Unreal Engine 4 Network Özeti

Türkçe Çeviri: Uğur Tunç

'Unreal Engine 4' Network Compendium

Created by Cedric 'eXi' Neukirchen
Blog: cedric-neukirchen.net

Co-Founder of Salty Panda Studios
You can hire us: saltypandastudios.com

Document-Version 1.5.4

İÇİNDEKİLER

DİKKAT

Bu Özet, sadece Unreal Engine 4 Oyun Geliştirme Altyapısı anlayışıyla kullanılmalıdır.

Yeni başlayanlara, genel olarak Unreal Engine 4'ün nasıl kullanılacağını ÖĞRETMEZ

Unreal Engine 4'e başlangıç seviyesi olarak başlamak için, Marcos Romero tarafından hazırlanan "Blueprint Compendium" bakınız.

Bu Network Özeti sizden, Blueprint'leri ve C++ dilini (C++ kod örnekleri) nasıl kullanılacağını bilmenizi beklemektedir.

Örnekler Unreal Engine 4.14.x sürümüyle oluşturuldu, bu nedenle kullandığınız yeni UE4 sürümlerinde bir miktar farklılıklar olabilir.

Bu doküman sadece UE4 Network konusuna başlamanız içindir!

GİRİŞ

Unreal Engine 4 Network Özet'ine hoş geldiniz **(**Umarım sizlere yararlı olacak**)**.

Resmi Dokümantasyon zaten oldukça iyi ama ben UE4 ve Multiplayer Oyunlar'da çalışırken son iki yılda öğrendiklerimi **Blueprint ve Kod Özeti** örnekleriyle tek bir çatı altında özetlemek istedim.

İlerleyen sayfalar size **Network Framework'ü; Classes**, **Replication**, **Ownership'i** ve daha fazlasının açıklamalarıyla tanıtacaktır. Elimden geldiğince bir Blueprint ve bir C++ örneği vererek, işlerin nasıl yürüdüğüne dair sizlere ışık tutacağım.

Bu Doküman **tabii ki;** Resmi Dokümantasyondan (Kaynak 1*) alınmış çok sayıda içerik içermektedir, çünkü sistem bu şekilde çalışıyor.

Bu Doküman, daha önce açıklanmış bir şeyin yazılı olarak yeniden tanımlanması olarak değil, bir özet niteliğindedir!

Unreal'da Network

Unreal Engine 4 standard bir **Server-Client** mimarisini kullanmaktadır.

Bu, Server'ın **yetkili** olduğu ve tüm verilerin önce Client'dan Server'a gönderilmesi gerektiği anlamına gelir. Ardından Server verileri doğrular ve kodunuza göre tepki verir.

Küçük bir örnek:

Bir Multiplayer Maç içinde Karakterinizi Client olarak hareket ettirdiğinizde, aslında karakterinizi kendiniz hareket **ettirmiyorsunuz** sadece Server'a hareket etmek istediğinizi söylüyorsunuz. Server daha sonra Karakterin konumunu siz dahil diğer herkeste günceller.

Not: **Lokal** Client'lardaki "**gecikme**" hissini önlemek için; ek olarak Oyuncunun kendi Karakterini direkt kontrol etmesine izin verir (genellikle Kodlayıcılarla) ancak Client hile yapmaya başladığında Server hala Karakterin konumunu geçersiz kılabilir (override)! Bu, Client'ın (neredeyse) hiçbir zaman diğer Client'larla doğrudan '**haberleşemeyeceği**' anlamına gelir.

Başka bir örnek:

Başka bir Client'a sohbet mesajı gönderirken aslında o mesajı, daha sonra iletmesini istediğiniz Client'a ulaştırması için Server'a gönderiyorsunuz. Bu aynı zamanda bir **Takım**, **Birlik**, **Grup** vb. için de olabilir.

ÖNEMLİ!

Client'a **asla** güvenme! Client'lara güvenmek: Client'ların eylemlerini, onlar çalıştırmadan test etmemeniz anlamına gelir. Bu işlem, **hile yapılmasına** imkan verir!

Ateş etmek ilgili basit bir örnek:

Server tarafında; Client'ın direkt olarak ateş etme işlemini yapmak yerine, Client'ın gerçekten mermisinin olup olmadığını ve atış yapmaya izninin olup olmadığını test ettiğinizden emin olun!

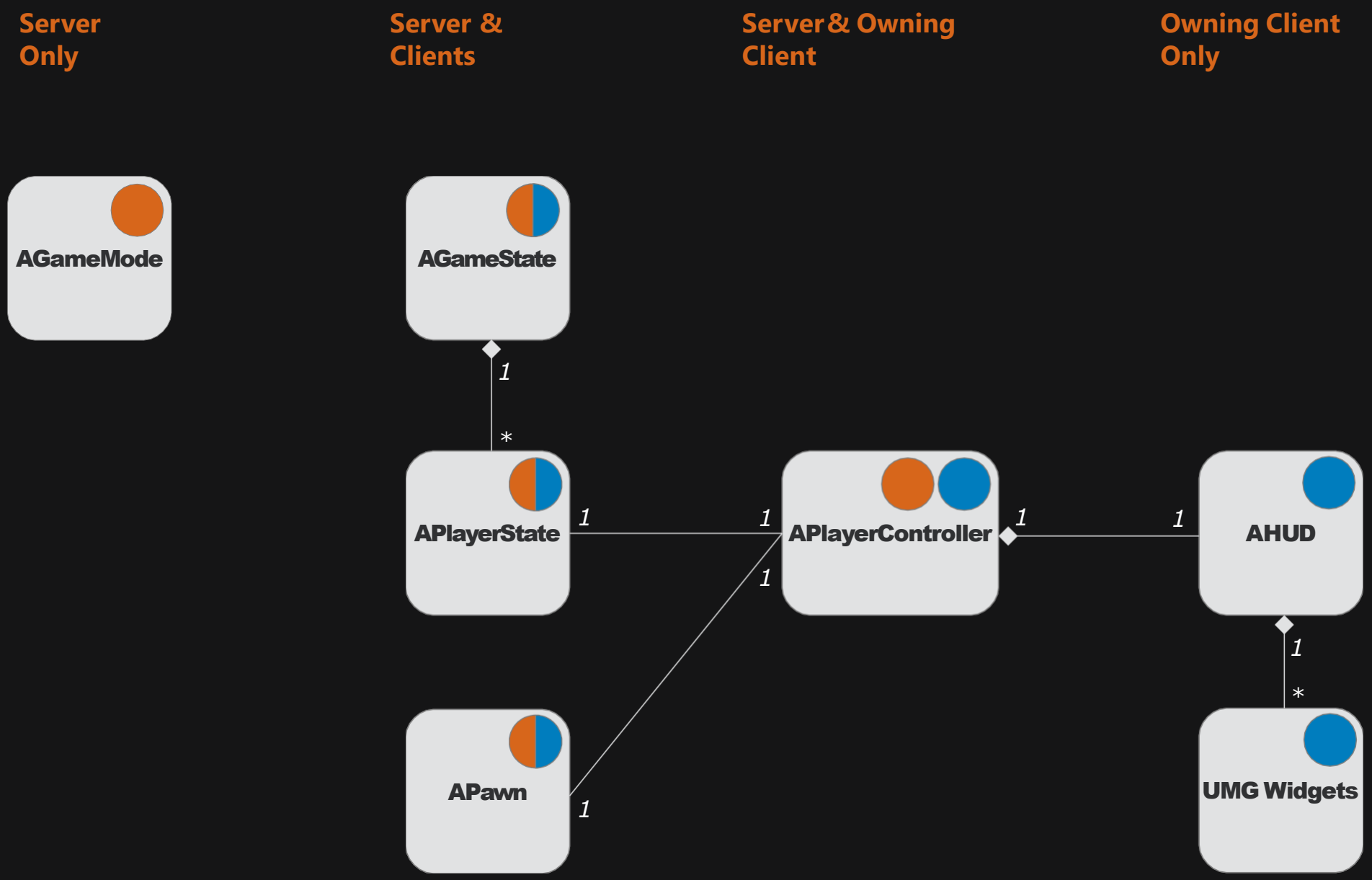
Framework & Network

Unreal Engine 4'ün Server-Client mimarisi hakkındaki bilgilerle Framework'ü 4 bölüme ayırabiliriz:

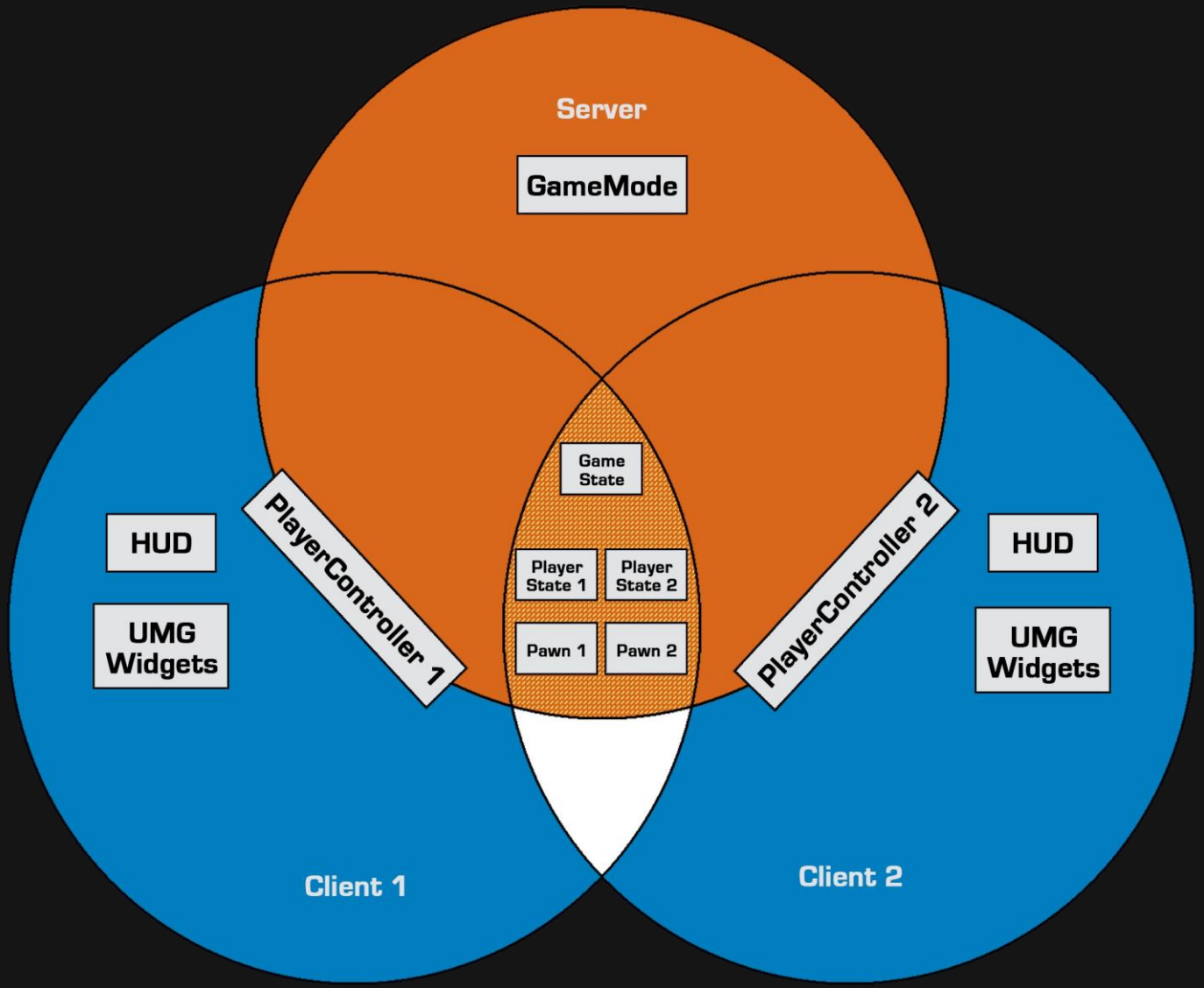
- **Server Only** – Bu objeler sadece Server'da bulunur
- **Server & Clients** – Bu objeler sadece Server'da ve tüm Client'larda bulunur
- **Server & Owning Client** – Bu objeler sadece Server'da ve sahip olan Client'da bulunur
- **Owning Client Only** – Bu objeler sadece Client'da bulunur

"**Owning Client**" Actor'ün sahibi olan Player/Client'dır. Bunu kendi bilgisayarınıza sahip olmanız gibi görebilirsiniz. Sahiplik konusu daha sonra "**RPCs (Remote Procedure Calls)**" konusu için önem arz etmektedir.

Bir sonraki sayfadaki tablo, size bazı ortak sınıfları ve bunların hangi bölümde bulunduğunu gösterir.



Bu kısım, bazı çok önemli sınıfların Network Framework'de nasıl bir düzende olduğunu göstermektedir. (Kaynak 2*)



Soldaki resim bu sınıfların nesnelerinin, Network Framework aracılığıyla nasıl dağıldığına dair başka bir örneği gösterir. (2 Client'lı özel Server)

Client 1 ile Client 2 arasındaki kesişme noktasında obje yoktur çünkü gerçekte sadece kendilerinin bildiği objeleri diğer Client'lara paylaşmazlar. (Kaynak 2*)

Yaygın Sınıflar

Aşağıdaki sayfalarda **Yaygın Sınıflar'dan** bazıları açıklanacaktır.
Ayrıca aşağıdaki sayfalar, bu sınıfların nasıl kullanılacağına dair küçük örnekler vermektedir.

Bu liste, tüm doküman boyunca genişletilecek ve yalnızca başlangıç yapmanız içindir.
Bu yüzden sınıflar hakkında detaylı açıklama yapmadan özgürce konuşabiliriz!

Listelenen örneklerin tümü Replication hakkında bilgi gerektirmektedir.
Anlamadığınız kısımlar olursa, **Replication** bölümünü okuyana kadar şimdilik onları görmezden gelin.

Not: Bazı oyun türleri bu Sınıfları farklı şekillerde kullanabilir.
Bu örnekler ve açıklamalar, onların tek kullanım şekilleri değildir.

Game Mode

(Bu sınıfın ayrıntılı API sayfasına gitmek için ismine tıklayınız)

NOT: Unreal Engine 4.14 ile GameMode sınıfı, GameModeBase ve GameMode adında iki sınıfa ayrıldı. GameModeBase daha az özelliğe sahiptir çünkü bazı oyunlar eski GameMode sınıfının içinde bulunan tüm özelliklere ihtiyaç duymayabilir.

AGameMode sınıfı oyununuzun **KURALLARINI** tanımlamak için kullanılır. Bu **APawn**, **PlayerController**, **APlayerState** gibi çokça kullanılan sınıfı içermektedir.

Yalnızca Server tarafında erişilebilir. Client'lar GameMode nesnesine sahip değildir, ona erişmeye çalıştıklarında nullptr değerini alırlar.

Örnek:

GameMode'ları **Deathmatch**, **Team Deathmatch** ya da **Capture the Flag** gibi yaygın kullanılan modlar ile tanıyabilirsiniz.

Bu demektir ki bir GameMode'un aşağıdaki gibi şeyleri tanımlayabileceği anlamına gelir:

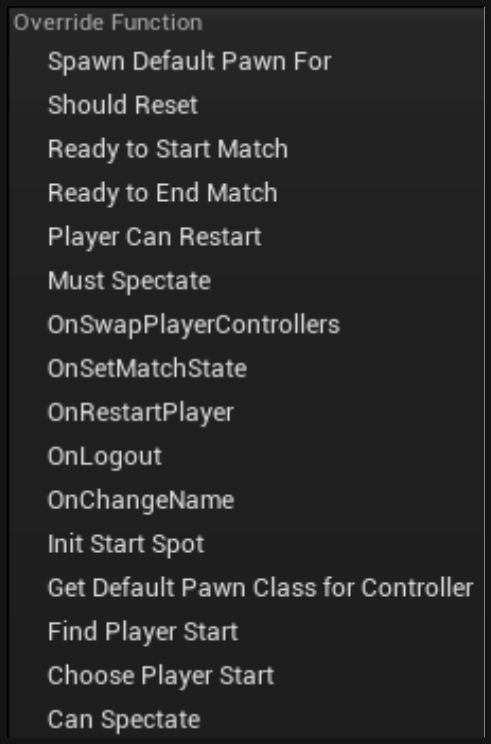
- Takımlarımız var mı yoksa herkes kendi skoru için mi onuyor?
- Oyunu kazanma koşulları nelerdir? Oyuncunun/Takımın kaç kişiyi öldürmesine ihtiyacı var?
- Oyuncu puanları nasıl kazanabilir? Birisini öldürerek mi yoksa bir bayrağı ele geçirerek mi?
- Hangi karakterler kullanılacak? Hangi silahlara erişilebilir? Sadece tabancalar mı?

Örnekler ve Kullanımları

GameMode, Multiplayer oyunlarda Oyuncuları ve genel maç akışını yönetmemize yardımcı olan bazı önemli fonksiyonlara da sahiptir.

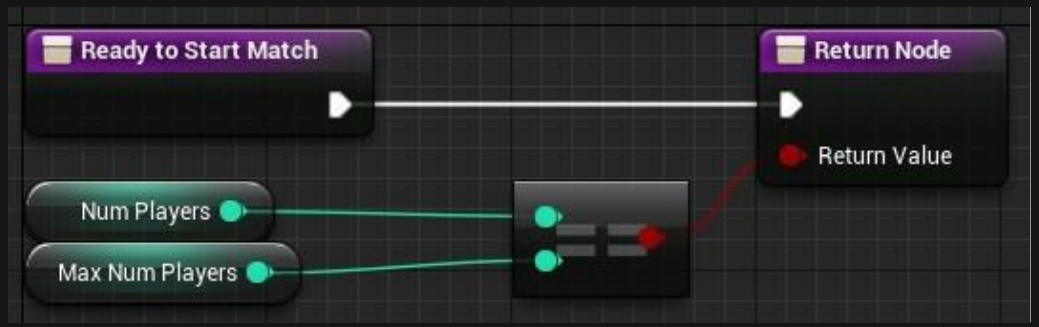
Blueprint

İlk durağımız, örneklerimizin Blueprint versiyonunda "**Override Function**" bölümü olacak.



Oyununuzun belirli kurallara uyması için bu fonksiyonları kendi kafanızdaki mantığa göre yazabilirsiniz. Bu fonksiyonlarla, oyununuza spawn olacak **DefaultPawn'ı** değiştirmeyi ya da oyunun **Ready to Start** durumunda ne yapmak istediğinizi gerçeklebilirsiniz.

Örneğin tüm Oyuncuların oyuna henüz katılıp katılmadığını ve hazır olup olmadığını kontrol etmek için:



Ayrıca maç boyunca gerçekleşen bazı şeylere tepki vermek için kullanabileceğiniz başka **Event**'ler de bulunmaktadır.

Sıklıkla kullandığım **Event OnPostLogin** buna güzel bir örnektir.

Oyuna yeni bir oyuncu **her katıldığında** bu Event çağrılır.

Daha sonra **Bağlantı Süreci (Connecting Process)** hakkında daha fazla bilgi edineceksiniz ama şimdilik bu konuyla devam edeceğiz.

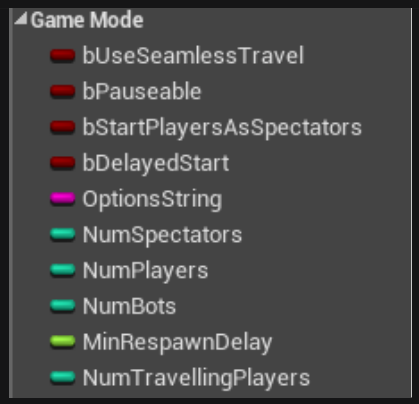
Bu Event size, oyuna bağlanan oyuncunun **sahip** olduğu geçerli (valid) bir Player Controller nesnesi verir. (daha sonra bu konuya bakacağız).

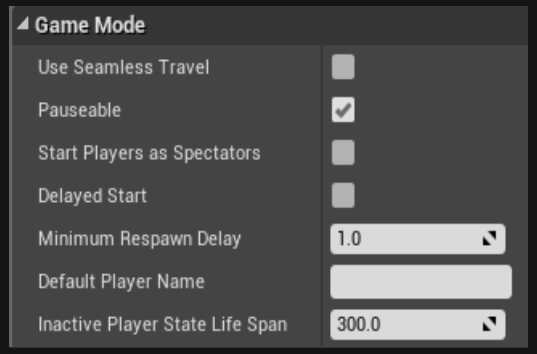
Hali hazırda etkileşime geçtiğiniz bu Oyuncuyla, onun için yeni bir Pawn spawn edebilirsiniz ya da daha sonra kullanmak için bir Player Controller Dizisinde tutabilirsiniz.

Daha önce de belirtildiği gibi, oyununuzun **Maç Akışını** yönetmek için GameMode'u kullanabilirsiniz.
Bu işlem, override edebildiğiniz fonksiyonlarla bağlantılıdır. (**Ready to start Match** gibi).

Bu Fonksiyonlar ve Event'ler, mevcut Maç Durumunuzu kontrol etmek için kullanılabilir. Bunların çoğu **Ready to..** fonksiyonu **TRUE** döndüğünde **otomatik olarak** çağrılır fakat onları **manuel olarak** da kullanabilirsiniz.

New State değişkeni **FName** tipindedir. "Bu işlem neden GameState sınıfında ele alınmıyor?" diye sorabilirsiniz.
Bu GameMode fonksiyonları aslında GameState ile birlikte çalışıyor. State'in Client'ın dışında yönetilmesi gerektiği için, GameMode'un neden sadece Server'da bulunduğunu anlayabiliriz.





GameMode, kullanmak isteyeceğiniz birçok önemli değişkene sahiptir.

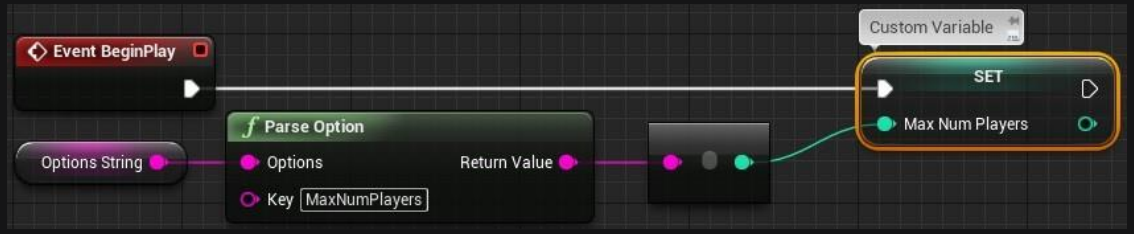
Yandaki görselde **Miraslama'dan (Inherited)** gelen değişkenlerin listesini görebilirsiniz. Bunlardan bazıları, GameMode Blueprint'inin **ClassDefaults** sekmesinde değiştirilebilmektedir:

Çoğu değişkenin adından ne için kullanıldığını anlayabilirsiniz Ör: **Default Player Name**. Bu size oyuna bağlanan her Oyuncunun varsayılan **PlayerName** değişkenine erişme olanağı sağlar. (PlayerName'e PlayerState sınıfı aracılığıyla erişilebilir)

Ya da **Ready to start Match** fonksiyonu diğer tüm kriterleri karşılasa bile oyunun başlamasını engelleyecek olan **bDelayedStart** değişkeni örnek olabilir.

Önemli olan değişkenlerden bir başkası: **Options String**. Başka bir haritayı açmak için çağrılan **OpenLevel** fonksiyonundaki ayarların '**?**' ile ayrıldığı parametredir. **Konsol Komutu'nda (Console Command) ServerTravel** komutunu çağırmanız gibi.

Fonksiyona gönderilen ayarları genişletmek için **Parse Option**'ı kullanabilirsiniz. **MaxNumPlayers** gibi:



UE4++

Önceki sayfalarda yapılan Blueprint işlemleri C++ dilinde de yapılabilir.

Aynı bilgileri tekrar yazmadan, önceki Blueprint örneklerini nasıl yapılacağına dair bazı kod örnekleri vereceğim.

ReadyToStartMatch bir **BlueprintNativeEvent'tidir**. Fonksiyonu kullanmak için C++ tarafında **ReadyToStartMatch_Implementation** anahtar sözcüğünü kullanırız. Böylece fonksiyonu override ederiz:

```
/* Sınıf deklarasyonun yapıldığı, kendi Game Mode Child Class'ımızın Header dosyası */
// Maç içerisinde gereken/izin verilen maksimum oyuncu sayısı
int32 MaxNumPlayers;

//ReadyToStartMatch fonksiyonun override edilmesi
virtual bool ReadyToStartMatch_Implementation() override;
```

```
/* Kendi GameMode Child Class'ımızın cpp dosyası*/
bool ATestGameMode::ReadyToStartMatch_Implementation(){
    Super::ReadyToStartMatch();

    return MaxNumPlayers == NumPlayers;
}
```

OnPostLogin fonksiyonu virtual'dır ve C++ dilinde basitçe **PostLogin** olarak çağrılır.

Hadi bu fonksiyonu da override edelim:

```
/* Sınıf deklarasyonunun yapıldığı, kendi Game Mode Child Class'ımızın Header dosyası */
// PlayerController'ların Listesi
TArray<class APlayerController*> PlayerControllerList;

// PostLogin fonksiyonun override edilmesi
virtual void PostLogin(APlayerController* NewPlayer) override;
```

```
/* Kendi GameMode Child Class'ımızın cpp dosyası */
void ATestGameMode::PostLogin(APlayerController* NewPlayer) {
	Super::PostLogin(NewPlayer);

	PlayerControllerList.Add(NewPlayer);
}
```

Tabii ki tüm **Maç-yönetme (Match-handling)** fonksiyonları override edilebilir ve değiştirilebilir. Burada hepsini listelemek istemiyorum. **GameMode Başlığındaki** linke tıklayarak **API'yi** inceleyebilirsiniz.

Son C++ örneğimiz GameMode için '**Options String**' :

```
/* Sınıf deklarasyonunun yapıldığı, kendi Game Mode Child Class'ımızın Header dosyası */
// Maç içerisinde gereken/izin verilen maksimum oyuncu sayısı
int32 MaxNumPlayers;

// Blueprint versiyonunundaki BeginPlay'in override edilmesi
virtual void BeginPlay() override;
```

```
/* Kendi GameMode Child Class'ımızın cpp
dosyası */

void ATestGameMode::BeginPlay() {
    Super::BeginPlay();

    //OptionsString'den doğru Key'i almak için 'UGameplayStatics' Sınıfının statik 'ParseOption' foksiyonunu kullanırız.
    // FCString::Atoi fonksiyonu Fstring'i 'int32' türüne çevirir.

    MaxNumPlayers = FCString::Atoi( *(UGameplayStatics::ParseOption(OptionsString, "MaxNumPlayers")) );
}
```

Game State

(Bu sınıfın ayrıntılı API sayfasına gitmek için ismine tıklayınız)

NOT: Unreal Engine 4.14 ile GameState sınıfı, AGameStateBase ve AGameState adında iki sınıfa ayrıldı. GameStateBase daha az özelliğe sahiptir çünkü bazı oyunlar eski GameState sınıfının içinde bulunan tüm özelliklere ihtiyaç duymayabilir.

AGameState sınıfı, Server ve Client'lar arasında paylaşılan bilgiler için muhtemelen en önemli sınıftır.

GameState, mevcut Oyununuzun Durumunu takip etmek için kullanılır. Multiplayer oyunlar için önemli olan **Bağlı Oyuncular Listesi** (**APlayerState türünde**) dizisini içerisinde barındırır.

GameState, tüm Client'lara replike edilir (replicated). Böylece herkes ona erişebilir. Bu özellik GameState'i, Multiplayer Oyunlar için en merkezi sınıflardan biri yapar.

GameMode; oyunu kazanmak için her Oyuncunun ve/veya Takımın ne kadar oyuncu öldürmesi gerektiğini söylerken, GameState bu mevcut miktarı takip eder!

Burada hangi bilgileri saklayacağınız tamamen size kalmış. Grupları ve Birlikleri takip etmek için bir puan dizisi veya kendi hazırladığınız özel bir struct dizisi kullanabilirsiniz.

Örnekler ve Kullanımları

Multiplayer oyunda GameState sınıfı; **Oyunun Current State'ini (O anki), Player'ları** ve onların **PlayerState'lerini** takip etmek için kullanılır. GameMode, GameState'in **Maç Durumu (Match State)** fonksiyonlarının çağrılmasını sağlar, GameState'in kendisi de bunları Client'larda kullanır.

GameMode ile GameState'i karşılaştırdığımızda, GameState bize çalışmamız için fazla bir şey vermiyor. Ancak yine de Client'lara **dağıtılması gereken bilgi** için kendi mantığımızı yazmamıza izin verir.

Blueprint

Temel GameState sınıfından kullanabileceğimiz **Değişkenleri** alabiliriz. **PlayerArray* (Oyuncu Dizisi)**, **MatchState (Maç Durumu)** ve **ElapsedTime (Geçen Zaman)** değişkenleri **replikedir (replicated)**. Böylece bu değişkenlere Client'lar da erişebilir.
Aynı özellik **AuthorityGameMode** için geçerli değildir. GameMode sadece Server'da bulunduğu için ona sadece Server erişebilir.

*PlayerArray direkt olarak replike (replicated) değildir. Fakat her PlayerState, Construction aşamasında Oyuncuları PlayerArray'e ekler.

PlayerArray oluşturulduğunda GameState bu diziye bir kez erişir.

C++ tarafında PlayerArray'in nerede "replike" (replicated) edildiğini anlamak için kod parçası

PlayerState sınıfın içeriği:

```
void APlayerState::PostInitializeComponents() {
	[...]
	UWorld* World = GetWorld();
	//Bu PlayerState'i Oyun'un Replikasyonla kaydedilmesi
	if(World->GameState != NULL) {
		World->GameState->AddPlayerState(this);
	}
	[...]
}
```

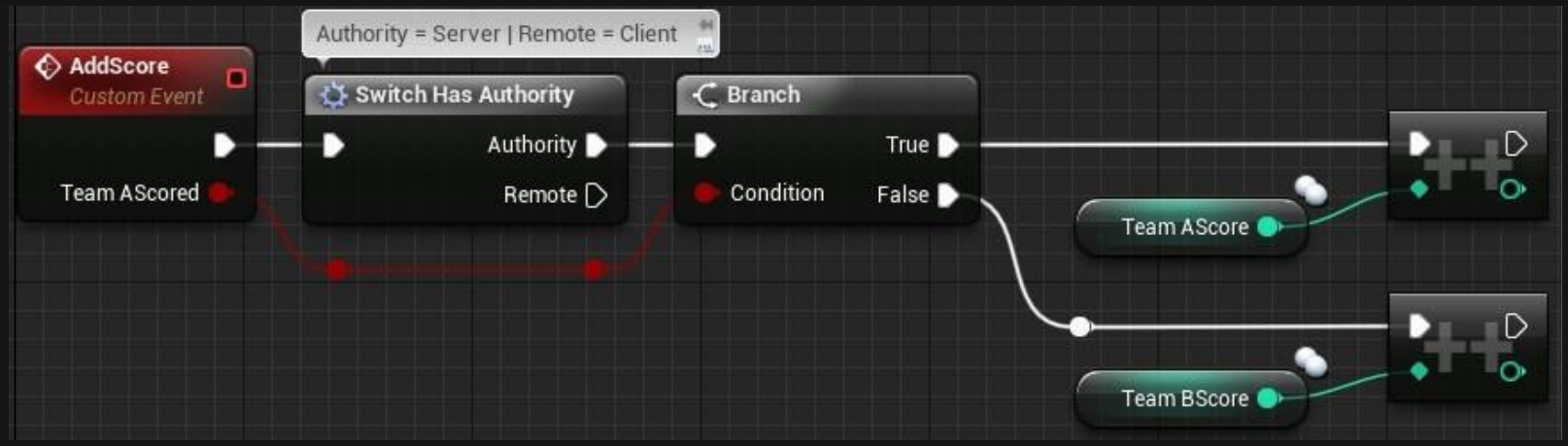
GameState'in içeriği:

```
void AGameState::PostInitializeComponents() {
	[...]
	for(TActorIterator<APlayerState> It(World); It; ++It){
		AddPlayerState(*It);
	}
}
void AGameState::AddPlayerState(APlayerState* PlayerState) {
	if(!PlayerState->bIsInactive) {
		PlayerArray.AddUnique(PlayerState);
	}
}
```

Tüm bunlar Server'da ve Client'lardaki PlayerState ve GameState kopyalarında gerçekleşir!

Takım 'A' ve **Takım 'B'** adında iki takımın **Skorlarını takip etmek** için küçük bir örnek verelim. Diyelim ki bir Takım skor aldığında elimizde bir **Custom Event (Özel Event)** olsun.

Herhangi bir takım skor alırsa bu bilgiyi bir boolean değişkenle gönderiyoruz. Daha sonra işleyeceğimiz "**Replikasyon**" **(Replication)** Bölümü'nde, değişkenleri **sadece** Server'ın replike edebilmesi "Kuralı" hakkındaki bilgileri okuyacaksınız. Bu yüzden bu Event'i sadece Server'ın çağırdığından emin oluyoruz. Bu Event başka bir sınıftan çağrılıyor. Bu olayın (daima!) Server tarafında olması gerekiyor (Ör: Bir silahla birini öldürdüğümüzde). Bu yüzden burada **RPC (Remote Procedure Calls)** işlemlerine ihtiyacımız yok.



Bu değişkenler ve GameState replike (replicated) olduğundan bu iki değişkeni istediğiniz başka bir Sınıf'ta alıp kullanabilirsiniz. Ör: **Skor Tablosu'**nda **(Scoreboard Widget)** gösterebilirsiniz..

UE4++

Aynı örneği C++ tarafında oluştururken, kendi içinde bulunan fonksiyonlarına rağmen biraz daha kod yazmamıza ihtiyacımız var.
Her sınıf başına bir kez yapılması gereken Replication ayarlamalarını yazmamız gerekiyor.

```
/* Sınıf deklarasyonun yapıldığı, kendi GameState Class'ımızın Header dosyası */
// Replication'ın çalışması için bu kütüphaneyi  dahil etmeniz gerek. Projenin Header'ında eklemeniz daha iyi olacaktır!
#include "UnrealNetwork.h"

// Değişkeni replike etmek için Replicated Belirleyicisi (Specifier)
UPROPERTY(Replicated)
	int32 TeamAScore;
UPROPERTY(Replicated)
	int32 TeamBScore;

// Bir Takımın skorunu arttıran Fonksiyon
void AddScore(bool TeamAScored);
```

Replikasyon (Replication) Bölümü'nde bu işlemler hakkında daha fazla bilgi edineceksiniz!

```
/* Kendi GameState Sınıfımızın cpp dosyası */
//UPROPERTY Macro'sunda deklere edilen Replicated Belirleyicisi (Specifier) yüzünden bu Fonksiyon
zorunludur.
void ATestGameState::GetLifetimeReplicatedProps(TArray<FLifetimeProperty>& OutLifetimeProps) const {

	Super::GetLifetimeReplicatedProps(OutLifetimeProps);

	DOREPLIFETIME(ATestGameState, TeamAScore);
	DOREPLIFETIME(ATestGameState, TeamBScore);
}
```

```
/* Kendi GameState Sınıfımızın cpp dosyası */
void ATestGameState::AddScore(bool TeamAScored){

	if(TeamAScored)
		TeamAScore++;
	else
		TeamBScore++;
}
```

Player State

(Bu sınıfın ayrıntılı API sayfasına gitmek için ismine tıklayınız)

APlayerState spesifik bir Player için en önemli sınıftır.
Bu sınıf, Player hakkında güncel bilgileri tutmak içindir. Her Player'ın kendine ait PlayerState'i vardır. PlayerState diğer Client'lardaki verileri alıp görüntülemek için herkese replike (replicated) edilmiştir. GameState sınıfının içindeki **PlayerArray,** mevcut Oyun'daki tüm PlayerState'lere erişmenin kolay bir yolu yoludur.

PlayerState'den alıp kaydetmek isteyebileceğiniz bazı bilgiler:

- **PlayerName** – Bağlanan Oyuncunun mevcut Adı (Current Name)
- **Score** – Bağlanan Oyuncunun mevcut Skoru (Current Score)
- **Ping** – Bağlanan Oyuncunun mevcut Ping'i (Current Ping)
- **GuildID** – Oyuncun dahil olabileceği birliğin ID'si
- Ya da diğer Oyuncuların bilmesi gerekebilecek diğer Replike (Replicated) bilgiler

Örnekler ve Kullanımları

Multiplayer oyunda PlayerState, **bağlı Oyuncu'nun Durumunu (State)** tutmak için kullanılmaktadır.

Buna **Adı (Name)**, **Skoru (Score)**, **Pingi (Ping)** ve kendi özel değişkenleriniz de dahildir.

PlayerState sınıfı replike (replicated) olduğundan, kolayca bir Client'dan başka bir Client'a **verileri almak** için kullanılabilir.

Blueprint

Ne yazık ki bunu yazdığımda çok önemli ana fonksiyonlar Blueprint'te tutulmuyordu.

Bu yüzden PlayerState'in C++ örneklerinde onları açıklayacağım. Ancak bazı değişkenlere bir göz atabiliriz:

Bu değişkenlerin tamamı replikedir (replicated). Bu yüzden bu değişkenler **tüm Client'larda senkronizedir (sync)**.

Ne yazık ki bu değişkenler Blueprint tarafında kolayca **ayarlanamazlar** fakat hiçbir şey sizi kendi modelinizi oluşturmaktan alıkoyamaz.

Oyuncu'nun PlayerController'ını "**ChangeName**" adında **bir GameMode fonksiyonuna** parametre vererek, Oyuncu'nun **PlayerName** değişkenini set edebileceğiniz bir örnek.

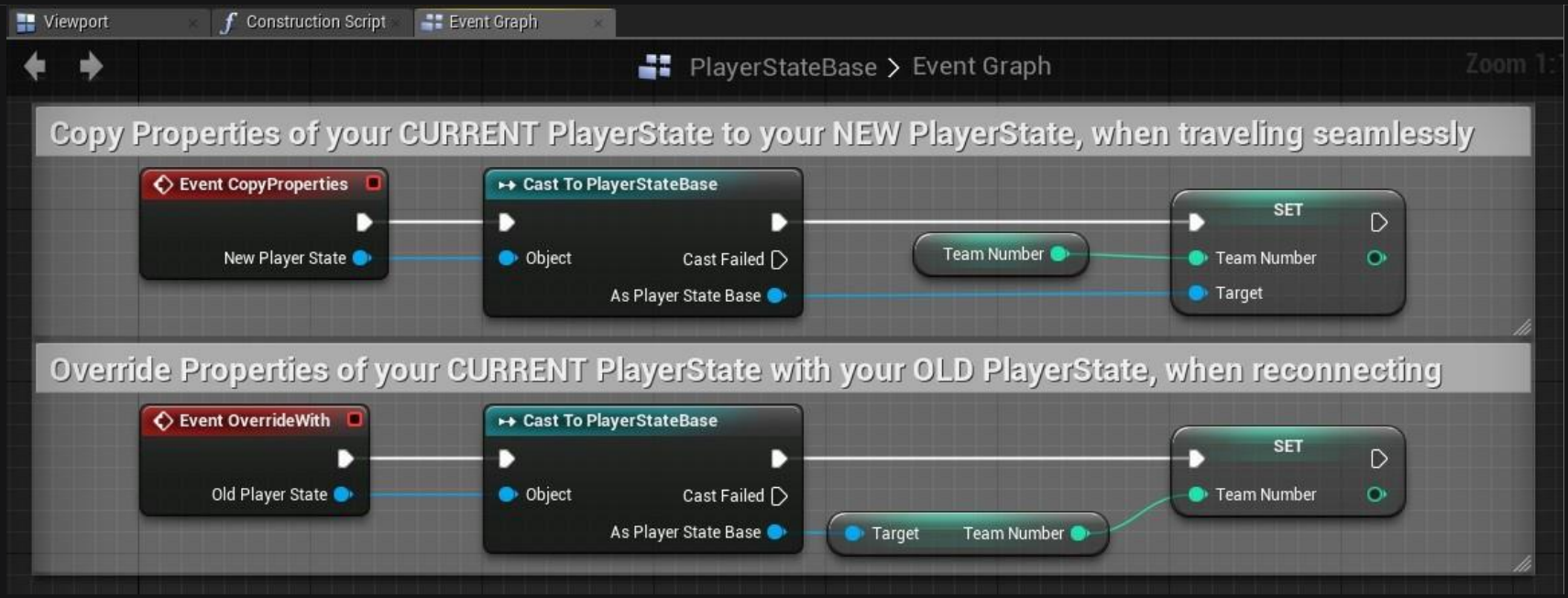
PlayerState Level Değişikliklerinde ya da beklenmeyen bağlantı sorunları esnasında verilerin kalıcı kalmasını sağlamak için de kullanılmaktadır.

PlayerState **yeniden bağlanan Oyuncular** ve Server'la birlikte yeni haritaya **taşınan (traveled) Oyuncular** için iki fonksiyonu da içerisinde barındırır.

Yaptığım Pull Request kabul edilip, UE 4.13 yayımlanınca bu fonksiyonlar Player State'den türetilmiş Blueprint'lerde erişilebilir duruma

gelmiştir. PlayerState halihazırda elinde bulundurduğu bilgileri, yeni PlayerState'e kopyalama işlemini de yapmaktadır.

Her iki işlem, **Level Değişikliği** esnasında veya Oyuncu **yeniden bağlanıyorken** oluşturulmaktadır.



UE4++

Aynı işlem tabii ki C++ dilinde de bulunuyor.

```
/* Sınıf deklarasyonun yapıldığı, kendi PlayerState Child Class'ımızın Header dosyası */
//Mevcut PlayerState'den geçirilecek olan PlayerState'e özellikleri
kopyalan fonksiyon
virtual void CopyProperties(class APlayerState* PlayerState);

//Mevcut PlayerState'den geçirilecek olan PlayerState'e özellikleri üzerine yazan
fonksiyon
virtual void OverrideWith(class APlayerState* PlayerState);
```

Bu fonksiyonlar, verileri yönetmek için kendi özel C++ PlayerState Child-Sınıfınızın içerisinde gerçeklenebilir. Orijinal gerçeklemenin etkin kalması için (eğer istiyorsanız) fonksiyonun sonuna "override" eklediğinizden ve fonksiyonu çağırırken "Super::" belirteçlerini kullandığınızdan emin olunuz.

Fonksiyonların gerçeklemesi şu şekildedir:

```
/* Kendi PlayerState Child Sınıfımızın cpp dosyası */

void ATestPlayerState::CopyProperties(class APlayerState* PlayerState) {
	Super::CopyProperties(PlayerState);

	if(PlayerState) {
		ATestPlayerState* TestPlayerState = Cast<ATestPlayerState>(PlayerState);

		if(TestPlayerState)
			TestPlayerState->SomeVariable = SomeVariable;
	}
}

void ATestPlayerState::OverrideWith(class APlayerState* PlayerState){
	Super::OverrideWith(PlayerState);

	if(PlayerState) {
		ATestPlayerState* TestPlayerState = Cast<ATestPlayerState>(PlayerState);

		if(TestPlayerState)
			SomeVariable = TestPlayerState->SomeVariable;
	}
}
```

Pawn

(Bu sınıfın ayrıntılı API sayfasına gitmek için ismine tıklayınız)

APawn sınıfı Oyuncu'nun gerçekten kontrol ettiği **Actor**'dür. Pawn çoğu zaman insandır fakat aynı zamanda bir kedi, uçak, gemi, blok vb. olabilir. Oyuncu aynı anda sadece bir Pawn'a sahip (possess) olabilir ama Pawn'lar arasında, sahiplikten çıkma (un-possessing) ve tekrar sahip olma (re-possessing) işlemleriyle kolaylıkla geçiş yapabilirsiniz.

Pawn çoğunlukla tüm Client'larda replikedir (replicated).

Oyuncunun Karakterinin Pozisyonunu, Rotasyonunu vb. ele almak için sıklıkla, Pawn sınıfının child'ı olan **ACharacter** sınıfı kullanılır. Çünkü Karakter sınıfı içerisinde halihazırda network üzerinde replike edilmiş (replicated) **MovementComponent** beraberinde gelir.

Not: Client Karakteri hareket ettirmez. Server, Client'dan hareket girdilerini (move-inputs) alır, Karakteri hareket ettirir ve replike (replicating) eder!

Örnekler ve Kullanımları

Multiplayer Oyun'da Karakterin görüntülenmesi ve bazı bilgilerin diğerleriyle paylaşmak için çoğunlukla Pawn'ın **Replication** Bölümü'nü kullanırız. Karakterin **Health (Sağlık)** değeri buna bir örnektir.

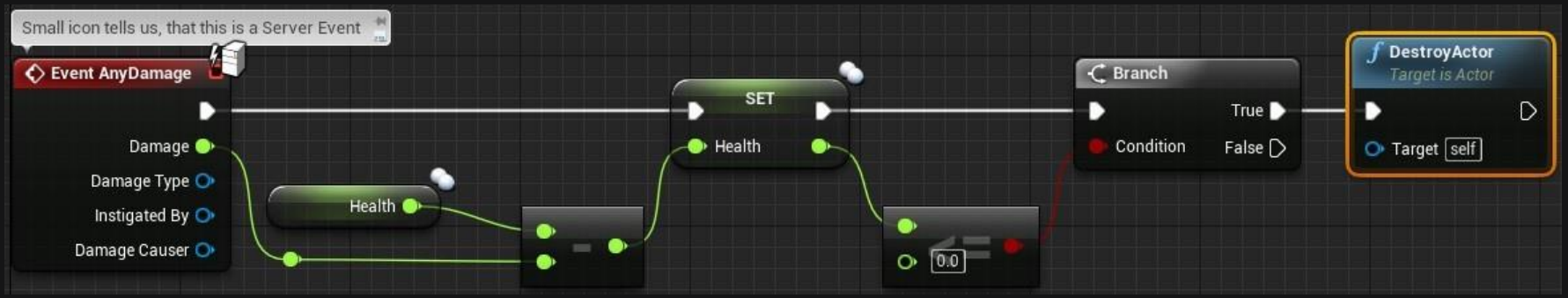
Health (Sağlık) değişkenini, sadece diğer Oyuculardaki erişebilir olması için replike (replicate) etmiyoruz. Aynı zamanda da Server'ın bu değişken üzerinde yetki sahibi olması (authority) ve Client'ın hile yapamaması için replike ediyoruz.

Blueprint

'**Standard**' üzerine yazılabilir (overridable) fonksiyonlara karşın Pawn, sahip olmaya (possessing) ve sahiplikten çıkmaya (un-possessing) yarayan iki fonksiyonu da içerir. Bu fonksiyonlar, Pawn sahipsiz kaldığında (un-possessed) Pawn'ı gizlemek için kullanılabilirler:

Not: Yukarıdaki ekran görüntüsünde gördüğünüz gibi, Pawn'ın görünürlüğünü değiştirme işlemi **MulticastRPCFunction'a** ihtiyaç duyar. Sahip olma işlemleri Server tarafında yapıldığı için bu Multicast Event'ler sadece **Server Versiyon'unda (Server Event)** çağrılır!

İkinci örneğimiz **Health (Sağlık)** değişkenini ele alacaktır. Aşağıdaki resimde, Oyuncu'nun replike (replicated) **Health (Sağlık)** değişkenini düşürmek için **EventAnyDamage** Event'inin nasıl kullanacağınızı gösterecektir. Bu işlem Client'da değil Server tarafında olur!



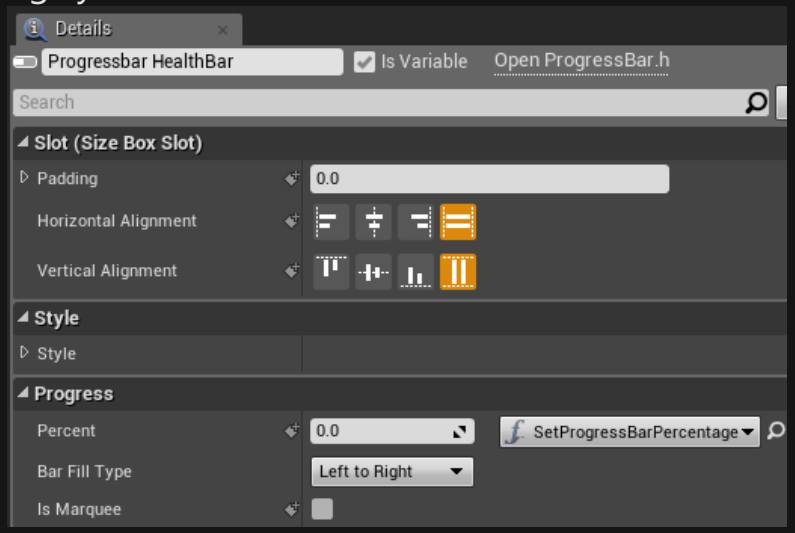
Pawn replike (replicated) olmasıyla birlikte, **DestroyActor** Node'u Server tarafından çağrılırsa Client tarafındaki Actor'ü de yok edecektir (destroy). Client tarafında replike (replicated) **Health (Sağlık)** değişkenini kendi HUD'ımızda (Head-up Display) ya da oyuncuların kafasının üstündeki **HealthBar (Sağlık Barı)** üzerinde kullanabiliriz.

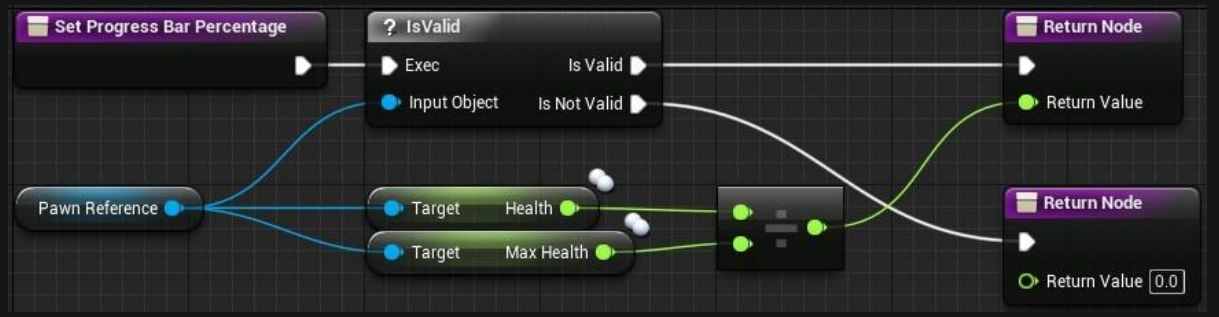
Bunu **Progressbar'lı Widget** ve **Pawn'ın Referansı** ile kolaylıkla yapabilirsiniz.

Bu doküman, Network Özeti olduğundan Widget'lar hakkında bilgi sahibi olmanızı ya da en azından onlar hakkında başka yerlerdeki bilgileri okumanızı bekliyorum. Aksi takdirde Widget konusu, bu doküman için fazla olacaktır.

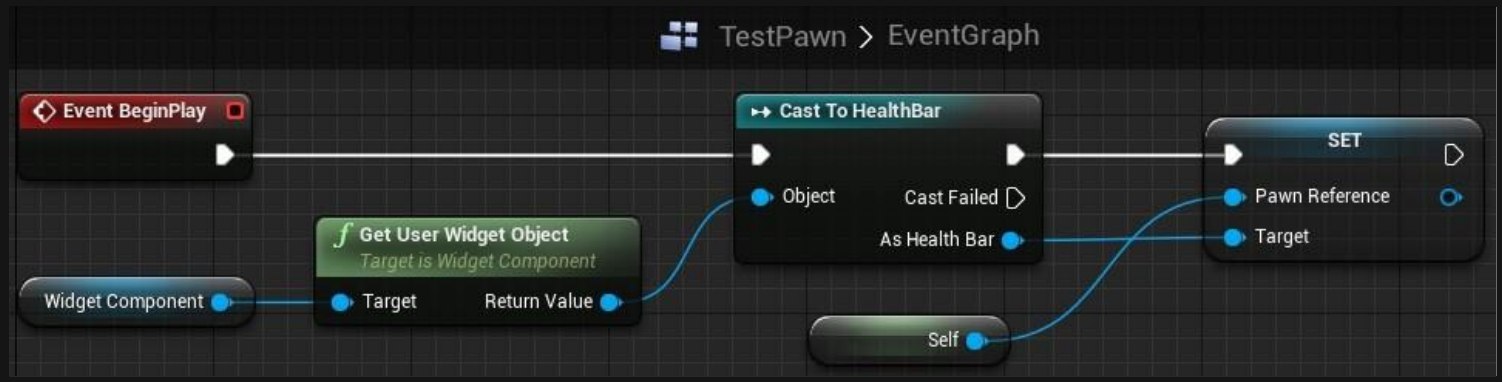
TestPawn sınıfımızda **Health (Sağlık)** ve **MaxHealth (Maksimum Sağlık)** adlarında replike (replicated) edilmiş değişkenlerimiz olduğunu varsayalım.

Widget'ın içerisinde **TestPawn** referansını ve **ProgressBar** oluşturduktan sonra can barı için aşağıdaki fonksiyonu bağlayabiliriz:





WidgetComponent'i ayarladıktan sonra bu Widget sınıfını, **BeginPlay'de HealthBar (Can Barı)** için kullanabiliriz:



BeginPlay, Pawn'ın **tüm** instance'larında (örneklerinde) çağrılır, yani Server ve tüm Client'larda. Böylelikle **her** instance Pawn, Widget'daki Pawn Referansına kendisini (self) kaydeder.

Pawn ve Health değişkeni replike (replicated) olduğundan , her Pawn'ın üstünde doğru sağlık yüzdesini görürüz.

Bu noktaya kadar **Replication** işlemi **kafanıza tam olarak oturmadıysa** sorun değil, **okumaya devam edin.** Dokümanın sonunda bu olayın nasıl bu kadar kolay çalıştığını anlayacaksınız!

C++ örneklerinde Widget Blueprint'leri tekrar oluşturmayacağım. C++ tarafında arka planda yapılması gereken çok iş var ve bunu burada ele almak istemiyorum. Bu yüzden burada **Possess (Sahip olma)** ve **Damage (Hasar)** Event'lerine odaklanacağız.

C++ dilinde iki Possess Event'i şu şekildedir:

```
virtual void PossessedBy(AController* NewController);

virtual void UnPossessed();
```

UnPossessed (Sahiplikten çıkma) Event'i beraberinde önceki (old) PlayerController'ı parametre olarak vermez.

```
/* Sınıf deklarasyonun yapıldığı, kendi Pawn Child Class'ımızın Header dosyası */
// SkeletalMesh Component, gizlemek istediğimiz bir durum
var
class USkeletalMeshComponent* SkeletalMesh;

//UnPossessed Event'inin override
edilmesi
 virtual void UnPossessed() override;
```

Ayrıca burada **MulticastRPCFunction'a** ihtiyacımız var. Bu konu hakkındaki bilgileri **RPC** Bölümü'nde okuyacaksınız:

```
/* Sınıf deklarasyonun yapıldığı, kendi Pawn Child Class'ımızın Header
dosyası */

UFUNCTION(NetMulticast, unreliable)
	void Multicast_HideMesh();
```

```
/* Kendi Pawn Child Sınıfımızın cpp
dosyası */

void ATestPawn::UnPossessed() {
	Super::UnPossessed();

	Multicast_HideMesh();
}

/* RPC konusunu ve neden '_Implementation' olduğunu daha sonra
anlayacaksınız */
void ATestPawn::Multicast_HideMesh_Implementation() {
	SkeletalMesh->SetVisibility(false);
}
```

Health örneğini C++ tarafında yapalım. Dediğim gibi, şu ana kadar Replikasyon (Replication) adımlarını anlamıyorsanız endişelenmeyin. **Gelecek** bölümler size bu konuyu açıklayacaktır.

Replikasyon (Replication) terimleri size karmaşık geliyorsa bu örnekleri atlayın.

TakeDamage fonksiyonu **EventAnyDamage** nodunun eşdeğeridir. Hasar aldığınızda Actor üzerinden **TakeDamage** fonksiyonu çağırılır ve hasar alma işlemi yönetilir. Eğer bu Actor fonksiyonu gerçeklemişse (implement) Actor, örnekteki gibi bu işleme tepki verir.

```
/* Sınıf deklarasyonun yapıldığı, kendi Pawn Child Class'ımızın Header dosyası */
//Replicated Health
değişkeni
UPROPERTY(Replicated)
      int32 Health;

//Damage Event'inin override edilmesi
virtual float TakeDamage(float Damage, struct FDamageEvent const& DamageEvent,
                                    AController* EventInstigator, AActor* DamageCauser) override;
```

```cpp
/* Kendi Pawn Child Sınıfımızın cpp dosyası */
//UPROPERTY Macro'sunda kullandığımız Replicated belirleyicisinden dolayı bu fonksiyon gereklidir.
void ATestPawn::GetLifetimeReplicatedProps(TArray<FLifetimeProperty>& OutLifetimeProps) const{

	Super::GetLifetimeReplicatedProps(OutLifetimeProps);

	//Değişkeni replike (replicated) etmek için
	DOREPLIFETIME(ATestPawn, Health);
}

float ATestPawn::TakeDamage(float Damage, struct FDamageEvent const& DamageEvent,
                            AController* EventInstigator, AActor* DamageCauser) {
	float ActualDamage = Super::TakeDamage(Damage, DamageEvent, EventInstigator, DamageCauser);

	Health -= ActualDamage;		//Oyuncunun Health değerini düşürmek için

	if(Health <= 0.0)		Health değişkeni 0'dan küçük veya eşitse yok et (destroy)
		Destroy();

	return ActualDamage;
}
```

Player Controller

(Bu sınıfın ayrıntılı API sayfasına gitmek için ismine tıklayınız)

APlayerController sınıfı, karşılaştığımız en ilginç ve karmaşık sınıf olabilir. Client'ın **sahip olduğu** ilk sınıf olduğu için birçok Client işleminde merkezi konumdadır.

PlayerController, Oyuncunun girdisi (input) olarak görülebilir. Oyuncunun Server'la arasındaki bağlantıdır. Bunun anlamı her Client bir adet PlayerController'a sahiptir.

Bir Client'ın PlayerController'ı sadece kendisinde VE Server'da bulunur fakat diğer Client'ların PlayerController'ı hakkında bilgi sahibi değildir.

Her Client sadece kendi PlayerController'ı hakkında bilgi sahibidir!

Bunun sonucunda, Server tüm Client'ların PlayerController'ının referansına sahiptir!

Girdi (Input) terimi; PlayerController'da olması gereken, doğrudan tüm asıl girdiler (Butona basmak, mouse hareketleri, kontrolcü eksenleri vb.) anlamına gelmez.

PlayerController; karakter nesnesi geçerli olmadığında (not valid) bile tüm karakterlerle çalışabilmektedir. Bundan dolayı Character/Pawn sınıflarınızdaki özel girdileri, (Arabalar insanlardan farklı çalışır) PlayerController'a yerleştirmek daha iyi olacaktır!

Önemli bir not:

Girdiler, her zaman önce PlayerController'dan geçer. Eğer PlayerController bu girdiyi KULLANMAZSA, bu işlem aynı girdiyi kullanabilecek diğer sınıflarda işlenir. İşlenen girdi ihtiyaç halinde deaktif (deactivate) edilebilir.

Ayrıca bilinmesi gereken önemli bir bilgi:

Doğru olan PlayerController nasıl alabilirim?

Sıkça kullanılan **GetPlayerController(0)** nodu ya da '**UGameplayStatics::GetPlayerController(GetWorld(), 0);** kod satırı Server'da ve Client'da farklı çalışır fakat bu o kadar da zor değildir.

- Listen-Server'da çalıştırmak: Listen-Server'ın PlayerController'ını döner
- Client'da çalıştırmak: Client'ın PlayerController'ını döner
- Dedicated Server'da çalıştırmak: İlk Client'ın PlayerController'ını döner

Client için '0' dışındaki diğer sayılar başka bir Client'ı dönmez. Bu index, burada anlatmayacağımız lokal Oyuncular içindir (Bölünmüş ekran=Splitscreen).

Örnekler ve Kullanımları

İçerisinde varsayılan çok fazla bir şey olmamasına rağmen, PlayerController Network için **en önemli** sınıflardan birisidir. Biz de PlayerController'a neden ihtiyacımız olduğunu anlamak için küçük bir örnek vereceğiz. **Ownership (Sahiplik)** bölümünde PlayerController'ın RPCs (Remote Procedure Calls) için neden önemli olduğunu anlayacaksınız. Aşağıdaki örnek, bir Widget butonuna basarak GameState'deki replike (replicated) bir değişkeni arttırmak için PlayerController'ın nasıl kullanıldığını gösterecektir.

Bunun için neden **PlayerController'a** ihtiyacımız var?

RPC ve **Ownership** bölümlerini tekrardan buraya yazmak istemiyorum, kısa bir açıklama yapacak olursak:

Widget'lar Client tarafından sahip olunsa bile, **sadece** Client'larda / Listen-Server'da bulunur. Bir Server RPC'sinin Server'da çalıştıracak bir örneği (instance) yoktur.

Basite indirgersek, Widget'lar replike **değildir!**

Butona basarak Server'daki değişkenin değerini arttırmak için bir yol bulmalıyız.

Neden direkt olarak RPC'i GameState üzerinden 'doğrudan' çağırmıyoruz?

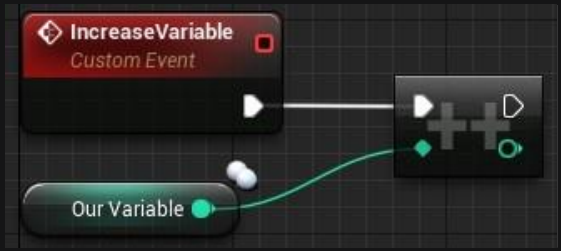
Çünkü onun sahibi (owner) Server'dır.

Bir Server RPC'si, sahip olarak Client'a ihtiyacı vardır!

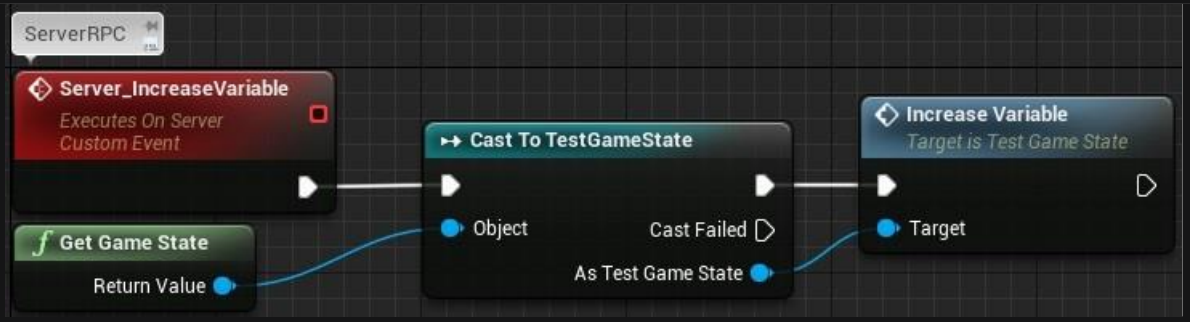
Blueprint

İlk olarak, basabileceğimiz bir **Button'u** olan **Widget'a** ihtiyacımız var. Bu bir **Network Özetidir**, bu yüzden lütfen Widget işlerini kendiniz öğreniniz. Bunu yapabileceğinizi biliyorum!

Ekran resimlerini **ters sırayla** vereceğim, hangi Event'lerin hangisini çağırdığını ve nerede bittiğini önceki resimlerde görebilirsiniz. Hedefimiz olan GameState'den başlayalım. GameState, replike (replicated) bir Integer (tam sayı) değişkenini arttıran Event'e sahiptir:



Bu Event PlayerController'ımızdaki ServerRPC'mizin içerisinde **Server tarafında** çağrılır:



Son olarak, **ServerRPC'ni** çağıran bir butonumuz var:

Böylece butona **(Client tarafındaki)** tıkladığımızda, Server tarafına ulaşmak için PlayerController'ımızdaki ServerRPC'yi kullanmış oluruz (bu mümkündür çünkü PlayerController'a Client **sahip!**). Daha sonra replike integer'ı (tam sayı) arttırmak için GameState'in **IncreaseVariable** Event'ini çağırırız.

Bu Integer (tam sayı), Server tarafından replike edildiği ve ayarlandığı için GameState'in tüm örneklerinde (instance) güncellenecek ve Client'lar da bu güncellemeyi görebilecektir!

Bu örneğin C++ versiyonunda Widget'ı, PlayerController'ın **BeginPlay'i** ile yer değiştireceğim. Bu çok mantıklı değil fakat Widget'ın C++ kodunda, burada göstermek istemediğim biraz daha fazla koda ihtiyacı var. Bu doküman için biraz fazla olacaktır!

```
/* Sınıf deklarasyonun yapıldığı, kendi PlayerController Child Class'ımızın Header dosyası */
/*Server RPC. RPC bölümünde bu konuda daha fazla bilgi
edineceksiniz */
UFUNCTION(Server, unreliable, WithValidation)
	void Server_IncreaseVariable();

// Bu örnek için BeginPlay fonksiyonunun override edilmesi
virtual void BeginPlay() override;
```

```
/* Sınıf deklarasyonun yapıldığı, kendi GameState Child Class'ımızın Header dosyası */
//Replike Integer değişkeni
UPROPERTY(Replicated)
	int32 OurVariable;

public:
	// Değişkeni arttıran fonksiyon
	void IncreaseVariable();
```

```
/* Kendi PlayerController Child Sınıfımızın cpp dosyası */
//Aksi takdirde GameState fonksiyonlarına erişemeyiz
#include "TestGameState.h"
/* RPC'ler ve '_Validate' şeyi hakkında daha sonra bilgi sahibi
olacaksınız */
bool ATestPlayerController::Server_IncreaseVariable_Validate() {
	return true;
}
/* RPC'ler ve '_Implementation' şeyi hakkında daha sonra bilgi sahibi
olacaksınız */
 void ATestPlayerController::Server_IncreaseVariable_Implementation() {
	ATestGameState* GameState = Cast<ATestGameState>(UGameplayStatics::GetGameState(GetWorld()));

	GameState->IncreaseVariable();
}
void ATestPlayerController::BeginPlay() {
	Super::BeginPlay();

	//* Bu PlayerController'ın Client versiyonunda ServerRPC'sini çağırdığından
	emin olun*/
	if(Role < ROLE_Authority)
		Server_IncreaseVariable();
}
```

```cpp
/* Kendi GameState Child Sınıfımızın cpp dosyası */
//UPROPERTY Macro'sunda kullandığımız Replicated belirleyicisinden dolayı bu fonksiyon gereklidir.
void ATestGameState::GetLifetimeReplicatedProps(TArray<FLifetimeProperty>& OutLifetimeProps) const {
	Super::GetLifetimeReplicatedProps(OutLifetimeProps);

	// Değişkeni replike (replicated) etmek için
	DOREPLIFETIME(ATestGameState, OurVariable);
}

void ATestGameState::IncreaseVariable(){
	OurVariable++;
}
```

Burada oldukça uzun bir kod var. Bazı işlevlerin kullanımını ve adlandırılmalarını henüz anlamıyorsanız, **endişelenmeyin**. **Gelecek** Bölümler, bunun neden böyle yapıldığını anlamanıza yardımcı olacaktır.

HUD

(Bu sınıfın ayrıntılı API sayfasına gitmek için ismine tıklayınız)

AHUD sınıfı, yalnızca her Client'da ayrı kullanılabilen ve PlayerController aracılığıyla erişilebilen bir sınıftır. Otomatik olarak yaratılır (spawned).

UMG (Unreal Motion Graphics) yayımlanmadan önce HUD sınıfı Client'larda metin yazma, resim ekleme ve daha fazlasını eklemek için kullanılmıştır.

Şimdi ise çoğunlukla Widget'lar, HUD sınıfın yerini almıştır.

HUD sınıfını hala hata ayıklamak (debug), Widget'ları oluşturma, gösterme, gizleme ve yok etme işlemlerini izole bir alanda yönetmek için kullanabilirsiniz.

HUD direkt olarak Network ile bağlantılı olmadığından, onları es geçeceğim. Örnekler yalnızca Tek Oyunculu (Singleplayer) işlemlerini göstermektedir.

Widgets (UMG)

(Bu sınıfın ayrıntılı API sayfasına gitmek için ismine tıklayınız)

Widget'lar Epic Games'in yeni UI Sistemini (User Interface System) kullanmaktadır. **U**nreal **M**otion **G**raphics.

UE 4 Editör'ünün kendisinde de kullanılan, C++ tarafında UI oluşturmak için kullanılan **Slate** sınıfından miras alan bir

sınıftır.

Widget'lar yalnızca lokal olarak Client'larda kullanılabilir (Listen-Server).

Replike **DEĞİLLERDİR** ve Replike sınıfların replikasyon işlemleri yapılırken bu işlemler daima ayrılmalıdır. Ör: Butona basmak

UMG ve Widget'lar hakkında daha fazla bilgi edinmek için lütfen yukarıda verilen API linkini kullanınız.

APawn örneklerinde Widget'ların kullanımı için küçük bir örnek vermiştir. Bu yüzden örnekleri burada atlayacağım.

Dedicated vs Listen Server

Dedicated Server

Dedicated Server, Client **GEREKTİRMEYEN** bağımsız bir Server'dır.

Game Client'ından ayrı olarak çalışır ve çoğunlukla oyuncuların her zaman katılabileceği / ayrılabileceği bir Server çalışması için kullanılır.

Dedicated Server'lar **Windows** ve **Linux** için derlenebilir ve Oyuncuların sabit IP adresi üzerinden bağlanabileceği **Sanal Server'larda (Virtual Servers)** çalıştırılabilir.

Dedicated Server'ların görsel bir elementi yoktur bu yüzden de ne bir UI'a ne de bir PlayerController'a ihtiyacı yoktur. Ayrıca Oyunda Dedicated Server'ı temsil eden Karakter veya benzeri elemanın hiçbirisi yoktur.

Listen-Server

Listen-Server aslında Client olan bir Server'dır.

Bunun anlamı Server'a her zaman bağlı en az **BİR** Client'a sahip olduğu anlamına gelir.

Bu Client Listen-Server olarak adlandırılır ve eğer oyundan çıkarsa Server kapanır.

Aynı zamanda bir Client olması nedeniyle Listen-Server UI'a ihtiyacı vardır ve Listen Server'ın Client tarafını temsil eden bir PlayerController'a sahiptir. Listen-Server'da **PlayerController(0)** kodunu çalıştırdığımızda tam da o Client'ın PlayerController'ını döndürür.

Listen-Server, Client'ın kendisi üzerinde çalıştığından dolayı, başka oyuncuların bağlanması gereken IP bu Client'lardan biridir. Dedicated Server ile karşılaştırıldığında, bu genellikle İnternet Kullanıcılarının statik (sabit) bir IP'ye sahip **OLMAMASI** sorununu beraberinde getirir.

Ancak **OnlineSubsystem** (daha sonra anlatılacak), bu değişen IP sorununu çözebilir.

Replication

Replication (Replikasyon) nedir?

Replication, Server'ın Client'lara bilgi / veri aktarma eylemidir.

Bu işlem belirli birimler (entities) ve gruplarla (groups) sınırlı olabilir. Blueprint'ler çoğunlukla etkilenen **AActor'un** ayarlarına göre replikasyon gerçekleştirir.

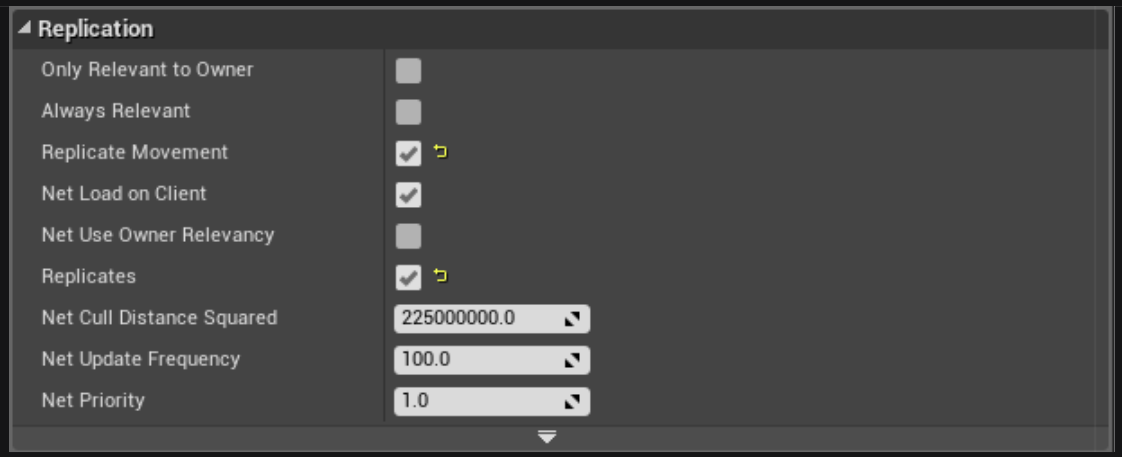
Replication niteliklerini (properties) kullanabilen ilk sınıf, Actor sınıfıdır.

Daha önce bahsedilen sınıfların tümü bir noktada Actor'den miras alır ve gerektiğinde nitelikleri replike etme olanağı verir.

Ama hepsi, bu işlemi aynı şekilde yapmaz.

Ör: GameMode, herkese replike **edilmez** ve **sadece** Server'da bulunur.

Replication nasıl kullanılır:

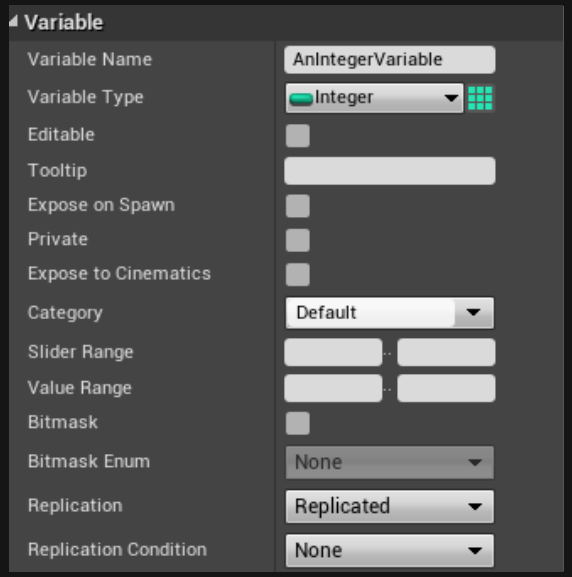


Replication, Actor'den türetilmiş child sınıfta **Class Defaults/Constructor'da (Sınıf Varsayılanları / Yapıcısında)** aktifleştirilebilir:

```
ATestCharacter::ATestCharacter() {
    //Network oyunu, Replication ayarı
    bReplicates = true;
    bReplicateMovement = true;
}
```

'**bReplicates**' değişkeni TRUE olarak ayarlanmış bir Actor, Server tarafında spawn edilirse bu Actor tüm Client'larda oluşturulur ve replike edilir. **SADECE** Server tarafından. Eğer bir Client bu Actor'ü spawn ederse, Actor **SADECE** o Client'da bulunur.

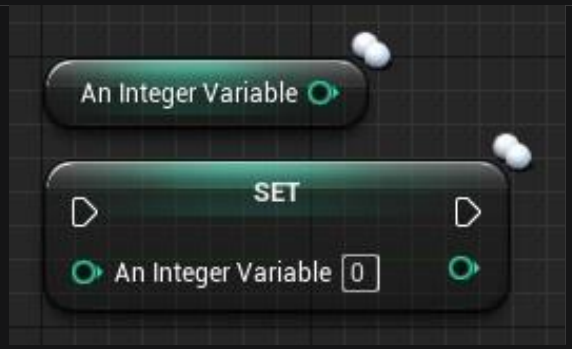
Replicating Nitelikleri



Replication'ı etkinleştirdiğimizde değişkenleri replike edebiliriz. Bunu yapmak için birçok yol vardır. Biz en kolay olanla başlayacağız:

"**Replication**" açılan menüsünden "**Replicated**" ayarını yapmak bu değişkeni, bu aktörün tüm instance'larında (örneklerinde) replike eder . Tabii ki, bu işlem sadece replike edilen aktörler için geçerlidir.

UE 4.14 sürümü ile değişkenler -Blueprint'lerin içerisi dahil- belirli koşullar altında replike edilebilir. Koşullar hakkında daha fazla bilgi edinmek için ileriki sayfalara gidiniz..



Replike edilmiş Değişkenler **2** beyaz **daire** ile belirtilmiştir.

C++ dilinde bir değişkeni replike etmek başlangıçta biraz daha uğraştırır. Fakat C++ aynı zamanda bu değişkeni kime replike etmek istediğimizi belirtmemize de izin vermektedir.

```
/* Sınıf deklarasyonun yapıldığı, Class'ımızın Header dosyası */
// Replike edeceğimiz Health değişkeni
UPROPERTY(Replicated)
float Health;
```

.cpp dosyası '**GetLifetimeReplicatedProps**' fonksiyonunu alır. Bu fonksiyonun Header deklarasyonunda bazı UE4 makrosu aracılığıyla zaten sağlanmıştır, bu sebeple gelen parametreleri önemsememize gerek yoktur. Tam olarak burada replike değişkenlerinizin kurallarını tanımlarsınız.

```
void ATestPlayerCharacter::GetLifetimeReplicatedProps(TArray<FLifetimeProperty>& OutLifetimeProps) const {
	Super::GetLifetimeReplicatedProps(OutLifetimeProps);

	//Burada replike etmek istediğimiz değişkenleri/koşulları listeliyoruz.
	DOREPLIFETIME(ATestPlayerCharacter, Health);
}
```

Koşullu replike işlemini bu şekilde yapabilirsiniz:

```
//Değişken, sadece bu nesnenin/sınıfın sahibine replike edilir.
DOREPLIFETIME_CONDITION(ATestPlayerCharacter, Health, COND_OwnerOnly);
```

Koşul	Açıklama
COND_InitialOnly	Bu nitelik yalnızca **ilk gruba** gönderilmeye çalışılır
COND_OwnerOnly	Bu nitelik yalnızca **Actor'ün sahibine** gönderilir
COND_SkipOwner	Bu nitelik sahibi **DIŞINDA** tüm bağlantılara gönderilir
COND_SimulatedOnly	Bu nitelik yalnızca **simulated** Actor'lere gönderilir
COND_AutonomousOnly	Bu nitelik yalnızca **autonomous** Actor'lere gönderilir
COND_SimulatedOrPhysics	Bu nitelik simulated YA DA **bRepPhysics** Actor'lere gönderilir
COND_InitialOrOwner	Bu nitelik **ilk pakette** veya **Actor'ün sahibine** gönderilir
COND_Custom	Bu niteliğin belirli bir koşulu yoktur fakat **SetCustomIsActiveOverride** ile açma/kapama özelliğini ister

Tüm bu replike etme işleminin yalnızca **Server'dan Client'a** çalıştığını ve tersi yönde **ÇALIŞMADIĞINI** anlamanız gerekmektedir.

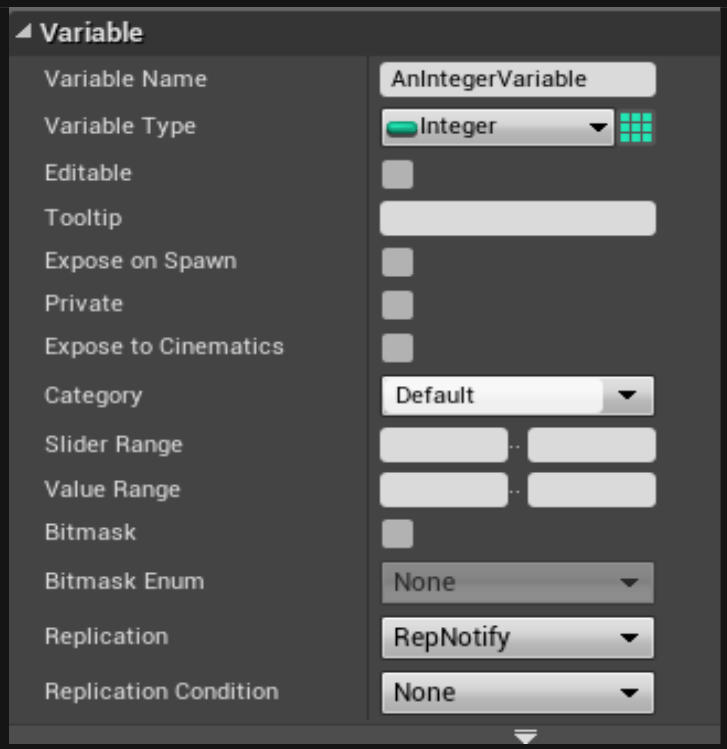
Client'ın başkalarıyla paylaşmak istediği bir şeyi, Server'ın bunu nasıl yapacağını daha sonra öğreneceğiz (Ör: **PlayerName**).

Replikasyon işleminin farklı bir sürümü, "**RepNotify**'dır.

Güncellenmiş bir değişeni alırken **tüm instance'larda** çağrılacak bir fonksiyon kullanır.

It makes use of a function that will be called on when receiving the updated value.

Bununla birlikte değişkenin değeri replike edildikten **SONRA** çağrılması gereken fonksiyonu çağırabilirsiniz.



Blueprint'de Replication menüsünden "**RepNotify**" seçtiğinizde, bu fonksiyon **otomatik olarak** oluşturulur:

C++ tarafında biraz daha fazla işlem vardır fakat aynı şekilde çalışıyor:

```
/* Sınıf deklarasyonun yapıldığı, Class'ımızın Header dosyası */
//RepNotify ayarlı Health değişken
UPROPERTY(ReplicatedUsing=OnRep_Health)
float Health;

//OnRep fonksiyonu | UFUNCTION() Macrosu önemlidir! | virtual though olmasına gerek yok
UFUNCTION()
virtual void OnRep_Health();
```

```
/* Class'ımızın cpp dosyası*/
void ATestCharacter::OnRep_Health(){
      if(Health < 0.0f)
        PlayDeathAnimation();
}
```

'**ReplicatedUsing=FUNCTIONAME**' ile Değişken başarıyla replike edildiğinde çağrılması gereken fonksiyonu belirtiriz.
Bu fonksiyonun içerisi boş olsa bile '**UNFUNCTION()**' macrosuna sahip olması gerekir!

Remote Procedure Calls

Replikasyon için bir başka yol ise "**RPC**"s. "**R**emote **P**rocedure **C**all" kısaltması.

Başka bir instance (örnek) üzerinde bir şey çağırmak için kullanılırlar. Televizyonunuzun Uzaktan Kumandası da Televizyonunuza aynı işlemi yapar.

Unreal Engine 4'de "**RPC**"ler, **Client'dan Server'a**, **Server'dan Client'a** ya da **Server'dan belirli gruba**

fonksiyon çağırmak için kullanılır.

RPC'lerin dönüş değeri (return value) olmaz! Bir şeyi return etmek için ikinci bir RPC kullanmanız gerekir.

Fakat bu işlem belirli kurallara göre çalışır.

Aşağıda resmi dokümantasyonda bulabileceğiniz liste bulunmaktadır:

- **Run on Server** – Actor'ün Server instance'ında (örneğinde) çalıştırmak içindir
- **Run on owning Client** – Bu Actor'ün sahibi üzerinde çalışır
- **NetMulticast** – Bu Actor'ün tüm instance'larında çalışır

Gereksinimler ve Uyarılar

RPC'lerin tamamen işlevsel olması için karşılanması gereken birkaç gereksinim vardır:

1. Actor'ler tarafından çağrılmalıdır.
2. Actor **replike (replicated)** olmalı.
3. Eğer RPC **Client'da** çalıştırılmak üzere **Server** tarafından çağrılıyorsa, sadece bu Actor'ün gerçek **sahibi** olan Client fonksiyonu çalıştırır.
4. Eğer RPC **Server'da** çalıştırılmak üzere **Client** tarafından çağrılıyorsa, Client'ın **RPC'nin** çağrıldığı Actor'e **sahip** olması gerekir.
5. **Multicast RPC'ler** istisnadır:
   - Eğer Server tarafında çağrılırsa, Server bunu lokal olarak çalıştırdığı gibi aynı zamanda **o anda bağlanmış Client'larda** da çalıştırır
   - Eğer Client'lardan çağrılırsa, sadece lokal olarak çalıştırılır ve Server'da çalıştırılmaz.
   - Multicast Event'ler için basit dar boğaz olma durumlarımız bulunuyor:
     - Bir Multicast fonksiyonu belirli bir işlevde iki defadan fazla replike edilmez.
     - Actor'ün network güncelleme periyodu. Uzun vadede bu konuda geliştirme bekliyoruz.

Server'dan çağrılan RPC

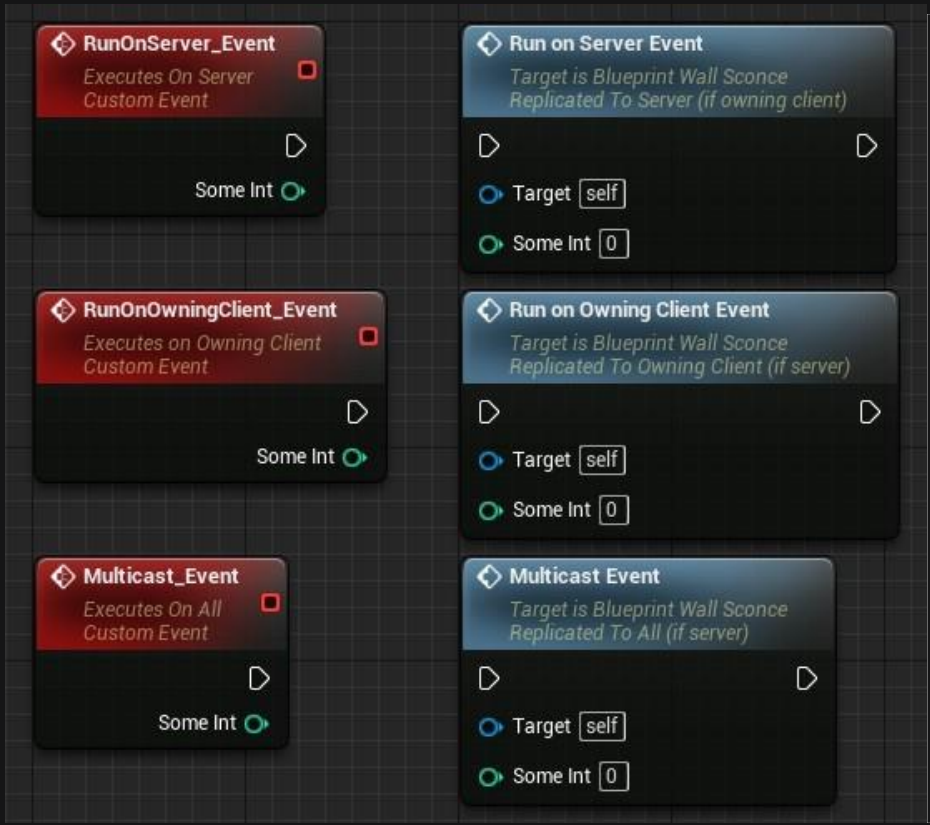
Actor Sahipliği (Ownership)	Replike değilken	NetMulticast	Server	Client
Client-sahipli Actor	Server'da çalışır	Server'da ve tüm Client'larda çalışır	Server'da çalışır	Actor'ün sahibi olan Client'da çalışır
Server-sahipli Actor	Server'da çalışır	Server'da ve tüm Client'larda çalışır	Server'da çalışır	Server'da çalışır
Sahipsiz Actor	Server'da çalışır	Runs onServer and all Clients	Server'da çalışır	Server'da çalışır

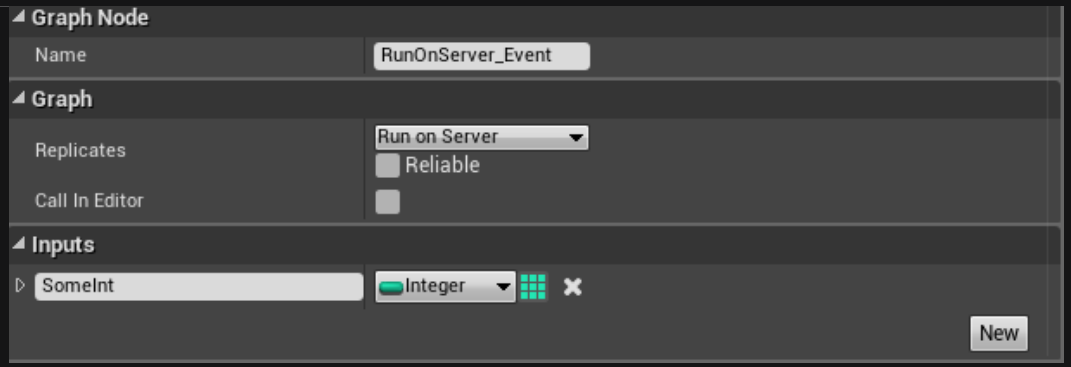
Bir Client'dan çağrılan RPC

Actor Sahipliği (Ownership)	Replike değilken	NetMulticast	Server	Client
Çağıran Client tarafından sahipli	Çağıran Client'da çalışır	Çağıran Client'da çalışır	Server'da çalışır	Çağıran Client'da çalışır
Başka Client tarafından sahipli	Çağıran Client'da çalışır	Çağıran Client'da çalışır	Çöker (Dropped)	Çağıran Client'da çalışır
Server-sahipli Actor	Çağıran Client'da çalışır	Çağıran Client'da çalışır	Çöker (Dropped)	Çağıran Client'da çalışır
Sahipsiz Actor	Çağıran Client'da çalışır	Çağıran Client'da çalışır	Çöker (Dropped)	Çağıran Client'da çalışır

Blueprint'lerde RPC

Blueprint'lerde RPC, **CustomEvents** oluşturarak ve onları **Replicate** yaparak oluşturulur. RPC'ler bir geri dönüş değerine (return value) sahip olamazlar! Bu yüzden onları fonksiyonlarla oluşturamayız.





Reliable check box'ı RPC'i **önemli** yapmak için kullanılır. Bunu işaretlersek, 99.99% çalıştırılacağından ve bağlantı sorunları nedeniyle düşmeyeceğinden (dropped) emin oluruz.

Not: Her RPC'i **Reliable** olarak **işaretlemeyin!**

C++ tarafında RPC'ler

Tüm Network işlemlerini yapmak için projenizin Header'ında "**UnrealNetwork.h**" kütüphanesini dahil etmeniz gereklidir! RPC'lerin C++ tarafında oluşturmak nispeten kolaydır. Sadece **UFUNCTION()** Macrosunu eklememiz gerekiyor.

```
// Bu Server RPC'dir, unreliable olarak işaretlenişve WithValidation (gerekli!)
UFUNCTION(Server, unreliable, WithValidation)
void Server_PlaceBomb();
```

CPP dosyası farklı bir fonksiyon oluşturur. Bunun da **_Implementation** sonuna ek olarak eklenmeli.

```
// Burası asıl gerçekleme kısmıdır (Server_PlaceBomb değil). Fakat çağırırken "Server_PlaceBomb" adıyla çağırırız
void ATestPlayerCharacter::Server_PlaceBomb_Implementation() {
    //BOOM!
}
```

CPP dosyasının **_Validate** son ekli fonksiyona da ihtiyacı vardır. Daha sonra göreceğiz.

```
bool ATestPlayerCharacter::Server_PlaceBomb_Validate() {
    return true;
}
```

Kalan iki çeşit RPC'ler şu şekilde oluşturulur:

'**reliable**' ya da '**unreliable**' olarak işaretlenmesi gereken **Client RPC**!

```
UFUNCTION(Client, unreliable)
void ClientRPCFunction();
```

ve '**reliable**' ya da '**unreliable**' olarak işaretlenmesi gereken **Multicast RPC**!

```
UFUNCTION(NetMulticast, unreliable)
void MulticastRPCFunction();
```

Tabii ki RPC'e '**reliable**' anahtar kelimesini de ekleyerek güvenilir hale getirebiliriz.

```
UFUNCTION(Client, reliable)
void ReliableClientRPCFunction();
```

Validation (C++)

Validation (Doğrulama) fikri şudur**:** Bir RPC için validation fonksiyonu parametrelerden herhangi birinin hatalı olduğunu algılarsa, RPC çağrısını oluşturan Client/Server'ın bağlantısını **kesilmesi** için sistem bilgilendirilir.

Validation artık tüm ServerRPC fonksiyonları için gereklidir. Bunun için **UFUNCTION** Macrosunda '**WithValidation**' anahtar sözcüğü kullanılır.

```
UFUNCTION(Server, unreliable, WithValidation)
void SomeRPCFunction(int32 AddHealth);
```

İşte '**_Validate**' fonksiyonunun nasıl kullanılabileceğine dair bir örnek:

```
bool ATestPlayerCharacter::SomeRPCFunction_Validate(int32 AddHealth) {
    if(AddHealth > MAX_ADD_HEALTH){
        return false;                    //Burası, çağıranın bağlantısını keser!
    }
    return true;                         // Burası, RPC'nin çağrılmasına izin verir!
}
```

Not: Client'dan Server'a RPC'ler, güvenli ServerRPC fonksiyonlarını desteklemek ve her parametrenin bilinen tüm girdi kısıtlamalarına karşı geçerli olup olmadığını kontrol ederken kod eklemeyi mümkün olduğunca kolaylaştırmak için '**_Validate**' anahtar sözcüğünü gerektirmektedir!

Ownership

Ownership anlamamız gereken önemli bir kavramdır. Yukarıdaki **Client-sahipli Actor** gibi şeyler içeren tabloyu zaten gördünüz.

Server ya da Client'lar bir Actor'e 'sahip olabilir' (own).

Ör: Client'ın sahip olduğu PlayerController (ya da Listen-Server).

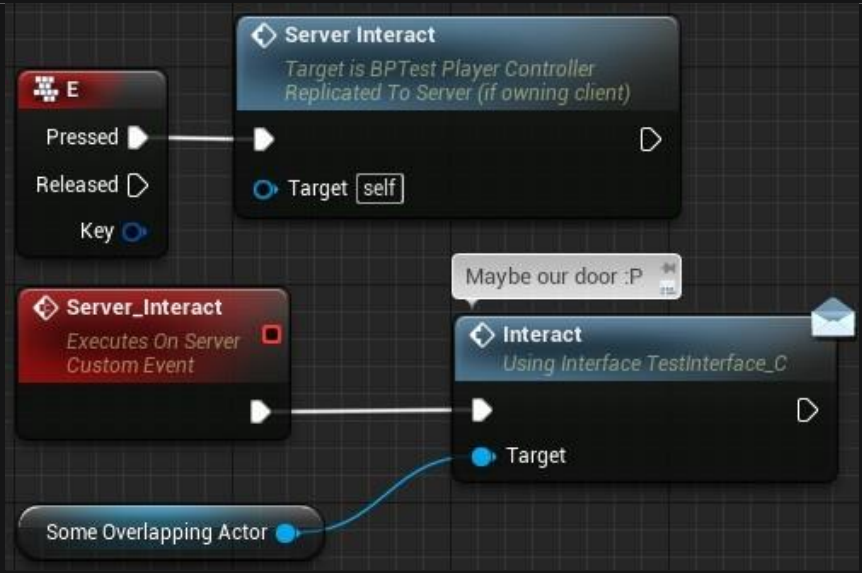
Başka bir Ör: Sahneye spawn edilmiş / yerleştirilmiş bir Kapı. Bu çoğunlukla Server'a ait olacaktır.

Bu neden bir problem?

Eğer tabloyu tekrar kontrol ederseniz; bir Client, kendisinin **SAHİP OLMADIĞI** bir Actor üzerinden ServerRPC çağırırsa bu işlemin çökeceğini (dropped) fark edeceksiniz.

Bu yüzden Client, Server'ın sahip olduğu kapı üzerinden **Server_OpenDoor**'u çağıramaz. Peki bunun üstesinden nasıl geleceğiz?

Client'ın gerçekten sahip olduğu Sınıfı/Actor'ü kullanırız. Bu da PlayerController'ın parlamaya başladığı yerdır.



Bu nedenle, kapı üzerinde Input'u etkinleştirmeye çalıştırmak için ServerRPC çağırmak yerine, PlayerController'da ServerRPC oluştururuz ve Server'ın kapı üzerinde bir Interface fonksiyonunu çağırmasına izin veririz. (Ör: **Interact()**).

Eğer Kapı doğru Interface'i gerçeklemişse (implement) daha önceden tanımlanmış lojiği çağırır ve açık/kapalı durumu doğru şekilde replike edilir.

Not: Interface'ler Multiplayer oyunlar için özel bir şey değildir ve onların ne olduklarını anlamıyorsanız şiddetle göz atmanızı tavsiye ederim!

Actor'ler ve Sahip Oldukları Bağlantılar (Owning Connections)

Ortak Sınıflar başlığında da belirtildiği gibi PlayerController, Oyuncunun aslında '**sahip olduğu**' ilk sınıftır. Fakat bu ne anlama geliyor?

Her **Bağlantının,** bu bağlantı için özel olarak oluşturulmuş bir PlayerController'ı vardır. Bu nedenle oluşturulan bir PlayerController, bu bağlantıya aittir. Dolayısıyla bir Actor'ün birine ait olup olmadığını belirlediğimizde, aslında **en dış** sahibine kadar sorgularız. Bu bir PlayerController ise o zaman PlayerController'ın sahibi olan Bağlantı, da o Actor'ün sahibidir.

Biraz basit mi, yoksa? Peki bunun için bir örnek ne olabilir?

Pawn/Character. Bunlar PlayerController tarafından sahip olunurlar (possessed). Tam da bu anda PlayerController, bu Pawn/Character'in sahibidir. Bunun anlamı bu PlayerController'a sahip olan Bağlantı, Pawn/Character'e de sahiptir.

Bu sadece PlayerController Pawn'a **SAHİPKEN (Possess)** geçerlidir. Sahiplik çıkma (Un-possessing) işlemi Client'ın artık Pawn'a sahip olmamasına neden olur.

Peki bu neden önemli ve bunun için neye ihtiyacım var?

- RPC'lerin **hangi** Client'ın bir Run-On-Client RPC çalıştıracağınız belirlemesi gerekiyor
- Actor **Replication** ve bağlantı **Relevancy (ilişkisi)**
- Actor'ün Sahibinin dahil olduğu, **Replication nitelikleri**

RPC'lerin sahip oldukları Bağlantı'ya (Connection) bağlı olarak Client/Server tarafından çağrıldığında farklı tepki verdiğini zaten gördünüz.

Ayrıca **Koşullu Replication,** C++ tarafında değişkenlerin yalnızca belirli bir koşul altında replike edildiğini de okudunuz.,

Aşağıdaki konu, Relevancy (İlgilik/İlişkilik) Bölümü'nü açıklamaktadır.

Actor Relevancy ve Priority

Relevancy (İlişki)

Relevancy nedir ve neden buna ihtiyacımız var?

Düşünün ki çok büyük Bölüm/Harita'ya sahip bir oyununuz var ve alan o kadar büyük ki Oyuncuların başka

oyunculara karşı **önemsiz** olabilecek duruma sahip**. A Oyuncusu B Oyuncusundan** kilometrelerce uzaktaysa

neden **B Oyuncusunun** mesh'ini görsün ki**?**

Bant genişliğini (bandwidth) arttırmak için **Unreal'ın Network Code'u**, Server'ın Client'lara yalnızca o Client'ın **ilişki kümesindeki (relevant set)** Actor'ler hakkında bilgi vermesini mümkün kılar.

Unreal, bir Oyuncu için **ilgili Oyuncu kümesini (relevant set)** belirlemek üzere aşağıdaki kuralları (sırayla) uygular. Bu testler **AActor::IsNetRelevantFor()** virtual function'da uygulanır.

1. Eğer Actor'ün **bAlwaysRelevant** açıksa Pawn ya da PlayerController tarafından sahipse, Pawn'ın ya da Instigator'ın bazı aksiyonları (Ör: ses ya da hasar) **ilişkilidir (relevant)**
2. Eğer Actor'ün **bNetUserOwnerRelevancy** açıksa ve bir **Sahibi (Owner)** varsa, **Sahibi'nin ilişkisini (Owner's relevancy)** kullanır.
3. Eğer Actor'ün **bOnlyRelevantToOwner** açıksa ve 1. maddedeki kontrolü geçemezse **ilişkili değildir (not relevant).**
4. Eğer Actor başka bir Actor'ün **İskeletine (Skeleton) bağlıysa (attached)**, **İlişkisi (Relevancy)** bağlı olduğu Actor'ün ilişkine göre belirlenir.
5. Eğer Actor gizliyse ('**bHidden == true**') ve **root component'i** başka bir Actor'le **çarpışmazsa, İlişkili değildir (not relevant).**
    - Eğer **root component** yoksa, **AActor::IsNetRelevantFor()** bir warning log'lar ve Actor'ün şu şekilde ayarlanması gerekip gerekmediğini sorar: '**bAlwaysRelevant = true**'
6. Eğer **AGameNetworkManager mesafeye dayalı ilişkiyi (based relevancy distance)** kullanacak şekilde ayarlamışsa, Actor **net cull distance** mesafesinden daha yakınsa **ilişkilidir (relevant).**

Not: Pawn ve PlayerController **AActor::IsNetRelevantFor()** fonksiyonunu override ederek ilişki düzeyi (relevancy) için farklı koşullara sahip olur.

Prioritization (Önceliklendirme)

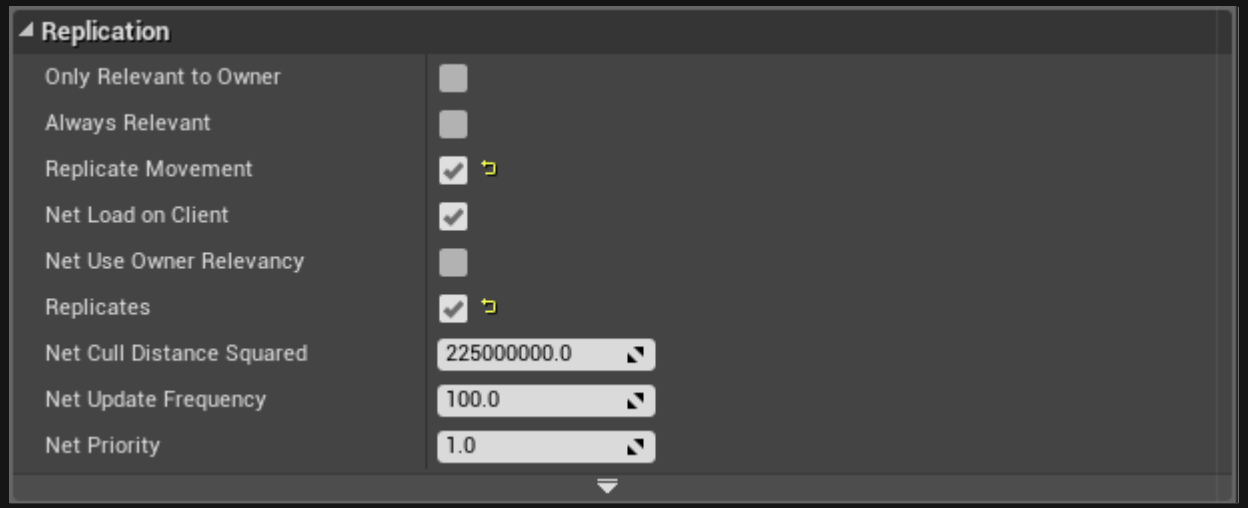
Unreal, **öncelik vermek (prioritizes)** için tüm Actor'lere **yük-dengeleme tekniğini (load-balancing technique)** kullanır ve Oyun için ne kadar önemli olduğuna bağlı olarak her birine bant genişliğini adil şekilde paylaştırır.

Actor'lerin **NetPriority** adında bir float değişkeni vardır. Bu sayının **yüksek** olması, bu Actor'ün diğer Actor'lerden **daha fazla bant genişliğine** sahip olduğunu ifade eder. **NetPriority** değişkeni **2.0** olan bir Actor**,** **NetPriority** değişkeni **1.0** olan diğer bir Actor'e göre tam olarak 2 kat daha sık güncellenir.

Öncelik vermede önemli olan **tek** şey onların **Oranıdır (Ratio).** Açıkçası, tüm Actor'lerin öncelik oranını arttırarak **Unreal'ın Network Performansını** arttıramazsınız.

Actor'ün anlık (current) öncelik değerini almak için **AActor::GetNetPriority()** virtual function kullanılır.
AActor::GetNetPriority() fonksiyonu, Actor'ün son replike edildiği andan şimdiye kadar geçen süre ile **NetPriority** değerini çarpar.

GetNetPriority fonksiyonu, **Actor** ve **İzleyici (Viewer)** arasındaki **göreli konumu (relative location)** ve mesafeyi de dikkat alır.



Bu **Ayarların** çoğu **Blueprint'in Class Defaults** sekmesinde bulunuyor. Tabii ki C++ tarafında her bir Actor'ün Child Sınıfında da ayarlanabilir.

```
bOnlyRelevantToOwner = false;
bAlwaysRelevant = false;
bReplicateMovement = true;
bNetLoadOnClient = true;
bNetUseOwnerRelevancy = false;
bReplicates = true;

NetUpdateFrequency = 100.0f;
NetCullDistanceSquared = 225000000.0f;
NetPriority = 1.0f;
```

Actor Role ve RemoteRole

Actor Replikasyonu için iki **önemli özelliğimiz** daha var.

Bu iki özellik siz şunları söyler:

- Actor üzerinde kimin **Yetkisi** var **(Authority)**
- Actor'ün **replike (replicated) olup** veya **olmadığı**
- Replikasyon **Modu**

Belirlememiz gereken ilk şey, belirli bir Actor üzerinde kimin Yetkisinin (Authority) olduğudur.

Bunu belirlerken, Oyun Motor'un şu an çalışan instance'ı yetkiye sahip olup olmadığını belirlemek için the **Rol** özelliğinin **ROLE_Authority** olup olmadığını kontrol edin. Eğer yetkisi varsa, o zaman Oyun Motor'un bu instance'ı bu Actor'den sorumludur. (replike olsun ya da olmasın)

Not: Bu Sahiplik (Ownership) ile aynı şey değildir!

Role/RemoteRole Reversal

Role ve **RemoteRole** bu değerleri kimin kontrol ettiğine bağlı olarak birbirini tersine çevrilebilir.

Örneğin Server'da şu konfigürasyona sahipseniz:

- Role == Role_Authority
- RemoteRole = ROLE_SimulatedProxy

Client bunu şöyle görecektir:

- Role == ROLE_SimulatedProxy
- RemoteRole == ROLE_Authority

Server, Actor'den sorumlu olduğundan ve Client'lara replike ettiği için bu mantıklıdır.

Client'lar sadece güncellemeleri almakla ve güncellemeler arasında Actor'ü simüle etmesiyle yükümlüdür.

Replication Modu

Server her güncellemede Actor'leri güncellemez. Bu çok fazla bant genişliği (bandwidth) ve CPU kaynağı gerektirecektir. Bunun yerine Server, **AActor::NetUpdateFrequency** Instead, niteliğinde belirlenen bir frekansta Actor'leri replike eder.

Bunun anlamı, Actor'ün güncellemeleri arasında Actor Client üzerinde biraz zaman geçirir. Bu, Actor'lerin hareketlerinde düzensiz veya değişken görünmelerine neden olabilir. Bunu telafi etmek için Client, güncellemeler arasında Actor'ü simüle eder.

Bunu oluşturan iki simülasyon tipi vardır:

ROLE_SimulatedProxy

ve

ROLE_AutonomousProxy

ROLE_SimulatedProxy

Bu tür standart simülasyon yoludur ve genellikle bilinen son hıza dayalı hareketi tahmin etmeye (extrapolating) dayanır bu işlemi yapar.

Server belirli bir Actor için bir güncelleme gönderdiğinde Client bu Actor'ün konumunu yeni konuma doğru düzenler (adjust). Ardından Client, güncellemeler arasında bu Actor'ü Server'dan gönderilen en son hıza göre hareket ettirmeye devam eder.

Bilinen son hızı kullanarak simüle etmek, genel simülasyonun nasıl çalıştığının sadece bir örneğidir.

Server güncellemeleri arasında başka bir bilgiyi tahmin etmek (extrapolate) için kendi kodunu yazabilirsin. Buna engel bir durum söz konusu değil.

ROLE_AutonomousProxy

Bu tür genellikle yalnızca PlayerController tarafından sahip olunan Actor'lerde kullanılır.

Bunun anlamı, bu Actor'ün gerçek bir İnsan tarafından girdi aldığıdır. Eksik bilgiyi doldurmak için gerçek insanın girdilerini kullanırız ve hareketi tahmin ederken biraz daha fazla bilgiye sahip oluruz (bilinen son hıza göre tahmin etmeye göre).

Multiplayer'da Traveling

Non-/Seamless travel

Seamless (Kesintisiz) ve **Non-seamless (Kesintisiz Olmayan)** yolculuk arasındaki fark basittir.

Seamless (Kesintisiz) yolculuk engellenmeyen bir işlemdir. **Non-seamless (Kesintisiz Olmayan)** ise bunu engelleyen bir çağrı olacaktır.

Bir Client için **Non-seamless (Kesintisiz Olmayan)** yolculuk, Server'dan bağlantısının koptuğu ve daha sonra aynı Server'a yeni Harita yüklendiğinde yeniden bağlandığı anlamında gelir.

Epic Game travel işleminin mümkün olduğunca **Seamless (Kesintisiz)** kullanılmasını önermektedir. Böylece daha pürüzsüz, akıcı bir deneyim sağlayacaktır. Yeniden bağlanma işlemi sırasında oluşabilecek sorunlardan kaçılacaktır.

Non-seamless (Kesintisiz Olmayan) travel işleminin gerçekleşmesi için 3 yol vardır:

- Haritayı ilk kez yüklerken
- Bir Server'a ilk kez Client olarak bağlanırken
- Multiplayer bir oyunu bitirip ve yeniden bir oyuna başlamak istediğinizde

Temel Traveling Fonksiyonları

Travel işlemini sağlayan 3 ana fonksiyon vardır:

UEngine::Browse

- Yeni harita yüklenirken hard reset yapılacak mı
- Her zaman **Non-seamless (Kesintisiz Olmayan)** bir travel ile sonuçlanacaktır
- Hedef haritaya gitmeden önce Server'ın o anki (current) Client'ların bağlantısını kesmesine neden olur
- Client'lar o anki (current) bağlı olduğu Server'dan bağlatıyı kesecektir
- Dedicated Server diğer Server'lara travel işlemini yapamaz. Bu nedenle haritanın lokal olması gerekir (URL olamaz).

UWorld::ServerTravel

- Yalnızca Server için
- Server, yeni bir World/Level'a atlayacak mı
- Tüm bağlı Client'lar Server'ı takip eder
- Bu, Multiplayer oyunların haritadan haritaya geçiş yoludur ve Server bu fonksiyonu çağırmaktan sorumludur
- Server, bağlı olan tüm Client Oyuncuların **APlayerController::ClientTravel** fonksiyonunu çağırır

APlayerController::ClientTravel

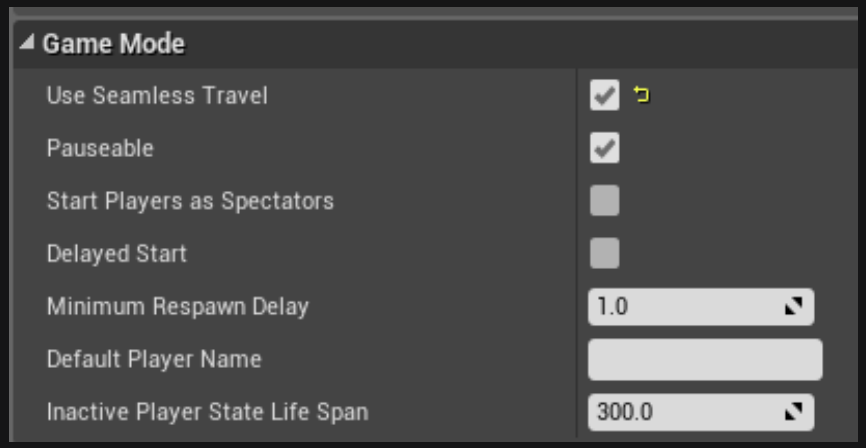
- Eğer bir Client tarafından çağrılırsa, **yeni** bir Server'a travel işlemi olur
- Eğer bir Server tarafından çağırılırsa, belirli Client'a yeni haritaya gitmesini (Ancak o anki Server'a bağlı kalmasını) bildirir

Seamless Travel Aktifleştirmek

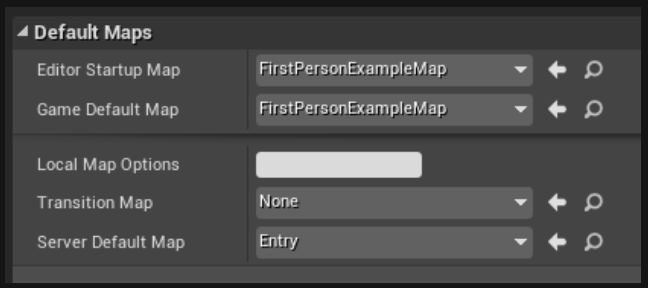
Seamless (Kesintisiz) travel, **Transition Map (Geçiş Haritası)** ile birlikte gelir. Bu ayar, **UGameMapsSettings::TransitionMap** özelliği aracılığıyla yapılandırılır. Varsayılan olarak bu özellik boştur. Oyununuzda bu özelliği boş bırakırsanız, motor bunun için boş bir harita oluşturur.

Transition Map (Geçiş Haritasının) var olmasının nedeni, her zaman yüklü bir World olması gerektiğidir (Haritayı tutan). Bu yüzden yenisini yüklemeden önce eski haritayı bırakamayız. Haritaların boyutu çok olabileceğinden dolayı eski ve yeni haritanın aynı bellekte olması kötü bir fikir olacaktır. Bu nedenle Geçiş Haritası kullanılmaktadır.

Transition Map (Geçiş Haritası) ayarını yapmak için **AGameMode::bUseSeamlessTravel** ayarını true işaretlersiniz. Böylece **Seamless (Kesintisiz)** travel çalışır!



Bu ayar tabii ki GameMode Blueprint'inden ve **Project Settings'deki** "**Maps and Nodes**" Tabından ayarlanabilir.



Persisting Actors / Seamless Travel

Seamless (Kesintisiz) travel kullandığınızda Actor'leri o anki (current) Level'dan yeni Level'a taşımak (persist: devam etmek, sürdürmek) mümkündür. Bu, Envanter Öğeleri, Oyunlar vb. gibi belirli Actor'ler için kullanışlıdır.

Varsayılan olarak bu Actor'ler otomatik olarak taşınır (persist):

- GameMode Actor'ü (Sadece Server)
  - **AGameMode::GetSeamlessTravelActorList** ile eklenen Actor'ler
- Geçerli **(valid)** bir PlayerState'e sahip tüm Controller'lar (Sadece Server Server)
- Tüm PlayerController'lar (Sadece Server)
- Tüm **lokal** PlayerController'lar (Server ve Client)
  - **Lokal** PlayerController'larda çağrılan **APlayerController::GetSeamlessTravelActorList** ile eklenen tüm Actor'ler

Seamless travel gerçekleştirirken genel akış:

1. Transition Level'a (Geçiş Haritası'na) taşınacak Actor'leri işaretleyin (yukarıda okuyun)
2. Transition Level'a (Geçiş Haritası'na) travel işlemini yapın
3. Final Level'a taşınacak Actor'leri işaretleyin (yukarıda okuyun)
4. Final Level'a travel işlemini yapın

Online Subsystem Genel Bakış

Online Subsystem (Çevrimiçi Alt Sistem) ve **Interfaces (Arayüzleri)** exist belirli bir ortamda mevcut platformlar kümesi genelinde ortak çevrimiçi fonksiyonellik için temiz bir soyutlama (abstraction) sağlamak için vardır.
Bu bağlamdaki **Platformların; Steam**, **Xbox Live**, **Facebook**, vb. ana hedeflerinden biri **Taşınabilirlik'tir.**

Ne için bir Subsystem'e ihtiyacım var?

Varsayılan olarak **SubsystemNULL** kullanıyor olacaksınız. Bu **LAN** Oturumlarını (Sessions) host'lamanıza (Böylece Oturumları, bir Server Listesi aracılığıyla bulabilir ve LAN ağınızda bunlara katılabilirsiniz) veya direkt **IP** olanak üzerinden katılmanıza olanak tanır. Bu ayarla birlikte Internet üzerinden Oturum açmanıza izin verilmez.

Neden?

Çünkü Client'lara Server'ların/Oturumların listesini sağlayan **Master Server'ınız** yok.

Alt Sistemler (Ör: Steam) Internet üzerinden görülebilen Server'ları/Oturumları da host'lamanıza izin verecektir. Bu kısmı anlamak önemlidir!

Kendi Alt Sisteminizi / Master Server'ınızı da oluşturabilirsiniz fakat bu UE4 dışında çok fazla kodlama gerektirmektedir.

Online Subsystem Modülü

Temel Tasarım

Temel modül **Online Subsystem,** platforma özgü modüllerin Engine ile nasıl tanımlandığını ve kaydedildiğini düzenlemekten sorumludur. Bir platform servisine tüm erişim, bu modülden geçecektir. Modül yüklendiğinde, bu modül **Engine.ini** dosyasında belirtilen varsayılan platform server modülünü (DefaultPlatformService) yüklemeye çalışacaktır:

```
[OnlineSubsystem]
DefaultPlatformService = <Default Platform Identifier>
```

Doğru bir şekilde çalışırsa; hiçbir parametre belirtilmediğinde varsayılan çevrimiçi arayüz, (default online interface) static accessor (statik erişimci) aracığıyla kullanılabilir.

```
static IOnlineSubsystem* Get(const FName& SubsystemName = NAME_None);
```

Uygun bir tanımlayıcısı olan fonksiyondan çağrıldığında, isteğe bağlı olarak ek hizmetler yüklenir.

Delegate'lerin Kullanımı

Online Subsystem, **Unreal Engine 3'e (UE3)** benzer olarak **Asenkron (Asynchronous)** yan etkileri olan fonksiyonları çağırırken **Delegate'lerden** yoğun şekilde yararlanacaktır. Delegate'lere uyum sağlamak ve işlem zincirindeki fonksiyonları çağırmadan önce uygun Delegate'in çağrılmasını beklemek önemlidir.

Asenkron bir task'ın (görev) beklenmemesi çökmelere, beklenmeyen tanımlanmamış davranışlara neden olabilir. Delegate'leri beklemek, özellikle kablonun çekilmesi gibi bağlantı hataları veya diğer bağlantı kopmaları sırasında önemlidir. Bir task'ın tamamlanması için gereken süre ideal durumda görünebilir ancak zaman aşımı durumunda bu süre neredeyse bir dakikaya kadar çıkabilir.

Delegate arayüzü oldukça basittir. Her Delegate, her arayüz header'ının en üstünde açıkça tanımlanmıştır. Her Delegate'in, bir **Add (Ekle)**, **Clear(Sil)**, ve **Trigger (Tetikleme)** fonksiyonu vardır. (Her ne kadar Delegate'leri **maneul olarak tektiklemek önerilmese de**).

Yaygın kullanım, uygun fonksiyonu çağırmadan hemen önce **Delegate'i Add()** ile eklemek daha sonra **Clear()** ile **Delegate'i** kendisiyle silmektir.

Interface'ler

Tüm platformlar tüm **Arayüz'leri (Interface'ler)** uygulamaz (implement) ve Oyun Kodları buna göre planlanmalıdır.

Profile

Çevrimiçi servislerin **Profil** servisleri için arayüz tanımıdır. Profil servisleri, belirli bir **Kullanıcı Profili** ve ilişkili meta verileriyle (metadata) (**Çevrimiçi Durumu**, **Erişim İzinleri**, vb.) ilgili herhangi bir şey olarak tanımlanır.

Friends

Çevrimiçi servislerin **Arkadaşlar** servisleri için arayüz tanımıdır. Arkadaş servisleri, Arkadaş ve Arkadaş Listelerinin İdamesi ile ilgili her şeydir.

Sessions

Çevrimiçi servislerin Oturum servisleri için arayüz tanımıdır.

Oturum servisleri, bir Oturumu ve durumu yönetmekle ilgili herhangi bir şey olarak tanımlanır.

Shared Cloud

Halihazırda bulutta bulunan **dosyaları paylaşmak** için Arayüz sağlar (diğer kullanıcılarınkine User Cloud bakınız).

User Cloud

Kullanıcı başına Bulut dosya depolama için Arayüz sağlar

Leaderboards

Çevrimiçi Skor Tablolarına erişmek için bir Arayüz sağlar.

Voice

Oyun sırasında ağ üzerinden **Voice Communication (Sesli İletişim)** için Arayüz sağlar.

Achievements

Başarımları **Okumak**/**Yazmak**/**Kilidini açmak** için Arayüz sağlar.

External UI

Belirli bir platformun **harici Arayüzlerine (external Interfaces)** erişmek için Arayüz sağlar. (Eğer varsa)

Sessions ve Matchmaking

Matchmaking, Oyuncuları Oturumlarla (Sessions) eşleştirme işlemidir. Bir Session temel olarak; belirli bir özellik kümesiyle Server'da çalışan ve Oyunu oynamak isteyen Oyuncular tarafından bulunabilmesi ve katılabilmesi için **bildirilen (advertised)** ya da **özel (private),** yani yalnızca davet edilen Oyuncular olan bir Oyunun Instance'ıdır (Örneğidir).

Şu anda oynanan tüm Oyunları listeleyen bir **Online Oyun Lobisi** hayal edin.

Listedeki her Oyun bir Session (Oturum) veya **ayrı (individual)** çevrimiçi maçtır. Oyuncular, aramayla veya başka yollarla Session'larla (Oturumlarla) eşleştirilir ve daha sonra maçı oynamak için Session'a (Oturuma) katılır.

Bir Session'ın Temel Yaşam Süresi

- **Create:** İstediğiniz ayarlarla yeni bir Session oluşturun
- **Wait:** Oyuncuların maça katılmak istemesini bekleyin
- **Register:** Katılmak isteyen Oyuncuları kaydedin
- **Start:** Session'ı başlatın
- **Play:** Maçı oynayın
- **End:** Session'ı bitirin
- **Un-register:** Oyucuların kaydını kaldırın. Ya da:
  - **Update:** Maç türünü değiştirmek ve Oyuncuların katılmasını **beklemeye (waiting)** geri dönmek istiyorsanız Session'ı güncelleyin
  - **Destroy:** Session'ı yok edin

Session Interface

Session Interface (Oturum Arayüzü), **IOnlineSession**, Oyuncuların çevrimiçi oyunları bulmasını ve katılmasına izin veren diğer yöntemlerin yanı sıra, Matchmaking yapmak için gerekli olan sahnenin arkasındaki işleri ayarlamak için platforma özel fonksiyonlar sağlar.

Buna **Session Management (Oturum Yönetimi)**; Session arama veya başka yollarla Session bulma, bu Session'lara katılma ve bu Session'lardan ayrılma da dahildir.

Session Interface, Online Subsystem tarafından oluşturulur ve ona aittir. Bu, yalnızca Server'da var olduğu anlamına gelir.

Bir seferde yalnızca bir Session Interface var olacaktır ve bu Engine'in şu anda üzerinde çalıştığı platformun Session Interface'idir. Session Interface tüm Session işlemlerini gerçekleştirirken, Oyun genellikle direkt olarak onunla etkileşime girmez.

Bunun yerine GameSession -**AGameSession**-, Session Interface etrafında oyuna özel bir sarmalayıcı görevi görür ve oyun kodu, Session ile etkileşime girmesi gerektiğinde ona çağrı yapar.

GameSession, GameMode tarafından oluşturulur ve ona aittir ve ayrıca sadece çevrimiçi bir oyun çalıştırılırken Server'da bulunur.

Her oyun potansiyel olarak birden fazla GameSession türü olabilir fakat aynı anda yalnızca biri kullanılacaktır.

Birden fazla GameSession türüne sahip bir oyun için en yaygın durum, oyunun Dedicated Server kullandığı zamanlar için bir GameSession türü eklemektir.

Session Settings

SessionSettings, **FOnlineSessionSettingsclass** tarafından tanımlanan oturumun özelliklerini belirleyen bir dizi özelliktir.

Temel uygulamada (implementation) bunlar aşağıdaki gibidir:

- İzin verilen Oyuncu sayısı
- Session bildirilmiş (advertised) ya da özel (private) mi
- Session bir LAN maçı mı
- Server Dedicated mı yoksa Listen-Server mı (Player-hosted)
- Davetlere izin veriliyor mu
- Vb.

Çevrimiçi Oyun Lobi örneğini kullanırsak, bu oyunların her biri bir Session'dır ve kendi Session Ayarları'na sahiptir. Ör: Bazı Session'lar, **Player versus Player (Oyuncuya karşı Oyuncu)** (PvP), diğerleri ise **Cooperative Multiplayer** (Co-Op) olabilir.

Farklı Session'lar farklı haritalar veya çalma listeleri (playlists), farklı sayıda oyuncu gerektirebilir vb. olabilir.

Session Management

Tüm Session Node'ları **Asenkron (Asynchronous)** task'lardır (Saat sembolü) ve tamamlandıktan sonra "**OnSuccess**" ya da "**OnFailure**" çağırırlar. Bu arada, en üstteki exec çıktısını da kullanabilirsiniz.

Creating Sessions

Oyuncuların bir Session bulup potansiyel olarak ona katılabilmeleri için bir Session oluşturup, özelliklerini ayarlamanız ve bu özelliklerden hangisinin görünür olacağına karar vermeniz gerekir. Blueprint aracılığıyla Dedicated Server Session kaydedilmesi, kullandığınız Engine sürümünde desteklenmeyebilir. C++ tarafında **RegisterServer** fonksiyonunu override ederek AGameSession içinde normal C++ Session oluşturma yapmalısınız!

Blueprint ile Session Oluşturma

Blueprint'de Session oluşturmak için, Epic'in halihazırda node'u **CreateSession**'ı kullanabilirsiniz. Bu node çok fazla seçenek sunmuyor fakat **Forum'daki Plugin** ile seçenekleri genişletebilirsiniz.

C++ tarafında Session Oluşturma

C+ ile Session kullanma hakkındaki Wiki postum!

Updating Sessions

Mevcut bir Session Ayarlarını değiştirmek istediğinizde, IOnlineSession::UpdateSession() fonksiyonunu kullanarak güncelleştirme yapabilirsiniz.

Ör: Session, bir sonraki maç için 12 Oyuncuya izin vermesi gerekirken, şu anda sadece 8 Oyuncuya izin verecek şekilde ayarlanmış olabilir. Session'ı güncellemek için UpdateSession() fonksiyonu, maksimum 12 Oyuncu belirten yeni Session Ayarları geçirilerek çağrılır.

Bir Session'ı güncelleme isteği tamalandığında, OnUpdateSessionComplete delegate'i tetiklenir. Bu, Session ayarlarının değiştirilmesini işlemek için gereken yapılandırma veya başlatma işlemlerini gerçekleştirme olanağı sağlar.

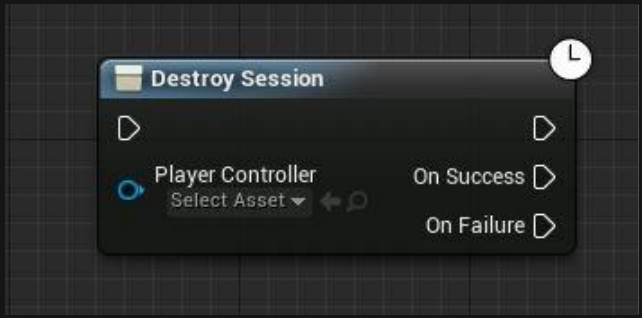
Bir Session'ın güncellenmesi genellikle Server üzerinde Maçlar arasında gerçekleştirilir ancak Session bilgilerini senkronize tutmam için Client'da da gerçekleştirilir.

Bu fonksiyonun henüz bir Blueprint Versiyonu yok ancak Forum'daki Plugin aracılığıyla bu işlemi kullanabilirsiniz.

Destroying Sessions

Bir Session sona erdiğinde veya artık ihtiyaç duyulmadığında, IOnlineSession::DestroySession() fonksiyonu kullanılarak yok edilir. Yok edilme işlemi tamamlandığında temizleme işlemlerini gerçekleştirmenizi sağlayan OnDestroySessionComplete delegate'i tetiklenir.

Blueprint ile Destroy Session



Blueprint tarafı için, Epic'in halıhazırda node'u DestroySession kullanabilirsiniz.

C++ ile Destroy Session

C+ ile Session kullanma hakkındaki Wiki postum!

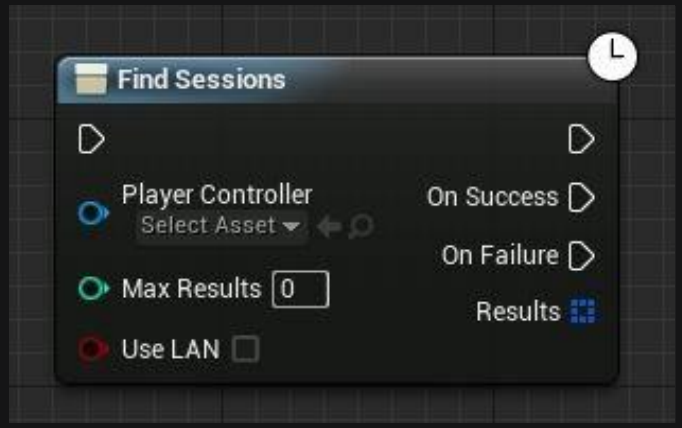
Searching Sessions

Session bulmanın en basit yolu, istenen bazı Ayar kümeleriyle eşleşen Session'ları aramaktır.

Bu işlem; Oyuncunun UI'ında (UI: Kullanıcı Arayüzü) bir filtre koleksiyonu seçmesiyle, Oyuncunun becerisine ve diğer faktörlere bağlı olarak sahnelerin arkasında otomatik olarak yapılabilir ya da her iki yöntemin bir kombinasyonu olabilir.

Session aramanın en temel şekli, mevcut oyunları gösteren ve Oyuncunun oynamak istediği oyun türüne göre filtrelemesine olanak veren klasik Server Tarayıcısıdır (Server Browser).

Blueprint ile Searching Session



Blueprint tarafı için, Epic'in halıhazırda node'u **FindSessions** kullanabilirsiniz. Aranacak Sessionların maksimum sayısını ve LAN veya Online Oyunlar aramak isteyip istemediğinizi belirtebilirsiniz.

Daha fazla ayar için **Forum'daki Plugin** kullanılabilir.

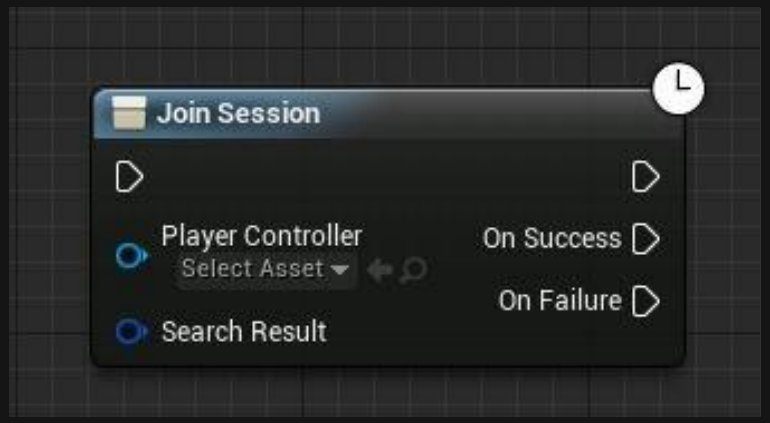
C++ ile Searching Session

C+ ile Session kullanma hakkındaki Wiki postum!

Joining Sessions

Oyuncunun katılacağı bir Session belirledikten sonra, **IOnlineSession::JoinSession()** fonksiyonuna **Oyuncunun Numarasını, Adını,** ve **Arama Sonucunu (Search Result)** parametrelerini girerek katılma işlemi başlatılır. Katılma işlemi tamamlandığında **OnJoinSessionComplete** Delegate'i tetiklenir. Burası Oyuncuyu Maç'a katılma mantığının gerçekleştiği yerdir.
Bu mantık hakkında daha fazla bilgiyi aşağıdaki Session **C++ Blog-Post'umda** bulabilirsiniz.

Blueprint ile Joining Session



Blueprint tarafı için, Epic'in halıhazırda kullandığı **JoinSession** node'unu kullanabilirsiniz. **FindSession** node'undan elde ettiğiniz bir SearchResult'a ihtiyaç duymaktadır.

FindSession size SearchResult dizisini döndürür. **JoinSession** nodu **OnSuccess** çıkışı ile doğrudan haritaya katılır. Bu kısımla uğraşmak zorunda değilsiniz.

C++ ile Joining Session

C+ ile Session kullanma hakkındaki Wiki postum!

Bulut-tabanlı Matchmaking

Bulut-tabanlı Matchmaking, mevcut ve genellikle platforma özgü **Matchmaking Servislerini** ifade eder. Bu servise örnek olarak, **Microsoft Xbox Live Servisi'nden** erişilebilen **TrueSkill Sistemi'ni** verebiliriz.

Bunu destekleyen Platformlarda Matchmaking'i başlatmak için, **IonlineSession::Startmatchmaking()** fonksiyonunu çağırıp parametre olarak da Oyuncu'nun Controller numarası, Session adı, yeni bir Session oluştururken kullanılacak Session Ayarları ve Arama Ayarlarını verirsiniz. Matchmaking tamamlandığında **OnMatchmakingComplete** Delegate'i tetiklenir. Bu, işlemin başarılı olup olmadığını ve bu durumda katılacak Session'ın adını belirten bir bool değişken verir.

Devam etmekte olan bir Matchmaking eylemi, **IOnlineSession::CancelMatchmaking()** fonksiyonunu çağırarak ve ilk etapta Matchmaking'i başlatmak için kullanılan fonksiyona parametre olarak gelen Oyuncu'nun Controller numarasını ve Session adını kullanarak iptal edebilir.

İptal işlemi tamamlandığında **OnCancelMatchmakingCompletedelegate** Delegate'i tetiklenir.

Following ve Inviting Friends

Arkadaş kavramını destekleyen platformlarda, Oyuncular bir Session'da **arkadaşlarını takip edebilir** veya arkadaşlarının bir Session'a katılması için **arkadaşlarına istek atabilir.**

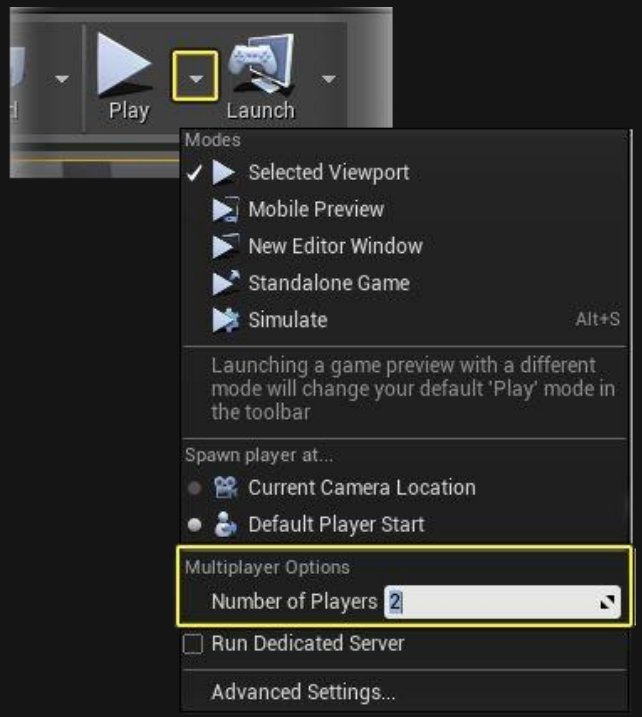
Session'dayken bir arkadaşınızı takip etmek için **IOnlineSession:FindFriendSession()** fonksiyonuna, Session'a katılmak isteyen Oyuncunun lokal Oyuncu numarası ve Session'daki arkadaşınızın ID'si parametre verilerek yapılır. Session bulunduğunda **OnFindFriendSessionComplete Delegate 'i** tetiklenir ve Session'a katılmak için kullanılabilecek bir Arama Sonucu (Search Result) verir.

Bir Oyuncu ayrıca **IOnlineSession::SendSessionInviteToFriend()** ya da **IOnlineSession::SendSessionInviteToFriends()** fonksiyonlarını kullanarak, davet edilecek **Oyuncu'ların Lokal Oyuncu Numarası, Session Adını** ve **ID'lerini** ileterek bir veya daha fazla arkadaşını **o anki** Session'larına katılmaya davet edebilir.

Bir Arkadaş daveti kabul ettiğinde, katılmak için Session'ın Arama Sonucunu (Search Result) içeren **OnSessionInviteAccepted Delegate'i** tetiklenir.

Multiplayer Oyuna Nasıl Başlanır?

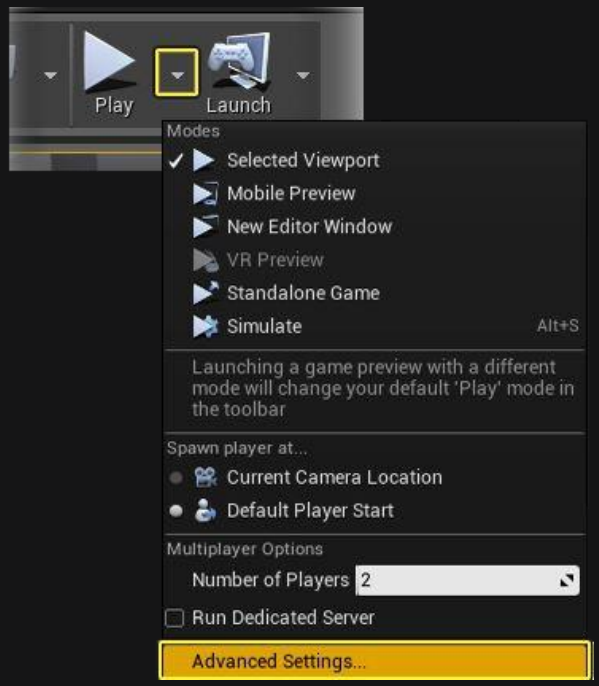
Multiplayer bir oyuna başlamanın en kolay ve anlaşılır yolu, Play Butonunun yanındaki menüden **Number of Players** seçeneğini 1'den yüksek değere ayarlamaktır.



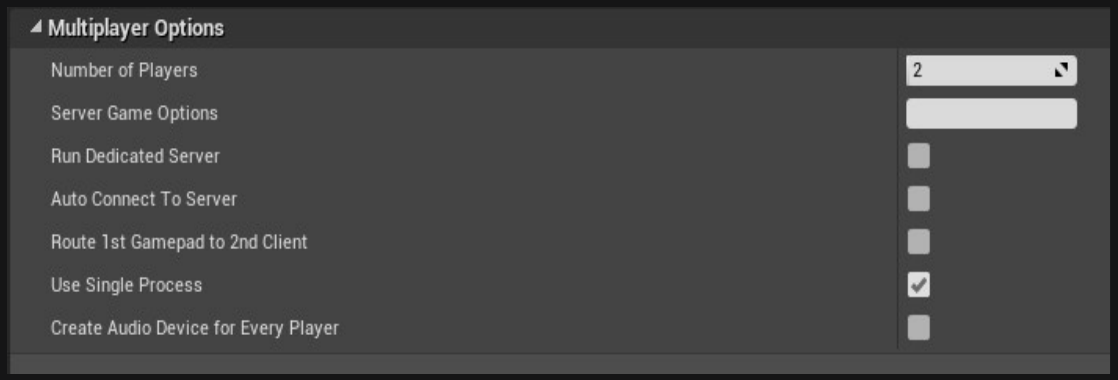
Bu, Server ve Client arasında otomatik olarak bir ağ bağlantısı oluşturacaktır. Böylece Oyunun Ana Menü bölümünüzde **Number of Players** seçeneğini 2+ olarak ayarlanmış şekilde başlatsanız bile Oyuna bağlanacaktır!

Bu her zaman bir network bağlantısıdır. Bu seçenek, bir lokal Co-op Multiplayer bağlantısı **DEĞİLDİR.** Bunun farklı şekilde ele alınması gerekiyor ve bu noktada ele **ALINMAYACAK.**

Advanced Settings



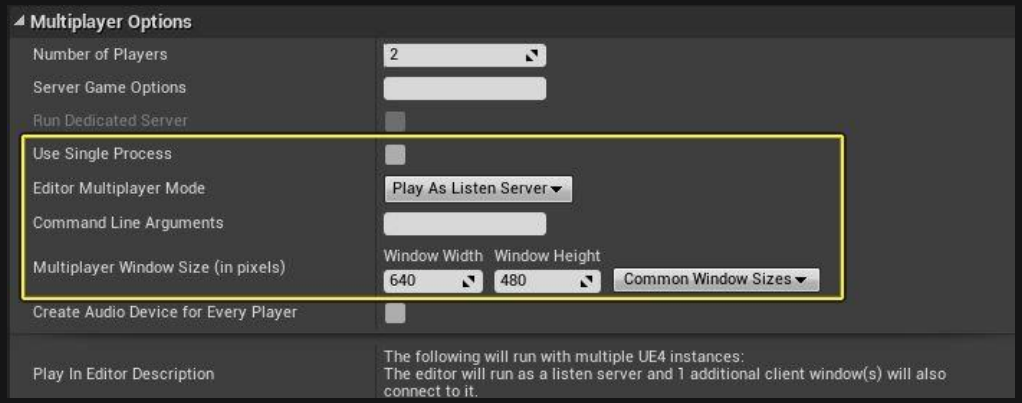
Advanced Settings Oynatma modu için birkaç seçenek daha belirlemenizi sağlar. Kategorilerden biri **Multiplayer Options (Multiplayer Seçenekleri)** ayarlamanıza olanak tanır:



Aşağıdaki tabloda **Multiplayer Options'ı** tek tek açıklayacağız.

Option (Seçenek)	Açıklama
Number of Players	Bu seçenek, oyun başlatıldığında oyunda görünecek oyuncu sayısını tanımlar. Editör ve Listen-Server, oyuncu olarak sayılır Dedicated server oyuncu olarak sayılmaz. Client'lar, oyuncuların geri kalanını oluşturur.
Server Game Options	Buraya Server'a URL parametreleri olarak iletilecek ek seçenekleri belirtebilirsiniz.
Run Dedicated Server	Bu seçenek işaretlenirse, ayrı bir Dedicated Server başlatılır. Aksi takdirde ilk oyuncu diğer tüm oyuncuların bağlanabileceği bir Listen-Server görevi görür.
Auto Connect To Server	Oyunun Client'ları doğrudan Server'a bağlanması gerekip gerekmediğini kontrol eder. Yani, sadece Oyunu test etmek istediğinizde kontrol edebileceğiniz ve bağlantı için Menü ayarlamayı önemsemeyeceğiniz anlamına gelir. Diğer taraftan eğer menünüz varsa, bu boolean değerini devre dışı bırakmak isteyebilirsiniz.
Route 1st Gamepad to 2nd Client	Tek bir işlemde birden çok oyuncu penceresi çalıştırırken, bu seçenek gamepad girişini nasıl yönlendirileceğini belirler. İşaretlenmezse (varsayılan) 1. gamepad 1. window'a (pencereye), 2. gamepad 2. window'a vb. bağlanır. İşaretlenirse, 1. gamepad 2. window'a gider. 1. window daha sonra iki kişi aynı bilgisayarda test yapıyorsa klavye / fare ile kontrol edilebilir.
Use Single Process	Bu seçenek, UE4'ün tek bir Instance'ında çoklu oyuncu penceresi oluşturur. This spawns multiple player windows in a single instance of Unreal Engine 4. Bu seçenekle oyun çok daha hızlı yüklenir ancak daha fazla sorun yaşama potansiyeli vardır. Bu seçenek işaretlenmediğinde ek seçenekler kullanılabilir hale gelir.
Create Audio Device for Every Player	Bunu etkinleştirmek, her oyuncunun bakış açısından doğru ses oluşturmaya izin verir fakat daha fazla CPU kullanır.
Play in Editor Description	Bu, halihazırda uygulanmakta olan Multiplayer ayarlara göre oynarken ne olacağının bir açıklamasıdır.

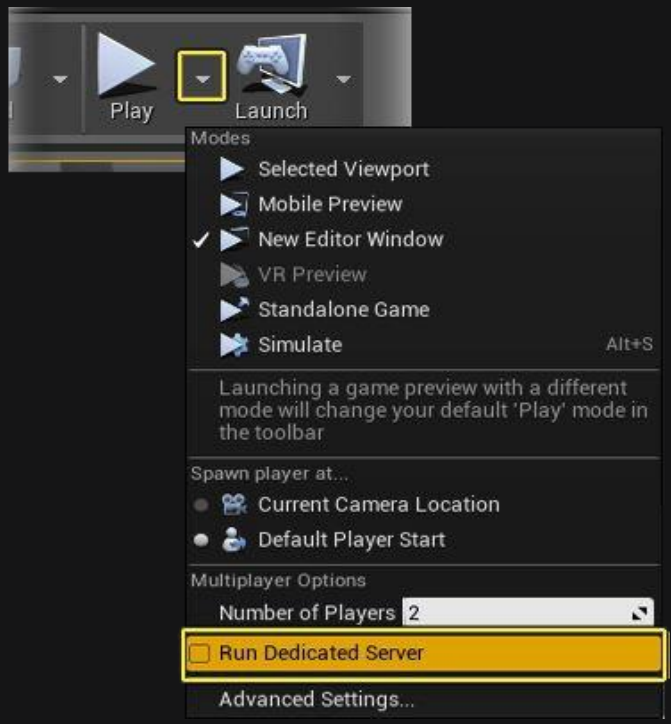
Use Single Process



Use Single Process işaretlendiğinde, Unreal Engine 4'ün tek bir Instance'ında birden çok pencere oluşturur.
Bu seçenek işaretlenmediğinde atanan her oyuncu için birden çok UE4 Instance'ı başlatılır ve ek seçenekler kullanılabilir hale gelir.

Option (Seçenek)	Açıklama
Editor Multiplayer Mode	Editör'de oynatmak için kullanılacak Network modudur (**Play Offline (Çevrimdışı oyna)**, **Play As Listen Server (Listen-Server olarak oyna)** ya da **Play As Client (Client olarak oyna)**).
Command Line Arguments	Burada standalone (bağımsız) oyun instance'larına aktarılacak ek komut satırı seçenekleri atayabilirsiniz.
Multiplayer Window Size (in pixels)	Ek olarak standalone (bağımsız) oyun instance'ları oluştururken kullanılacak pencerenin genişlik / yüksekliğini tanımlarsınız.

Dedicated Server Olarak Çalıştırma



"**Run Dedicated Server**" seçeneği işaretlenmezse, ilk Client bir Listen-Server olacaktır.
Öte yandan bu seçenek **İŞARETLENİRSE, tüm** Oyuncular Client olacaktır.

Server Başlatma ve Server'a Bağlanma

Alt Sistem ve Session Bölümleri, Session Sistemi ile bir Session / Server'ın nasıl başlatılacağı zaten açıklandı.
Ancak normal başlatma ve IP üzerinden bağlanma hakkında ne düşünüyorsunuz? Oldukça kolay!

Blueprint

Server'ı Başlatma

Session Sistemi olmadan Server'ı başlatmak için, **OpenLevel** Node'unda **Level Name (Bölüm Adı)** ve **listen (dinleme)** seçeneğini iletmeniz yeterlidir. GameMode sınıfında açıklandığı gibi parametreleri bir '?' ile ayırarak daha fazla seçenek ayarlanabilir.
Session Sistemi olmayan bir Dedicated Server, halihazırda Proje Ayarlarınızın (Project Settings) **Maps&Nodes** sekmesinde belirleyebileceğiniz haritada başlar.

Server'a Bağlanma

Bir Server'a bağlanmak için 'Execute Console Command' Node'unu **open IPADDRESS** komutuyla kullanmanız yeterlidir, burada **IPADDRESS**, Server'ın gerçek **IP ADRESİYLE** değiştirilir.

Bu alan, örneğin bir Widget Metin Kutusu (Text-Box) aracılığıyla doldurabilir.

Blueprint'deki kullanıma benzer şekilde, Blueprint Node'larıyla aynı sonucu veren bu iki fonksiyonu kullanabilirsiniz.

Server'ı Başlatma

```
UGameplayStatics::OpenLevel(GetWorld(), "LevelName", true, "listen");
```

Server'a Bağlanma

```
// PlayerController sınıfında olmadığınızı varsayarsak (eğer öyleyse, ClientTravel'ı doğrudan çağırın)
APlayerController* PlayerController = UGameplayStatics::GetPlayerController(GetWorld(), 0);
PlayerController->ClientTravel("IPADDRESS", ETravelType::TRAVEL_Absolute);
```

Komut Satırı Üzerinden Başlatma

Temel komut satırları (bu komutları Editör kullanır ve bu nedenle cooked data (build alınmış veri) gerektirmez:

Tür	Komut
Listen Server	UE4Editor.exe ProjeAdı HaritaAdı?Listen -game
Dedicated Server	UE4Editor.exe ProjeAdı HaritaAdı -server -game -log
Client	UE4Editor.exe ProjeAdı ServerIP -game

Not: Dedicated Server'lar varsayılan olarak başıboştur (headless). Eğer "-log" kullanmazsanız, Dedicated Server'ın sunmak için herhangi bir pencere görmezsiniz!

Bağlantı Süreci

Yeni bir Client ilk kez bağlandığında, birkaç şey olur:

İlk olarak Client Server'a bağlanmak için bir istek gönderir.

Server bu isteği işler ve Server bağlantıyı reddetmezse, devam etmek için uygun bilgilerle Client'a bir yanıt gönderir.

Aşağıdaki sayfada bağlantı sürecinin temel adımları gösterilecektir. Bu adımlar, Resmi Dokümantasyon'dan doğrudan alıntıdır.

Başlıca adımlar şunlardır

1. Client bir bağlantı isteği gönderir
2. Server kabul ederse, o anki (current) haritayı gönderir
3. Server, Client'ın bu haritayı yüklemesini bekleyecektir
4. Yüklendikten sonra Server lokal "**AgameMode::PreLogin**" fonksiyonunu çağırır
   - Bu işlem, GameMode'a bağlantıyı reddetme şansı verecektir
5. Kabul edilirse, Server daha sonra "**AgameMode::Login**" fonksiyonunu çağırır
   - Bu fonksiyonun rolü, daha sonra yeni bağlanacak Client'a replike edilecek bir PlayerController oluşturmaktır. Bir kere alındığında, bu PlayerController bağlantı işlemi sırasında tutucu (placeholder) olarak kullanılan Client'ların geçici PlayerController'ının yerini alacaktır. Burada "**APlayerController::BeginPlay**" fonksiyonunun çağrılacağını unutmayın. Bu Actor'de RPC fonksiyonlarını çağırmanın henüz güvenli (safe) **OLMADIĞINA** dikkat edilmelidir. "**AGameMode::PostLogin**" fonksiyonu çağrılana kadar beklemelisiniz.
6. Her şeyin yolunda gittiğini varsayarsak, "**AGameMode::PostLogin**" fonksiyonu çağrılır.
   - Tam bu noktada, Server'ın bu PlayerController'da **RPC** fonksiyonlarını çağırmaya başlaması güvenlidir.

Şimdilik bu kadar.

Bu doküman, ileriki zamanda genişletilebilir fakat şu anda Multiplayer Oyunlar için temel ve temel olmayan pek çok şeyi ele aldığımı düşünüyorum.

Umarım bu doküman, Unreal Engine 4'te Multiplayer ve Network konularına başlamanıza yardımcı olmuştur ve bu dokümanla birlikte gelecekte harika, yeni Multiplayer Oyunlar göreceğiz!

Kaynaklar:

*1: Kendi bilgilerimin yanında, Özet'in başında belirtildiği gibi Resmi Dokümantasyon'u kullandım.

*2: Ana Network Framework ve Ortak Sınıfların kümelerinin dağılımının (kendim yeniden oluşturdum) görüntüsü '**Nuno Afonso**' tarafından dokümante edilmiştir ve şu linkten erişilebilir: www.nafonso.com Bu müthiş sayfa için teşekkürler!

*3: Hemen hemen tüm örnekler, Blueprint görüntüleri ve C++ kodları benim tarafımdan oluşturulmuştur.

Dokümanın videolu anlatımına şu linkten erişebilirsiniz: https://www.youtube.com/channel/UC0RkDFVIqXO5PWn_luohJyQ