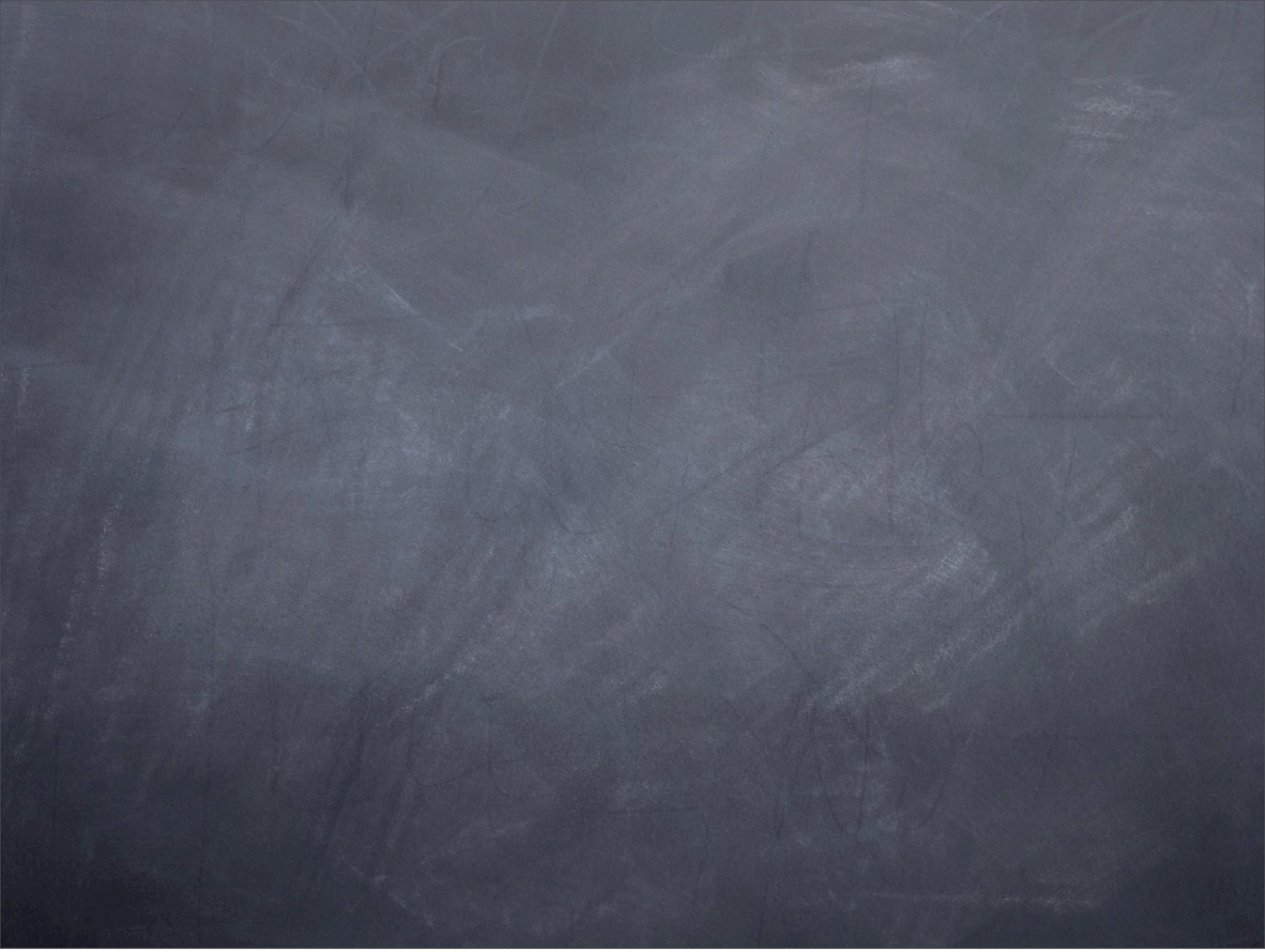


- I am: Senior Programmer,
- Currently developing - 4 Player Co-op Combat/Action RPG
  Using Unreal Engine

The Unreal Networking Model

Thaddaeus Frogley

CLIMAX

- I am: Senior Programmer,

- Currently developing - 4 Player Co-op Combat/Action RPG
  Using Unreal Engine

About This Talk

- Implementing Shared Reality
- Game State & Object Model
- Not: TCP/UDP, packets, protocols, or platform/hardware topics.

This talk is all about how programmers can give two or more people, sat on opposite sides of the world, the impression they are sharing an experience, a common environment, where their actions have consequences, be it in co-operation or in competition - that they are in the same place seeing the same things.

It is about how we can model the details of that experience using a high level programming language, and how we keep those separate game states in sync.

What is Unreal

- Epic Games' Licensed Game Engine Technology (Unreal Engine 3)
- PC, PS3, Xbox360
- Unreal, UT, Gears of War

I'm going to talk about Unreal, because it is a game engine that provides an interesting, powerful, flexible framework & toolset for doing that job.

You don't necessarily need to be using Unreal to benefit from this talk. Learning about how this problem is solved by other people may lead you towards improvements to your own custom solutions as well.

Keep in mind that I am not an official representative of Epic, just an employee of a licensee. My opinions are my own.

Its important to understand that although much of the game logic in an Unreal Game is written in the Unreal scripting language, which has support for networking, the language and engine just provide the framework. Writing game code in UnrealScript does not mean it is networked "for free". Deciding how objects you create work over the network is your job as a programmer. Not doing so up front will result in many headaches at the end.

Multi-player...

- Single Player Games
- Split Screen Multiplayer
- P2P Network
- Client/Server
- UnReal's "generalised client-server model"

When considering how multiplayer works with the unreal model we should look quickly at how distributed players introduces new problems to solve, and how epic have arrived at their solution.

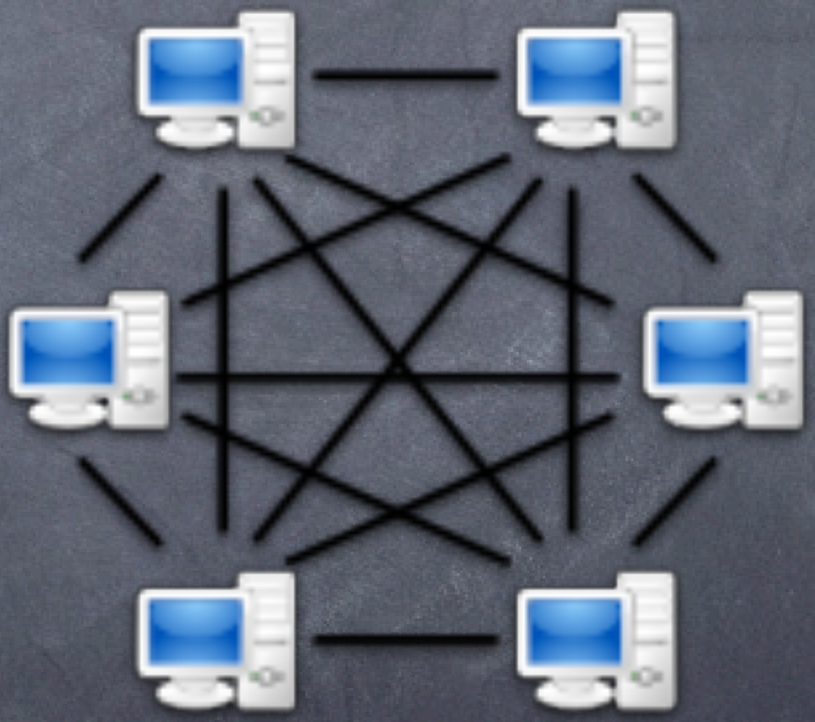
With single player games there is a single game model being seen by a single player - we can have a class "model, controller, view" structure, and there is no problem to solve.

With split-screen-multiplayer there is still only a single model, the only difference is that the model will be viewed from multiple perspectives. With model correctly de-coupled from view and controller, this transition is relativly simple.

Once you genuinely have multiple participants, with separately stored game states, you need to have a scheme for communicating to keep the models in sync, the two distinct approaches are:
 * P2P - which I will quickly talk about on the (NEXT SLIDE)
 * And Client/Sever - (FOLLOWING SLIDE)
The unreal model is a generalised C/S Model - (which is the REST OF THE TALK)

Peer-to-Peer

- MAZE, Doom, Duke Nukem, Magic & Mayhem...
- Start and stay strictly synchronised
- Fixed frame rate / lock-step ticks



The first P2P game (MAZE) was made in the early 70s, at NASA – two machines linked together, each one sending out it's players position.

In nontrivial applications, just sending positions isn't enough, since other game state changes happen depending on the players actions, so it is common with p2p systems for each machine / game instance to collects it's player input, distributes it to all participants, then once all input is gathered from all players the game model is stepped, and the process is repeated.

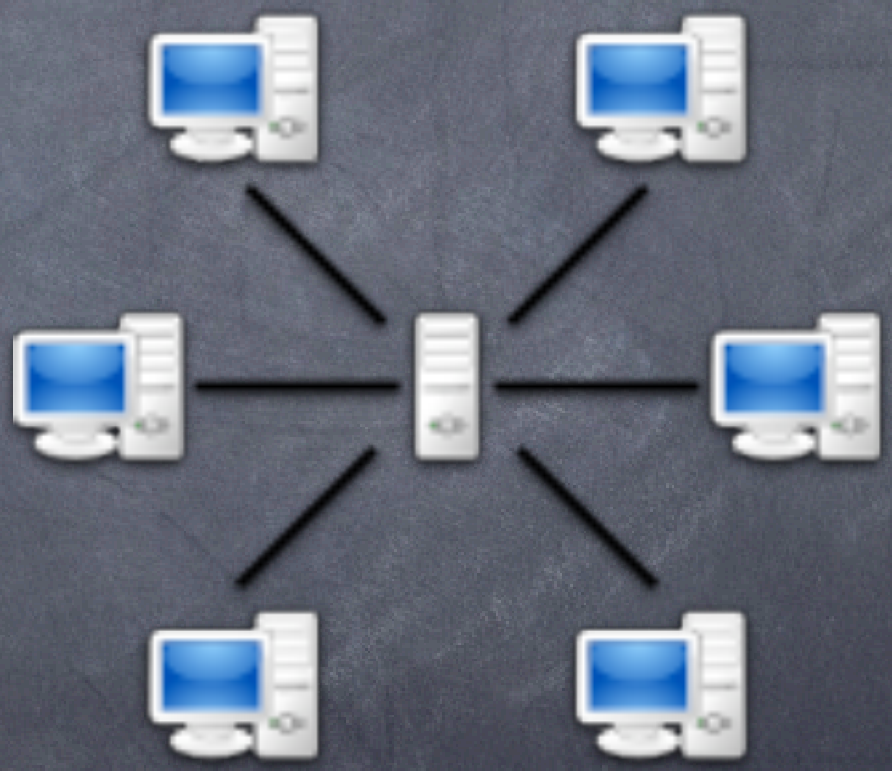
So game state is kept in sync, by synchronising all input, and keeping the game model deterministic.

This means frame rate (or rather, user input processing) is linked to ping, so how responsive the game feels to the players input is dependant on network responsiveness. Even on the fastest connection, every user is slowed down to the pace of the one on the slowest hardware.

LAN gaming.

Server-Client

- MUD, Quake, UT, MMOs...
- Join at anytime
- Client framerate not limited by network speed



The first, or one of the first, multiplayer games using a server/client architecture was Dr Richard Bartle's MUD, created in the late 70s, at Essex University.

With an ultra-thin client such as you'd have for a text based multi-user-dungeon the game state is only really modelled in one location - on the server, the the server tells each client exactly what to display. This isn't much different to the single player or split screen multiplayer we spoke of earlier. The next step from that, needed to support graphical games, was to send game state from which the players view point could be rendered on the client.

Quake, and later UT, however, took this concept, and extended it to include Client Side simulation, so that the server would distribute not only the positions of the other players to but their velocities, and allowing interpolation & extrapolation of movement.

With this in place the stage was set for allowing players some level of autonomy over their movement, giving the "lag free" control FPS plays now expect.

Key Concepts

- Participants
- Unreal Object Model
- Actor Relevancy
- Actor Roles

Before I move on to the details of how this is managed with UnReal's generalised client/server model, I'd like to just go over some of the key concepts and terminology used by Unreal Engine.

Some of this info might seem a bit fragmented and like I'm jumping around without going into details, but hopefully it'll all come together by the time I've finished.

Once I've summarised these 4 key concepts I'll explain the actual communication tools used to keep the game models in sync.

Participants

First of all, we have participants.

The first participant is always the server. The server is the definitive, or authoritative instance of the game world. Where different machines disagree, the server is the one that is always assumed to be correct. Furthermore, the server has additional workload in this model, and is of course, the hub for all communication.

All other participants in a game are clients, they are not authoritative and potentially only have a subset of the world at any one time. Each client is associated with a player (owner).

The Unreal model also supports a "meta-participant": The listen server (aka Host player), which combines the Server and Client participation roles into one - this is a single instance of the game running (not just a server and a client running on one machine).

Participants

- Server

First of all, we have participants.

The first participant is always the server. The server is the definitive, or authoritative instance of the game world. Where different machines disagree, the server is the one that is always assumed to be correct. Furthermore, the server has additional workload in this model, and is of course, the hub for all communication.

All other participants in a game are clients, they are not authoritative and potentially only have a subset of the world at any one time. Each client is associated with a player (owner).

The Unreal model also supports a "meta-participant": The listen server (aka Host player), which combines the Server and Client participation roles into one - this is a single instance of the game running (not just a server and a client running on one machine).

Participants

- Server
- Client(s)

First of all, we have participants.

The first participant is always the server. The server is the definitive, or authoritative instance of the game world. Where different machines disagree, the server is the one that is always assumed to be correct. Furthermore, the server has additional workload in this model, and is of course, the hub for all communication.

All other participants in a game are clients, they are not authoritative and potentially only have a subset of the world at any one time. Each client is associated with a player (owner).

The Unreal model also supports a “meta-participant”: The listen server (aka Host player), which combines the Server and Client participation roles into one - this is a single instance of the game running (not just a server and a client running on one machine).

Participants

- Server
- Client(s)
- Also: Listen Server (aka Host)
  Server & Client Combined

First of all, we have participants.

The first participant is always the server. The server is the definitive, or authoritative instance of the game world. Where different machines disagree, the server is the one that is always assumed to be correct. Furthermore, the server has additional workload in this model, and is of course, the hub for all communication.

All other participants in a game are clients, they are not authoritative and potentially only have a subset of the world at any one time. Each client is associated with a player (owner).

The Unreal model also supports a "meta-participant": The listen server (aka Host player), which combines the Server and Client participation roles into one - this is a single instance of the game running (not just a server and a client running on one machine).

Unreal Object Model

- Object rooted single inheritance class hierarchy
- Object graph with explicit ownership
- Event driven
- Actor as base class for network aware objects

Secondly – the basics of the unreal model, before going into the details of how it's used to create shared realities, are as follows:

There is an object model, using a single inheritance type hierarchy, with the object instances having explicit child/parent ownership relationships.

Execution of code is triggered by game engine events – for example a game "tick" (or frame) is an event that is triggered for every object in the model, every frame. The tick event is passed the amount of time that has passed since the last frame, thus giving every object in the model an opportunity to run periodic code.

Within the class hierarchy there is an Actor class, which is the base class for all potentially networked objects.

Actor Relevancy

- Determined by the Server for each Client
- Relevant Actors are Replicated & Run

Thirdly there is actor relevancy. Given that clients potentially only see a sub set of the world, how is that subset decided?

The answer is that the SERVER determines actor relevancy for each of the clients participating in the game.

Relevancy determination can be totally customised; though the default relevancy conditions cover most needs. I'll have a quick look at some of the details of that in a moment.

Server tells each client when an actor become relevant to them (creating an "actor channel"), and the client spawns an actor of that class (default properties), and keeps them active (events) so long as they remain relevant.

Actor Relevancy

- Owned by the client
- Visible or recently visible
- Flagged as always relevant (`bAlwaysRelevant`)
- Customisable by overriding `IsNetRelevantFor`

The purpose of relevancy is to determine a limited subset of actors which the client needs to know about, so that bandwidth is not wasted sending information about objects the client cannot see or interact with.

Sensible default behaviour for most actor classes is built in, but can be customised per object, or overriden on a per class basis.

This is powerful & important point of customisation for objects in the Unreal networking model.

Actor Roles

- Role & Remote Role
- Relative & Symmetric

In addition to relevancy, actors have Roles. Their roles are defined as a pair of values, “Role” and “Remote Role”.

I describe these as Relative, since their value changes depending on their context (that is which participant is asking the question, and in the case of the server, during replication - which I'll come to later - which client they are replicating too).

I say Symmetric, since the two roles actually swap during replication. The role on the client is set from the remote role on the server, and the remote role on the client, is the same as the role on the server.

Actor Roles

- **Authority**
  `ROLE_Authority`
- **Autonomous Proxy**
  `ROLE_AutonomousProxy`
- **Simulated Proxy**
  `ROLE_SimulatedProxy`
- **None**
  `ROLE_None`

The values that this pair of members can have are as follows:

Authority ->
For any given actor, it's role can be authority on only one participant.
Everything on the Server has the role Authority

Autonomous -> Locally Controlled by non-server Client (ie Player Pawn and his Controller)

Simulated -> Pawns controlled by "other" participants, can only run functions explicitly marked as "simulated"

None -> used for RemoteRole only; actors local to one machine

Communication

- Data Replication
- Remote Procedure Calls

So, given a system of actors with roles, replicated according to relevancy, how are the actors on the clients kept in sync with the server's view of the game world, and how are the players actions on the client side communicated to the server, and the other clients. What are the communication tools provided by this model, and how are they used to maintain the illusion of a responsive shared reality...

The two core tools used for this are: Data Replication and Remote Procedure Calls.

Data Replication

- One to many, from the server to all client(s)
- Controlled by replication statements
- Triggers `ReplicationEvent` via `repnotify`
- Newly Relevant actors spawn as defined by their "`defaultproperties`"

The server tells clients what actors are relevant, and replicates their data (according to priority, within the limits of bandwidth).

What data is replicated is controlled by replication statements.

Replication triggers events in code.

Data replicated to a client has the potential to overwrite any changes made locally by that client.

Not every state change is replicated immediately, but all replicated data should eventually be synchronised. This mostly only affects values that change multiple times a tick, since replication is done between game ticks, but where bandwidth is saturated it may take several seconds for a low priority actor to be updated.

Replication Statement

```
replication
{
     if (RemoteRole==ROLE_AutonomousProxy)
        Mana, GuardianMagic;

     if (RemoteRole==ROLE_SimulatedProxy)
        NetAnimInfo, bTraversalInProgress;

     if (NetRagdollMode!=RagdollMode_None)
        RepdRBState;
}
```

A replication statement consists of conditions, and comma separated member variables.

If the condition is true for the client under consideration, the class variable is replicated. Replication conditions are tested separately by the server for each client.

Replication Statement

```
replication
{
    if (RemoteRole==ROLE_AutonomousProxy)
       Mana, GuardianMagic;

    if (RemoteRole==ROLE_SimulatedProxy)
       NetAnimInfo, bTraversalInProgress;

    if (NetRagdollMode!=RagdollMode_None)
       RepdRBState;
}
```

This example shows an actor that has variables (on the first line) that are only needed on the server & owning client - in this case that's the magical energy the player has, and which magical defence he has available. Neither of which are seen by other players.

Replication Statement

```
replication
{
    if (RemoteRole==ROLE_AutonomousProxy)
       Mana, GuardianMagic;

    if (RemoteRole==ROLE_SimulatedProxy)
       NetAnimInfo, bTraversalInProgress;

    if (NetRagdollMode!=RagdollMode_None)
       RepdRBState;
}
```

It also shows data that is only replicated to clients where the pawn is being simulated – data needed for the animation system to show the players actions correctly, data that is set up by the controlling client and passed to the server using RPC, which we'll come to shortly.

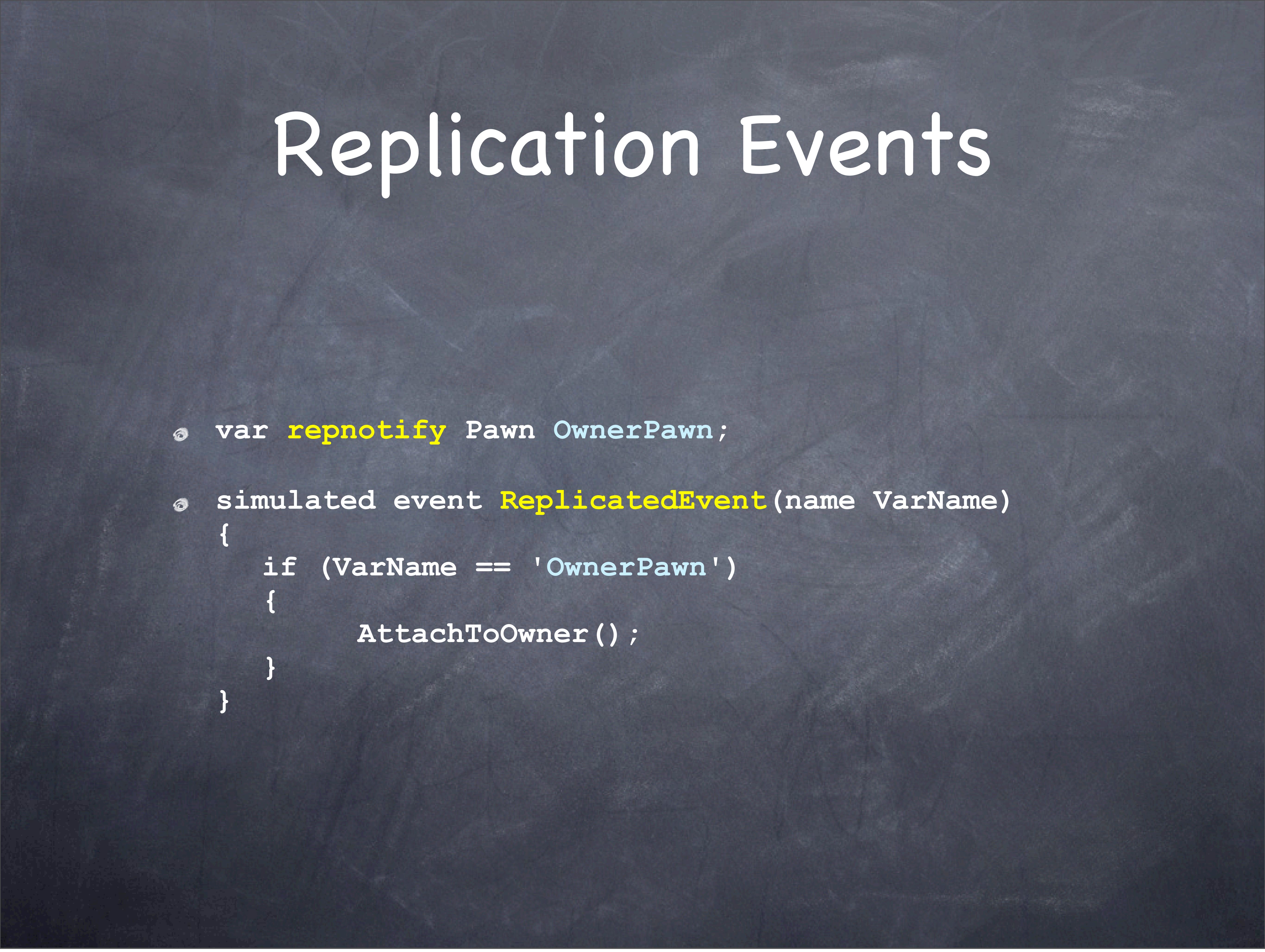
Replication Statement

```
replication
{
    if (RemoteRole==ROLE_AutonomousProxy)
       Mana, GuardianMagic;

    if (RemoteRole==ROLE_SimulatedProxy)
       NetAnimInfo, bTraversalInProgress;

    if (NetRagdollMode!=RagdollMode_None)
       RepdRBState;
}
```

Finally, it shows a condition not tied to roles, but to actor state – rigid body physics data only needs to be sent when the actor is in ragdoll.



Replication Events

- `var repnotify Pawn OwnerPawn;`
-
```
simulated event ReplicatedEvent(name VarName)
{
   if (VarName == 'OwnerPawn')
   {
         AttachToOwner();
   }
}
```

For any member variable we can also have the engine fire off events when replicated data arrives at the client, by marking the member as “repnotify”.

For instance, in this example, taken from the UTJumpBootEffect, the replication event is used to trigger the creation of particle effects on clients in the AttachToOwner function.

Remote Procedure Calls

RPCs are built into the scripting language directly, using a simple function declaration syntax.

It supports:
    server functions, where the client sends a message to the server

And
    client functions, where the server sends a message to the client

For client functions, the object ownership hierarchy determines which client a messages get sent to
---
UnrealScript also requires the function declaration to specify the whether the RPC needs to be reliable, or can be sent via an unreliable transport.

Remote Procedure Calls

- server functions
- client functions

RPCs are built into the scripting language directly, using a simple function declaration syntax.

It supports:
    server functions, where the client sends a message to the server

And
    client functions, where the server sends a message to the client

For client functions, the object ownership hierarchy determines which client a messages get sent to
---
UnrealScript also requires the function declaration to specify the whether the RPC needs to be reliable, or can be sent via an unreliable transport.

Remote Procedure Calls

- server functions
- client functions
- unreliable functions
- reliable functions

RPCs are built into the scripting language directly, using a simple function declaration syntax.

It supports:
    server functions, where the client sends a message to the server

And
    client functions, where the server sends a message to the client

For client functions, the object ownership hierarchy determines which client a messages get sent to
---
UnrealScript also requires the function declaration to specify the whether the RPC needs to be reliable, or can be sent via an unreliable transport.

Example

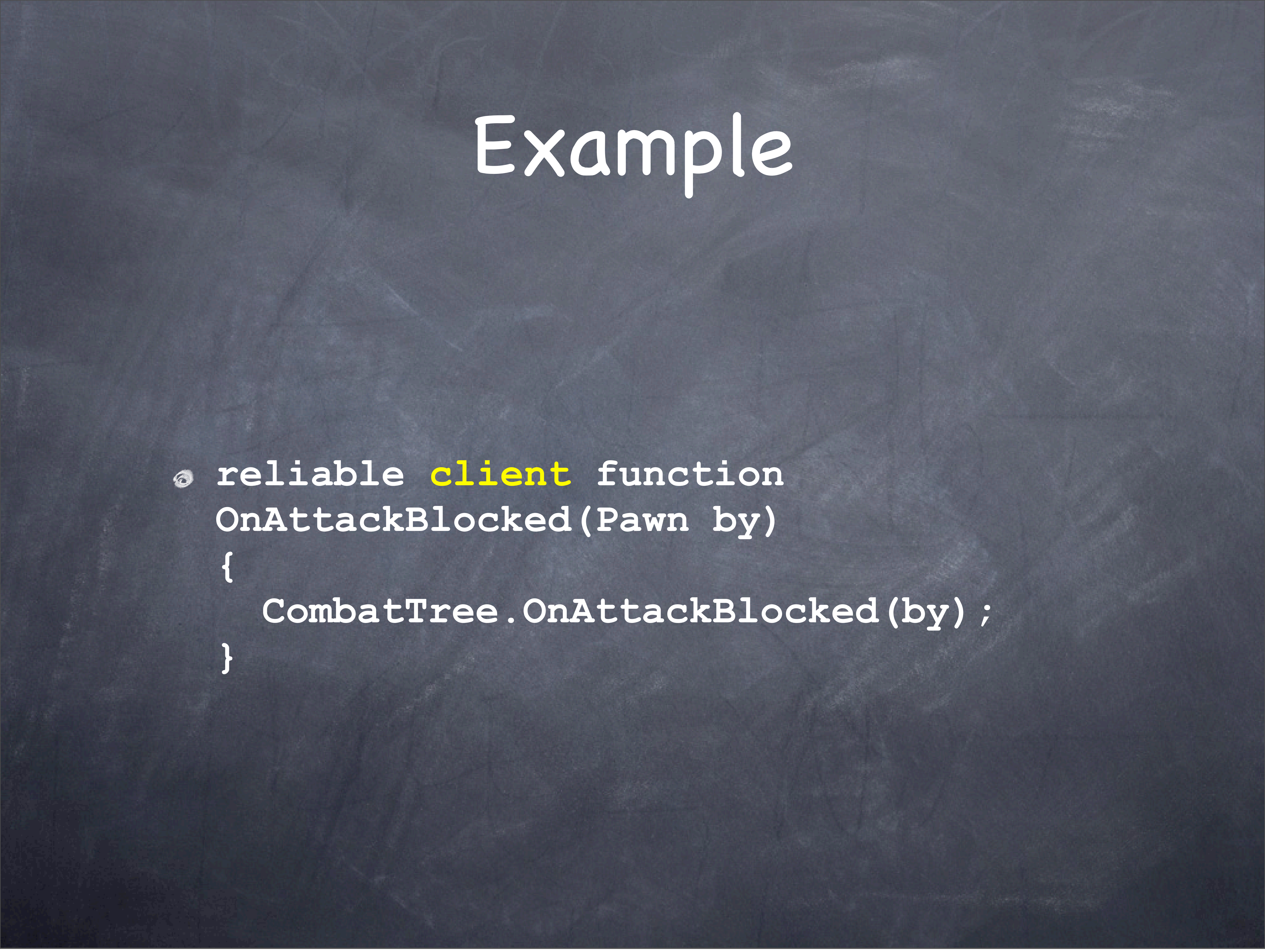
-
  ```
  unreliable server function
  ServerSendGroundSpeed(float newspeed)
  {
      Pawn.GroundSpeed = newspeed;
  }
  ```

A server function only executes locally if the actor's role is "Authority"

On a client (where Role < Authority) the function "call" wraps up a packet with the function arguments, and sends it to the server. It is not executed locally, and returns immediately. If a function invokes RPC and has a return value it will be none or zero.

The function is then executed on the server (in between ticks), once the packet has arrived.

Since there is no call context on the server, the return value is not used, nor is it passed back to the client, so server functions do not tend to have return values.



Example

```
reliable client function
OnAttackBlocked(Pawn by)
{
  CombatTree.OnAttackBlocked(by);
}
```

This example shows a client function, in this case, used by the server to tell a client that his attack has been blocked.

A client function runs locally if the actor it is a member of is owned (recursively up the graph) by the locally controlled pawn/player controller. If the owning pawn is being autonomously controlled by a client the server creates an RPC packet describing the call and sends it to that client, where it is executed on arrival.

If the function is called on a simulated actor, it is just ignored.

If a client function is passed an Actor reference which is not relevant or not (yet) replicated on the destination client it gets resolved (de-serialised) as “none” (null) on the client.

Reliable vs Unreliable

- **`reliable`** - always sent
- **`unreliable`** - sent bandwidth permitting

Functions replicated with the `unreliable` keyword are not guaranteed to reach the other party and, if they do reach the other party, they may be received out-of-order.

In practice the only things which can prevent an unreliable function from being received are network packet-loss, and bandwidth saturation.

Reliable functions should be used for sending information about game state change events, specifically player actions.

Unreliable functions should be used rarely - their main role is when sending regular but low priority value updates to the server. For most uses reliable functions should be preferred.

Communication Patterns

- Local / Server / Net
- Execute on ReplicatedEvent

So given these two primary tools - controlled replication of game state from server to client, and remote procedure calls from server to client and client to server, how do we use them in practice.

Whilst working with the unreal network model, I've identified 2 very common communication patterns, which are useful for understanding how the tools are used.

Local / Server / Net

- Local / Server / Net
- Local_foo -> implements the state change
- Server_foo -> a **`reliable server function`**, calls Local_foo
- Net_foo -> calls Local_foo, and if (**`Role<Role_Authority`**) Server_foo

This pattern is used when an action by the player can cause a state change, which needs to be duplicated on the server and seen by the player without them waiting for a network round trip.

A simple function with no RPC specifications is created to implement the state change, by convention I prefix this with “Local”.

A reliable server function is created named with the prefix Server, which simply calls Local_foo. By doing this we separate the implementation of the state change from the mechanism for RPC.

A 3rd function, prefixed Net, is created that calls both Local_foo and, if needed, Server_foo.

Target Lock-on

This is a Player Controller example from our game

showing the Local / Server / Net pattern - “Local” implements the state change, the server function makes sure the same state change happens on the server, and the “Net” function acts as a general front to the function that ensures the right calls are made depending if it was called on the client, or on the server.

When the player target-locks an enemy they see the consequences of their action immediately in game, without lag, and inform the server of the change.

If other clients, where this player is a simulated pawn, need to see the results of this action then that would need to be implemented separately via data replication.

The Local/Server/Net pattern is a Client to Server synchronisation pattern for keeping the server up to date, whilst avoiding client action lag.

Target Lock-on

```
private function LocalSetLockOnTarget(Pawn t)
{
    // ...
}
```

This is a Player Controller example from our game

showing the Local / Server / Net pattern – “Local” implements the state change, the server function makes sure the same state change happens on the server, and the “Net” function acts as a general front to the function that ensures the right calls are made depending if it was called on the client, or on the server.

When the player target-locks an enemy they see the consequences of their action immediately in game, without lag, and inform the server of the change.

If other clients, where this player is a simulated pawn, need to see the results of this action then that would need to be implemented separately via data replication.

The Local/Server/Net pattern is a Client to Server synchronisation pattern for keeping the server up to date, whilst avoiding client action lag.

Target Lock-on

-
  ```
  private function LocalSetLockOnTarget(Pawn t)
  {
      // ...
  }
  ```

-
  ```
  reliable server function ServerSetLockOnTarget(Pawn t)
  {
      LocalSetLockOnTarget(t);
  }
  ```

This is a Player Controller example from our game

showing the Local / Server / Net pattern – “Local” implements the state change, the server function makes sure the same state change happens on the server, and the “Net” function acts as a general front to the function that ensures the right calls are made depending if it was called on the client, or on the server.

When the player target-locks an enemy they see the consequences of their action immediately in game, without lag, and inform the server of the change.

If other clients, where this player is a simulated pawn, need to see the results of this action then that would need to be implemented separately via data replication.

The Local/Server/Net pattern is a Client to Server synchronisation pattern for keeping the server up to date, whilst avoiding client action lag.

Target Lock-on

-
  ```
  private function LocalSetLockOnTarget(Pawn t)
  {
      // ...
  }
  ```

-
  ```
  reliable server function ServerSetLockOnTarget(Pawn t)
  {
      LocalSetLockOnTarget(t);
  }
  ```

-
  ```
  function NetSetLockOnTarget(Pawn t)
  {
      ServerSetLockOnTarget(t);
      if (Role < Role_Authority)
      {
          LocalSetLockOnTarget(t);
      }
  }
  ```

This is a Player Controller example from our game

showing the Local / Server / Net pattern – "Local" implements the state change, the server function makes sure the same state change happens on the server, and the "Net" function acts as a general front to the function that ensures the right calls are made depending if it was called on the client, or on the server.

When the player target-locks an enemy they see the consequences of their action immediately in game, without lag, and inform the server of the change.

If other clients, where this player is a simulated pawn, need to see the results of this action then that would need to be implemented separately via data replication.

The Local/Server/Net pattern is a Client to Server synchronisation pattern for keeping the server up to date, whilst avoiding client action lag.

Execute on ReplicatedEvent

- Trigger variable marked as repnotify
- Server changes the variable & replicates that change to clients
- ReplicatedEvent on clients triggers execution of the required code

Execute on replicated event is a method for triggering the execution of code on any client that receives a value change via replication.

Its useful for spawning local objects, such as mesh attachments or particle effects that either do not need to exist on the server, would be to expensive to fully replicate and/or do not have to be kept in detailed sync.

We use this pattern to spawn weapon attachment and armour meshes, by replicating the attachment class type, and spawning an object of that type locally if the replicated value is a different type to the one we have currently on that client.

The UTJumpBootsEffect is an example of this pattern.

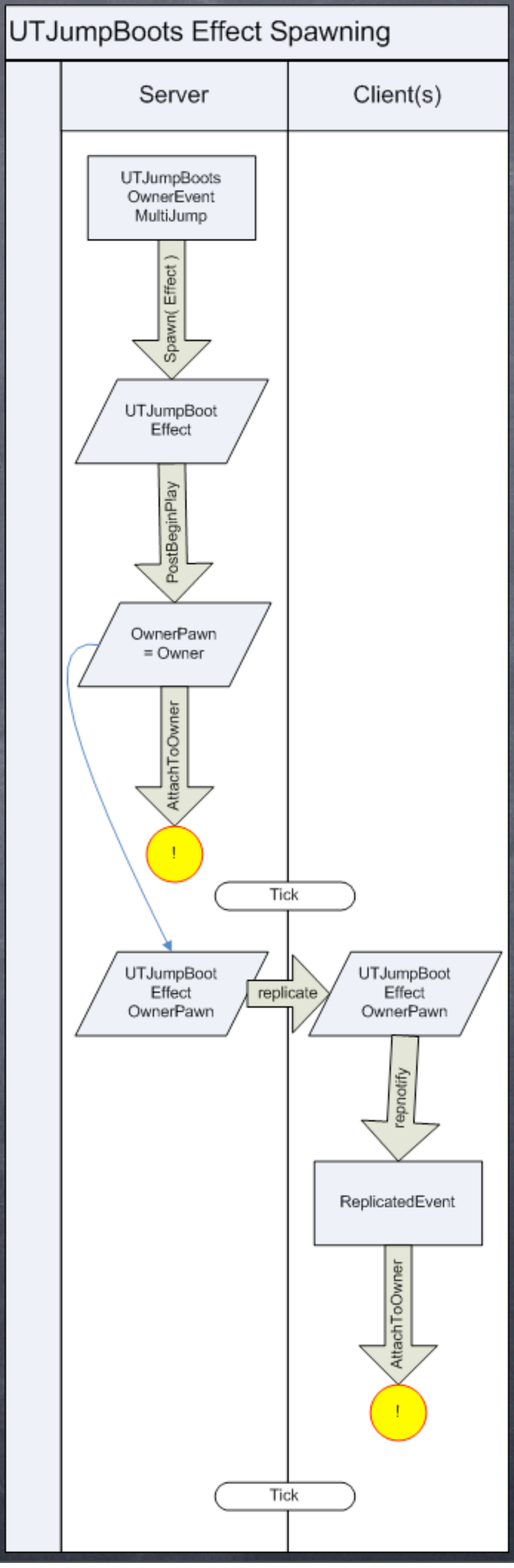
Examples

- Temporary Effect: UT Jump Boots
- Combat Collision in our current project

UT JumpBoots

```
/** pawn to spawn effects for */
var repnotify UTPawn OwnerPawn;

replication
{
    if (bNetInitial)
        OwnerPawn;
}
```



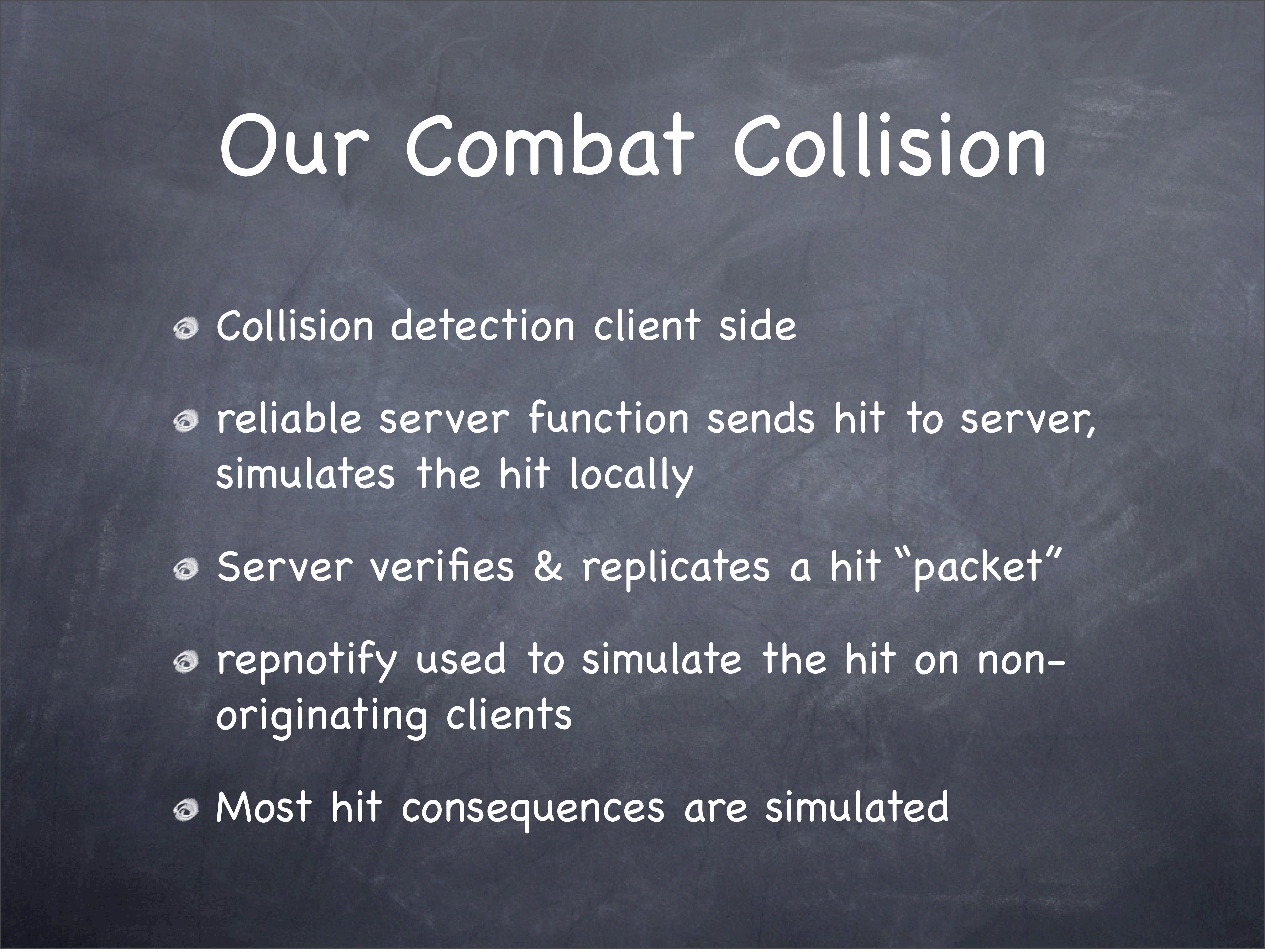
An event function in `UTJumpBoots` (an inventory item) spawns the `UTJumpBootEffect` on the server.

In the `PostBeingPlay` event (called on the server after an instance of the object is spawned) the `OwnerPawn` variable is set. This is a replicated variable (its listed in the replication statement) and its marked as `repnotify`. The `PostBeingPlay` also calls the `AttachToOwner` function. It is `AttachToOwner` that spawns the actual particle effects.

The `ReplicatedEvent` is triggered when the `OwnerPawn` is replicated to a client. That is used to trigger a call to `AttachToOwner` on each of the clients the object is replicated too.

As a diagram. the yellow circle represent the visible particle effect. It appears in the server column to show the case of a listen server, there is a specific condition tested to prevent that from actually happening on a dedicated server.

Note it is `bNetTemporary=true`, meaning that after initial replication, the Actor channel is closed and the Actor is never updated again. The actor will be destroyed by client once it's LifeSpan (0.2) has expired.



Our Combat Collision

- Collision detection client side
- reliable server function sends hit to server, simulates the hit locally
- Server verifies & replicates a hit "packet"
- repnotify used to simulate the hit on non-originating clients
- Most hit consequences are simulated

On my current project we are working hard to create a co-op multiplayer game, with the control responsiveness and collision accuracy of an arcade beat-em-up. To that end we've taken special care over how our combat collision system works, since up to 4 players can be swinging weapons in close proximity.

We've put the collision detection itself on the client, so that it frame-for-frame matches what the controlling player sees. Hits are passed to the server, and and hit-results are immediately simulated on the client. This is possible because any differences in the results on server and originating client are corrected automatically via replication.

Other clients that should see the hit & it's consequences do so during the repnotify event, when the "hit packet" is replicated back out by the server.

Limitations and Workarounds

- Client to Client Messages
- Multicast RPC
- Passing Interfaces across the network
- Changing Replication Conditions
- Subclassing for repnotify

Having gone through how the unreal networking model is set up, and what synchronisation tools it provides, along with some examples of how it can be used, I’d like to now review some if it’s limitations, and how they are avoided or worked around.

The first 3 are relatively simple to work around, as long as you are aware of them.

The last 2 can be more serious design constraints.

Client to Client Messages (RPC)

- All client to client communication goes via the server:
- `reliable client function ReceiveP2P( ... )`
  `{ ... }`
- `reliable server function SendP2P( Pawn to, ... )`
  `{ to.ReceiveP2P( ... ); }`

Client to client communication is not directly supported, but – as with all client/server architectures, the functionality can be easily supported by routing any client-to-client messages via the server.

Remember all clients in the game are associated with a player-controller, so that to make a remote procedure call from client A to client B you need a pair of functions in the player controller class, one server and one client, as shown here.

Client A makes a remote procedure call to the server by calling SendP2P, which in turn calls the client function ReceiveP2P, resulting in the code being executed on the client controlling the target player controller.

Multicast RPC

- From the server, manually call the (client) function on each of the player controllers:

-
  ```
  foreach PlayerController
      target.Function(...) //reliable client function ...
  ```
- example: `GameInfo.Broadcast()`

Similarly multi-casting an RPC from the server to all clients requires you explicitly call a client function for each participant.

There is an example of this in the unreal GameInfo class.

Though in I'd advise against this use of RPC, and favour data replication, since over use of remote procedure calls can saturate your bandwidth.

Passing Interfaces Across the Network

- Interfaces are not Actors

-
  ```
  interface IFoo extends Interface;
  // ...
  simulated function LocalDoFooThing( IFoo foo ) {...}
  simulated function NetDoFooThing( IFoo foo )
  {
    LocalDoFooThing( foo );
    if( Role < Role_Authority )
    {
      ServerDoFooThing(foo);
    }
  }
  ```

For those of you unfamiliar with UnrealScript, interfaces here are similar to interfaces in .NET or Java, and provide a mechanism similar to multiple inheritance in C++.

That is, they allow objects of different types to share common sets of functions, and be passed around as if the interface was a type itself.

So, given we have an some types that share an interface and we need to support player actions on those objects, we start to implement the Local / Server / Net pattern, like so –

– this slide shows the Local and Net parts of that, which will work as expected.

Passing Interfaces Across the Network

Unfortunately, for an object to be relevant at all it must be an Actor, and Interfaces are not necessarily Actors, thus this first attempt at implementing the “Server” part of the pattern will only ever pass “none” to the Local function on the server side.

So, since Actors are the base class for all network aware objects, where an object of uncertain hierarchy can be legitimately passed across the network it MUST be an Actor, so we can pass it as an Actor, and then cast back to the interface we know it supports on the other side of the connection.

Thus, we end up with the “Server” part looking like **this**, with it’s argument being an Actor, and a cast back to the interface we know this actor supports inside the implementation.

Passing Interfaces Across the Network

- Interfaces are not Actors

-
  ```
  // doesn't work (foo==none always):
  reliable server function ServerDoFooThing( IFoo foo )
  { LocalDoFooThing(foo); }
  ```

Unfortunately, for an object to be relevant at all it must be an Actor, and Interfaces are not necessarily Actors, thus this first attempt at implementing the “Server” part of the pattern will only ever pass “none” to the Local function on the server side.

So, since Actors are the base class for all network aware objects, where an object of uncertain hierarchy can be legitimately passed across the network it MUST be an Actor, so we can pass it as an Actor, and then cast back to the interface we know it supports on the other side of the connection.

Thus, we end up with the “Server” part looking like **this**, with it’s argument being an Actor, and a cast back to the interface we know this actor supports inside the implementation.

Passing Interfaces Across the Network

- Interfaces are not Actors

```
// doesn't work (foo==none always):
reliable server function ServerDoFooThing( IFoo foo )
{ LocalDoFooThing(foo); }
```

```
// foo MUST to be an actor
// even if your working on it as an IFoo on both sides
reliable server function ServerDoFooThing( Actor foo )
{
   // casting is usually bad practice,
   // but it's a necessary work around here
   LocalDoFooThing( IFoo(foo) );
}
```

Unfortunately, for an object to be relevant at all it must be an Actor, and Interfaces are not necessarily Actors, thus this first attempt at implementing the “Server” part of the pattern will only ever pass “none” to the Local function on the server side.

So, since Actors are the base class for all network aware objects, where an object of uncertain hierarchy can be legitimately passed across the network it MUST be an Actor, so we can pass it as an Actor, and then cast back to the interface we know it supports on the other side of the connection.

Thus, we end up with the “Server” part looking like **this**, with it’s argument being an Actor, and a cast back to the interface we know this actor supports inside the implementation.

Changing Replication Conditions

- Defined in the class that defines the variable
- Native Replication
  `GetOptimizedRepList`

Where as the previous 3 limitations are easily worked around, this one is not.

Since the replication conditions for a variable must be defined in the class that declares the member variable, there is no way to change that for sub class from scripts, instead you need to use native-replication, which involves overriding the GetOptimizedRepList in C++.

This is not for the faint hearted.

Subclassing repnotify

- `repnotify` is part of the variable declaration
- Add it to the base class
- To remove - override ReplicationEvent and do not pass up back up to the base

Similarly, the triggering of events on replication, via repnotify, is not something that can be changed from a sub class, since it is part of the variable declaration.

If you need to ADD an event, however, this can easily be done by changing the base class, and doing so has little or no negative repercussions.

If you need to prevent an event you could override the ReplicationEvent in your sub class, and not chain the call up to the base class. This doesn’t strictly prevent the event from being triggered, but does prevent it from being handled.

Be warned however, that this may have unexpected repercussions and could result in more refactoring than you expected.

Summary

Summary

- Object Oriented Simulation
- Generalised Server / Client

Summary

- Object Oriented Simulation
- Generalised Server / Client
- Data Replication
- Remote Procedure Calls (RPC)

Further Reading

- https://udn.epicgames.com/Three/NetworkingTome

Q&A

- Any Questions?