Unreal Networking Guide

Created by Zach Metcalf
zachmetcalf@gmail.com

Note: This guide is compiled from Unreal Documentation, Unreal Example Content, Shooter Game Demo, and experiences while porting the ITP380 UnrealShmup Game to Multiplayer.

There are several great tutorials on Unreal Networking for Blueprints, so this guide will bridge the gap and provide more C++ examples for developers.

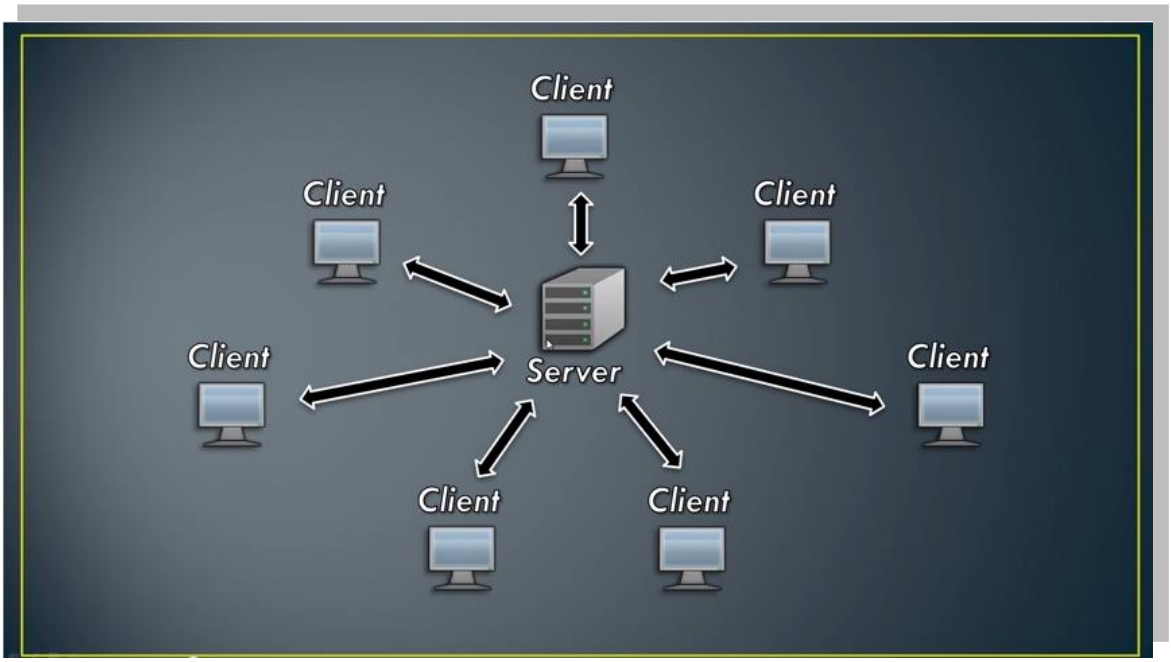
There may be features pertinent to Multiplayer Game Development, such as Match-Making, Network Protocols, etc., that are not covered, but this serves as the base requirements for creating network-ready multiplayer gameplay code.

Contents

Part 0—Unreal Networking Model 1
Part 1—Property Replication 3
Part 2—Actor Replication 6
Part 3—RPC Introduction, More Syncing 8
Part 4—Custom Object Replication via RPCs 10
Part 5—Complex RPC Relationships and Examples 12

Part 0—Unreal Networking Model

Before we get started, let's look at the Unreal Networking Paradigm—**The Client-Server Model**:



In this model, Clients send their data to the Server, the Server processes the data, and forwards this information and any results to all other Clients. Whenever a new Client wants to join, they communicate only with the Server, who then replicates this new player to all other players.

Now, There are many networking models—Peer-to-Peer, Modified Peer-to-Peer (Master Client or Rendezvous Server), Client-Server, etc. So I want to give some justifications for why Unreal uses Client-Server model and why this will be helpful for a large-scale multiplayer game:

Network Requirements—A peer to peer game requires n^2 bandwidth, because each client needs to communicate with each and every other client. However, Client-Server only requires 2n bandwidth since Clients only need to upload their information to the server. With each client only communicating to the Server, each client requires less upstream bandwidth.

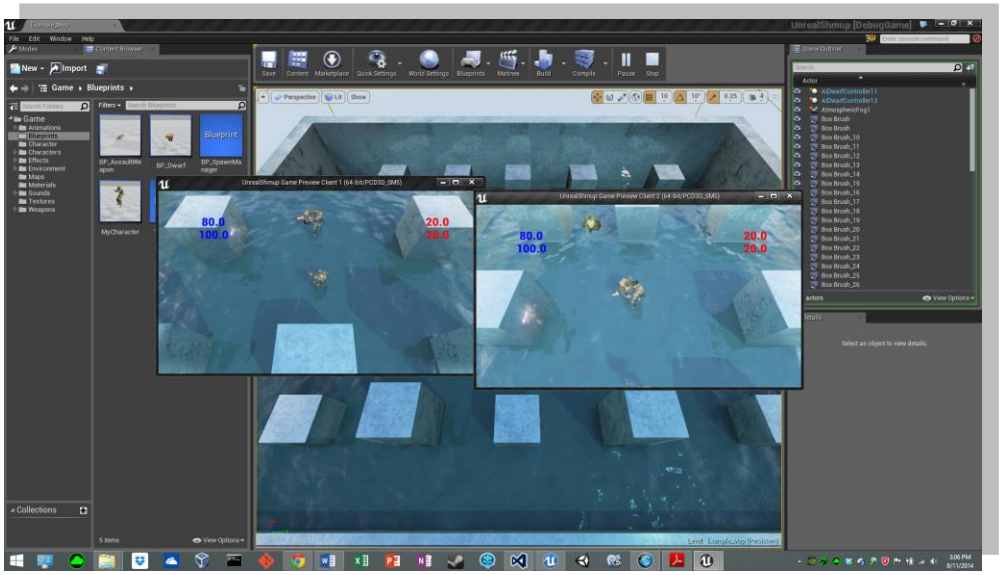
Gameplay Logic—The Server, both physically and responsibility-wise, serves as a central host for all logic. Thus, we can choose to optimize client-side code by doing processing on the server only. For instance, only the Server needs to process whether not a player has taken damage. Once processed, it can simply transmit the results. By moving code to the server, we decrease run-time calculations on the client-side. It also helps prevent Clients from cheating!

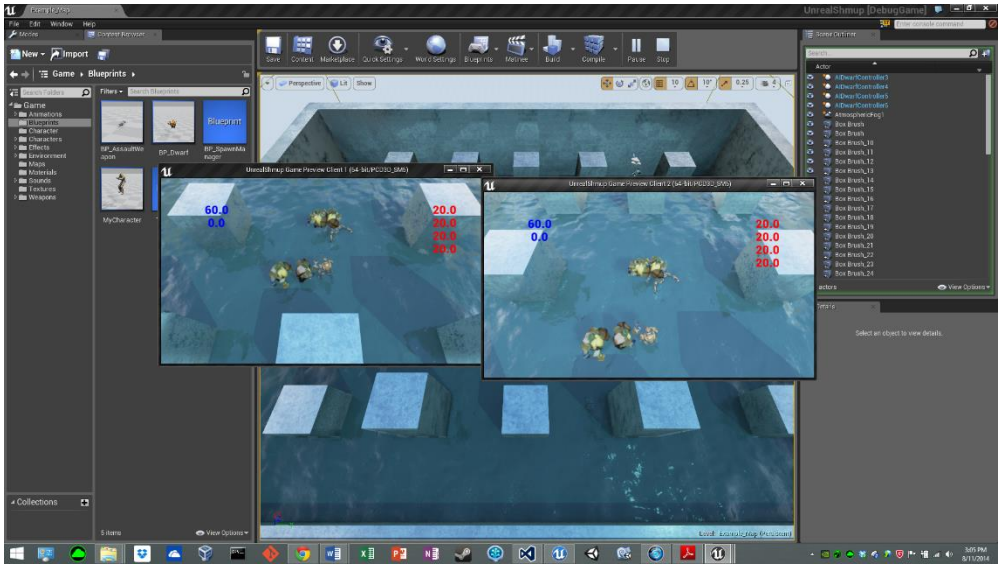
Graphics Processing—One of Unreal's more powerful features is the option to use a dedicated server. A dedicated server is a server that simulates much of the game world and other logic,

except it does no graphics. Thus, by putting more computations and mathematical simulations server-side, the clients only need to focus on sending input data to the server and processing the corresponding graphics of the results (this is known as the Dumb-Client model).

With less processing required on the client, less upstream bandwidth needed client-side, and the ability to adapt code to be network optimized and prevent cheating, Client-Server models are more suited to large-scale multiplayer games. One thing to remember though. The Server is King (or Queen), so we'll always need to keep this in mind.

Network Code within this guide comes from a Multiplayer version of the Unreal SHMUP for ITP380. I will be going through most of that code, but if there's something I've missed, I can provide any needed source code.





Part 1—Property Replication

First, let's enable networking! In ***UnrealShmup.h****:*

…

```
#include "Engine.h"
#include "UnrealNetwork.h"
```

…

That's it! We're done. Just kidding…

Let's start with the easiest: syncing simple properties to clients. A good example for when this would be required is **Health**. Property replication is simple in theory: Every time a variable changes, we want our network to notify all clients of the change and update the variable. In this case, if a player gets shot, everyone should see their health decrease.

When you register a variable to stream like this, it's important that these variables **Only be Modified by the Server** and replicated from the Server. In this case, Property Replication is a one-way from Server to Client, so if we want to change a variable, do it server-side. So let's replicate the Unreal SHMUP Player's Health! In ***UnrealShmupCharacter.h***:

```
class AUnrealShmupCharacter : public ACharacter
{
	…
	// Health
	UPROPERTY(Replicated, EditAnywhere, Category = Player)
	float Health;
	// IsDead
	UPROPERTY(Replicated, BlueprintReadOnly, Category = Player)
	bool IsDead;
	…
}
```

I have chosen to register both Health and IsDead Properties for this game. The procedure for other properties is identical, so whatever variable you want to replicate will follow this method.

Anyway, by adding the UPROPERTY Param ***Replicated*** we have told Unreal that we want these variables to be constantly replicated from **Server to Client**. Now, means 2 things. 1) We need to implement this class for variable syncing, and 2) We have inherently moved this variable to the server's representation of our character only. We'll walk through what each means.

Enabling Syncing—If you try to compile right now, likely your build will break. To fix this, we need to add a function to ***UnrealShmupCharacter.cpp***. There is no need to declare the function prototype in the header file, just copy and paste the code like this:

```
// [Server to AllClient] Multiplayer Replication
void AUnrealShmupCharacter::GetLifetimeReplicatedProps(TArray<FLifetimeProperty>&
OutLifetimeProps) const
{
	Super::GetLifetimeReplicatedProps(OutLifetimeProps);

	DOREPLIFETIME(AUnrealShmupCharacter, Health);
	DOREPLIFETIME(AUnrealShmupCharacter, IsDead);
}
```

And in the constructor (I'll explain these parts later):

```
AUnrealShmupCharacter::AUnrealShmupCharacter(const class FObjectInitializer& PCIP) :
Super(PCIP)
{
	…
	// Replication
	bReplicates = true;
	bReplicateMovement = true;
	bReplicateInstigator = true;
	bNetUseOwnerRelevancy = true;
}
```

And that's it! The `DOREPLIFETIME` macro will set everything up so that when ***Health*** changes on the server, it makes all Clients call ***mClientHealth = mServerHealth***.

For the most part, this is good enough. However, if you are bandwidth crazy and want more control, there are addition parameters you can provide that give you more control as to when a variable replicates. For instance: *COND_OwnerOnly* will only send to the actor's owner, *COND_SkipOwner* will do the opposite. There are several more available at https://www.unrealengine.com/blog/network-tips-and-tricks, and I will be creating a followup guide soon about these further optimizations.

And if you're converting your SHMUP Game to multiplayer, the next step would be to do the same thing for the ***ADwarfCharacter::Health*** variable.

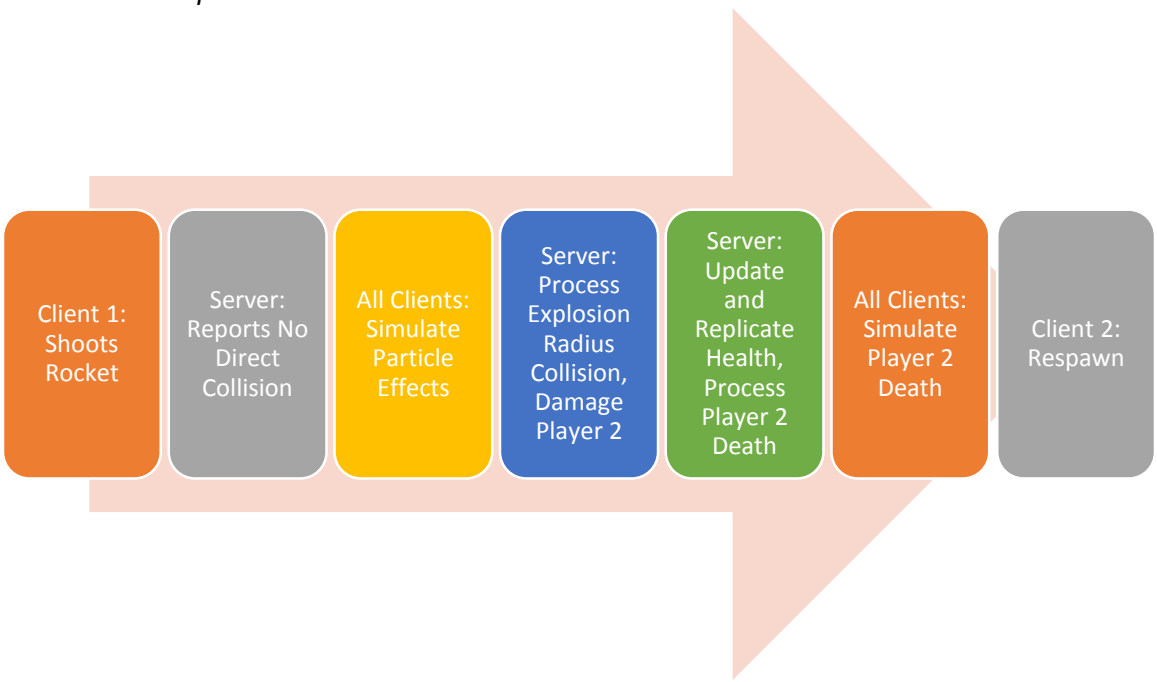
Commenting Style:

```
// [Server to AllClient] Multiplayer Replication to AllClients
```

This style was used to implement Unreal's ShooterGame sample and we will continue to use it within this guide. Essentially, the ***[…]*** will encapsulate who is physically calling and what Server-Client relationship is within code.

As code bases get bigger, the ties between sections of code grow more dependent on the networking relationship. With these complex relationships, documentation, commenting, and code cleanliness is crucial! Before writing a single line of code, it might be good to go through the process of visualizing the Client-Server relationships via a Flow Chart.

Common example: Rocket Launcher



Part 2—Actor Replication

Now that we've decided to replicate, let's make sure that our visible objects will properly replicate across a network. This next section will refer to anything spawned that one or more clients will have a relationship with. In the SHMUP game, this will be: AUnrealShmupCharacter, AAssaultWeapon, and ADwarfCharacter. We will add this code to their constructors:

```
AAssaultWeapon::AAssaultWeapon(const class FObjectInitializer & PCIP) : Super(PCIP)
{
	…
	// Replication
	bReplicates = true;
	bReplicateInstigator = true;
	bNetUseOwnerRelevancy = true;
}

AUnrealShmupCharacter::AUnrealShmupCharacter(const class FObjectInitializer & PCIP) :
Super(PCIP)
{
	…
	// Replication
	bReplicates = true;
	bReplicateMovement = true;
	bReplicateInstigator = true;
	bNetUseOwnerRelevancy = true;
}

ADwarfCharacter::ADwarfCharacter(const class FObjectInitializer & PCIP) : Super(PCIP)
{
	…
	// Replication
	bReplicates = true;
	bReplicateMovement = true;
	bReplicateInstigator = true;
}
```

Now, let's discuss these 4 properties:

bReplicates—This property is actually required for the variable streaming that we covered in the last section. If there is data to be synced, we enable this bool.
bReplicateMovement—This property will only be relevant to actors like the player or dwarf. For instance, the AddMovementInput() and MoveToActor() functions are already network-ready. So once we enable this and move our player, this will stream player movement correctly.
bReplicateInstigator—This property syncs the **Ownership** of an object. This is important for 2 reasons: 1) it lets the server know who owns the object in case it's trying to call TakeDamage() or a similar server-side function. 2) it enables us to write code within the class that is Server-specific and Client-specific.
bNetUseOwnerRelevancy (Optional)—Allows us to stream object with the priority of parent.

So, *bReplicates* is required for variable streaming, *bReplicateMovement* helps with movement, *bNetUseOwnerRelevancy* is for update frequencies. Let's demystify *bReplicateInstigator* and discuss Network Ownership and code strategies.

Now that we've enabled ***bReplicateInstigator***, we can separate Server and Client code. In ***UnrealShmupCharacter.h***, we override the following functions and place these in ***UnrealShmupCharacter.cpp***. You may notice these functions are changed slightly from the ITP380 SHMUP Game. We needed to move some code around for networking readiness. If you're familiar with Unity, this function corresponds to Start(). BeginPlay() is essentially the first called Tick() update.

```
// BeginPlay Override
void AUnrealShmupCharacter::BeginPlay()
{
	Super::BeginPlay();

	// [Server]
	if (Role == ROLE_Authority)
	{
		SpawnWeapon();
	}
}
```

Looking at the familiar commenting style, we can see that the ***Role*** variable is how we will determine whether or not code should be run on a client or a server. The ***ROLE_Authority*** corresponds to an enum within Unreal's Engine (it's the highest value of 4 enum values). There are a few different options here, but this is a quick summary:

If (Role == ROLE_Authority), you're the server.
If (Role < ROLE_Authority), you're a client.

So what does this mean for us? Well, if we look at this again, we see that only the Server will run the code to spawn the weapon. Here is another example:

```
// PlayerTick Override
void AUnrealShmupPlayerController::PlayerTick(float DeltaTime)
{
	Super::PlayerTick(DeltaTime);

	// [Client] Update Rotation
	if (Role < ROLE_Authority)
	{
		UpdateMouseLook();
	}
}
```

Now we have modified the player Tick() function so only clients will do the mouse looking/rotating. Once we learn about RPCs, we will go into more in depth examples. But for now, we have learned how to separate code.

Part 3—RPC Introduction, More Syncing

So we have the player movement synced up and some variables like health streaming. But this is not enough. For instance, the first bug I discovered was that the *bReplicateMovement* does not replicate rotation! So to fix this, we will discuss RPCs and more strategies on how to sync objects. Let's add our first RPC to ***UnrealShmupPlayerController.h***:

```
class AUnrealShmupPlayerController : public APlayerController
{
	…
	// [Client] Use Mouse to Rotate Player View
	void UpdateMouseLook();

	// [Server] SetRotation
	UFUNCTION(Reliable, Server, WithValidation)
	void SyncRotation(FRotator Rotation);
	…
}
```

Let's discuss some of the UFUNCTION Parameters that we just added.
Reliable or Unreliable—*Unreliable* says that we're all right dropping this packet's information for a few frames if we have a slow connection. *Reliable* says that we want it guaranteed to get there, however long that takes. In this case, I chose *Reliable* because my weapon hit logic is based on the player's rotation. But this decision should be made for each variable.

Server, Client, or Multicast—This one gets tricky. ***Server*** RPCs can be called from a client or the server but only within classes where *bReplicateInstigator*=true. These functions are then run by the server only. ***Client*** RPCs are called only for the specific client who owns the object we are working on (again, we need *bReplicateInstigator*=true). ***Multicast*** RPCs are the most powerful. First, the *Server* must call a Multicast Function and run the code *directly on the server*. Then, it forwards this code to all *Clients*, who then *run the functions on all clients*. Whenever we need to simulate graphics after server processing. Multicast RPCs are our best friends.

WithValidation—This parameter (required for all Server functions, optional otherwise) allows us to have a callback function that gets called whenever someone receives the data. A good example for this would be, once a Client gets the weapon fire validated, we play a particle effect explosion just after. The other option here is to leave the parameter blank.

More info on these parameters is available here:
https://docs.unrealengine.com/latest/INT/Resources/ContentExamples/Networking/1_5/index.html, https://wiki.unrealengine.com/Networking/Replication

Back to our examples: In the UpdateMouseLook() that is only processed Client-side, we are going to add a function call and declare our new RPC functions right below it.

```
// [Client] Use Mouse to Rotate Player View
void AUnrealShmupPlayerController::UpdateMouseLook()
{
	APawn* const Pawn = GetPawn();
	…
	// [Client]
	Pawn->SetActorRotation(fRotatorToImpact);
	// [Server]
	if (Role < ROLE_Authority)
	{
		SyncRotation(fRotatorToImpact);
	}
}

// [Server] SyncRotation
bool AUnrealShmupPlayerController::SyncRotation_Validate(FRotator Rotation)
{
	return true;
}
void AUnrealShmupPlayerController::SyncRotation_Implementation(FRotator Rotation)
{
	APawn* const Pawn = GetPawn();
	if (Pawn)
	{
		Pawn->SetActorRotation(Rotation);
	}
}
```

Now, we will walk through these functions: In UpdateMouseLook(), the Client sets its own rotation (but the Server and All Other clients can't see the changes). Thus, the client makes sure it's not the server, and then calls an RPC for the server to call. The SyncRotation_Validate() is the extra function we are required to implement whenever we use the ***WithValidation*** parameter. It returns true, and if we wanted to process something on validation, we do it here. The SyncRotation_Implementation is the RPC function itself. And if you've done some Blueprint/C++ tutorials, the _Implementation and _Validate is Unreal doing its RPC magic.

The thing that's worth noting is in SyncRotation_Implementation() and UpdateMouseLook() we first fetch the Pawn via GetPawn() of this class before setting the rotation. This is, in fact, the same pawn, just from different perspectives, i.e., it is the same object because we set up our Character as *bReplicates*=true, but when the Server updates its version, other clients are able to see it. In other words, **All Changes Sync From Server to Clients**.

This is the power of RPCs. If the client needs to update other clients, it must use this technique, and update via the server.

Part 4—Custom Object Replication via RPCs

Before we go further into RPCs and Client-Server relationships, we should discuss how we get full classes to sync. In this example, we're going to return to spawning the character's weapon:

```
// PostInitializeComponents Override
void AUnrealShmupCharacter::PostInitializeComponents()
{
	Super::PostInitializeComponents();

	// [Server]
	if (Role == ROLE_Authority)
	{
		SpawnWeapon();
	}
}
```

Here, we spawned the weapon on the Server, but we still need to set it up so that the particular Client **Owns** the weapon. So let's go back to property syncing in ***UnrealShmupCharacter.h:***

```
class AUnrealShmupCharacter : public ACharacter
{
	…
	// Health
	UPROPERTY(Replicated, EditAnywhere, Category = Player)
	float Health;
	// IsDead
	UPROPERTY(Replicated, BlueprintReadOnly, Category = Player)
	bool IsDead;

	// Replicated Weapon
	UFUNCTION() void OnRep_MyWeapon();
	UPROPERTY(Transient, ReplicatedUsing = OnRep_MyWeapon)
	class AWeapon* MyWeapon;
	…
}
```

The new UPROPERTY ***ReplicatedUsing = OnRep_MyWeapon*** tells Unreal that I actually want to create my own **custom object syncing function**. With all of the ownership issues we could have, it's not enough just to say "*MyClientWeapon = MyServerWeapon"* when syncing.

And this new ***UFUNCTION void OnRep_MyWeapon()*** has nothing special, I just declared it here to show that it's only job is to sync the forward-declared class AWeapon* MyWeapon. Now back to ***UnrealShmupCharacter.h***:

```
// [Server to AllClient] Multiplayer Replication
void AUnrealShmupCharacter::GetLifetimeReplicatedProps(TArray<FLifetimeProperty>&
OutLifetimeProps) const
{
	Super::GetLifetimeReplicatedProps(OutLifetimeProps);

	DOREPLIFETIME(AUnrealShmupCharacter, Health);
```

```
	DOREPLIFETIME(AUnrealShmupCharacter, IsDead);
	DOREPLIFETIME(AUnrealShmupCharacter, MyWeapon /* OnRep_MyWeapon */);
}
// Replicated Weapon
void AUnrealShmupCharacter::OnRep_MyWeapon()
{
	EquipAndSyncWeapon();
}

// [Server] [AllClients] EquipAndSyncWeapon
void AUnrealShmupCharacter::EquipAndSyncWeapon()
{
	if (MyWeapon)
	{
		// Equip PreSpawned Weapon
		// This is attached to "WeaponPoint" Defined in Skeleton
		MyWeapon->WeaponMesh->SnapTo(Mesh, TEXT("WeaponPoint"));
		MyWeapon->SetActorRotation(FRotator(0.0f, -90.0f, 0.0f));
		MyWeapon->SetOwningPawn(this);
	}
}
```

Now we have created our own custom replication function OnRep_MyWeapon that is called whenever syncing occurs, giving us complete control across a network (note: this is called OnRep_Notify in blueprinting). But what was the point of doing this? The point is the function that I bolded above. SetOwningPawn(). Here is ***AWeapon::SetOwningPawn().*** This is crucial.

```
// [Server] [AllClients] SetOwner for RPC Calls
void AWeapon::SetOwningPawn(AUnrealShmupCharacter* NewOwner)
{
	if (MyPawn != NewOwner)
	{
		Instigator = NewOwner;
		MyPawn = NewOwner;
		// [Net Owner] for RPC Calls
		SetOwner(NewOwner);
	}
}
```

Even though we set *bReplicateInstigator*=true, ***if we want to sync an entire object, like a weapon, these 2 lines of bolded code ensure that RPC network calls are done correctly***. Without this little function (which I recommend within all Base classes of replicate-able objects), ownership/RPCs won't always work like we want.

Why is this important? Well, because Unreal has always implemented Client-Server models, some of their functions are built directly into the networking model. The best example is TakeDamage(). After diving deep within the engine code, there is a line that Zeroes out all Damage *if (Role < ROLE_Authority)*. And if we don't setup ownership/RPCs correctly, Unreal will get confused and Zero out our damage. It won't even return an error/exception. It will simply just not work/not call the RPC, and it will cost many wasted hours trying to figure out why nothing got called.

Part 5—Complex RPC Relationships and Examples

All right, we're almost done! I promise the rest of this guide is mostly code and not my lecturing anymore. But thanks for sticking with it so far!

Dwarf Example—In the SHMUP Game, is there any control the player has over the dwarves? Not really. Therefore, let's move all spawning and processing of the dwarves to the Server! Let's start with ***SpawnManager.h*** and ***SpawnManager.cpp***:

```
class UNREALSHMUP_API ASpawnManager : public AActor
{
	…
	// [Server] Spawn NextDwarf
	UFUNCTION(Reliable, Server, WithValidation)
	void SpawnNextEnemy();
	…
}

// Begin Play Override
void ASpawnManager::BeginPlay()
{
	Super::BeginPlay();

	// [Server]
	if (Role == ROLE_Authority)
	{
		GetWorldTimerManager().SetTimer(this, &ASpawnManager::SpawnNextEnemy,
MinSpawnTime, false);
	}
}

// [Server] Spawn NextDwarf
//UFUNCTION(Reliable, Server, WithValidation)
bool ASpawnManager::SpawnNextEnemy_Validate() { return true; }
void ASpawnManager::SpawnNextEnemy_Implementation()
{
	FActorSpawnParameters SpawnParams;
	SpawnParams.Owner = this;
	SpawnParams.Instigator = Instigator;
	…

	// Spawn Character
	UWorld* World = GetWorld();
	if (World)
	{
		ACharacter* Character = World->SpawnActor<ACharacter>(CharacterToSpawn,
Position, Rotation, SpawnParams);
		if (Character)
		{
			Character->SpawnDefaultController();
		}
	}

	// [Server] Next Spawn
```

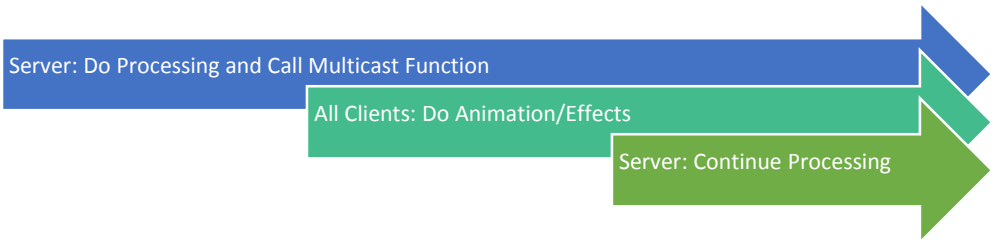
```
	float NextRandomSpawn = FMath::FRandRange(MinSpawnTime, MaxSpawnTime);
	GetWorldTimerManager().SetTimer(this, &ASpawnManager::SpawnNextEnemy,
NextRandomSpawn, false);
}
```

By catching that first call in BeginPlay() for the Server RPC, we have trapped SpawnManager so that all function calls following will only occur Server-side. This, and with the Dwarves being *bReplicated*=true, we moved the dwarf objects directly to the server. Now, they'll look odd since they won't animate now, but let's fix that. In ***DwarfCharacter.h*** and ***DwarfCharacter.cpp***:

```
class UNREALSHMUP_API ADwarfCharacter : public AEnemyCharacter
{
	…
	// [Server] [AllClients] StartAttack
	UFUNCTION(Reliable, NetMulticast, WithValidation)
	void StartAttack();
	// [Server] [All Clients] StopAttack
	UFUNCTION(Reliable, NetMulticast, WithValidation)
	void StopAttack();
	// [Server] [All Clients] ProcessDeath
	UFUNCTION(Reliable, NetMulticast, WithValidation)
	void ProcessDeath();
	…
}

// [Server] Take Damage Override
float ADwarfCharacter::TakeDamage(float Damage, struct FDamageEvent const& DamageEvent,
AController* EventInstigator, AActor* DamageCauser)
{
	float ActualDamage = Super::TakeDamage(Damage, DamageEvent, EventInstigator,
DamageCauser);
	if (ActualDamage > 0.0f)
	{
		Health -= ActualDamage;
		if (Health <= 0.0f)
		{
			// [Server] [AllClients]
			Health = 0.0f;
			bCanBeDamaged = false;
			StopAttack();
			ProcessDeath();
		}
	}
	return ActualDamage;
}

// [Server] [AllClients] Start Attack
//UFUNCTION(Reliable, NetMulticast, WithValidation)
bool ADwarfCharacter::StartAttack_Validate() { return true; }
void ADwarfCharacter::StartAttack_Implementation()
{
	// [Server] [AllClients] Play Attack Animation
	float AnimDuration = PlayAnimMontage(AttackAnim);

	// [Server]
	if (Role == ROLE_Authority)
```

```
	{
		// Do Damage
		GetWorldTimerManager().SetTimer(this, &ADwarfCharacter::DamageActor,
AnimDuration - 0.10f, true);
	}
}

// [Server] [All Clients] Stop Attack
//UFUNCTION(Reliable, NetMulticast,WithValidation)
bool ADwarfCharacter::StopAttack_Validate() { return true; }
void ADwarfCharacter::StopAttack_Implementation()
{
	// [Server] [AllClients] Stop Attack Animation
	StopAnimMontage();

	// [Server]
	if (Role == ROLE_Authority)
	{
		// Stop Damage
		GetWorldTimerManager().ClearTimer(this, &ADwarfCharacter::DamageActor);
	}
}

// [Server] [All Clients] Process Death
//UFUNCTION(Reliable, NetMulticast, WithValidation)
bool ADwarfCharacter::ProcessDeath_Validate() { return true; }
void ADwarfCharacter::ProcessDeath_Implementation()
{
	// [Server] [AllClients] Death Animations
	GetWorldTimerManager().ClearAllTimersForObject(this);
	StopAnimMontage();
	float AnimDuration = PlayAnimMontage(DeathAnim);

	// [Server]
	if (Role == ROLE_Authority)
	{
		GetWorldTimerManager().SetTimer(this, &ADwarfCharacter::DestroySelf,
AnimDuration - 0.25f, false);
		Controller->UnPossess();
	}
}
```

This procedure is often how the interactions between Server and Client will go:



AssaultWeapon Example—This is just to give one final clear picture of RPC relationships, and to show you that the RPC function calls of these classes will follow a very similar pattern every time. In ***AssaultWeapon.h*** and ***AssaultWeapon.cpp***:

```
class UNREALSHMUP_API AAssaultWeapon : public AWeapon
{
	…
	// StartFire Override
	void StartFire() override;
	// StopFire Override
	void StopFire() override;

	// [Server] ProcessWeaponStart
	UFUNCTION(Reliable, Server, WithValidation)
	void ProcessWeaponStart();
	// [Server] ProcessWeaponStop
	UFUNCTION(Reliable, Server, WithValidation)
	void ProcessWeaponStop();

	// [Server] WeaponTrace
	void WeaponTrace();

	// [AllClients] StartHitEffects
	UFUNCTION(Unreliable, NetMulticast)
	void StartHitEffects(FHitResult Hit);
	…
}

// StartFire Override
void AAssaultWeapon::StartFire()
{
	Super::StartFire();

	// [Client] Hand Off to [Server]
	if (Role < ROLE_Authority)
	{
		ProcessWeaponStart();
	}
}

// StopFire Override
void AAssaultWeapon::StopFire()
{
	Super::StopFire();

	// [Client] Hand Off to [Server]
	if (Role < ROLE_Authority)
	{
		ProcessWeaponStop();
	}
}

// [Server] ProcessWeaponStart
//UFUNCTION(Reliable, Server, WithValidation)
bool AAssaultWeapon::ProcessWeaponStart_Validate() { return true; }
void AAssaultWeapon::ProcessWeaponStart_Implementation()
{
	// [AllClients]
	StartWeaponEffects();
```

```
	// [Server] Start WeaponTrace Timer
	GetWorldTimerManager().SetTimer(this, &AAssaultWeapon::WeaponTrace, FireRate,
true);
}

// [Server] ProcessWeaponStop
//UFUNCTION(Reliable, Server, WithValidation)
bool AAssaultWeapon::ProcessWeaponStop_Validate() { return true; }
void AAssaultWeapon::ProcessWeaponStop_Implementation()
{
	// [AllClients]
	StopWeaponEffects();

	// [Server] Stop WeaponTrace Timer
	GetWorldTimerManager().ClearTimer(this, &AAssaultWeapon::WeaponTrace);
}

// [Server] WeaponTrace
void AAssaultWeapon::WeaponTrace()
{
	…
	// Test For Collision
	if (Hit.bBlockingHit)
	{
		// [AllClients] Effects
		StartHitEffects(Hit);

		// [Server] Damaging
		ADwarfCharacter* Dwarf = Cast<ADwarfCharacter>(Hit.GetActor());
		if (Dwarf)
		{
			Dwarf->TakeDamage(DamagePower, FDamageEvent(),
GetInstigatorController(), this);
		}
		AUnrealShmupCharacter* Player =
Cast<AUnrealShmupCharacter>(Hit.GetActor());
		if (Player)
		{
			Player->TakeDamage(DamagePower, FDamageEvent(),
GetInstigatorController(), this);
		}
	}
}

// [AllClients] StartHitEffects
//UFUNCTION(Unreliable, NetMulticast)
void AAssaultWeapon::StartHitEffects_Implementation(FHitResult Hit)
{
	UGameplayStatics::SpawnEmitterAtLocation(GetWorld(), HitFX, Hit.Location);
}
```

And that's it! There's still more to implement, but if there are any follow up questions or if you'd like to see the full source code, please feel free to ask! My contact is on the cover page. And if you fully implement the rest of this Unreal SHMUP as Multiplayer, it should look something like the pictures near the start of this guide!