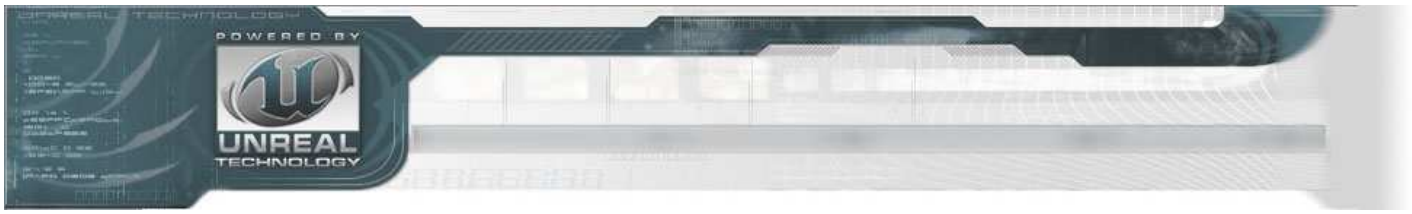

2.2 - Unreal Networking Architecture

Document Summary: Overview of Unreal Networking Architecture, originally written by Tim Sweeney.

Document Changelog: Ported and updated by Steve Polge; updated and maintained by Richard Nalezynski.

- Unreal Networking Architecture
  - Overview
    - Peer-to-Peer Model
    - Client-Server Model
    - Unreal Networking Architecture
  - Basic Concepts
  - Actors
    - Roles
    - Relevancy
    - Prioritization
  - Replication
    - Replication Overview
    - UnrealScript: the Replication Statement
    - Replication Conditions
    - Variable Replication
    - Function Call Replication
    - Replication Patterns
  - Simulated Functions and States
  - Gameplay
    - ReplicationInfo classes
    - The WorldInfo class
    - The GameInfo class
    - Player Movement and Prediction
    - Player Animation (Client-Side)
    - Dead Bodies
    - Weapon Firing

Overview

Multiplayer gaming is about shared reality: that all of the players feel they are in the same world, seeing from differing viewpoints the same events transpiring within that world. As multiplayer gaming has evolved from the little 2-player modem games that characterized *Doom*, into the large, persistent, more free-form interactions of games like *Quake 2*, *Unreal*, and *Ultima Online*, the technologies behind the shared reality have evolved tremendously.

One important thing to realize is that if you plan to support networked multiplayer in your game, build it in and test it as you develop your game! Building an efficient networking implementation will have a significant impact on your game object design decisions. Retrofitting a solution is a hard and costly effort!

Peer-to-Peer Model

In the beginning, there were peer-to-peer games like *Doom* and *Duke Nukem*. In these games, each machine in the game was an equal. Each exactly synchronized its input and timing with the others, and each machine carried out the same exact game logic on exactly the same inputs. In conjunction with completely deterministic (i.e. fixed-rate, non-random) game logic, all players in the machine perceived the same reality.

The advantage of this approach was simplicity. The disadvantages were:

- **Lack of persistence**. All of the players had to start the game together, and new players couldn't come and go as they pleased.

- **Lack of player scalability**. Because of the lock-step nature of the networking architecture, the coordination overhead and chance of network-induced failure increases linearly with the number of players.
- **Lack of frame rate scalability**. All of the players had to run at the same internal frame rate, making it difficult to support a wide variety of machine speeds.

Client-Server Model

Next came the monolithic client-server architecture, pioneered by *Quake*, and later used by *Ultima Online*. Here, one machine was designated "server", and was responsible for making all of the gameplay decisions. The other machines were "clients", and they were regarded as dumb rendering terminals, which sent their keystrokes to the server, and received a list of objects to render. This advancement enabled large-scale Internet gaming, as game servers started springing up all over the Internet. The client-server architecture was later extended by *QuakeWorld* and *Quake 2*, which moved additional simulation and prediction logic to the client side, in order to increase visible detail while lowering bandwidth usage. Here, the client receives not only a list of objects to render, but also information about their trajectories, so the client can make rudimentary predictions about object motion. In addition, a lock-step prediction protocol was introduced in order to eliminate perceived latency in client movement.

Still, there were some disadvantages to this approach:

- **Lack of open-endedness**. When users and licensees create new types of objects (weapons, player controls, etc), glue logic must be authored to specify the simulation and prediction aspects of those new objects.

Difficulties of the prediction model: In this model, the network code and the game code are separate modules, yet each must both be fully aware of the implementation of the other, in order to keep the game state reasonably synchronized. A strong coupling between (ideally) separate modules is undesirable because it makes extensibility difficult.

Unreal Networking Architecture

Unreal introduces into multiplayer gaming a new approach termed the **generalized client-server model**. In this model, the server is still authoritative over the evolution of the game state. However, the client actually maintains an

accurate subset of the game state locally, and can predict the game flow by executing the same game code as the server, on approximately the same data, thus minimizing the amount of data that must be exchanged between the two machines. Servers send information about the world to clients by replicating relevant actors and their replicated properties. Clients and servers also communicate through replicated functions, which are replicated only between the server and the client which owns the Actor on which the function is called.

Further, the *game state* is self-described by an extensible, object-oriented scripting language - UnrealScript - which fully decouples the game logic from the network code. The network code is generalized in such a way that it can coordinate any game which can be described by the language. This achieves a goal of object-orientation which increases extensibility, the concept that the behavior of an object should be fully described by that object, without introducing dependencies on other pieces of code which are hard-wired to know about the internal implementation of that object.

Basic Concepts

Goal

The goal here is to define Unreal's networking architecture in a fairly rigorous manner, because there is a fair amount of complexity involved that is easy to misinterpret if not defined exactly.

Basic Terminology

We define our basic terminology precisely:

- A **variable** is an association between a fixed name and a modifiable value. Examples of variables include integers such as `X=123`, floating point numbers such as `Y=3.14`, strings such as `Team="Rangers"`, and vectors such as `V=(1.5,2.5,-0.5)`.
- An **object** is a self-contained data structure consisting of a fixed set of variables.
- An **actor** is an object capable of independently moving around in a level and interacting with other actors in that level.
- A **level** is an object which contains a set of actors.

- A **tick** is an operation that updates the entire game state given that a variable amount of time called DeltaTime has passed.
- The **game state** of a level refers to the complete set of all actors that exist in that level and the current values of all of their variables at a time when a tick operation is not currently in progress.
- A **client** is a running instance of `Unreal.exe` which maintains an approximate subset of the game state suitable for approximately simulating the events that occur in the world, and for rendering an approximate view of the world for a player.
- A **server** is a running instance of `Unreal.exe` which is responsible for ticking a single level and communicating the game state authoritatively to all of the clients.

The Update Loop

All of the above concepts are well-understood with the possible exception of the tick and game state. So, these will be described in more detail. First of all, here is a simple description of Unreal's update loop:

- If I am a *server*, communicate the current game state to all of my clients.
- If I am a *client*, send my requested movement to the server, receive new game state information from the server, render my current approximate view of the world to the screen.
- Perform a *tick* operation to update the game state, given that a variable amount of time `DeltaTime` has passed since the previous tick.

A tick operation involves updating all of the Actors in the level, carrying out their physics, informing them of interesting game events that have occurred, and executing any necessary script code. Unlike many past games such as *Doom* and *Quake*, all of the physics and update code in Unreal is designed to handle a variable amount of time passing.

For example, _Doom_'s movement physics looks like:

```
Position += PositionIncrement
```

while Unreal's looks like:

```
Position += Velocity * DeltaTime
```

This enables greater frame rate scalability.

While a tick operation is in progress, the game state is being continually modified by the code which executes. The game state can change in exactly three ways:

- A variable in an Actor can be modified.
- An Actor can be created.
- An Actor can be destroyed.

The Server Is The Man

From the above, the server's game state is completely and concisely defined by the set of all variables of all actors within a level. Because the server is authoritative about the gameplay flow, the server's game state can always be regarded as the one true game state. The version of the game state on client machines should always be regarded as an approximation subject to many different kinds of deviations from the server's game state. Actors that exist on the client machine should be considered proxies because they are a temporary, approximate representation of an object rather than the object itself.

When a client loads a Level to use in a networked multiplayer game, it deletes all Actors in the Level except those that have either `bNoDelete` set to `TRUE` or `bStatic` set to `TRUE`. Other Actors relevant to that client (as determined by the server) will be replicated from the server to the client. Some Actors (such as the GameInfo Actor) are never replicated to a client.

Bandwidth Limitation

If network bandwidth were unlimited, the network code would be very simple: at the end of each tick, the server could just send each client the complete and exact game state so that the client always renders the exact view of the game as is occurring on the server. However, the Internet reality is that 28.8K modems have only perhaps 1% of the bandwidth necessary to communicate complete and exact updates. While consumers' Internet connections will become faster in the future, bandwidth is growing at a rate far lower than Moore's law which defines the rate of improvement in games and graphics. Therefore, there is not now and never will be sufficient bandwidth for complete game-state updates.

So, the main goal of the network code is to enable the server to communicate a reasonable approximation of the game state to the clients so that the clients can render an interactive view of the world which is as close to shared reality as is reasonable given bandwidth limitations.

Replication

Unreal views the general problem of "coordinating a reasonable approximation of a shared reality between the server and clients" as a problem of "replication". That is, a problem of determining a set of data and commands that flow between the client and server in order to achieve that approximate shared reality.

Actors

Roles

In general, every Actor has a `Role` and a `RemoteRole` property, which have different values on the server and the client. Every Actor on a server has a `Role` set to `ROLE_Authority`.

Actors on the server may have as their `RemoteRole`:

- `ROLE_AutonomousProxy` (PlayerControllers and the Pawn they control, when replicated to the owning client)
- `ROLE_SimulatedProxy` (All other replicated Actors)
- `ROLE_None` (Actors which are never replicated to any client)

The `RemoteRole` of an Actor on the server is the `Role` of that Actor on the client. All Actors replicated to a client have `RemoteRole` set to `ROLE_Authority`.

Definition

The `Actor` class defines the `ENetRole` enumeration and two variables, `Role` and `RemoteRole`, as follows:

```
// Net variables.
enum ENetRole
{
   ROLE_None,              // No role at all.
   ROLE_SimulatedProxy,    // Locally simulated proxy of this actor.
```

```
   ROLE_AutonomousProxy,   // Locally autonomous proxy of this
actor.
   ROLE_Authority,         // Authoritative control over the actor.
};
var ENetRole RemoteRole, Role;
```

The `Role` and `RemoteRole` variables describes how much control the local and remote machines, respectively, have over the actor:

- `Role==ROLE_SimulatedProxy` means the actor is a temporary, approximate proxy which should simulate physics and animation. On the client, simulated proxies carry out their basic physics (linear or gravitationally-influenced movement and collision), but they don't make any high-level movement decisions. They just go. They can only execute script functions with the *simulated* keyword; and they can only enter states marked as *simulated*.

This situation is only seen in network clients, never for network servers or single-player games.

- `Role==ROLE_AutonomousProxy` means the actor is the local player. Autonomous proxies have special logic built in for client-side prediction (rather than simulation) of movement. They can execute any script functions on the client; and they can enter any state.

This situation is only seen in network clients, never for network servers or single-player games.

- `Role==ROLE_Authority` means this machine has absolute, authoritative control over the Actor.

This is the case for all actors in single-player games. They can execute any script functions; and they can enter any state.

This is the case for all Actors on a server. On a client, this is the case for Actors that were locally spawned by the client, such as gratuitous special effects which are done client-side in order to reduce bandwidth usage. On the server side, all Actors have `Role==ROLE_Authority`, and `RemoteRole` set to one of the proxy types. On the client side, the `Role` and `RemoteRole` are always the exactly reversed relative to the server's value. This is as expected from the meaning of `Role` and `RemoteRole`.

Most of the meaning of the `ENetRole` values is defined by the replication statements in the UnrealScript classes such as `Actor` and `PlayerPawn`. Here are several examples of how the replication statements define the meanings of the various role values:

- The `Actor.AmbientSound` variable is sent from the server to clients because of this replication definition in the `Actor` class: `if( Role==ROLE_Authority ) AmbientSound;`
- The `Actor.AnimSequence` variable is sent from the server to clients, but only for actors rendered as meshes, because of this replication definition in the Actor class: `if( DrawType==DT_Mesh && (RemoteRole<=ROLE_SimulatedProxy) ) AnimSequence;`
- The client replicates his `Fire` and `AltFire` function calls to the server because of this replication definition in the `PlayerPawn` class: `if( Role<ROLE_Authority ) Fire, AltFire;`
- The server sends clients the `Velocity` of all simulated proxies when they are initially spawned and all moving brushes because of this replication definition in the `Actor` class: `if( (RemoteRole==ROLE_SimulatedProxy && (bNetInitial || bSimulatedPawn)) || bIsMover ) Velocity;`

By studying the replication statements in all of the UnrealScript classes, you can understand the inner workings of all of the roles. There is really very little "behind-the-scenes _magic_" going on with respect to replication: At a low C++ level, the engine provides a basic mechanism for replicating actors, function calls, and variables. At the high UnrealScript level, the meanings of the various network roles are defined by specifying what variables and functions should be replicated based on the various roles. So, the meaning of the roles is almost self-defining in UnrealScript, with the exception of a small amount of behind-the-scenes C++ logic which conditionally updates physics and animation for simulated proxies.

What Exactly Is Happening Behind The Scenes

The answer is for licensees to do a "Find in Files" in Visual C++ for all occurrences of `ROLE_` in the C++ and UnrealScript files. While documentation provides a general understanding of how network roles are interpreted, the exact impact of roles on all aspects of the code can only be completely described by the source code.

Relevancy

Definition

An Unreal Level can be huge, and at any time a player can only see a small fraction of the Actors in that level. Most of the other Actors in the level aren't visible, aren't audible, and have no significant effect on the player. The set of Actors that a server deems are visible to or capable of affecting a client are deemed the relevant set of Actors for that client. A significant bandwidth optimization in Unreal's network code is that the server only tells clients about Actors in that client's relevant set.

Unreal applies the following rules in determining the relevant set of Actors for a player:

- If the Actor has `bStatic=true` or `bNoDelete=true`, then it is relevant.
- If the Actor belongs to the `ZoneInfo` class, then it is relevant.
- If the Actor is owned by the player (`Owner==Player`), then it is relevant.
- If the Actor is a `Weapon` and is owned by a visible actor, then it is relevant.
- If the Actor is hidden (`bHidden=true`) and it doesn't collide (`bBlockPlayers=false`) and it doesn't have an ambient sound (`AmbientSound==None`) then the actor is not relevant.
- If the Actor is visible according to a line-of-sight check between the actor's `Location` and the player's `Location`, then it is relevant.
- If the Actor was visible less than 2 to 10 seconds ago (the exact number varies because of some performance optimizations), then it is relevant.
- Actors whose `RemoteRole!=ROLE_None` are checked for relevancy for each client.
    - Virtual function `AActor::IsNetRelevantFor()` is called to determine whether the Actor is relevant to that client.
    - `bAlwaysRelevant` (relevant to all clients, always).
    - `bOnlyRelevantToOwner` (used for Inventory).
    - Relevancy based on visibility (using line checks).
    - If skeletally attached, relevancy of base is used.

Note that `bStatic` and `bNoDelete` Actors (which remain on the client) can also be replicated.

These rules are designed to give a good approximation of the set of Actors which really can affect a player. Of course, it is imperfect: the line-of-sight check can

sometimes give a false negative with large Actors (though we use some heuristics to help it out), it doesn't account for sound occlusion of ambient sounds, and so on. However, the approximation is such that its error is overwhelmed by the error inherent in a network environment with such latency and packet loss characteristics as the Internet.

Changing the definition of the relevant set in C++

The set of relevant Actors for a client is determined by the C++ function `UWorld::ServerTickClients` from `UnLevTic.cpp`. Licensees who want to implement different gameplay rules are free to modify the definition of the relevant set by modifying `UWorld::ServerTickClients` or (better yet) subclassing `UWorld`, and overriding the function.

Prioritization

In deathmatch games on modem-based Internet connections, there is almost never enough bandwidth available for the server to tell each client everything it desires to know about the game state, Unreal uses a load-balancing technique that prioritizes all actors, and gives each one a *fair share* of the bandwidth based on how *important* it is to gameplay.

Each Actor has a floating point variable called `NetPriority`. The higher the number, the more bandwidth that Actor receives relative to others. An Actor with a priority of 2.0 will be updated exactly twice as frequently as an Actor with priority 1.0. The only thing that matters with priorities is their ratio; so obviously you can't improve Unreal's network performance by increasing all of the priorities. Some of the values of `NetPriority` we have assigned in our performance-tuning are:

- `Actor` = 1.0
- `Pawns` = 2.0
- `PlayerController` = 3.0
- `Projectiles` = 2.5
- `Inventory` = 1.4
- `Vehicule` = 3.0

To avoid starvation `AActor::GetNetPriority()` multiplies `NetPriority` with the time since the Actor was last replicated. `APawn::GetNetPriority()` also considers replicated location error, and whether the Pawn is behind the viewer.

Replication

Replication Overview

The network code is based on three primitive, low-level replication operations for communicating information about the game state between the the server and clients:

1. Actor replication
2. Variable replication
3. Function replication

Actor replication. The server identifies the set of "relevant" Actors for each client (Actors which are either visible to the client or are likely to somehow affect the client's view or movement instantaneously), and tells the client to create and maintain a "replicated" copy of that Actor. While the server always has the authoritative version of that Actor, at any time many clients might have approximate, replicated versions of that Actor.

When a replicated Actor is spawned on a client, only `Location` and `Rotation` (valid if `bNetInitialRotation` is set to `TRUE`) are valid during `PreBeginPlay()` and `PostBeginPlay()`. Replicated Actors can only be destroyed because the server closes their replication channel, with the exception being if the Actor properties `bNetTemporary` and `bTearOff` have been set to `TRUE`.

Actor property replication is reliable. This means that the property of the client version of the Actor will eventually reflect the value on the server, not that all property value changes will be replicated. In any case, Actor properties are only replicated from the server to the client; and such properties are replicated only if they are included in the replication definition of the Actor class which defines that property.

The replication definition specifies replication conditions, which describe when and if to replicate a given property to the client currently being considered. Even if an Actor is relevant, not all of its properties are replicated. Careful specification of replication conditions can substantially reduce bandwidth use.

There are three Actor properties which are only valid during replication, and change values depending on the client for which the server is determining replication:

1. `bNetDirty is true` if any replicated properties have been changed by UnrealScript (or this flag has been set in C++). This is used as an optimization (no need to check UnrealScript replication conditions, or to check whether properties which are only modified in script have been changed if `bNetDirty` is false). Don't use `bNetDirty` to manage replication of frequently updated properties!
2. `bNetInitial` remains true until initial replication of all replicated Actor properties is complete.
3. `bNetOwner` is true if the top owner of the Actor is the PlayerController owned by the current client.

Variable replication. Actor variables that describe aspects of the game state which are important to clients can be "replicated". That is, whenever the value of the variable changes on the server side, the server sends the client the updated value. (The variable may have changed on client side too - in which case the new value will overwrite it.)

Function call replication. A function that is called on the server in a network game can be routed to the remote client rather than executed locally. Alternatively, a function called on the client side may be routed to the server, rather than called locally. To give a concrete example, consider the case where you are a client in a network game. You see two opponents running toward you, shooting at you, and you hear their shots. Since all of the game state is being maintained on the server rather than on your machine, why can you see and hear those things happening?

- You can see the opponents because the server has recognized that the opponents are *relevant* to you (i.e. they are visible) and the server is currently replicating those Actors to you. Thus, you (the client) have a local copy of those two player Actors who are running after you.
- You can see that the opponents are running toward you because the server is replicating their `Location` variable to you.
- You can see them animating because the server is replicating their animation variables to you. In other words, the server is continually feeding you new values of their Location and animation parameters, at the rate of several times per second.
- You can hear their gunshots because the server is replicating the `ClientHearSound` function to you. The `ClientHearSound` function is called for a `PlayerPawn` whenever that `PlayerPawn` hears a sound.

So, by this point, the low-level mechanisms by which Unreal multiplayer games operate should be clear. The server is updating the game state and making all of the big game decisions. The server is replicating some Actors to clients. The server is replicating some variables to clients. And, the server is replicating some function calls to clients.

It should also be clear that not all Actors need to be replicated. For example, if an Actor is halfway across the level and way out of your sight, you don't need to waste bandwidth sending updates about it. Also, all variables don't need to be updated. For example, the variables that the server uses to make AI decisions don't need to be sent to clients; the clients only need to know about their display variables, animation variables, and physics variables. Also, most functions executed on the server shouldn't be replicated. Only the function calls that result in the client seeing or hearing something need to be replicated. So, in all, the server contains a huge amount of data, and only a small fraction of it matters to the client--the ones which affect things the player sees, hears, or feels.

Thus, the logical question is, "How does the Unreal engine know what Actors, variables, and function calls need to replicated?"

The answer is, the programmer who writes a script for an actor is responsible for determining what variables and functions, in that script, need to be replicated. And, he is responsible for writing a little piece of code called a "replication statement", in that script, to tell the Unreal engine what needs to be replicated under what conditions. For a real-world example, consider some of the things defined in the `Actor` class.

- The `Location` variable (a vector) contains the Actor's location. The server is responsible for maintaining the location, so the server needs to send that to clients. So the replication condition basically says "Replicate this if `I` am the server".
- The `Mesh` variable (an object reference) references the mesh that should be rendered for the actor. The server needs to send that to clients, but it only needs to be sent if the Actor is being rendered as a mesh, i.e. if the Actor's `DrawType` is `DT_Mesh` . So the replication condition basically says "Replicate this if `I` am the server and the `DrawType` is `DT_Mesh` ".
- In the `PlayerPawn` class, there are a bunch of boolean variables that define keypresses and button presses, such as `bFire` and `bJump`. These are generated on the client side (where the input happens), and the server

needs to know about them. So the replication condition basically says "Replicate this if I am a client".
- In the `PlayerController` class, there is a `ClientHearSound` function that tells the player that he or she hears a sound. It's called on the server but, of course the sound needs be heard by the actual person playing the game, who is on the client side. So the replication condition for this function might be "Replicate this if I am the server".

From the above examples, several things should be apparent. First of all, every variable and function that might be replicated needs to have a "replication condition" attached to it, that is, an expression which evaluates to True of False, depending on whether the thing needs to be replicated. Second, these replication conditions need to be two-way: the server needs to be able to replicate variables and functions to the client, and the client needs to be able to replicate them to the server. Third, these "replication conditions" can be complex, such as "Replicate this if I'm the server and this is the first time this Actor is being replicated across the network."

Therefore, we need a general-purpose way of expressing (complex) conditions under which variables and functions should be replicated. What is the best way to express these conditions? We looked at all of the options, and concluded that UnrealScript--which is already a very powerful language for authoring classes, variables, and code--would be a perfect tool for writing replication conditions.

UnrealScript: the Replication Statement

In UnrealScript, every class can have one replication statement. The replication statement contains one or more replication definitions. Each replication definition consist of a replication condition (a statement that evaluates to `True` or `False`), and a list of one or more functions and variables to which the condition applies.

The replication statement in a class may only refer to variables defined in that class, and functions defined first in that class (that is, it cannot apply to functions defined in a superclass but overridden in that class). This way, if the `Actor` class contains a variable `DrawType`, then you know where to look for its replication condition: it can only reside there in the Actor class.

It's valid for a class to not contain a replication statement; this simply means that the class doesn't define any new variables or functions that need to be replicated. In fact, most classes do not need replication statements because

most of the "interesting" variables that affect display are defined in the `Actor` class, and are only modified by subclasses. In Unreal, we have about 500 classes, and only about 10 of them need replication statements.

If you define a new variable or function in a class, but you don't list it in a replication definition, that means that your variable or function is absolutely never replicated. This is the norm; most variables and functions don't need to be replicated.

Here is an example of the UnrealScript syntax for the replication statement. This is taken from the `PlayerReplicationInfo` class:

```
replication
{
   // Things the server should send to the client.
   if ( bNetDirty && (Role == Role_Authority) )
      Score, Deaths, bHasFlag, PlayerLocationHint,
      PlayerName, Team, TeamID, bIsFemale, bAdmin,
      bIsSpectator, bOnlySpectator, bWaitingPlayer, bReadyToPlay,
      StartTime, bOutOfLives, UniqueId;
   if ( bNetDirty && (Role == Role_Authority) && !bNetOwner )
      PacketLoss, Ping;
   if ( bNetInitial && (Role == Role_Authority) )
      PlayerID, bBot;
}
```

The key things you see here are:

- The replication statement is enclosed by a `replication {}` block.

Reliable vs. Unreliable

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. The only things which can prevent an unreliable function from being received are network packet-loss, and bandwidth saturation. So, you need to understand the odds, which we will grossly approximate here. The results vary wildly among different types of network, so we can make no guarantees:

LAN In a LAN game, we guesstimate that unreliable data is received successfully approximately 99% of the time. However, in the course of a game, hundreds of thousands of things are replicated, so you can be sure that some unreliable data will be lost. Therefore, even if you're aiming for LAN performance only, your code needs to handle your unreliably replicated variables gracefully in the case that they are lost over the wire.

Internet In a typical low quality 28.8K ISP connection, unreliable data is generally received 90%-95% of the time. In other words, it is very frequently lost.

To get a better feeling for the tradeoffs between reliable and unreliable functions, check out the replication statements in the Unreal scripts, and gauge their importance vs. the reliability decision we made. Be cautious and use reliable functions only when absoluately necessary.

Variables are always reliable

The `reliable` and `unreliable` keywords are ignored for variables. Variables are always guaranteed to reach the other party eventually, even under packet-loss and bandwidth saturation conditions. Changes in such variables are not guaranteed to reach the other party in the same order in which they were sent.

Summary

Here, we have documented the syntax of the replication statement thoroughly, without saying much about the meaning of the expressions like `Role==ROLE_Authority` and `bNetOwner`. This is covered in the next section.

Replication Conditions

Here is a simple example of a replication condition within a class's script:

```
replication
{
   if( Role==ROLE_Authority )
      Weapon;
}
```

This replication condition, translated into English, is "If the value of this Actor's `Role` variable is equal to `ROLE_Authority`, then this Actor's `Weapon` variable should be replicated to all clients for whom this Actor is relevant".

A replication condition may be any expression that evaluates to a value of `True` or `False` (that is, a boolean expression). So, any expression you can write in UnrealScript will do, including comparing variables; calling functions; and combining conditions using the boolean `!`, `&&`, `||`, and `^^` operators.

An Actor's `Role` variable generally describes how much control the local machine has over the Actor. `ROLE_Authority` means "this machine is the server, so it is completely authoritative over the proxy Actor". `ROLE_SimulatedProxy` means "this machine is a client, and it should simulate (predict) the physics of the Actor". `Role` is described in more detail in a later section, but the quick summary is this:

- `if(Role==ROLE_Authority)` : means "If I am the server, I should replicate this to clients".
- `if(Role<ROLE_Authority)` : means "If I am a client, I should replicate this to the server".

The following variables are very often used in replication statements because of their high utility:

- `bIsPlayer`: Whether this Actor is a player. `True` for players, `False` for all other Actors.
- `bNetOwner`: Whether this Actor is owned by the client for whom the replication condition is being evaluated. For example, say "Fred" is holding a `DispersionPistol`, and "Bob" isn't holding any weapon. When the `DispersionPistol` is being replicated to "Fred", its `bNetOwner` variable will be `True` (because "Fred" owns the weapon). When it is being replicated to "Bob", its `bNetOwner` variable will be `False` (because "Bob" does not own the weapon).
- `bNetInitial`: Valid only on the server side (i.e. if `Role==ROLE_Authority`). Indicates whether this Actor is being replicated to the client for the first time. This is useful for clients with `Role==ROLE_SimulatedProxy`, because it enables the server to sent their location and velocity just once, with the client subsequently predicting it.

Replication condition guidelines:

Since variables are typically replicated one-way (either from the client to the server, or from the server to the client, but never both), all replication conditions generally start with a comparison of `Role` or `RemoteRole`: for example, `if(Role==ROLE_Authority)` or `if(RemoteRole<ROLE_SimulatedProxy)`. If a replication condition doesn't contain a comparison of `Role` or `RemoteRole`, there is probably something wrong with it.

Replication conditions are evaluated very, very frequently on the server during network play. Keep them as simple as possible, but no simpler.

While replication conditions are allowed to call functions, try to avoid that because it could be a big slowdown.

Replication conditions shouldn't have any side-effects, because the network code may choose to call them at any time including times when you don't expect. For example if you do something like `if(Counter++ > 10)` ..., good luck trying to figure out what is going to happen!

Variable Replication

Update mechanism

After every tick, the server checks out all Actors in its relevant set. All of their replicated variables are examined to see if they have changed since the previous update, and the variables' replication conditions are evaluated to see if the variables need to be sent. As long as there is bandwidth available in the connection, those variables are sent across the network to the other machine.

Thus, the client receives updates of the *important* events that are happening in the world, which are visible or audible to that client. The key points to remember about variable replication are:

Variable replication occurs only after a tick completes. Therefore, if in the duration of a tick, a variable changes to a new value, and then it changes back to its original value, then that variable will not be replicated. Thus, clients only hear about the state of the server's Actor's variables after its tick completes; the state of the variables during the tick is invisible to the client.

Variables are only replicated when they change, relative to their previously known value.

Variables for an Actor are only replicated to a client when they are in the client's relevant set. Thus, the client does not have accurate variables for Actors that are not in its relevant set.

UnrealScript has no concept of global variables; so they only variables that can be replicated are instance variables belonging to an Actor.

Variable type notes:

- **Vectors** and **Rotators**: To improve bandwidth efficiency, Unreal quantizes the values of vectors and rotators. The `X`, `Y`, `Z` components of vectors are converted to 16-bit signed integers before being sent, thus any fractional values or values out of the range -32768...32767 are lost. The `Pitch`, `Yaw`, `Roll` components of vectors are converted to bytes, of the form `(Pitch>>8)&255`. So, you need to be careful with vectors and rotators. If you absolutely must have full precision, then use int or float variables for the individual components; all other data types are sent with their complete precision.

General structs are replicated by sending all of their components. A struct is sent as an "all or nothing" type of thing.

- **Arrays of variables** can be replicated, but only of the size of the array (in bytes) is less than 448 bytes.

- **Arrays** are replicated efficiently; if a single element of a large array changes, only that element is sent.

Actor Properties

Actor property replication is reliable. This means that the property of the client version of the Actor will eventually reflect the value on the server, not that all property value changes will be replicated.

- Properties are only replicated from the server to the client.
- Properties are replicated only if they are included in the replication definition of the class which defines that property.

Function Call Replication

Remote Routing Mechanism

When an UnrealScript function is called during a network game, and that function has a replication condition, the condition is evaluated and execution progresses as follows:

- If the function's replication condition evaluates to `True`, the function call is sent to the machine on the other side of the network connection for execution. In other words, the function's name, and all of its parameters, are crammed together into a packet of data, and transmitted to the other machine for later execution. When this occurs, the function returns immediately and execution continues on. If the function were declared to return a value, then its return value is set to zero (or the equivalent of zero in some other type, i.e. `0,0,0` for vectors, `None` for objects, etc). Any out parameters are left unaffected. In other words, UnrealScript never sits around waiting for a replicated function call to complete, so it can never deadlock. Rather, replicated function calls are sent off for the remote machine to execute, and the local code continues executing.
- If the replication condition evaluates to `False`, the function is executed normally on the local machine.

Unlike replicated variables, a function call on an Actor can only be replicated from the server to the client (player) who owns that Actor. So, replicated functions are only useful in subclasses of `PlayerController` (i.e. players, who own themselves), `Pawn` (i.e. player avatars that are owned by a Controller that is controlling them), and subclasses of `Inventory` (i.e. weapons and pickup items, who are owned by the player who is currently carrying them). That is to say, a function call can only be replicated to one Actor (the player who owns it); they cannot be multicast.

If a function marked with the *server* keyword is called on a client, it will be replicated to the server. Conversely, when a function marked with the *client* keyword is called on the server, it will be replicated to the client which owns that Actor.

Unlike replicated variables, replicated function calls are sent to the remote machine immediately when they are called, and they are always replicated regardless of bandwidth. Thus, it is possible to flood the available bandwidth if

you make too many replicated function calls. Replicated functions suck away however much bandwidth is available, and then whatever bandwidth is left over is used for replicating variables. Therefore, if you flood the connection with replicated functions, you can starve the replication of variables, which visually results in not seeing other Actors update, or seeing them update in an extremely choppy motion.

In UnrealScript, there are no global functions, so there is no concept of "replicated global functions". A function is always called in the context of a particular Actor.

Replicated Function Calls vs. Replicated Variables

Too many replicated functions can flood the available bandwidth (because the are always replicated, regardless of available bandwidth), whereas replicated variables automatically are throttled and parceled out according to bandwidth available.

Function calls are replicated only during UnrealScript execution when they are actually called, whereas variables are replicated only at the end of the current tick when no script code is executing.

Function calls on an actor are only replicated to the client who owns that actor, whereas an Actor's variables are replicated to all clients for whom that actor is relevant.

Quantization Gotchas

To improve bandwidth efficiency, Unreal quantizes the values of vectors and rotators. The `X`, `Y`, `Z` components of vectors are converted to integers before being sent, thus any fractional values are lost. The `Pitch`, `Yaw`, `Roll` components of vectors are converted to bytes, of the form `(Pitch>>8)&255`. If you need complete accuracy on rotators and vectors, send them as individual **`float`** or **`int`** components.

Replication Patterns

The goals for common replication patterns in the Unreal Engine and games shipped by Epic are:

- Minimizing Server CPU utilization
- Minimizing Bandwidth use
- Minimizing perceived latency

Minimizing Server CPU Utilization

Minimize cost of replicating actors (Effectively, optimize execution of =UWorld::ServerTickClients=())

- Minimize number of potentially replicated Actors (those with `RemoteRole = ROLE_None`).
- Minimize number of Actors that need to be checked for relevancy per client any given tick.
- Minimize number of actually relevant Actors per client any given tick.
- Minimize number of replicated properties that need to be checked per replicated actor per client any given tick.
- Avoid unnecessarily setting `bNetDirty`.

Minimize Actor tick cost

- Avoid spawning Actors not needed on server (particle effects, etc.).
- Avoid executing code if it has no game play relevance.

Minimize cost of processing received replicated functions

- Minimize number of functions received and processing required.

As the number of players increases, the cost of replicating Actors becomes the dominant part of server execution time, since it tends to increase geometrically rather than linearly with the number of players (since the number of potentially replicated Actors tends to be proportional to the number of players).

Minimizing Bandwidth Use

Minimize the following factors:

- the number of relevant Actors per client
- the frequency of property updates
- the number of packets sent

Unsuppress `DevNetTraffic` to see logging of all replicated Actors and properties. The console command `Stat Net` is also useful.

Minimizing Perceived Latency

Have client predict behavior of client owned actors based on player inputs; simulate this behavior before receiving confirmation from the server (and correcting if necessary). We use this model for Pawn movement and Weapon handling, but not for Vehicles, as the complexity of saving and replaying the physics simulation outweighs the benefit of reduced latency for vehicles handling, where typical internet response latencies aren't that different from typical real world vehicle control response latencies.

Simulated Functions and States

On the client side, many Actors exist in the form of "proxies", meaning approximate copies of Actors created by the server, and sent to the client to provide a visually and aurally reasonable approximation of what the client sees during gameplay.

On the client, these proxy Actors are often moving around using client-side physics and affecting the environment, so at any time their functions can potentially be called. For example, a simulated proxy `TarydiumShard` projectile might run into a dumb proxy `Tree` Actor. When Actors collide, the engine attempts to call their `Touch` functions to notify them of the collision. Depending on context, the client desires to execute some of these functions calls, but ignore others. For example, a `Skaarj='s =Bump` function should not be called on the client side, because his `Bump` function attempts to carry out gameplay logic, and gameplay logic should only occur on the server. So, the `Skaarj='s =Bump` function should not be called. However, a `TarydiumShard` projectile's `Bump`

function should be called, because it stops the physics and spawns a client-side special effect Actor.

UnrealScript functions can optionally be declared with the *simulated* keyword to give programmers fine-grained control over which functions should be executed on proxy actors. For proxy actors (that is, actors with `Role<ROLE_Authority`), only functions declared with the *simulated* keyword are called. All other functions are skipped.

Here is an example of a typical simulated function:

```
simulated function HitWall( vector HitNormal, actor Wall )
{
        SetPhysics(PHYS_None);
        MakeNoise(0.3);
        PlaySound(ImpactSound);
        PlayAnim('Hit');
}
```

So, *simulated* means "this function should always be executed for proxy Actors".

NOTE: Make sure that subclass implementations of simulated functions also have the *simulated* keyword in their definition!

Simulated states are similar to simulated functions.

Gameplay

Below are some gameplay considerations regarding Infos, Players, Weapons, etc.

`ReplicationInfo` classes

ReplicationInfo classes have `bAlwaysRelevant` set to TRUE. Server performance can be improved by setting a low `NetUpdateFrequency`. Whenever a replicated property changes, explicitly change `NetUpdateTime` to force replication. Server performance can also be improved by setting `bSkipActorPropertyReplication` and `bOnlyDirtyReplication` to TRUE (if no C++ updated properties).

See: `PlayerReplicationInfo`, `GameReplicationInfo`.

The `WorldInfo` class

Every game world instance in a networked game has a NetMode. The `WorldInfo` class defines the `ENetMode` enumeration and an associated **`NetMode`** variable as follows:

```
var enum ENetMode
{
   NM_Standalone,        // Standalone game.
   NM_DedicatedServer,   // Dedicated server, no local client.
   NM_ListenServer,      // Listen server.
   NM_Client             // Client only, no local server.
} NetMode;
```

The `NetMode` property is often used to control what code will execute on different game instance types.

The `GameInfo` class

The UnrealScript `GameInfo` class implements the game rules. A server (both dedicated and single-player) has one `GameInfo` subclass, accessible in UnrealScript as `Level.Game`. For each game type in Unreal, there is a special `GameInfo` subclass. For example, some existing classes are `DeathmatchGame`, `SinglePlayer`, `TeamGame`.

A client in a network game does not have a `GameInfo`. That is, `Level.Game==None` on the client side. Clients should not be expected to have a `GameInfo` because the server implements all of the gameplay rules, and the generality of the code calls for the client not knowing what the game rules are.

`GameInfo` implements a broad set of functionality, such as recognizing players coming and going, assigning credit for kills, determining whether weapons should respawn, and so on. Here, we will only look at the `GameInfo` functions which are directly related to network programming.

InitGame

```
event InitGame( string Options, out string ErrorMessage );
```

Called when the server (either in network play or single-player) is first started up. This gives the server the opportunity to parse the startup URL options. For example, if the server was started with "Unreal.exe MyLevel.unr?game=unreali.teamgame", the `Options` string is `"?game=unreali.teamgame"`. If `Error` is set to a non-empty string, the game fails with a critical error.

PreLogin

```
event PreLogin( string Options, string Address, out string
ErrorMessage, out string FailCode );
```

Called immediately before a network client is logged in. This gives the server an opportunity to reject the player. This is where the server should validate the player's password (if any), enforce the player limit, and so on.

Login

```
event PlayerController Login( string Portal, string Options, out
string ErrorMessage );
```

The `Login` function is always called after a call to `PreLogin` which does not return an error string. It is responsible for spawning the player, using the parameters in the `Options` string. If successful, it should return the `PlayerPawn` Actor it spawned.

If the `Login` function returns `None` indicating that the login failed, then it should set Error to a string describing the error. Failing a `Login` should be used only as a last resort. If you are going to fail a login, it is more efficient to fail it in `PreLogin` rather than `Login`.

PostLogin

```
event PostLogin( PlayerController NewPlayer );
```

The `PostLogin` function is called after a successful login. This is the first point at which replicated functions can be called.

Player Movement and Prediction

Overview

If a pure client-server model were used in Unreal, player movement would be very laggy. On a connection with 300 msec ping, when you push the forward key, you wouldn't see yourself move for 300 msec. When you pushed the mouse left, you wouldn't see yourself turn for 300 msec. This would be really frustrating.

To eliminate client-movement lag, Unreal uses a prediction scheme similar to that pioneered by QuakeWorld. It must be mentioned that the player prediction scheme is implemented entirely in UnrealScript. It is a high-level feature implemented in the `PlayerPawn` class, rather than a feature of the network code: Unreal's client movement prediction is layered entirely on the general-purpose replication features of the network code.

Inner Workings

You can see exactly how Unreal's player prediction works by examining the `PlayerPawn` script. Since the code is somewhat complex, its workings are briefly described here.

The approach can best be described as a lock-step predictor/corrector algorithm. The client takes his input (joystick, mouse, keyboard) and physical forces (gravity, buoyancy, zone velocity) into account and describes his movement as a 3D acceleration vector. The client sends this acceleration along with various input related information and his current timestamp (the current value of `Level.TimeSeconds` on the client side) to the server in a replicated `ServerMove` function call:

```
server function ServerMove
(
    float TimeStamp,
    vector InAccel,
    vector ClientLoc,
    byte MoveFlags,
    byte ClientRoll,
    int View
```

)

Then the client calls his `MoveAutonomous` to perform this same identical movement locally, and he stores this movement in a linked list of remembered movements using the `SavedMove` class. As you can see, if the client never heard anything back from the server, the client would be able to move around with zero lag just as in a single-player game.

When the server receives a `ServerMove` function call (replicated across the network), the server carries out the exact same movement on the server immediately. It deduces the movement's `DeltaTime` from the current `ServerMove='s =TimeStamp` and the previous one's. In this way, the server is carrying out the same basic movement logic as the client. However, the server might see things slightly different than the client. For example, if there's a monster running around, the client might have thought it was in a different position than the server (because the client is only in rough approximate sync with the server). Thus, the client and the server might disagree about how far the client actually moved as a result of the `ServerMove` call. At any rate, the server is authoritative, and he is completely responsible for determining the client's position. Once the server has processed the client's `ServerMove` call, it calls the client's `ClientAdjustPosition` function which is replicated across the network to the client:

```
client function ClientAdjustPosition
(
    float TimeStamp,
    name newState,
    EPhysics newPhysics,
    float NewLocX,
    float NewLocY,
    float NewLocZ,
    float NewVelX,
    float NewVelY,
    float NewVelZ,
    Actor NewBase
)
```

Now, when the client receives a `ClientAdjustPosition` call, he must respect the server's authority over his position. So, the client sets his exact location and velocity to that specified by the `ClientAdjustPosition` call. However, the

position the server specifies in `ClientAdjustPosition` reflects the client's actual position at some time in the past. But, the client wants to predict where he is supposed to be at the present moment. So, now the client goes through all of the `SavedMove='s in his linked list.  All moves earlier than the =ClientAdjustPosition` call's `TimeStamp` are discarded. All moves that occurred after `TimeStamp` are then re-run by looping through them and calling `MoveAutonomous` for each one.

This way, at any point in time, the client is always predicting ahead of what the server has told him by an amount of time equal to half his ping time. And, his local movement is not at all lagged.

Advantages

This approach is purely predictive, and it gives one the best of both worlds: In all cases, the server remains completely authoritative. Nearly all the time, the client movement simulation exactly mirrors the client movement carried out by the server, so the client's position is seldom corrected. Only in the rare case, such as a player getting hit by a rocket, or bumping into an enemy, will the client's location need to be corrected.

Movement Pattern

The following diagrams help illustrate the movement pattern on the server and client, including error adjustment.

Server

Client

ReplicateMove()

Called instead of ProcessMove(). Performs pawn physics update based on player inputs, saves (in PlayerController SavedMoves) and replicates results. SavedMove can be subclassed to save game specific movement inputs and results. ReplicateMove() also attempts to combine replicated moves to save upstream bandwidth and improve server performance.

CallServerMove()

Sends one or two current moves (depending on frame rate and available bandwidth) with client clock timestamps. Sending two moves at a time saves bandwidth, but increases latency for corrections. Also possibly calls OldServerMove() to re-send most recent "important" move in case of packet loss.

ServerMove()

Performs pawn physics update based on received inputs, and compares results to results sent by the client. Note that movement update is done based on the client's clock. If client has accumulated a significant position error, request correction. Else, request good move ack.

SendClientAdjustment()

Client response deferred to end of PlayerController tick to avoid sending multiple responses if multiple ServerMoves() were received that tick. If no error, ack good move.

ClientAckGoodMove()

Update ping, based on round trip time of timestamp, and clear out savedmoves with earlier timestamp.

Server

SendClientAdjustment()

Client response deferred to end of PlayerController tick to avoid sending multiple responses if multiple ServerMoves() were received that tick. If Error, call ClientAdjustPosition(). ack good move.

Client

ClientAdjustPosition()

Clear SavedMoves with timestamps prior to the correction timestamp. Move Pawn to the position specified by the server, and set bUpdatePosition

ClientUpdatePosition()

Called from PlayerTick() when bUpdatePosition is true. Replays all outstanding SavedMoves to bring the Pawn back up to present client time.

Player State Synchronization

The PlayerController code assumes that the client and server always try to run the exact same state; ClientAdjustPosition() includes the state so that the client can be updated if it has entered a different state. In those cases where the server needs to change states but the client cannot simulate that state change itself, ClientGotoState() is used to force it into that state immediately. There is no support for handling/synchronizing UnrealScript's state stack functionality (PushState()/PopState() and we recommend not using it for PlayerControllers.

Player Animation (Client-Side)

If animation has no gameplay relevance (i.e. in a fighting game), it doesn't need to be executed on the server at all. This can be controlled by SkeletalMeshComponent's bUpdateSkelWhenNotRendered and bIgnoreControllersWhenNotRendered properties, as well as on a per-skeletal controller basis with SkelControlBase::bIgnoreWhenNotRendered. Client-side animation is driven by the inspection of Pawn state (physics, Pawn properties).

For animation-driven movement, the root bone motion is converted into acceleration/velocity, and that is what is being replicated. So the animation remains locked in place (relative to the Actor), and the root bone motion is transferred into acceleration/velocity moving the actor instead.

This is not heavier for the server/client than non root motion movement.

Dead Bodies

If `bTearoff` is true, this Actor is no longer replicated to new clients, and is *torn off* (becomes a `ROLE_Authority`) on clients to which it was being replicated. The `TornOff()` event is called when `bTearOff` is received. The default implementation calls `PlayDying()` on the dying Pawn.

Weapon Firing

Weapon firing is similar to the pattern for Player movement:

- Client plays firing effect (sound, animation, muzzleflash) immediately on player input requesting fire, and calls `ServerStartFire()` and `ServerStopFire()` to request server firing.
    - Client has enough state information (ammo count, weapon timing state etc.) to correctly predict whether weapon can be fired, except in rare cases where client and server version of the relevant properties aren't synchronized.
- The Server spawns projectiles/damage the weapon (spawn the projectile).

Projectiles

This example is useful for simple, predictable projectiles:

- `bNetTemporary` = **TRUE**
    - After initial replication, Actor channel is closed and Actor is never updated again. Actor will be destroyed by client.
    - Saves bandwidth, as well as server property replication tests.
- `bReplicateInstigator` = **TRUE**
    - So projectiles can interact properly with instigator.
- Client-side effect spawning
    - Note that Actors spawned on client have ROLE_Authority on the client, do not exist on server or any other client.
    - These effects don't need to be spawned on the server at all; nor replicated.

Drawback: May erroneously hit or miss target if client simulation of target and/or projectile is off. Don't use for *one shot kill* type projectiles for this reason.

Weapon Attachments

Avoid having interrelated groups of Actors all replicated, both because of performance and to minimize synchronization issues.

In *Unreal Tournament*, Weapons are only replicated to the owning client. Weapon Attachments are not replicated, but spawned clientside and controlled through some replicated Pawn properties. Pawns replicate `FlashCount` and

`FiringMode`, and UT Pawns replicate `CurrentWeaponAttachmentClass`. The `ViewPitch` property in Pawn is another example of this pattern in use.

Sounds

The function `ClientHearSound()` is called on every PlayerController for which the sound is audible. The function `ClientCreateAudioComponent()` is called on the Actor responsible for the sound. If the Actor doesn't exist on the client, the sound will be played at the replicated position, with the audio component created by `WorldInfo`. The function `ClientPlaySound()` in PlayerController plays non-positional sound on client.

Try to play sounds on the client whenever possible!

Physics

Replication

Physics simulation is run on both the client and the server. Updates are sent to the client from the server.

The following struct is used to describe the physical state of a rigid body, and is replicated (as defined in `Actor`):

```
struct RigidBodyState
{
   var vector   Position;
   var Quat   Quaternion;
   var vector   LinVel; // RBSTATE_LINVELSCALE times actual
(precision reasons)
   var vector   AngVel; // RBSTATE_ANGVELSCALE times actual
(precision reasons)
   var int      bNewData;
};
```

A struct used so that all properties change at the same time. The vectors are compressed to integer resolution, so that they are scaled before sending. Quats are compressed to only send 3 values; the 4th value is inferred from the other 3.

For Physics Replication, there are two types of correction:

- Small corrections and object moving: 20% position adjust, 80% additional velocity to target
- Large correction or object stopped: 100% position adjust

Simulation

The following scenarios describe physics simulation:

- `ROLE_SimulatedProxy` Actor simulation
    - The client continuously updates the simulated actor position based on the received position and velocity.
    - If bUpdateSimulatedPosition is true, *authoritative* position updates are continuously sent from the server to the client (otherwise, no position updates are sent after the initial replication of the Actor).
- Pawns on other clients
    - Unlike other Actors, simulated Pawns do not execute normal physics functions on the client. This means that physics events, like the `Landed()` event, are never called for pawns on non-owning clients.
    - The physics mode of the Pawn is inferred from its position, and the `bSimulateGravity` flag, and its predicted position is updated based on the replicated velocity.
        - The `bSimGravityDisabled` flag is set, temporarily turning off gravity simulation, if Pawn didn't fit at the replicated position, and is in danger of falling through the world on the client.
    - Pawns by default set `bReplicatePredictedVelocity` flag to true and implement `AActor::ExtrapolateVelocity()`, which is used to calculate an average velocity to send to clients for simulating movement, taking current acceleration into account. Current implementation improves pawns stopping without a noticeable correction.
- `PHYS_RigidBody` Actors (Vehicles, KActors, etc.)
    - Both client and server simulate the objects, but the server sends *authoritative* updates to the client periodically (when the object is awake). The client then moves the object to match the server version using the code in `AActor::ApplyNewRBState()`

        - Attempts to do so smoothly, by altering velocity to bring about convergence in positions rather than snapping the position if the error is below an acceptable threshold
    - Use `RigidBodyState` struct for atomic replication, when all properties must be received in synch.

For Ragdoll physics, only the hip location is replicated. It is often possible to *tear off* completely and not replicate at all.

For Vehicles (`PHYS_RigidBody` Actors), there is the following network flow:

1. Press key on client
2. Send inputs (throttle, sterring, rise) to server - replicated function `ServerDrive` called
3. Generate ouptut (OutputBrake, OutputGas, etc.); pack into replicated structs that can be sent to the client - function `ProcessCarInput` called on server
4. Update vehicle on server and client; use outputs (OutputBrake, OutputGas, etc.) to apply forces/torques to wheels/vehicle - function `UpdateVehicle` called on client and server

Bandwidth Performance Tips

Optimization goal

The goal here is to maximize the amount of visibly important detail that is sent with a given bandwidth limit. With the bandwidth limit determined at runtime, your goal in writing scripts for Actors that are used in multiplayer games is to keep bandwidth use to a minimum. The techniques we use in our scripts include:

Use `ROLE_SimulatedProxy` and simulated movement whenever possible. For example, nearly all of the Unreal projectiles use `ROLE_SimulatedProxy`. The one exception is the Razorjack alt-fire blades, which the player can steer during gameplay, thus the server must continually update the position to clients.

For quick special effects, spawn the special effects Actors purely on the client side. For example, many of our projectiles use a simulated `HitWall` function to spawn their effects on the client-side. Since these special effects are just decorative rather than affecting gameplay, there is no drawback to doing them completely on the client side.

Execute code based on received property changes via `PreNetReceive()` and `PostNetReceive()`. This is required for updates of Actor properties that shouldn't be directly modified (`Location`, `DrawScale`, `Collision`). The advantage is C++ performance for frequently replicated properties. In addition, the old value is known, so you can check against it. When properties marked with the `Repnotify` keyword are replicated, the UnrealScript `ReplicatedEvent()` event is called with the name of the modified property as a parameter.

Fine tune the default `NetPriority` for each class. Projectiles and players need to have high priorities, purely decorative effects can have lower priorities. The defaults that Unreal provides are good first-pass guesses, but you can always gain some improvement by fine-tuning them. When an actor is first replicated to a client, all of its variables are initialized to their class default values. Subsequently, only variables that differ from their most recent known values are replicated. Thus, you should design your classes so that as many variables as possible are automatically set to their class defaults. For example, if an Actor always should have a `LightBrightness` value of 123, there are two ways you can make that happen: (1) set the class default's value of `LightBrightness` to 123, or (2) in the Actor's `BeginPlay` function, initialize `LightBrightness` to 123. The first approach is more efficient, because the `LightBrightness` value will never need to be replicated. With the second approach, the `LightBrightness` needs to be replicated each time an Actor first becomes relevant to the client.

Also be aware of the following situations:

- `bNetInitial` and `bNetDirty` don't get cleared if the Actor reference can't be serialized. See native replication code for replicating Base, Controller, and Instigator.
- Avoid replicating arrays.
- Some properties (Vectors and Rotations) are truncated for replication.

Native (C++)

Use Native Replication

- `NativeReplication` keyword for the class definition.
- Implement your class' version of `AActor::GetOptimizedRepList()`.
- Still need script `Replication{}` definition to generate `RepIndex` for each replicated property.

- In some cases, you can heavily optimize `GetOptimizedRepList()`, such as APickupFactory version.
- `AActor::IsNetRelevantFor()`
    - Net relevancy caching for Pawns (during a single pass of one client only, since some things like Weapon Attachment are relevant based on their Pawn's relevancy): `bCachedRelevant` flag.

Cheat Detection and Prevention

We have encountered the following types of network-related cheats in *Unreal Tournament*:

- Speedhack
    - Takes advantage of the fact we use the client's clock for movement updates.
    - Built-in detection by verifying client and server clock don't move at excessively different rates.
    - False positives with substantial packet loss
- Aimbots - UnrealScript and external versions
- Wall hacks and radars - UnrealScript and external versions

Traffic Monitoring

To monitor network traffic, licensees can compile the engine with:

`#define SUPPORT_SUPPRESSED_LOGGING 1` (UnBuild.h)

This enables network statistic gathering via `DevNetTraffic` (automatic in *Debug*). To unsuppress network statistic, comment out the line **Suppress[#]=DevNetTraffic** in your game Engine configuration, which is generated from the `DefaultEngine.ini` configuration file.

With `DevNetTraffic`, a complete summary of network data received by the machine will be written to the log. In other words, do this on the client side to see the data sent by the server. Do it on the server side to see the data sent by the client. The data is written to the log in a very verbose format, giving timestamps for all packets, along with a summary of all replicated actors, variables, and function calls.

The following console command can also be used to unsuppress network statistic gathering:

```
unsuppress DevNetTraffic
```

Network Driver Implementation

Plug-in Network Drivers

Unreal's network support is generalized through a plug-in interface based on the C++ `UNetDriver` class. The Unreal engine deals with all networking issues through the `UNetDriver` class, and dynamically loads the appropriate network driver specified in the `Unreal.ini` configuration file, which defaults to:

```
[Engine.Engine]
NetworkDevice=IpDrv.TcpNetDriver
```

Unreal's current Internet support is the UDP network driver, named `TcpNetDriver`. However, it is quite possible to support new kinds of networks by creating new subclasses of `UNetDriver`, and using the `TcpNetDriver` source as a guideline for how to implement the new functions. In fact, the Maverick Software team has created an AppleTalk driver for the Mac version of Unreal using this interface.

The network driver is responsible for opening connections, closing connections, sending data, and receiving unreliable data. However, the contents of the data is defined by the Unreal Wire Protocol, and is completely opaque to the network driver itself. Thus, `UNetDriver` has no idea what information it's communicating, it just sends and receives it blindly. Its characteristics are as follows:

Connection-oriented. `UNetDriver` is responsible for maintaining a list of connections defined by `UNetConnection` subclasses. When `UNetDriver` is layered on top of a connectionless protocol (such as UDP), it must contain its own internal logic for maintaining connections, handling their timeouts, etc. `UNetDriver` is packet-oriented, rather than stream-oriented. All data sent and received through `UNetDriver` exists in discrete packets of size `0...MAX_PACKET_SIZE`.

Packets are sent unreliably. All reliable sending and receiving of packets is performed at a higher level which is invisible to `UNetDriver`. A packet sent from one machine to another might not arrive, might arrive once, or might arrive

multiple times. Additionally, packets might arrive out of order or with significant delay.

Packets are never corrupted. When a packet is received, it is guaranteed that the size and contents received are identical to the size and contents that were sent.

Internet Driver

Unreal's Internet support is based on UDP, the standard Internet protocol for unreliable, connectionless communication. All Unreal gameplay coordination between a client and server goes through one constant, unchanging pairing of UDP addresses (where the client and server each have a 32-bit Internet address and a 16-bit port number). The default port number can be seen in the `Unreal.ini` configuration file.

Thus, Unreal's UDP packets are extremely easy to proxy, and they can be proxied transparently without the proxy having to know anything about their contents.

Unreal Wire Protocol

The Unreal Wire Protocol defines the contents of packets as a stream of bits. The protocol deals with a variety of concepts, such as replicating variables and functions, creating actors, destroying Actors, transmitting files, and exchanging packets reliably and unreliably (both types are used in various circumstances).

Although the protocol has undergone some major changes to improve efficiency and reliability, there aren't any plans to document the packet format any further.

It must be mentioned that the protocol is closely tied to the data structures of the Unreal package file format described in the Unreal Packages Documentation, so it would be quite difficult for an external application to parse and do anything useful with. The source code and comments are the best guide.

The Present and The Future

The words one might use to describe Unreal's networking architecture are "powerful", "general", "complex" and "hard to master". The architecture went through a tremendous amount of evolution during the game's development, to

arrive at the current solution which is a balance between power, simplicity, and the pragmatic need to solve certain problems.

The ideal networking architecture, from an ease-of-programming standpoint, is the pure client-server model, where the client truly acts as a dumb rendering terminal, and receives a display list from the server. However, as discussed previously, the analysis of the rate of bandwidth improvement versus hardware improvement indicates that we live in a world where the pure client-server model will be impractical for the foreseeable future.

Thus, the ability to provide rich client-side simulation of physics and script execution in general is tantamount in any next-generation networking engine. In Unreal we have defined a generalized networking model based on the generalized replication of objects, object variables, and object function calls. This model provides rich simulation capabilities without being hardcoded to any particular simulation model.

I'm pretty sure that this broad networking architecture for Unreal is the right one. Some of the research areas I investigated while determining the current direction include database replication, Unix RPC, the CORBA distributed object model, and Java RPC. My general conclusion is that, though there are some cool ideas out there, nobody else is really pushing the limitations of Internet technology as much as game developers. The same can probably be said about other research areas, such as real-time 3D.

As for the future, there are a lot of areas where this model can be extended and improved. The addition of multicast replicated functions would increase the amount of simulation that can be done without variable replication. An extended peer proxy model could enable the creation of a server backbone enabling the realistic real-time flow of NPC objects between servers. The replicated object model could be extended to other areas of the engine, such as the user-interface and chat system. There is a tremendous amount of promise for this architecture to be applied to much richer, more persistent game environments such as the worlds of _Ultima Online, while retaining the distributed, user-modifiable nature of the system. In the coming years, networking will be an extremely interesting area of game development!_

- Tim Sweeney

Useful links

Much of the original information from this document was updated to include the information from the Networking presentation at the GDC 2006 Licensee Seminar.