Unreal Networking Architecture

Tim Sweeney
Epic MegaGames?, Inc.
http://www.epicgames.com/

Audience: Advanced UnrealScript? Programmers, C++ Programmers.
Last Updated: 07/21/99

Note: References to Unreal's C++ code are only relevant to Unreal licensees who have access to the Unreal source.

History

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.

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.

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 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. Further, the "game state" is self-described by an extensible, object-oriented scripting language 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:

1. If I am a **server**, communicate the current **game state** to all of my **clients**.
2. 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.
3. 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 += PositionIncrment[?]" 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.

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.

The Relevant Set of Actors

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.

1. The actor belongs to the ZoneInfo[?] class, then it is relevant.
2. If the actor has static=true or bNoDelete=true, then it is relevant.
3. If the actor is owned by the player (Owner==Player), then it is relevant.
4. If the actor is a Weapon and is owned by a visible actor, then it is relevant.
5. 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.
6. 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.
7. 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.

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 ULevel::GetRelevantActors. This way, licensees who want to implement different gameplay rules are free to modify our definition of the **relevant set** by modifying ULevel::GetRelevantActors or (better yet) subclassing ULevel, and overriding the function.

Actor 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:

- Bots: 8.0
- Movers: 7.0
- Projectiles: 6.0
- Pawns: 4.0
- Decorative creatures (such as fish): 2.0
- Decorations: 0.5

Overview of Replication

Three Types of Replication

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**:

- **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.

- **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.

- **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 PlayerPawn? class, there is a ClientHearSound? function that tells the player that 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 my DrawType? is DT_Mesh".

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.

The UnrealScript? 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 Pawn class:

```
replication
{
        // Variables the server should send to the client.
```

```
        reliable if( Role==ROLE_Authority )
                Weapon;
        reliable if( Role==ROLE_Authority && bNetOwner )
                 PlayerName, Team, TeamName, bIsPlayer, CarriedDecoration, SelectedItem,
                 GroundSpeed, WaterSpeed, AirSpeed, AccelRate, JumpZ, MaxStepHeight,
                 bBehindView;
        unreliable if( Role==ROLE_Authority && bNetOwner && bIsPlayer && bNetInitial )
                ViewRotation;
        unreliable if( Role==ROLE_Authority && bNetOwner )
                Health, MoveTarget, Score;

        // Functions the server calls on the client side.
        reliable if( Role==ROLE_Authority && RemoteRole==ROLE_AutonomousProxy )
                ClientDying, ClientReStart, ClientGameEnded, ClientSetRotation;
        unreliable if( Role==ROLE_Authority )
                ClientHearSound, ClientMessage;
        reliable if( Role<ROLE_Authority )
                NextItem, SwitchToBestWeapon, bExtra0, bExtra1, bExtra2, bExtra3;

        // Input sent from the client to the server.
        unreliable if( Role<ROLE_AutonomousProxy )
                bZoom, bRun, bLook, bDuck, bSnapLevel, bStrafe;
        unreliable always if( Role<=ROLE_AutonomousProxy )
                bFire, bAltFire;
}
```

The key things you see here are:

- The replication statement is enclosed by "replication {}".
- Each replication definition begins with "reliable if (condition)" or "unreliable if (condition)".

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:

- 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.

- 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.

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
{
        reliable 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 reliably 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_DumbProxy means "this machine is a client and it should not do any simulation on this proxy 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. Is 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[?], i.e. 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 the hell is going to happen..

Variable Replication

Update mechanism

After every **tick**, the client and the server check out all actors in their **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 his 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.

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 to the player who owns that actor. So, replicated functions are only useful in subclasses of PlayerPawn? (i.e. players, who own themselves) 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.

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 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 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. If you need complete accuracy on rotators and vectors, send them as individual float or int components.

Actor Roles

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

```
// Net variables.
enum ENetRole
{
    ROLE_None,            // Means the actor is not relevant in network play.
    ROLE_DumbProxy,       // A dumb proxy.
    ROLE_SimulatedProxy,  // A simulated proxy.
    ROLE_AutonomousProxy, // An autonomous proxy.
    ROLE_Authority,       // The one authoritative version of the actor.
};
var ENetRole Role;
var(Networking) ENetRole RemoteRole;
```

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

- **Role==ROLE_DumbProxy** means the actor is a temporary, approximate proxy which should not simulate any physics at all. On the client, dumb proxies just sit around and are only moved or updated when the server replicates a new location, rotation, or animation information.
    - This situation is only seen in network clients, never for network servers or single-player games.
- **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*.
    - 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.
    - 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.
    - 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 AmbientSound[?] variable is sent from the server to clients because of this **replication definition** in the Actor class: reliable if (Role==ROLE_Authority) AmbientSound[?];
- The 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: unreliable 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: unreliable 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: unreliable 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.

Simulated Functions

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".

Player 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 TimeSeconds[?] on the client side) to the server in a replicated ServerMove[?] function call:

```
function ServerMove
(
    float TimeStamp,
    float AccelX,
    float AccelY,
    float AccelZ,
    float LocX,
    float LocY,
    float LocZ,
    byte MoveFlags,
    eDodgeDir DodgeMove,
    rotator Rot,
    int ViewPitch,
    int ViewYaw
);
```

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:

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

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.

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[120] Options, out string[80] Error );
```

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[120] Options, out string[80] Error );
```

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 playerpawn Login( string[32] Portal, string[120] Options, out string[80] Error, class<playerpawn> SpawnClass );
```

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.

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.
- 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.

Traffic Monitoring

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

```
#define DO_SLOW_GUARD 1
```

This enables network statistic gathering. Then, comment out the line Suppress[#]=DevNetTraffic in Unreal.ini. Then, 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.

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 Unreal.ini, 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 Unreal.ini.

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).

The protocol will undergo some major changes in the coming months to improve efficiency and reliability, so it would not be useful to document it here and now.

It must be mentioned that the protocol is closely tied to the Unreal package file format described in the Package Documentation so it will be quite difficult to parse and do anything useful with.

UnrealScript[?]: UdpLink[?], TcpLink[?]

Unreal's TcpLink[?] and UdpLink[?] scripts enable you to write, in UnrealScript[?], code for actors which are able to communicate with external programs. These are the optimal way of extending Unreal with cool new networking features. For example, some of the cool things one could do are:

- Write "event gateways" which forward events (such as players pressing buttons, walking into zones, etc) to other servers.
- Write server-side programs to interface the server to master servers, statistic trackers, and so on.
- Write server-side programs to maintain persistent player statistics or inventory. For example, you could write a Java/C++ database style "player account manager" program that keeps track of player names, passwords, and their inventory for players. Then, you

can write an UnrealScript? class to interface to the account manager by TCP or UDP and validate players that are trying to log in. See GameInfo?.Login, TcpLink?, and UdpLink? for ideas.
- Write client-side chat programs.

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

End