `--ueport` will override from what port UE server will start spinning
- `--ip` will override the static server ip
- `--acceptconnection` will specify if this zone host will accept connections or not
- `--jsonconfig` will specify which json file to use for configurations
- `--apiconfig` if used, configuration will be fetched from a GET API instead of JSON

c. EasyKafka

EasyKafka is a Kafka/Redpanda client sub-system for unreal engine. It supports producing and consuming records through blueprint and C++. This sub-system eases the asynchronous data sharing between all the shards. Full documentation at https://github.com/sha3sha3/UE-EasyKafka.



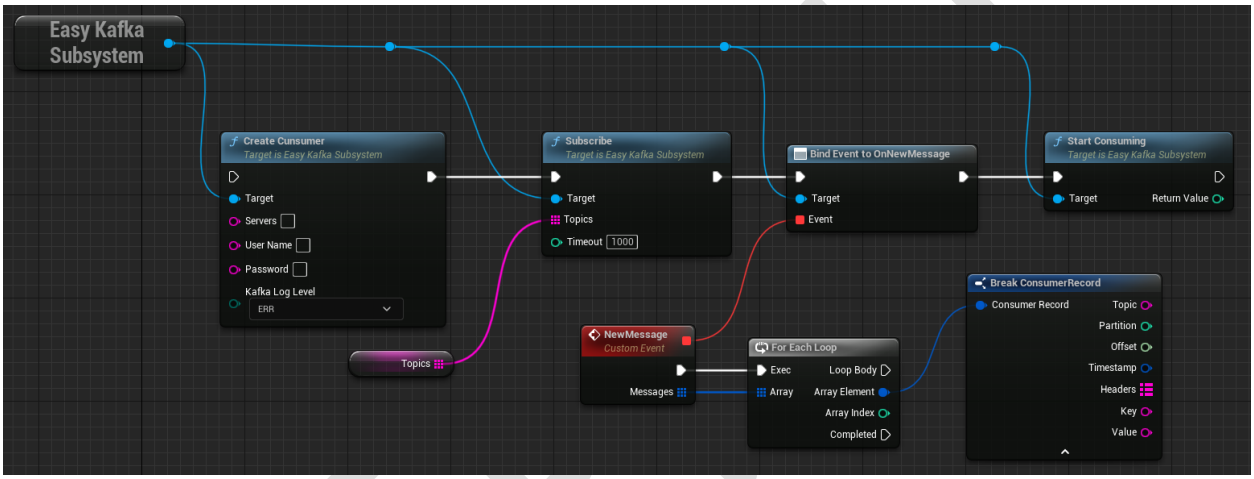


Figure 14: Consuming Messages over Blueprints

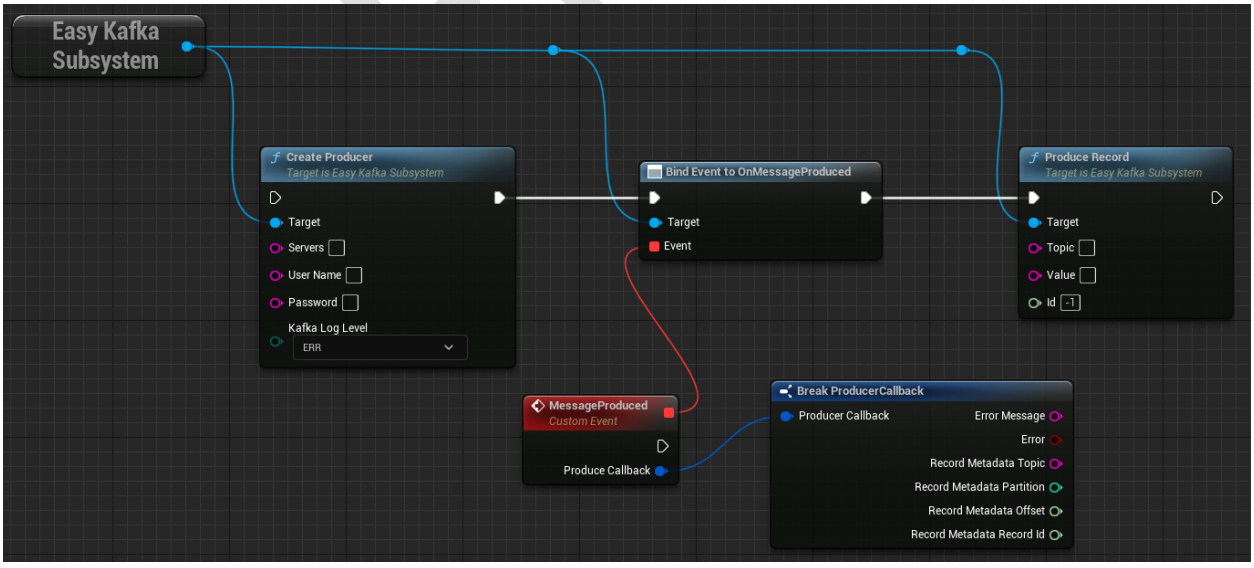


Figure 15:Producing messages over Blueprints

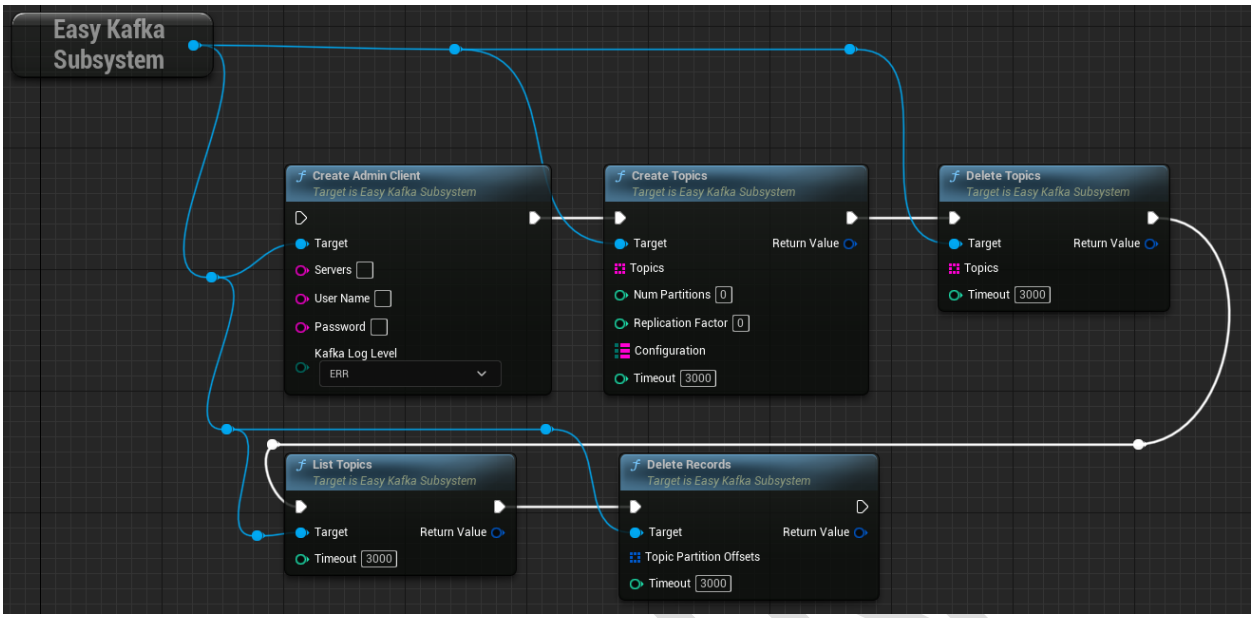
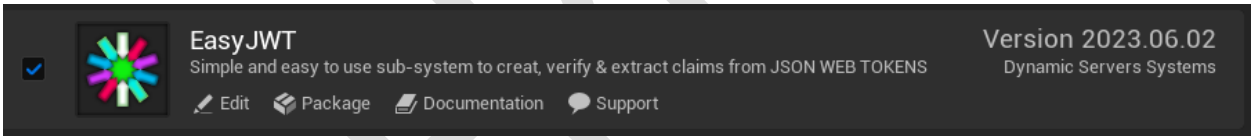


Figure 16: Manage topics and records.

d. EasyJWT

EasyJwt is a JSON web tokens engine sub-system for Unreal Engine 4/5, that provides a c++ and bluprint interface to Generate, Sign, Verify and manage claims of JWT. Full documentation at https://github.com/sha3sha3/UE-EasyJWT



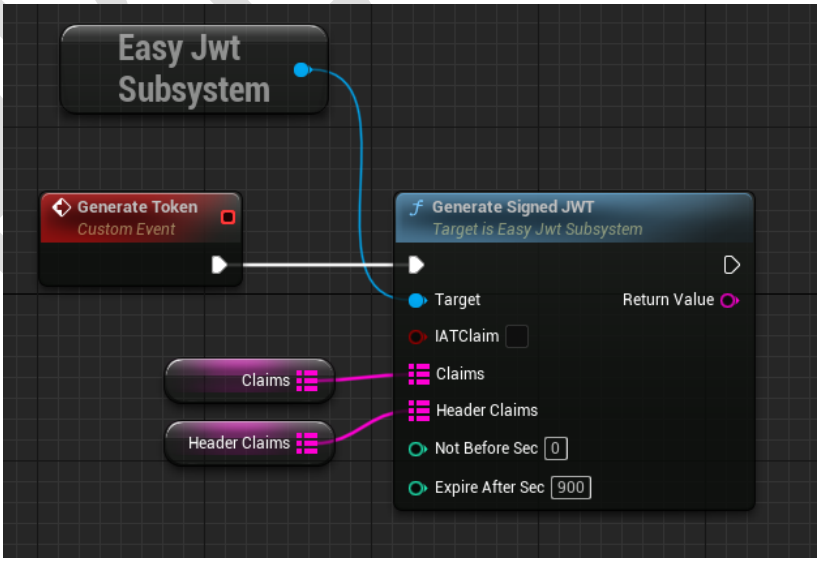


Figure 17: Generate signed token

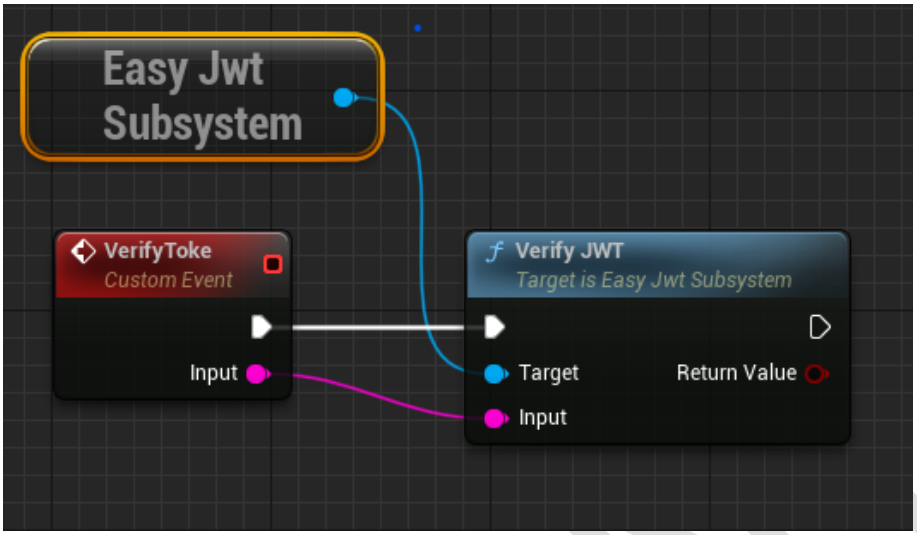


Figure 18: Verify JWT

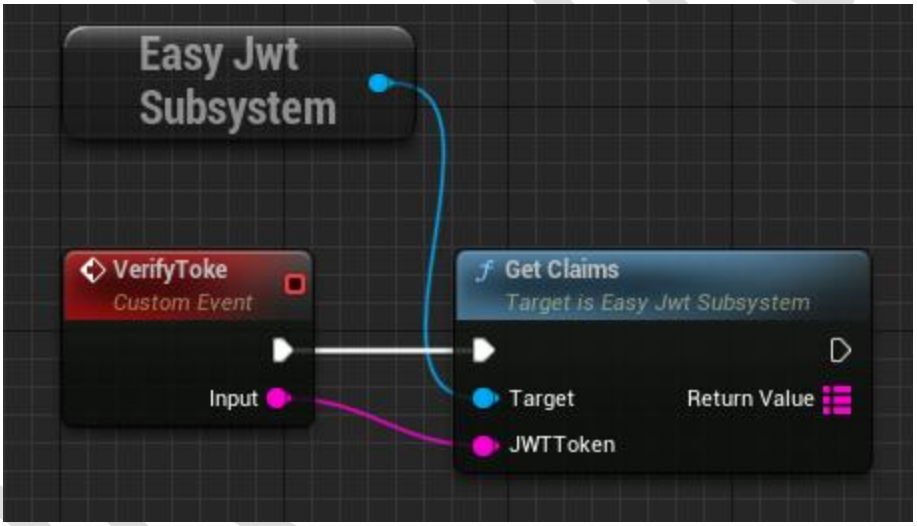


Figure 19: Extract claims from JWT

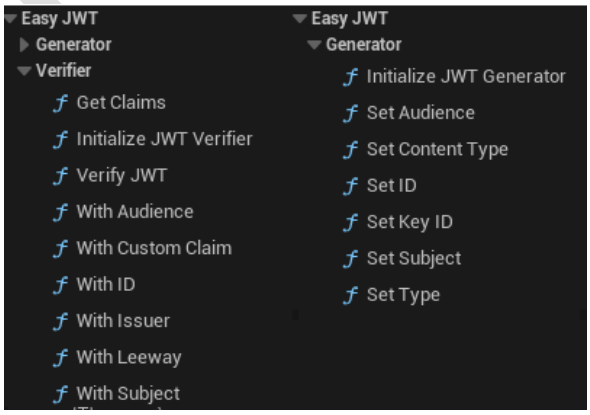


Figure 20: General API

e. EasyGRPC

EasyGrpc is a set of automation scripts that allows to build and generate a GRPC sub-system for Unreal Engine. The automation scripts also allow to generate files for protobuf.

grpcsource	8/10/2023 7:34 PM	File folder	
protobuf	8/10/2023 7:55 PM	File folder	
UBT	8/10/2023 5:26 PM	File folder	
UE-EasyGrpc	8/10/2023 5:26 PM	File folder	
.gitignore	8/10/2023 5:26 PM	Git Ignore Source ...	1 KB
build	8/10/2023 5:27 PM	Windows Batch File	15 KB
CrossCompile	8/10/2023 5:26 PM	CMake Source File	2 KB
LICENSE	8/10/2023 5:26 PM	File	2 KB
Linux64	8/10/2023 5:26 PM	Windows Batch File	14 KB
LinuxArm64	8/12/2023 9:56 AM	Windows Batch File	14 KB
README	8/10/2023 5:26 PM	Markdown Source...	1 KB

Figure 21:Automation scripts

build	8/10/2023 7:55 PM	File folder	
headers	8/10/2023 5:26 PM	File folder	
linux64	8/10/2023 7:50 PM	File folder	
proto	8/10/2023 5:27 PM	File folder	
protoc	8/10/2023 5:26 PM	File folder	
win64	8/10/2023 6:52 PM	File folder	
build	8/10/2023 5:26 PM	Windows Batch File	2 KB

Figure 22: Building protobuf for different platforms

f. Distributed chat server

Since players are distributed on multiple shards and zone hosts, there must be a service that allows players to chat between each other across all the shards. This subsystem support C++ and Blueprint and it scales horizontally to provide best cost to performance ratio. This chat service supports profanity filtering based on reconfigurable multi-language dictionary. It also tackles the issue of retaining the messages while traveling between shards.

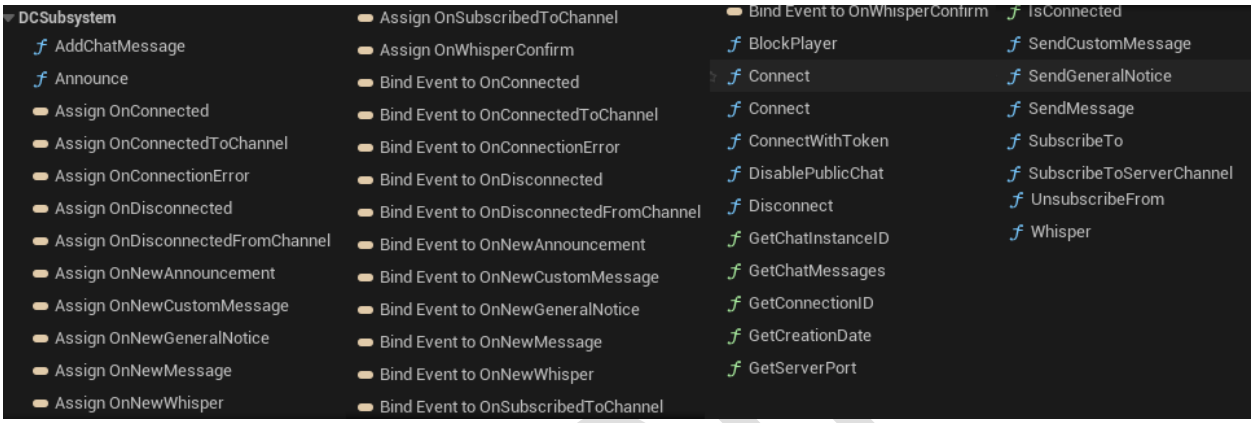


Figure 23:DCS Blueprint API

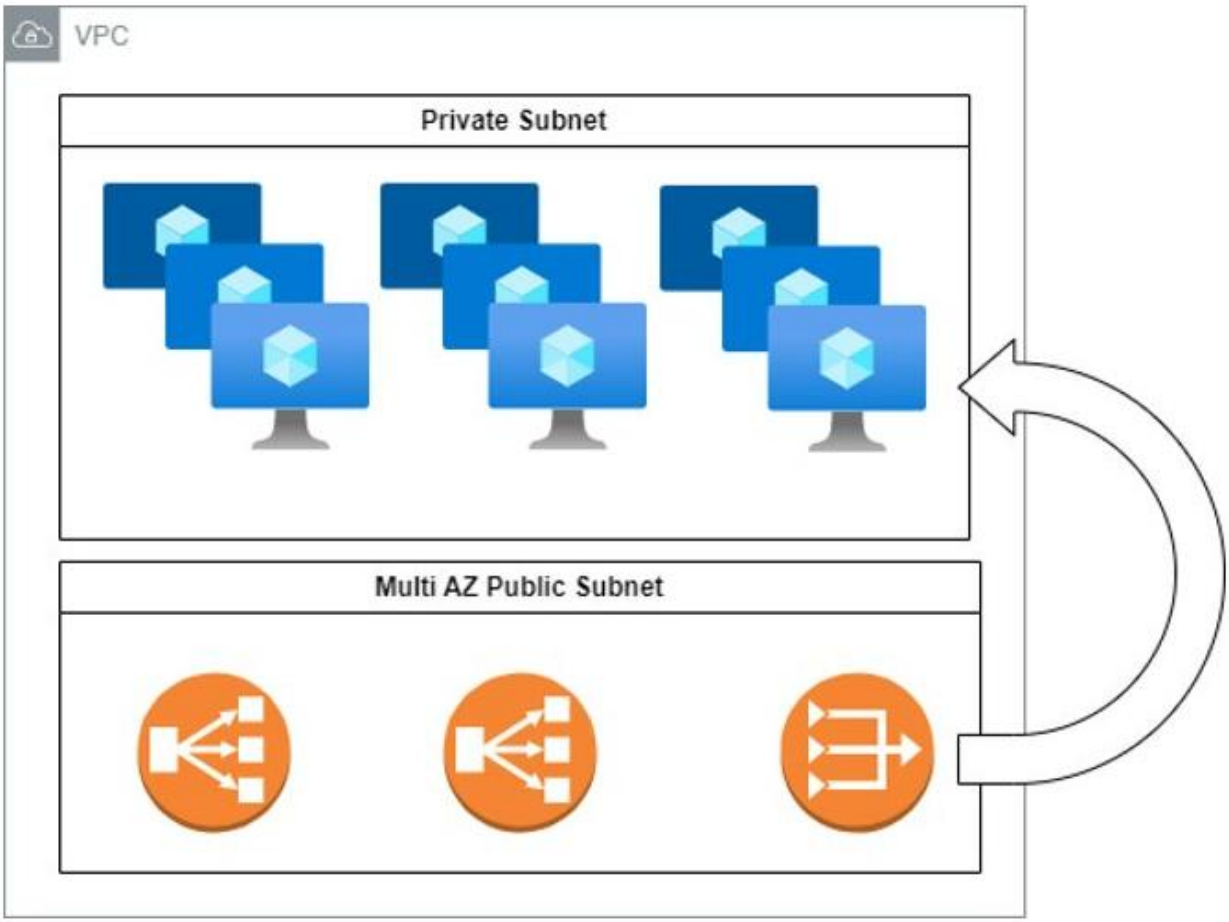
- Party: send message to all the players in a party
- Clan: send message to all the players in a Clan
- Level: send message to all the players in zone
- Public: send message to all the players
- Server Instance: send message to all the players in a shard
- Whisper: send a private message
- Custom: subscribe and send to a custom channel on the go
- Announce: Send by GM to all players

III. Cloud Architecture

DSS is made up of a set of micro-services that are hosted pretty much the same way in the cloud. The whole VPC access is protected behind a VPN such as DSS API, VMs….

a. DSS Services

- DSS Servers service: Handles the zones scaling and players distribution, expose full control over HTTP1.1. Running on Linux OS, integrating with systemd
- DSS Chat service: A distributed chat service that scales horizontally with the players load. It provides a modular API to create and subscribe to channels on the go.
- DSS background services service: Handles repetitive services and events.
- DSS WebApi Service: Horizontally scalable WebApi that deal with persistent storage and cache.
- DSS Worker Service: Horizontally scalable worker that process events published by different components asynchronously.



b. Cloud Services

- **Networking Load balancers**: Distributes traffic on zone hosts.
- **Set of containers and VMs**: Hosting the different services and dynamically scaling
- **Elastic search**: All the services are logging to Elastic Search cluster.
- **Redis server**: Stores important data frequently accessed by DSS services.
- **Kafka stream service**: For events publishing and processing.
- **SQL & NoSql databases**: for persistent data storage
- **A virtual private cloud**: Provide the required security and networking level.