UEオンライン機能実装ガイド：基礎と鉄則

はじめに

オンラインゲーム開発において最も重要なのは、**「サーバーが絶対的な正解（権限/Authority）を持っている」**ということです。クライアント（プレイヤーの画面）は、あくまでサーバーの結果を映す鏡にすぎません。

実装は**「命令・同期・演出」**の3ステップで考えるとスムーズです。

実装の3ステップ

1. 命令：アクションはまずサーバーに頼む **(Server RPC)**

プレイヤーが操作をした際、勝手に自分の画面でHPを減らしたり弾を発射してはいけません。まずサーバーに「処理をお願い」します。

- **Run On Server (Server RPC):**
  - クライアントから呼び出され、サーバー上で実行される関数です。
  - ここで弾の発射判定やHPの計算など、ゲームの「重要な処理」を行います。

2. 同期：結果を全員に配る **(Replicated変数)**

サーバーで処理が行われ、変数の値（例：HPが100→90）が変わったら、それを参加している全クライアントに通知して書き換えます。

- **Replicated（Repnotify）変数:**
  - サーバーで値が変更されると、自動的にクライアント側の変数も同じ値に書き換わる機能です。
  - 重要: これは「サーバー ⇒ クライアント」の一方通行です。クライアントが変数を書き換えても他には伝わりません。

3. 演出：値が変わった瞬間に見た目を変える **(RepNotify)**

変数の値が同期されただけでは、画面上の「HPバーの減少アニメーション」や「ダメージエフェクト」は再生されません。そこで、変数が書き換わったタイミングを検知する機能を使います。

- **RepNotify (On_Rep関数):**
  - 変数が同期（更新）された瞬間に、自動的に呼び出される関数（`On_Rep_変数名`）を設定できます。

- 使い方のコツ: 「**HP**を減らす計算」はサーバーで行い、「減った**HP**に合わせて**UI**を揺らす、音を鳴らす」という見た目の処理はこの**On_Rep**関数の中に記述します。

データの流れ（処理イメージ）

1. 【**Client**】攻撃ボタンを押す ⇒ `Server_Attack()` を呼ぶ **(RPC)**
2. 【**Server**】`Server_Attack()` が実行される
   - ダメージ計算を行う
   - **HP**変数を **100** から **90** に変更する
3. 【**Server**】変更された **HP=90** を全クライアントに送信 **(Replication)**
4. 【**Client**】各クライアントで以下が起こる
   - **HP**変数が **90** に更新される
   - 自動的に `On_Rep_HP()` が呼ばれる
   - `On_Rep_HP()` 内で「**HP**バーのアニメーション」が再生される **(**演出**)**

補足：移動の同期について

キャラクターの移動に関しては、上記のような複雑な実装をしなくても、**Unreal Engine**が標準で強力な機能を用意しています。

- **Replicates Movement:**
  - アクター（**Character**など）のこの設定にチェックを入れるだけで、位置と回転がスムーズに同期されます。
  - 背景では「位置の予測」や「補間」という高度な計算が自動で行われています。

DarkFactoryのプロジェクトでの実装例

1.命令ステップについて

まず、上の 説明ではなかった大事なルール

「**1.** 命令」ステップにおいて、サーバーに処理をお願いできるのは自分が操作している**Character**（このプロジェクトだと**BP_Player**だけ）からだけというルールがある。（正確には所有権を持つアクターからのみ？）

実際のサーバーに処理を頼む関数：

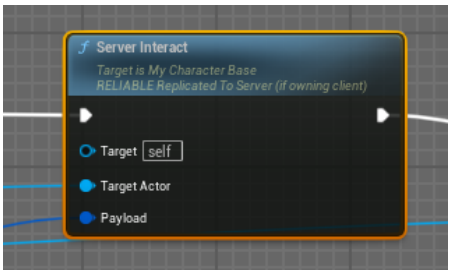
サーバーに処理を頼むための関数にはUFUNCTION(Server, Reliable…) という指定子 をつける。この関数はBP_PlayerのベースクラスのMyCharacterBaseクラスに定義されている。
つまり、以下の関数からであれば問題なくサーバーに処理をお願いできる。

```
/**
* @brief [Server RPC] サーバーへインタラクトを要求します。
* クライアントから呼び出されると、サーバー上で_Implementation関数が実行されます。
* @param TargetActor インタラクトしたい対象のアクター
* @param InteractionType 実行したいアクションの種類
*/
UFUNCTION(Server, Reliable, BlueprintCallable, Category = "Interaction")
void Server_Interact(AActor* TargetActor, const FInteractionPayload& Payload);
```

一方で、BP_PlayerとそのベースクラスのMyCharacterBaseクラス以外のプログラムではServer指定子をつけた関数を作成して呼んでも失敗する。

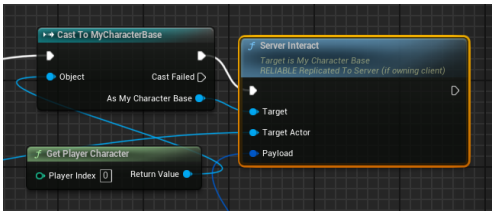
実際の使用例：

パターン1 （BP_Playerからサーバーに処理を頼みたい場合）
以下のノードはBP_Playerのイベントグラフにあるノード。



引数「Target」がこの関数を実装しているプログラムのことだけど、BP_Playerのイベントグラフで実装しているから「self」でいい。
※他の2つの引数「Target Actor」「Payload」については後述

パターン2 （BP_Player以外がサーバーに処理を頼みたい場合）
以下のノードはBP_Caseのイベントグラフにあるノード



引数「Target」にはSeverInteractを実装しているMyCharacterBase(BP_Playerの親CPP)をつなげる必要があるから、GetPlayerCharacterノードでBP_Playerを取得し、そこからベースクラスにキャスト。

2.同期ステップについて

「**ServerInteract**」関数の詳細について記すことで、どのような流れでサーバー側の変数が変化するかについて説明

```
// ================================================================
// ▼ 【サーバー側の処理】 RPCを受け取って実行する部分 ▼
// ================================================================
void AMyCharacterBase::Server_Interact_Implementation(AActor* TargetActor,const FInteractionPayload& Payload)
{
    // 引数で受け取ったアクターが有効か、インターフェースを実装しているかをサーバー側で再検証
    if (TargetActor == nullptr || !TargetActor->Implements<UInteractableInterface>())
    {
        return;
    }

    // 検証をパスしたら、対象アクターのインターフェース関数を呼び出す
    // これにより、実際の処理をプレイヤーキャラクターから対象アクターへと委譲する
    IInteractableInterface::Execute_RequestInteraction(TargetActor, Payload);
}
```

ServerInteract関数の処理

以上がServerInteract関数の処理であるが、具体的な内容は、
第一引数のアクター（TargetActor）に
インターフェース（InteractableInterface）が実装されていれば
そのインターフェースの関数（RequestInteraction）関数を呼ぶというだけの処理。

※サーバーに処理を頼む関数は_Implementationが後ろにつく。インターフェースの関数は手前にExecute_がつく。

以上の設計によって、移動以外の自分で同期処理(アクターの状態変化など)をさせたいアクター（**TargetActor**）には必ず（**InteractableInterface**）を実装させるというルールがこのプロジェクトにはあるので覚えておいてほしい。

InteractableInterfaceの使い方

処理をするアクター側

基本的にアクターのベースCPPクラスに実装する。
以下実装例（製品アクター）：

1．ヘッダファイル
・IInteractableInterfaceを実装する。
・RequestInteraction関数を宣言する。(_Implementationが付いているのはBlueprintNativeEventであるから)
※GetLifetimeReplicatedProps関数は同期したい変数を登録みたいな処理がある(これについてはUEの決まりごとで調べればすぐに出てくるし、すぐに理解できると思うから説明省略)

```cpp
UCLASS(Blueprintable)
class ILLEGALFACTORY_API AProductActorBase : public AActor, public IInteractableInterface
{
    GENERATED_BODY()

public:
    AProductActorBase();
```

```cpp
public:
    // レプリケーション登録
    virtual void GetLifetimeReplicatedProps(TArray<FLifetimeProperty>& OutLifetimeProps) const override;

    // インタラクション処理（IInteractableInterface 実装）
    virtual void RequestInteraction_Implementation(const FInteractionPayload& Payload) override;
```

2. CPPファイル

・RequestInteraction関数は以下のようにFInteractionPayload引数（後述）に入っているInteractionTypeの情報から、様々な処理を条件分岐する形で作成する。

・分岐の中に具体的な処理を書くとコードが大変なことになるのでインターフェース(BPで実装している)に逃がしたり、関数を作ったりして、RequestInteraction関数はすっきりさせたい。

```cpp
// ========================================
// ◇ インタラクション処理
// ========================================
void AProductActorBase::RequestInteraction_Implementation(const FInteractionPayload& Payload)
{
    switch (Payload.InteractionType)
    {
    case EInteractionType::Use:
        if (this->Implements<UUsable>())
        {
            IUsable::Execute_Use(this, Payload.TargetActor, Payload.InstigatorActor);
        }
        break;

    case EInteractionType::Pickup:
        if (this->Implements<UCarryable>())
        {
            ICarryable::Execute_Carry(this, Payload.ParentActor, Payload.TargetLocation);
        }
        break;

    case EInteractionType::Place:
        if (this->Implements<UCarryable>())
        {
            ICarryable::Execute_Release(this, Payload.TargetLocation);
        }
        break;

    case EInteractionType::Hide:
        if (this->Implements<UDetectable>())
        {
            IDetectable::Execute_SetIsHidden(this, true);
        }
        break;

    case EInteractionType::Reveal:
        if (this->Implements<UDetectable>())
        {
            IDetectable::Execute_SetIsHidden(this, false);
        }
        break;
```

FInteractionPayloadとEInteractionType

・FInteractionPayloadは目標座標や対象のアクターなど処理に必要な情報を持つ。すべての変数を毎回入れる必要はなく、EInteractionTypeごとに必要なものだけを入れる。また、他に必要な変数ができたらその都度追加してほしい。

```cpp
// インタラクションの種類を定義するEnum
UENUM(BlueprintType)
enum class EInteractionType : uint8
{
    // デフォルト、または何もしない場合
    Default   UMETA(DisplayName = "デフォルト"),

    // オブジェクトを使用する（ドアを開ける、スイッチを押すなど）
    Use       UMETA(DisplayName = "使う"),

    // オブジェクトを拾い上げて持つ
    Pickup    UMETA(DisplayName = "持つ/拾う"),

    // 持っているオブジェクトを置く
    Place     UMETA(DisplayName = "置く"),

    // オブジェクトを隠す、またはキャラクターが隠れる
    Hide      UMETA(DisplayName = "隠す"),

    // 隠されたオブジェクトを見つける、または隠れている状態から出る
    Reveal    UMETA(DisplayName = "見つける/出す"),

    // オブジェクトを移動させる（押す、引く、動かすなど）
    Move      UMETA(DisplayName = "移動させる"),

    // オブジェクトを回転させる
    Rotate    UMETA(DisplayName = "回転させる"),

    // オブジェクトを移動させた後削除する(例：製品を売る処理、箱に入れる処理)
    DeliverAndRemove UMETA(DisplayName = "移動後削除"),

    // オブジェクトをピュア状態にする（正常な製品に戻す）
    MakePure  UMETA(DisplayName = "ピュア状態にする"),

    // オブジェクトをジャンク状態にする（不良品に変える）
    MakeJunk  UMETA(DisplayName = "ジャンク状態にする"),

    // オブジェクトをON状態にする（電源を入れる、起動するなど）
    TurnOn    UMETA(DisplayName = "オンにする"),

    // オブジェクトをOFF状態にする（電源を切る、停止するなど）
    TurnOff   UMETA(DisplayName = "オフにする"),
};
```

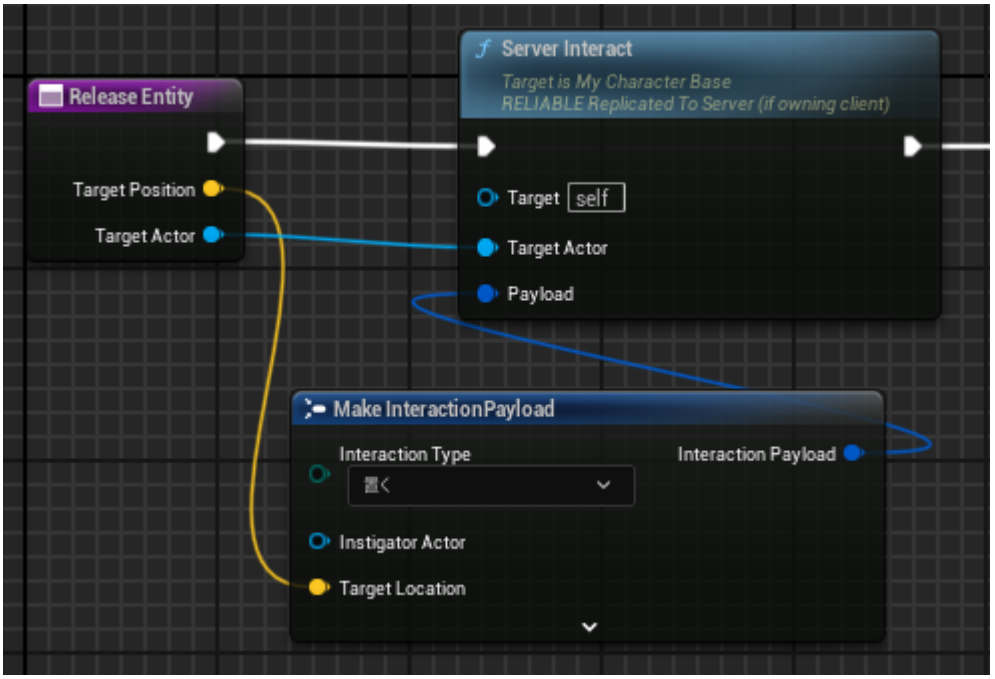
```cpp
USTRUCT(BlueprintType)

struct FInteractionPayload
{
    GENERATED_BODY()

    // どのアクションか
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
    EInteractionType InteractionType = EInteractionType::Default;

    // 誰がインタラクトしてきたか
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
    TObjectPtr<AActor> InstigatorActor = nullptr;

    // 目標座標
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
    FVector TargetLocation = FVector::ZeroVector;

    // 目標回転
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
    FRotator TargetRotation = FRotator::ZeroRotator;

    // 親にするアクター
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
    TObjectPtr<USceneComponent> ParentActor = nullptr;

    // 布アニメーションを実行するためのGeometricData
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
    FActorGeometricData GeometricData;

    // 対象アクター（何に対して使うか）
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
    TObjectPtr<AActor> TargetActor = nullptr;

    // 以降データを追加したい場合は以上を参考に...
    // UPROPERTY(EditAnywhere, BlueprintReadWrite)
    // float DamageAmount = 0.f;

    FInteractionPayload() = default;
};
```

処理を要請する側

・例１:

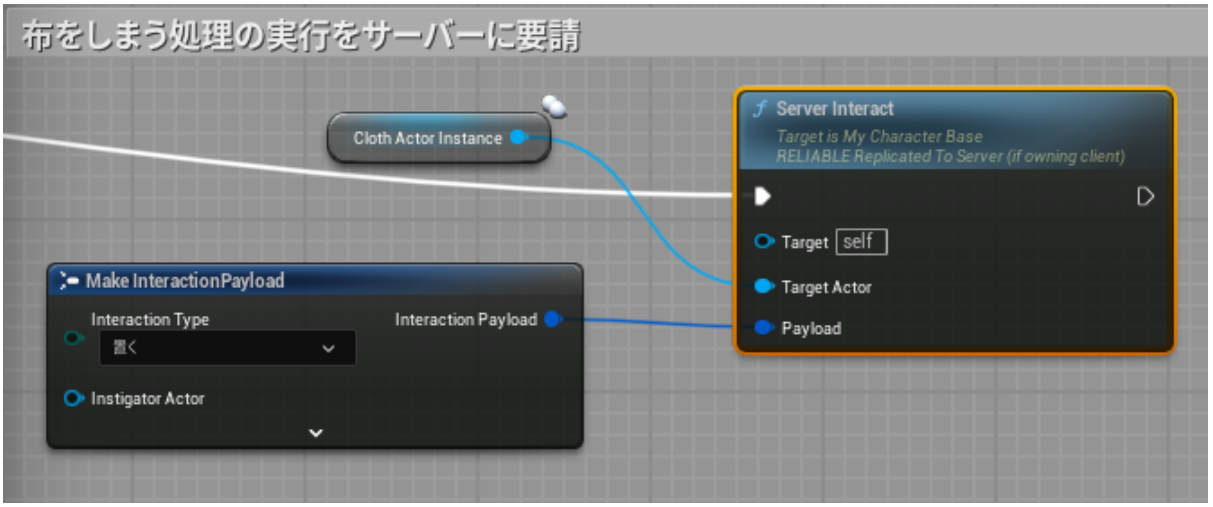
「置く」という処理は目標座標が必要なのでInteractionPayloadのTargetLocationに置く場所を入れる。

※引数「TargetActor」は IInteractableInterfaceを実装しているアクターを指定する



・例２:

布をしまうという処理であるが、しまうというInteractionTypeがないので「置く」というInteractionTypeを指定している。これは布に対して他に「置く」というようなアクションがないからできること。もしちゃんとしたいなら「しまう」というInteractionTypeを追加してもいい。



3.演出ステップについて

変数が同期されただけではゲーム上には何も影響が出ていない。

同期された変数の内容をゲームに反映させるための方法についてまとめる

その方法は大きく分けて2つある。**On_rep**関数と**Multicast RPC**。
現在、**DarkFactory**プロジェクトでは**On_rep**関数しか使っていない。
理由は**Multicast RPC**がなんか上手くいかないことがあったからと**On_rep**関数の方が手間はかかるがメリットは多いから。

On_rep関数

Repnotify変数と呼ばれる変数を宣言する。Repnotify変数の宣言時には、変数の値が変化したときに呼びたい関数（On_rep関数）を登録するところがある。
（ReplicateUsing = On_rep関数）

```
UPROPERTY(EditAnywhere, BlueprintReadWrite, ReplicatedUsing = OnRep_IsActive, Category = "Product State")
bool bIsActive = false;
```

※注意点
On_rep関数の処理はクライアント側でしか呼ばれないため、サーバーではOn_rep関数と同じ処理を呼ぶ必要がある。(On_rep関数をそのままサーバーで呼ぶのは関数の意味的に良くないらしい…On_rep関数にさらに共通の関数を用意するのがベストだが、まあ、面倒ならOn_rep関数をそのまま呼んでもいいかも)

Multicast RPC

サーバーとクライアントに対して一斉に同じ処理を実行させることができる。
Repnotify(Replicated)変数を用意する必要がないため手間が省けるが、
処理負荷が少し大きいのと、その場で一回しか処理が行われないため、プレイヤーが途中参加した場合に対して弱い。（このゲームで途中参加は想定してないが、気を付けておくに越したことはない？）

4.アニメーションの同期について

Discordで共有した動画を見てもらうのが早い

4.まとめ（このプロジェクトのルール）

1. クライアント側がサーバーに処理を頼みたいときは**MyCharacterBase**（**BP_Player**）の**ServerInteract**関数を使う。
2. 同期処理が必要なアクターには、**InteractableInterface**を実装する。
3. 基本**On_rep**関数を使う。**Multicast RPC** はできるだけ使わない。
4. 継続的なアニメーションには**AnimationBP**を使い、単発のアニメーションには**GAS**を使う。