# Character Movement Component Replication

| | |
|---|---|
| Created | @March 8, 2022 3:55 PM |
| Tags | |

- 이 문서는 UE4.27 기준으로 작성되었습니다.
- 이 문서는 언리얼 공식 문서인 Character Movement Component의 번역으로 주로 이루어져 있지만 이해를 돕기 위해 코드를 추가하기도 하였습니다.
- Character Movement Component
  - https://docs.unrealengine.com/4.27/en-US/InteractiveExperiences/Networking/CharacterMovementComponent/
- 이 문서와 Character Movement Component 한국어 문서를 병행하여 읽으시면 이해에 도움이 되리라 생각합니다.

## Basics of Character Movement

- 캐릭터 무브먼트는 먼저 클라이언트에서 자기 자신의 이동을 시뮬레이션(예측)합니다.
  - 여기에서 시뮬레이션이라고 하는 이유는 오직 서버만이 권한을 갖기 때문입니다.
  - 즉, 플레이어의 입력에 따라 움직이는 것은 서버이고, 클라이언트는 그저 시뮬레이션하는 것에 지나지 않는 것입니다.
  - 시뮬레이션이라고도 할 수 있고 예측(Predict)라고 할 수 있습니다.
- 그 후, 클라이언트는 자신의 이동 정보(입력 정보, 시뮬레이션 결과)를 서버에 보내게 되고 서버는 입력 정보를 토대로 움직임을 계산합니다.
- 서버는 클라이언트가 보내온 시뮬레이션 결과가 맞는지 확인하고 맞다면 좋은 이동이었다는 사인을 보냅니다.
- 그렇지 않다면 서버의 이동 결과를 클라이언트에게 보내고 클라이언트는 서버로부터 온 데이터를 통해 움직임을 보정합니다.
- CharacterMovementComponent를 통한 네트워크 게임에서의 매 프레임 플레이어 이동 예측, 리플리케이션, 보정 방식은 다음과 같습니다.
  1. TickComponent() 함수가 호출됩니다.
  2. 해당 프레임에서의 가속도를 계산하고, MovementMode에 따라 캐릭터를 이동시키는 PerformMovement() 함수가 호출됩니다.
  3. 이동했던 정보들 (위치, 회전, MovementMode 등등)이 SavedMoves 배열에 저장됩니다.
  4. 캐릭터의 이동 정보들을 서버로 리플리케이션합니다.
  5. 서버에서는 리플리케이션 정보를 가지고 다시 클라이언트의 움직임을 계산합니다.
  6. 서버에서 완료된 움직임에 대한 정보를 클라이언트에 리플리케이션시키고 그 정보와 클라이언트에서의 움직임 정보와 비교합니다.
  7. 차이가 크면 서버가 알려준 정보로 다시 위치를 갱신합니다.

## PerformMovement and Movement Physics

- 싱글플레이 게임에서 UCharacterMovementComponent는 PerformMove()를 매틱마다 직접적으로 호출하게 됩니다.
- 반면에 멀티플레이 게임에서는 ReplicateMoveToServer()함수에서 PerformMovement()가 실행되고 이동을 마친후, 이 프레임에서의 이동에 필요했던 정보들을 서버로 리플리케이트, 서버에서도 똑같이 PerformMovement()를 실행합니다.
  - PerformMovement() 함수는 월드에서 캐릭터를 물리적으로 움직이는데 대한 책임을 가지고 있습니다.

```
// 이해에 불필요한 코드는 모두 생략하였습니다.
// 밑의 코드는 실제 코드와 반드시 일치하지는 않습니다.

void UCharacterMovementComponent::TickComponent(float DeltaTime, enum ELevelTick TickType, FActorComponentTickFunction *ThisTickFunction)
{
  // ConsumeInputVerctor()를 하면 AddMovementInput()으로 입력했던 이동 벡터를 가져 옵니다.
  // 아래 코드를 다음과 같습니다.
  // AddMovementInput(DirectionToMove, MoveAmount);
  // const Vector InputVector = DirectionToMove * MoveAmount;
  const FVector InputVector = ConsumeInputVector();

  Super::TickComponent(DeltaTime, TickType, ThisTickFunction);

  if (CharacterOwner->GetLocalRole() > Role_SimulatedProxy)
  {
    // PredictionData는 시뮬레이션한 이동 정보들의 모임입니다.
    // UCharacterComponent는 클라이언트에서의 이동정보들과 서버에서의 이동정보들을 저장하고 비교하게 됩니다.
    FNetworkPredictionData_Client_Character* ClientData = GetPredictionData_Client_Character();

    if (ClientData && ClientData->bUpdatePosition)
    {
      // 서버로부터 받은 이동정보를 이용, 클라이언트가 조종하는 캐릭터의 위치를 보정합니다.
      ClientUpdatePositionAfterServerUpdate();
    }
      ControlledCharacterMove(InputVector, DeltaTime);                  // Simulate Autonomously...
  }
  else if (CharacterOwner->GetLocalRole() == ROLE_SimulatedProxy)
  {
     SimulatedTick(DeltaTime);                                          // Simulate Other Client's Character...
  }
}

void UCharacterMovementComponent::ControlledCharacterMove(const FVector& InputVector, float DeltaSeconds)
{
  // 점프 실행
  CharacterOwner->CheckJumpInput(DeltaSeconds);

  // Acceleration은 InputVector로부터 정해집니다.
  // InputVector는 APawn::AddMovementInput()에서 결정됩니다.
  // InputVector * MaxAccelration이 최종 Acceleration이 됩니다.
  // 즉, UE에서의 Acceleration은 폰이 입력으로 받는 방향벡터인 것입니다.
  Acceleration = ScaleInputAcceleration(InputVector)

  if (CharacterOwner->GetLocalRole() == ROLE_Authority)
  {
    // 싱글플레이 게임에서는 여기로 들어옵니다.
    PerformMovement(DeltaSeconds);
  }
  else if (CharacterOwner->GetLocalRole() == ROLE_AutonomousProxy && IsNetMode(NM_Client))
  {
    // 멀티플레이 게임에서는 여기로 들어옵니다.
    ReplicateMoveToServer(DeltaSeconds, Acceleration);
  }
}
```

- ReplicateMoveToServer()로 들어가고 PerformMovement()를 호출하게 되는데 PerformMovement()에서는 다음 사항들을 조절, 실행합니다.
  - 힘과 중력같은 외부 물리적 요소들을 적용합니다.
  - 애니메이션 루트모션으로부터 움직임을 계산합니다.
  - StartNewPhysics()를 호출하는데 이 함수는 캐릭터가 이용하는 MovementMode에 따라 Phys*() 함수를 선택하여 실행합니다.

```
void UCharacterMovementComponent::StartNewPhysics(float deltaTime, int32 Iterations)
{
  //  코드 생략...

  switch ( MovementMode )
  {
  case MOVE_None:
    break;
  case MOVE_Walking:
    PhysWalking(deltaTime, Iterations);
    break;
  case MOVE_NavWalking:
```

```
    PhysNavWalking(deltaTime, Iterations);
    break;
  case MOVE_Falling:
    PhysFalling(deltaTime, Iterations);
    break;
  case MOVE_Flying:
    PhysFlying(deltaTime, Iterations);
    break;
  case MOVE_Swimming:
    PhysSwimming(deltaTime, Iterations);
    break;
  case MOVE_Custom:
    PhysCustom(deltaTime, Iterations);
    break;
  default:
    UE_LOG(LogCharacterMovement, Warning, TEXT("%s has unsupported movement mode %d"), *CharacterOwner->GetName(), int32(MovementMode));
    SetMovementMode(MOVE_None);
    break;
  }

  // 코드 생략...
}
```

- 각각의 Phys*() 함수는 물리적으로 이동을 책임지게 되는데 Acceleration을 가공하고(예를 들면 PhysWalking()의 경우 Acceleration.Z를 0으로 만듭니다.) Velocity를 계산합니다.
- 만약 캐릭터가 떨어지기 시작하거나, 어떤 물체에 충돌하는 것과 같이 틱 도중에 MovementMode가 바뀌게 되면, Phys*() 함수들은 StartNewPhysics() 함수를 다시 호출하여 새로운 MovementMode에 따른 캐릭터의 이동을 계산합니다.
- 그래서 StartNewPhysics()와 Phys*() 함수들은 StartNewPhysics가 몇번 호출되었는지에 카운팅하게 되는데 MaxSimulationIterations에 따라 재귀적으로 몇번 호출하는지를 제한하게 됩니다.

# Movement Replication Summary

- UCharacterMovementComponent는 CharacterOwner의 Network Role에 따라서 리플레케이션 방식을 결정합니다.

```
// EngineTypes.h

/** The network role of an actor on a local/remote network context */
UENUM()
enum ENetRole
{
  /** No role at all. */
  ROLE_None,
  /** Locally simulated proxy of this actor. */
  ROLE_SimulatedProxy,
  /** Locally autonomous proxy of this actor. */
  ROLE_AutonomousProxy,
  /** Authoritative control over the actor. */
  ROLE_Authority,
  ROLE_MAX,
};
```

| Network Role | Description |
|---|---|
| Simulated Proxy | 다른 플레이어 혹은 AI가 컨트롤하는 캐릭터일 것입니다. |
| Autonomous Proxy | 플레이어가 컨트롤하는 캐릭터일 것입니다. |
| Authority | 게임을 호스팅하는 서버에 이 캐릭터가 존재하는 경우입니다. 서버는 이 캐릭터를 컨트롤할 권한을 가지고 있습니다. |

- 리플리케이션 프로세스는 TickComponent()에서 매틱마다 순환되게 됩니다. 캐릭터가 이동을 하면 Remote Procedure Calls(RPCs) 함수를 이용, 서버와 다른 클라이언트에 플레이어가 컨트롤하는 캐릭터 움직임이 리플리케이트되고, 동기화시킵니다.
- 다음 테이블은 UCharacterMovementComponent가 어떻게 각각의 클라이언트에 캐릭터 이동을 리플리케이션하는지 스텝-바이-스텝 개요입니다.

| | | |
|---|---|---|

| Step | Subject | Description |
|---|---|---|
| 1 | Autonomous Proxy(Owning Player's Client) | 내가 조종하는 캐릭터의 움직임을 계산합니다. |
| 2 | Autonomous Proxy(Owning Player's Client) | 캐릭터의 이동정보를 저장하는 FSavedMove_Character를 생성하고 SavedMoves에 넣습니다. SavedMoves는 배열이지만 큐처럼 작동합니다. |
| 3 | Autonomous Proxy(Owning Player's Client) | CallServerMove_Packed()함수를 호출하여 이동 정보들을 서버에 전송합니다. 이동 정보에는 이동을 마친 후의 위치, 회전값, MovementMode와 이동을 하기전에 필요한 값(ClientTimeStamp, Acceleration, MoveFlags, ClientControlRotation)을 포함합니다. |
| 4 | Authoritative Actor(Server) | 전송받은 정보를 이용해서 PerformMovement()를 실행, 클라이언트의 이동을 재현합니다. |
| 5 | Authoritative Actor(Server) | 서버가 재현한 클라이언트의 이동후의 정보와 클라이언트가 보고한 위치 사이의 차이를 알아봅니다. |
| 6 | Authoritative Actor(Server) | 차이가 충분히 작은 경우에는 클라이언트에게 움직임이 유효하다라는 신호를 보내고 그렇지 않으면 보정된 위치를 ClientAdjusstPosition() RPC 함수를 호출하여 전송합니다. |
| 7 | Authoritative Actor(Server) | 또 서버는 이 캐릭터의 위치, 회전값, 상태를 다른 클라이언트에 ReplicatedMovement를 리플리케이트 합니다. ReplicatedMovement는 AActor의 멤버로서 리플리케이트된 정보를 담습니다. |
| 8 | Autonomous Proxy(Owning Player's Client) | 클라이언트가 ClientAdjustPosition() RPC 함수를 통해 보정값을 받게되면 움직임을 보정하고 SavedMoves를 이용, 보정 이후의 이동을 다시 재생하여 최종 위치를 업데이트합니다. 움직임이 최종적으로 완료되면 그 움직임 정보를 SavedMoves에서 제거합니다. |
| 9 | Simulated Proxy(All Other Clients) | 그저 ReplicatedMovement를 적용하여 Position을 업데이트합니다. NetworkSmoothing() 함수는 비주얼적으로 마지막 모션을 부드럽게 처리하게 합니다. |

- 위 일련의 과정들이 멀티플레이 게임에서 세가지의 타입(Authority, AutonomousProxy, SimulatedProxy)의 머신들을 동기시킵니다. 유저가 컨트롤하는 캐릭터는 서버로부터의 간섭을 최소한으로 받아야 하고 다른 유저들의 근사된 움직임을 볼수있어야 합니다.
- 유저들이 자신의 캐릭터를 컨트롤하는데 최대한 부드러운 경험을 할 수 있게 하기 위해서 이 프로세스들의 복잡함은 자율 프록시와 서버사이에서의 이동 예측과 보정을 조정하는데 초점을 맞추고 있습니다.
- 반면에 시뮬레이티드 프록시는 그저 서버가 말하는 대로 움직이면 될 것입니다.

# Replicated Character Movement In-Depth

- 이 섹션에서는 위에서 다루었던 과정들을 자세하게 다룹니다.
- 대부분의 프로젝트들은 UCharacterMovementComponent의 함수들을 오버라이드할 필요가 없으나 어떤 기능만들거나 수정하기 위해서 자세하게 들여볼 필요가 있기 때문에 이 섹션을 기술합니다.
- 이 섹션은 캐릭터의 보통의 이동(에픽이 정의해둔 이동들) 리플리케이션을 다룹니다.
- 루트모션이나 다른 액터를 기반으로 움직이는 경우에는 다른 코드패스(Code Path)를 통과할 것이지만 이 섹션에서 기술하는 단계와 유사할 것입니다.

## Local Movement on the Owning Client

- 자율 프록시는 TickComponent()에서 이동을 처리하고 기록합니다. 그리고 서버에 이동 정보를 보내고 서버에서는 자율프록시에서 처리된 이동을 재현하고 보정합니다. 이 섹션은 매 틱마다 자율프록시가 어떻게 움직임을 처리하는지에 대해 기술합니다.

### Building Client Prediction Data

- 자율프록시는 ClientPredictionData로 이름지어진 FNetworkPredictionData_Client_Character 객체를 만듭니다.
- ClientPredictionData는 이동 프로세스의 일부로서 이동정보를 기록하고 서버로부터의 보정을 수행하게 됩니다.
- FNetworkPredictionData_Client_Character는 다음의 멤버들을 포함합니다.
  - 언제 서버와 커뮤니케이션했는지를 기록하는 Timestamps
  - TArray<FSavedMovePtr> SavedMoves
  - 서버로부터 보정된 정보
  - 어떻게 보정할 것인지를 표시하는 Flags

    - 부드러운 이동을 처리하는데 이용되는 변수들
- 아래의 코드들은 많은 부분이 생략되어 있고 꼭 필요하다고 생각되는 것들을 포함합니다.

```
// UCharacterMovementComponent.h

UCLASS()
class ENGINE_API UCharacterMovementComponent : public UPawnMovementComponent, public IRVOAvoidanceInterface, public INetworkPredictionInter
{
  GENERATED_BODY()

  // ... 생략 ...
protected:
  class FNetworkPredictionData_Client_Character* ClientPredictionData;
  class FNetworkPredictionData_Server_Character* ServerPredictionData;
  // ... 생략 ...
}

typedef TSharedPtr<class FSavedMove_Character> FSavedMovePtr;

class ENGINE_API FNetworkPredictionData_Client_Character : public FNetworkPredictionData_Client, protected FNoncopyable
{
public:

  FNetworkPredictionData_Client_Character(const UCharacterMovementComponent& ClientMovement);
  virtual ~FNetworkPredictionData_Client_Character();

  /** Client timestamp of last time it sent a servermove() to the server. This is an increasing timestamp from the owning UWorld. Used for
  float ClientUpdateTime;

  /** Current TimeStamp for sending new Moves to the Server. This time resets to zero at a frequency of MinTimeBetweenTimeStampResets. */
  float CurrentTimeStamp;

  /** Last World timestamp (undilated, real time) at which we received a server ack for a move. This could be either a good move or a corre
  float LastReceivedAckRealTime;

  TArray<FSavedMovePtr> SavedMoves;   // Buffered moves pending position updates, orderd oldest to newest. Moves that have been acked by th
  FSavedMovePtr PendingMove;        // PendingMove already processed on client - waiting to combine with next movement to reduce client to
  FSavedMovePtr LastAckedMove;      // Last acknowledged sent move.

  uint32 bUpdatePosition:1; // when true, update the position (via ClientUpdatePosition)

  // ... 생략 ...
}
```

- FNetworkPredictionData_Client_Character API (https://docs.unrealengine.com/4.27/en-US/API/Runtime/Engine/GameFramework/FNetworkPredictionData_Client_Ch-/)

### Reproducing Server Corrections

- 플레이어의 인풋이나 힘(AddForce() 혹은 AddImpulse()등으로 생긴 캐릭터에 적용되는 힘을 말하는 것)을 처리하기 전에 자율프록시는 ClientUpdatePositionAfterServerUpdate()를 호출합니다. 여기에서는 서버가 보내온 보정이 있는지 확인합니다. 보정이 있으면 bool FNetworkPredictionData_Client_Character::bUpdatePosition이 참이 되고 캐릭터는 서버가 보내온 보정을 처리하게 됩니다.
- 더 많은 정보를 원하시면 밑에 Handling Client Error and Corrections 섹션을 참고하십시오.

### Performing and Recording Movement

- 자율 프록시에서는 TickComponent()에서 PerformMovement()를 바로 실행하지 않고 ReplicateMoveToServer()를 실행합니다. 이 함수에서 PerformMovement()를 호출하고 이동정보를 기록하기 위해 필요한 로직들을 처리하게되고 이 이동정보를 서버에 보내게 됩니다.
- FSavedMove_Character 구조체는 매 틱마다 자율프록시의 이동 시작 정보, 이동 완료 정보를 기록합니다.
- FSavedMove_Character 구조체는 다음의 멤버들을 포함합니다.
    - 캐릭터의 마지막 위치와 회전값에 대한 정보

- 플레이어가 어떤 이동 키를 입력했는지(이를테면 bPressedJump)
- 캐릭터의 Velocity와 Acceleration
- 루트모션 정보

```
// UCharacterMovementComponent.h

class ENGINE_API FSavedMove_Character
{
public:
  FSavedMove_Character();
  virtual ~FSavedMove_Character();

  ACharacter* CharacterOwner;

  uint32 bPressedJump:1;
  uint32 bWantsToCrouch:1;
  uint32 bForceMaxAccel:1;

  uint32 bWasJumping:1;

  float TimeStamp;    // Time of this move.
  float DeltaTime;    // amount of time for this move

  // Information at the start of the move
  uint8 StartPackedMovementMode;
  FVector StartLocation;
  FVector StartRelativeLocation;
  FVector StartVelocity;

  // Information after the move has been performed
  uint8 EndPackedMovementMode;
  FVector SavedLocation;
  FRotator SavedRotation;
  FVector SavedVelocity;

  FVector Acceleration;

  // ... 생략 ...
};
```

- FSavedMove_Character API (https://docs.unrealengine.com/4.27/en-US/API/Runtime/Engine/GameFramework/FSavedMove_Character/)

- PerformMovement()를 처리한 후 ReplicateMoveToServer() 함수 안에서 클라이언트가 예측한 이동정보를 FSavedMove_Character NewMove에 저장하고 SavedMoves에 추가합니다.

```
// CharacterMovementComponent.cpp
// 생략된 코드가 많습니다. 설명에 필요한 코드만 포함합니다...

void UCharacterMovementComponent::ReplicateMoveToServer(float DeltaTime, const FVector& NewAcceleration)
{
  FNetworkPredictionData_Client_Character* ClientData = GetPredictionData_Client_Character();

  FSavedMovePtr NewMovePtr = ClientData->CreateSavedMove();

  // 캐릭터가 이동하기 전의 정보를 담습니다. 위치나 속도 등...
  NewMove->SetMoveFor(CharacterOwner, DeltaTime, NewAcceleration, *ClientData);

  if (const FSavedMove_Character* PendingMove = ClientData->PendingMove.Get())
  {
    if (PendingMove->CanCombineWith(NewMovePtr, CharacterOwner, ClientData->MaxMoveDeltaTime * CharacterOwner->GetActorTimeDilation(*MyWorl
    {
      NewMove->CombineWith(PendingMove, CharacterOwner, PC, OldStartLocation);

      // Remove pending move from move list. It would have to be the last move on the list.
      if (ClientData->SavedMoves.Num() > 0 && ClientData->SavedMoves.Last() == ClientData->PendingMove)
      {
        const bool bAllowShrinking = false;
        ClientData->SavedMoves.Pop(bAllowShrinking);
      }
      ClientData->FreeMove(ClientData->PendingMove);
```

```
      ClientData->PendingMove = nullptr;
    }
  }

  PerformMovement(NewMove->DeltaTime);

  // 캐릭터가 이동한 후의 정보를 담습니다.
  NewMove->PostUpdate(CharacterOwner, FSavedMove_Character::PostUpdate_Record);

  if (CharacterOwner->IsReplicatingMovement())
  {
    check(NewMove == NewMovePtr.Get());
    ClientData->SavedMoves.Push(NewMovePtr);

    // 이동정보를 서버로 송신
    CallServerMovePacked(NewMove, ClientData->PendingMove.Get(), OldMove.Get());
  }

  // ... 생략 ...
}
```

- SavedMoves는 FSavedMove_Character들을 시간의 역순(FIFO)으로 정렬되어 있고 큐처럼 작동합니다. 서버와의 통신에서 대역폭을 줄이기 위해 비슷한 움직임의 정보들은 하나로 합쳐집니다(코드에서는 Combine이라고 되어 있습니다).
- FSavedMovePtr FNetworkPredictionData_Client_Character::PendingMove는 새로 들어올 이동 정보와 결합을 기다리고 있는 이동 기록인데 PendingMove와 NewMove가 비슷한 경우 NewMove로 합쳐지게 됩니다.
- 서버로부터 Acked 신호를 받은, 즉 좋은 움직임이라고 인정을 받거나 클라이언트에서 위치 보정 후 Ack를 한 이동정보들은 SavedMoves에서 지워지게 됩니다.
- 마지막으로 Acked된 이동정보는 LastAckedMove라는 이름의 변수로 저장되고 미래의 위치 보정에 사용됩니다.

### Submitting Moves to the Server

- ReplicateMoveToServer()에서 위의 과정을 끝내게 되면 CallServerMovePacked()함수가 호출됩니다.
- CallServerMovePacked()에서는 새로운 이동정보와 SavedMoves에 있던 가장 오래된 이동정보(Acked되지 않은)를 서버로 송신합니다.

```
// CharacterMovementComponent.cpp

void UCharacterMovementComponent::CallServerMovePacked(const FSavedMove_Character* NewMove, const FSavedMove_Character* PendingMove, const
{
  // Get storage container we'll be using and fill it with movement data
  FCharacterNetworkMoveDataContainer& MoveDataContainer = GetNetworkMoveDataContainer();

  // Container에 FSavedMove_Character들을 채워 넣습니다.
  MoveDataContainer.ClientFillNetworkMoveData(NewMove, PendingMove, OldMove);

  // Reset bit writer without affecting allocations
  FBitWriterMark BitWriterReset;
  BitWriterReset.Pop(ServerMoveBitWriter);

  // Extract the net package map used for serializing object references.
  APlayerController* PC = Cast<APlayerController>(CharacterOwner->GetController());
  UNetConnection* NetConnection = PC ? PC->GetNetConnection() : nullptr;
  ServerMoveBitWriter.PackageMap = NetConnection ? NetConnection->PackageMap : nullptr;

  // Container의 데이터를 ServerMoveBitWriter에 직렬화합니다.
  MoveDataContainer.Serialize(*this, ServerMoveBitWriter, ServerMoveBitWriter.PackageMap)

  // Copy bits to our struct that we can NetSerialize to the server.
  // 'static' to avoid reallocation each invocation
  static FCharacterServerMovePackedBits PackedBits;
  PackedBits.DataBits.SetNumUninitialized(ServerMoveBitWriter.GetNumBits());

  // ServerMoveBitWriter가 가지고 있는 비트스트림을 복사합니다.
  FMemory::Memcpy(PackedBits.DataBits.GetData(), ServerMoveBitWriter.GetData(), ServerMoveBitWriter.GetNumBytes());

  // Send bits to server!
  ServerMovePacked_ClientSend(PackedBits);
}
```

- CallServerMovePacked()는 NewMove, PendingMove, OldMove를 직렬화하여 서버로 송신합니다.
- ServerMovePacked_ClientSend()는 unreliable한 서버 RPC 함수를 호출하여 서버에 데이터를 송신합니다.
- 서버 RPC함수가 unreliable한 두가지 이유가 있습니다.
  1. 보통의 게임플레이에서 서버 RPC함수가 reliable하다면 너무 많이 서버 RPC함수가 호출되기 때문에 reliable한 함수들을 위한 버퍼가 overflow될 수 있고 이것은 플레이어와 서버의 연결을 끊어버리도록 강제하기 때문입니다.
  2. 이동 정보들을 저장하는 시스템은 송수신할 때 정보를 잃어버리면 다시 송수신할 수 있도록 이미 설계되어 있습니다.

## Evaluating Movement on the Server

- 서버에서는 게임의 틱 싸이클을 클라이언트와 동기화시키는데 TickComponent()를 사용하지 않습니다.
- 대신에 서버는 클라이언트가 보낸 정보를 송신하면 해당 정보들을 이용, 클라이언트의 이동을 재현하게 됩니다.

```
// CharacterMovementComponent.cpp
void UCharacterMovementComponent::ServerMovePacked_ClientSend(const FCharacterServerMovePackedBits& PackedBits)
{
  // Pass through RPC call to character on server, there is less RPC bandwidth overhead when used on an Actor rather than a Component.
  CharacterOwner->ServerMovePacked(PackedBits);
}

// Character.cpp
void ACharacter::ServerMovePacked_Implementation(const FCharacterServerMovePackedBits& PackedBits)
{
  GetCharacterMovement()->ServerMovePacked_ServerReceive(PackedBits);
}

// CharacterMovementComponent.cpp
void UCharacterMovementComponent::ServerMovePacked_ServerReceive(const FCharacterServerMovePackedBits& PackedBits)
{
  // Deserialize bits to move data struct...
  ServerMoveBitReader.SetData((uint8*)PackedBits.DataBits.GetData(), NumBits);
  ServerMoveBitReader.PackageMap = PackedBits.GetPackageMap();

  FCharacterNetworkMoveDataContainer& MoveDataContainer = GetNetworkMoveDataContainer();
  MoveDataContainer.Serialize(*this, ServerMoveBitReader, ServerMoveBitReader.PackageMap);

  ServerMove_HandleMoveData(MoveDataContainer);   // 이 함수에서 ServerMove_PerformMovement() 실행하여 자율프록시의 이동을 재현
}
```

### Building Server Prediction Data

- 서버에 존재하는 캐릭터의 캐릭터 무브먼트 컴포넌트는 ServerPredictionData라고 불리는 FNetworkPredictionData_Server_Character 객체를 생성하고 이 객체는 캐릭터의 생명주기와 동일한 생명주기를 갖습니다.
- ServerMove_PerformMovement()에서 ServerPredictionData는 추후에 클라이언트의 이동을 재현하기 위해 정보들을 저장합니다.
- 이 객체는 다음의 정보들을 저장합니다.
  - 서버의 delta time을 계산하기 위해 사용된 Timestamps
  - 어떻게 보정할지에 대한 정보
  - 시간 불일치를 해결할 Flags
  - 서버가 Ack할지 위치를 보정할지를 나타내는 Flags
- FNetworkPredictionData_Server_Character API (https://docs.unrealengine.com/4.27/en-US/API/Runtime/Engine/GameFramework/FNetworkPredictionData_Server_Ch-/)

### Verifying Client Timestamp And Calculating Delta Time

- 해킹을 방지하기 위해 서버와 클라이언트의 이동이 언제 발생했는지 timestamp를 확인하고 서버와 클라이언트의 timestamp의 불일치가 크면 클라이언트의 이동을 재현할 때 서버의 timestamp를 사용합니다. 이동 해킹은 보통 클라이언트의 시간을 빠르게 만들어서 하기 때문입니다.

### Evaluating the Move

- 서버는 캐릭터의 이동을 재현하기 위해 ServerMove_PerformMovement()함수안에서 MoveAutonomous() 함수를 호출하게 됩니다.
- MoveAutonomous()는 PerformMovement() 함수를 호출하게 됩니다. 클라이언트가 이동하기 시작한 위치가 아닌 서버에 기록된 캐릭터의 이전 위치를 사용하여 이동을 재현하게 됩니다.
- 만약 캐릭터가 루트모션 애니메이션을 재생하고 있다면 MoveAutonomous()는 애니메이션의 틱을 처리합니다.

## Handling Client Error and Corrections

- 서버 이동처리 작업은 클라이언트와 서버가 같은 위치에서 이동을 시작한 것으로 가정합니다.
- 클라이언트가 서버에 전송한 데이터로 서버가 이동처리를 하면 클라이언트나 서버에서의 캐릭터의 이동 후의 위치는 같을 것입니다.
- 그러나 만약 클라이언트의 이동정보가 연결 문제로 인해 드랍되거나 혹은 잘못된 정보를 전송했을 경우 클라이언트와 서버에서의 캐릭터의 이동 후의 위치는 다를 것입니다.
- 이렇게 되면 보정이 필요하게 됩니다.
- ServerMoveHandleClientError()함수가 이러한 보정에 책임이 있습니다.
- ServerMoveHandleClientError()함수는 ServerMove_PerformMovement() 함수 안에서 MoveAutonomous()가 호출 된 뒤에 호출됩니다.

### Determining if Adjustments Are Necessary

- 보정이 너무 자주 일어나게되면 대역폭에 부담을 주고 클라이언트로 하여금 너무 잦은 재현을 하도록 하기 때문에 ServerMoveHandleClientError()함수에서 AGameNetworkManager의 WithinUpdateDelayBounds()를 호출하여 이동 보정사이의 최소시간이 지났는지 확인합니다.
  - false를 반환하면 보정이 일어나지 않고 함수를 빠져나갑니다.
- 그리고 ServerCheckClientError() 함수를 호출하여 서버와 클라이언트의 이동 정보가 커서 보정이 필요한지 확인하게 됩니다.
  - true를 반환하거나 ServerData의 bForceClientUpdate가 true로 되어 있으면 나머지 작업을 진행합니다.
- 위의 작업들에 사용되는 파라미터들을 BaseGame.ini에서 확인할 수 있고 프로젝트에 맞춰서 DefaultGame.ini에 이런 파라미터들을 오버라이드할 수 있습니다.
- ClientErrorUpdateRateLimit는 초단위로 서버가 클라이언트에게 보정하는 최소한의 딜레이를 정의하고 있습니다.
- MAXPOSITIONERRORSQUARED는 최대 포지션 에러의 제곱으로 이 값을 넘어서면 보정이 일어나게 됩니다.
- 이 값들은 [/Script/Engine.GameNetworkManager] 섹션에서 찾아볼 수 있습니다.

```
; BaseGame.ini

[/Script/Engine.GameNetworkManager]
MAXPOSITIONERRORSQUARED=3.0f
ClientErrorUpdateRateLimit=0.0f

// CharacterMovementComponent.cpp
void UCharacterMovementComponent::ServerMoveHandleClientError(float ClientTimeStamp, float DeltaTime, const FVector& Accel, const FVector& 
{
  FNetworkPredictionData_Server_Character* ServerData = GetPredictionData_Server_Character();

  // Don't prevent more recent updates from being sent if received this frame.
  // We're going to send out an update anyway, might as well be the most recent one.
  APlayerController* PC = Cast<APlayerController>(CharacterOwner->GetController());
  if( (ServerData->LastUpdateTime != GetWorld()->TimeSeconds))
  {
```

```
    const AGameNetworkManager* GameNetworkManager = (const AGameNetworkManager*)(AGameNetworkManager::StaticClass()->GetDefaultObject());
    if (GameNetworkManager->WithinUpdateDelayBounds(PC, ServerData->LastUpdateTime))
    {
      return;
    }
  }

  if (ServerData->bForceClientUpdate || ServerCheckClientError(ClientTimeStamp, DeltaTime, Accel, ClientLoc, RelativeClientLoc, ClientMovem
  {
    // 클라이언트가 보정해야할 정보들을 준비합니다...
    // 보정이 필요하다고 판단되면 서버는 보정데이터를 FClientAdjustment PendingAdjustment 구조체에 담습니다.
    UPrimitiveComponent* MovementBase = CharacterOwner->GetMovementBase();
    ServerData->PendingAdjustment.NewVel = Velocity;
    ServerData->PendingAdjustment.NewBase = MovementBase;
    ServerData->PendingAdjustment.NewBaseBoneName = CharacterOwner->GetBasedMovement().BoneName;
    ServerData->PendingAdjustment.NewLoc = FRepMovement::RebaseOntoZeroOrigin(UpdatedComponent->GetComponentLocation(), this);
    ServerData->PendingAdjustment.NewRot = UpdatedComponent->GetComponentRotation();

    ServerData->PendingAdjustment.bBaseRelativePosition = MovementBaseUtility::UseRelativeLocation(MovementBase);
    if (ServerData->PendingAdjustment.bBaseRelativePosition)
    {
      // Relative location
      ServerData->PendingAdjustment.NewLoc = CharacterOwner->GetBasedMovement().Location;

      // TODO: this could be a relative rotation, but all client corrections ignore rotation right now except the root motion one, which wo
      //ServerData->PendingAdjustment.NewRot = CharacterOwner->GetBasedMovement().Rotation;
    }

    ServerData->LastUpdateTime = GetWorld()->TimeSeconds;
    ServerData->PendingAdjustment.DeltaTime = DeltaTime;
    ServerData->PendingAdjustment.TimeStamp = ClientTimeStamp;
    ServerData->PendingAdjustment.bAckGoodMove = false;
    ServerData->PendingAdjustment.MovementMode = PackNetworkMovementMode();
  }
  else
  {
    // acknowledge receipt of this successful servermove()
    // 보정이 필요하지 않다면 PendingAdjustment의 bAckGoodMove 값을 true로 설정합니다.
    ServerData->PendingAdjustment.TimeStamp = ClientTimeStamp;
    ServerData->PendingAdjustment.bAckGoodMove = true;
  }

  ServerData->bForceClientUpdate = false;
}

// GameNetworkManager.cpp
bool AGameNetworkManager::ExceedsAllowablePositionError(FVector LocDiff) const
{
  return (LocDiff | LocDiff) > GetDefault<AGameNetworkManager>(GetClass())->MAXPOSITIONERRORSQUARED;
}

// CharacterMovementReplication.h

// ClientAdjustPosition replication (event called at end of frame by server)
struct ENGINE_API FClientAdjustment
{
public:

  FClientAdjustment()
    : TimeStamp(0.f)
    , DeltaTime(0.f)
    , NewLoc(ForceInitToZero)
    , NewVel(ForceInitToZero)
    , NewRot(ForceInitToZero)
    , NewBase(NULL)
    , NewBaseBoneName(NAME_None)
    , bAckGoodMove(false)
    , bBaseRelativePosition(false)
    , MovementMode(0)
  {
  }

  float TimeStamp;
  float DeltaTime;
  FVector NewLoc;
  FVector NewVel;
  FRotator NewRot;
  UPrimitiveComponent* NewBase;
  FName NewBaseBoneName;
  bool bAckGoodMove;
  bool bBaseRelativePosition;
```

```
  uint8 MovementMode;
};
```

### Sending Client Adjustments or ACKing Moves

- 클라이언트에게 PendingAdjustment를 돌려 주기 위해 서버에서는 APlayerController::SendClientAdjustment() 함수를 호출합니다.
- 서버에서는 클라이언트의 위치를 계산하고 틱의 마지막에 UNetDriver::ServerReplicateActors()를 호출, 하고 거기에서 APlayerContoller::SendClientAdjustment() 함수를 호출합니다.
- APlayerController::SendClientAdjustment()에서는 PendingAdjustment를 ServerSendMoveResponse() 함수가 호출되어 클라이언트에 전달합니다.
- ServerSendMoveResponse() 함수에서는 PendingAdjustment를 비트 스트림으로 직렬화하여 클라이언트에 데이터를 전송합니다.
- 클라이언트는 PendingAdjustment를 받게 되고 이 데이터를 통해 클라이언트의 위치를 보정할 준비를 하게 됩니다.

```
// CharacterMovementComponent.cpp

void UCharacterMovementComponent::ClientHandleMoveResponse(const FCharacterMoveResponseDataContainer& MoveResponse)
{
  // 클라이언트에서 데이터를 수신 후 이 함수로 들어옵니다.
  // 루트모션 관련 로직은 해서는 모두 생략했습니다.
  // MoveResponse는 서버로부터 온 데이터를 역직렬화하여 저장되어 FClientAdjustment를 포함하고 있습니다.

  if (MoveResponse.IsGoodMove())
  {
    ClientAckGoodMove_Implementation(MoveResponse.ClientAdjustment.TimeStamp);
  }
  else
  {
    ClientAdjustPosition_Implementation(
        MoveResponse.ClientAdjustment.TimeStamp,
        MoveResponse.ClientAdjustment.NewLoc,
        MoveResponse.ClientAdjustment.NewVel,
        MoveResponse.ClientAdjustment.NewBase,
        MoveResponse.ClientAdjustment.NewBaseBoneName,
        MoveResponse.bHasBase,
        MoveResponse.ClientAdjustment.bBaseRelativePosition,
        MoveResponse.ClientAdjustment.MovementMode);
  }
}

void UCharacterMovementComponent::ClientAdjustPosition_Implementation
  (
  float TimeStamp,
  FVector NewLocation,
  FVector NewVelocity,
  UPrimitiveComponent* NewBase,
  FName NewBaseBoneName,
  bool bHasBase,
  bool bBaseRelativePosition,
  uint8 ServerMovementMode
  )
{
  // 생략된 코드가 많습니다. 이해를 돕기 위해 꼭 필요한 코드만 포함합니다.
  // 밑의 코드는 실제 클라이언트의 위치, 회전, 속도를 보정하는 코드입니다.

  FNetworkPredictionData_Client_Character* ClientData = GetPredictionData_Client_Character();

  // Trust the server's positioning.
  if (UpdatedComponent)
  {
    UpdatedComponent->SetWorldLocation(WorldShiftedNewLocation, false, nullptr, ETeleportType::TeleportPhysics);
  }

  Velocity = NewVelocity;

  // Trust the server's movement mode
  UPrimitiveComponent* PreviousBase = CharacterOwner->GetMovementBase();
  ApplyNetworkMovementMode(ServerMovementMode);

  // Set base component
  UPrimitiveComponent* FinalBase = NewBase;
```

```
  FName FinalBaseBoneName = NewBaseBoneName;
  if (bUnresolvedBase)
  {
    check(NewBase == NULL);
    check(!bBaseRelativePosition);

    // We had an unresolved base from the server
    // If walking, we'd like to continue walking if possible, to avoid falling for a frame, so try to find a base where we moved to.
    if (PreviousBase && UpdatedComponent)
    {
      FindFloor(UpdatedComponent->GetComponentLocation(), CurrentFloor, false);
      if (CurrentFloor.IsWalkableFloor())
      {
        FinalBase = CurrentFloor.HitResult.Component.Get();
        FinalBaseBoneName = CurrentFloor.HitResult.BoneName;
      }
      else
      {
        FinalBase = nullptr;
        FinalBaseBoneName = NAME_None;
      }
    }
  }
  SetBase(FinalBase, FinalBaseBoneName);

  // Update floor at new location
  UpdateFloorFromAdjustment();
  bJustTeleported = true;

  // Even if base has not changed, we need to recompute the relative offsets (since we've moved).
  SaveBaseLocation();

  LastUpdateLocation = UpdatedComponent ? UpdatedComponent->GetComponentLocation() : FVector::ZeroVector;
  LastUpdateRotation = UpdatedComponent ? UpdatedComponent->GetComponentQuat() : FQuat::Identity;
  LastUpdateVelocity = Velocity;

  UpdateComponentVelocity();
  ClientData->bUpdatePosition = true;
}
```

### Receiving Client Adjustments on Autonomous Proxies

- 클라이언트에서 보정은 TickComponent에서 ClientUpdatePositionAfterServerUpdate() 함수를 통해 이뤄집니다.

## Replicating Movement to Simulated Proxies

- Simulated Proxy인 캐릭터들은 그저 서버로부터 온 정보로 시뮬레이션합니다.
- 물리적으로 움직임을 계산하는 것 대신 이동 정보를 서버로부터 받아서 위치, 회전, 속도 등 업데이트해야 하는 이동정보를 업데이트합니다.
- Simulated Proxy의 움직임을 더 부드럽고 신뢰할 수 있게 해주는 여러 추가적인 프로세스들 또한 있습니다.

### Ticking Movement on Simulated Proxies

- UCharacterMovementComponent가 TickComponent를 돌릴 때 캐릭터가 Simulated Proxy이면 SimulatedTick()이 호출됩니다.
- SimulatedTick()에서는 최근에 리플리케이트된 이동데이터에 따라 움직임을 계속합니다.
- 하지만 그저 이동데이터를 리플리케이트 받아서 업데이트하기만 한다면 캐릭터가 마치 계속해서 텔레포트하는 것 처럼 보일 것입니다.
- 그 이유는 보통 네트워크로 데이터를 전송하는 속도가 로컬에서 머신이 렌더링하는 속도보다 느리기 때문입니다.
- 예를 들면 클라이언트는 240Hz의 refresh rate를 가진 모니터에서 렌더링을 할 때 리플리케이트된 이동정보는 30Hz의 속도로 전송될 수 있습니다.
- 이런 움직임을 부드럽게 하기 위해 Simulated Proxy는 받은 정보로 바로바로 움직이는 게 아니라 현재의 이동정보를 서버로부터 받은 이동정보로 보간을 시킵니다.

- 이러한 보간은 SmoothClientPosition()함수에서 책임을 지고 이 함수에서는 어떤 보간(Interpolate나 Extrapolate를 의미하는 듯)을 사용하는지 판단하기 위해 NetworkSmoothingMode()라는 함수를 호출하여 결정합니다.