# Replication De-Obfuscation



## Introduction

Ask any halfway decent UnrealScript programmer what the most confusing aspect of Unreal coding, and he'll be sure to tell you replication. (or perhaps PlayerMovement or pawn/bot AI :) I've been there myself, and I still have trouble with it sometimes, but I think I know enough to write some documentation. Lots of trial and error helped me get far, along with various insights from people that had the source code to the engine. I've tried to piece it all together to create this document. I have however, recently gotten a lot more information about exactly how replication works, which has helped me immensely in understanding replication. Since I can't post that information directly because of NDA, I'll try my best to explain it to you through this networking documentation. Hopefully I'll be able to expand upon what was described in Epic's documents at: NetworkingArchitecture and UTMods. Please read it first, as it will help a lot with background and context. In writing this document, I'm going to assume you've programmed in UnrealScript before, you have at the very least considered the problems of replication, you are reasonably intelligent, you know how to read, and you think my jokes are funny.

First, Unreal Tournament uses a client-server architecture. The client is basically a dumb terminal, that sends data to the server about where the player moved, if they're jumping, if they're shooting, etc. It displays to the user what is given to it by the server. The client's information is necessarily out of date with what exists on the server, because of lag, ping time, and other factors. So in order to present a believable world, the client has logic to simulate what it thinks the server will be sending it. It knows the laws of physics, and that if something is travelling in a straight line, then it will contine to travel with that same velocity. Likewise, if it's falling through the air, it's likely to continue to be accelerated under the force of gravity. The dumb terminal predicts where how the game state will be progressing, and so can provide the user with a more believable game. On the other end of the connection is the server, which holds the authoritative state of what's going on in the game. If the server thinks you are dead, then you are dead. Like your teachers, they're never wrong, no matter how much evidence you've seen to the contrary (like how you **thought** you dodged that rocket). The client gets information sent to it by the server (like where other people are, what things have been shot, etc) , and sends other information back to the server (like where the client wants to move, shoot, etc). In the client-server architecture, the client only communicates with the server, and the server communicates with all of the clients (people connected to the server).

## Server as Authority

With the recent brew-ha-ha regarding the ZeroPing mod, which aims to give the client more control, there have been two sides to the issue of why the server as authority is the best model.

A good description of the history of the networking architectures in games can be found at the top of the tech page's networking document. Go read the "History" section now, and then come back. The current model is that of a client-server architecture, where the client has prediction to minimize the effects of lag. In current versions of UT, the client tells the server all about it's input. Where is the user looking, in which direction are they moving, and are they firing primary or secondary fires.. The server then uses that data to calculate information about the player, and to tell other clients about that player. For the most part, that is all there is to it. ZeroPing pioneered an idea of giving the client more power. It allowed the client to decide whether they hit the other player. In theory, this sounds great. The user gets to decide what's happening, and if they hit the player, then they hit the player. Unfortunately, there are problems with this idea.

In a large-scale game like UT or Q3A, many people try to cheat. The people that make the game have to prevent that from happening. With these games, the client does so little logic, that there is little room to cheat. The client sends its data to the server, and the trusted server decides whether they hit the player or not. Cheats are extremely limited in what they can do, as they can only modify the direction the player is going, and whether to cheat or not. It's not something that is really worthwhile. However, when you have something like ZeroPing, you are letting the client decide whether or not you hit the player. A cheat can pretend that it hit every player, and can wreck havoc in such a game, where it is killing everyone the moment they enter view, and sometimes even getting repeated kills ten times in a row on a single player as they respawn and then are 'hit' again. Mods like ZeroPing are mods, and so do not ahve the same public appeal that a full game does. As such, mods are less likely to be hacked for cheats. But it does not make them immune. If game makers were to use the approach that ZeroPing uses, there would be a flood of cheats within weeks of the game's release. You can understand the resistance that the game companies have for such an approach. It is not because it's the 'not-invented-here' syndrome as many have suggested, but rather that it really is much more open to cheating.

As a side note, I guess the ZeroPing author does realize this. He has hidden his code to make it impossible to cheat with. He took the security through obfuscation approach. This will work for awhile, until code decompilers become available, or the code gets out. While I am not working on any of these myself, I do know of some UnrealScript code decompilers in the works. So this approach really will not work in the long run.

## Replication Overview

Replication deals with how information is sent from the server to the client and vice versa. Because of bandwidth limitations for modems, not everything can be sent, and because of the unpredictability of the Internet, not everything can be ensured of arriving on time. Only a subset of all the information is sent to the client, and replication deals with that subset that travels from the server to the client, and deals with how the client sends its relevant information back to the server.

While the definition of replication may sound simple, there is a lot of underlying assumptions and quirks that you must be aware of, which UT uses in the interest of optimum bandwidth and speed. It's an unmapped jungle out there, and many an explorer has been lost to the dangerous native inhabitants. This guide you are reading is your map and survival guide.

First, the Unreal server does some logic like the following for each client connected:

1. Run all events (like tick, timer, hitwall, etc)
2. Go through each client connected to the server
3. Get the list of relevant actors for this client.
4. For each actor, replicate variables that need to be replicated.
5. Replicate any function calls that should be replicated to that particular client, if they were called during the previous server-side events run above.

The client does the following logic:

1. Receive data from the server, and update its local copies of the actors
2. Run any replicated calls that were called on the server
3. Run any simulated events (like tick, timer, hitwall, etc), and do the local playerpawn handling
4. Find out what replicated were called on the client and need to be replicated to the server, and send them

If that didn't make any sense, don't worry about it. They just outline the main parts of replication, and they will all be explained later.

If you did understand that, there's still plenty to learn. UT has a lot of logic going on under the hood in regards to replication, most of which is not immediately apparent. That was the purpose of this document, to explain these mysteries to you, the reader, so that your networking development will go that much more smoothly. Again, all this will be explained in due time, assuming you read the whole document. :)

## Relevant Actors

First, we need to define what a relevant actor is. A client, as the dumb terminal, needs to know about everything that it will need to display on screen. It needs to know about the other players, and weapons in the level, if any projectiles are visible, if the flag is visible, health packs, etc. All of these actors constitute the relevant set of actors. how is the relevant set of actors determined? If an actor is not relevant, no amount of replication statements or voodoo magic will cause its information to be sent to the client. It must be relevant to that particular client in order for data to be sent to that client. For Unreal / Unreal Tournament, the method of determining relevant actors is as follows (ripped from the tech page's networking document):

The following logic is performed on each actor, and you should be able to tell whether it is relevant. The actor must pass two discrete sets of steps in order to be considered relevant. One it passes the first test, it must then pass the second. The rules higher in the list have precedence over rules that follow them:

1. If it is bNetTemporary, then it does not pass the first stage.
2. If it is not bStatic and not bNoDelete, then it passes the first stage.
3. If it is a LevelInfo, it passes the first stage.
4. If it is a temporary network actor, (as defined by bNetTemporary), and it's been sent to this client already, then it does not pass the first stage. These types of actors will be decribed in more detail later.
5. If its Role is ROLE_None, then does not pass the first stage.
6. If it is bAlwaysRelevant, and time since the last relevancy is less than 1 / NetUpdateFrequency (described farther below) seconds, then it does not pass the first stage. This basically means an actor will only update it's variables to the client NetUpdateFrequency times per second. (NetUpdateFrequency values: Actor: 100, ZoneInfo: 4, GameReplicationInfo: 4, PlayerReplicationInfo: 2, or Inventory: 8).
7. If it is bNetOptional, and it has a non-zero LifeSpan (meaning it does not exist forever), and it is not in the first 150 ms of its lifetime, then it passes the first stage. Any attempt at replicating a bNetOptional actor after the first 150 ms, (because it was pushed off the replication list due to bandwidth contraints for that client, etc,) will fail.
8. If the time since the last relevancy is greater than 1 / ~NetUpdateFrequency seconds, it passes the first stage. (This is a more finely tuned optimization of NetUpdateFrequency, tuned to the type of actor. It will only perform the calculations needed if it has passed through the weeding techniques above.)
9. It does not pass the first stage.

If it passed all of those steps, it's not through and clear to be relevant yet. It has another step to go through. If it is deemed relevant during the following steps, then it is relevant to this client.

1. If it is bAlwaysRelevant, then it is relevant.
2. If it is owned by the client or owned by the actor we are viewing through (as returned by PlayerCalcView), then it is relevant.
3. If it is the client or the actor we are viewing through (as returned by PlayerCalcView), then it is relevant.
4. If its Instigator is the client, then it is relevant.

5. If it has an ambient sound and is within 'audible distance', then it is relevant.
6. If it is a weapon that is owned by a visible Pawn owner, then it is relevant.
7. If it is bHidden or and not bBlockPlayers and the AmbientSound is none, then the actor is not relevant.
8. If it is not relevant (per the above tests), but if any of the above conditions made it relevant within the last RelevantTimeout seconds, (by default, five seconds, but is defined in UnrealTournament.ini:[IpDrv.TcpNetDriver],) then it is relevant.
9. It is not relevant.

Those rules, while seeming complicated, are exactly how UT determines what is relevant to each client. Many times such checks are not related to what you are doing, but they are always good to know. Thinking about it, it makes sense. If something is not visible, then the client probably does not need to know about it. If it's a player on the other side of CTF-LavaGiant, what use does this client have for it? Or if it's an invisible pathnode used for Bot AI, the client also has no use for it. Sending data and variables for such actors would be wasteful on a tiny little 28.8 kbps modem.

Working this discussion into a guideline for your own code, you should set your actors' bAlwaysRelevant to true if they are data storage units that the client needs to know about, like PlayerReplicationInfo or GameReplicationInfo. Things like PlayerReplicationInfo, which store information about the player, are always relevant. This allows a client to view the Scoreboard and see the information (name, score, ping, packetloss, etc) for everyone in the level, regardless of whether that player actor is relevant to the client. PlayerReplicationInfo's allow precious information that's small, bandwidth wise, to be relevant and replicated to each client, yet have information like the location, velocity, weapon, animations, etc of the player to not be replicated when it's not necessary (because it's in the Pawn/PlayerPawn, which is not always relevant.) Hopefully I've not lost you here. It's an important thing to know and remember about Unreal replication. Where to put your variables, and how to make them relevant to the client. If you needed some important game information to be seen by all clients, your best bet would be to subclass ReplicationInfo and utilize that for your data variables. Information that is only needed when it's seen is best put in regular actors, that will go in and out of relevancy as time passes.

In a nutshell, if the client needs to know about it, they will. All inventory is relevant to the client, as is all nearby players. All projectiles and weapon effects that are visible are relelvant. Anything that affects the player's world on the client, will be replicated. This includes things that block the player's movement, or actors that play ambient music. There is an overhead to spawning a new actor in netplay, and so UT does not immediately remove an actor from the relevancy list when it goes out of view. It gives it a few seconds before it is really considered not relevant. Oh, and UT takes into account the fact that a player may be viewing through another actor's eyes, as in the case of the GuidedWarhead, or the teammate views when you hit the number keys on the right side of your keyboard.

## NetPriorities

While that's the "simple" description, figuring out why things happen can get a bit more complicated. Unreal Tournament gives different priorities for different types of actors. NetPriority, which should typically be between 1 and 3, determines how to prioritize a given actor's bandwidth "allocation" in the stream being sent to each client. Actors are first sorted by a calculation based upon their priority (adjusted to take into account how close the client is looking at them, etc.) Actors marked bNetOptional are sent to the bottom of the list. So....actors with a higher priority are sent first in the bandwidth stream, and when the connection becomes saturated, any remaining actors are starved of bandwidth until later. (UT remembers which actors have not been replicated, and so it can give bNetOptional actors 150 ms to get replicated before they are discarded. See the above relevancy list for more information.)

Here's a wide and almost complete sampling of UT's NetPriorities:

| | |
|---|---|
| Bots | 3.0 |
| PlayerPawns | 3.0 |
| GuidedRedeemer | 3.0 |
| Movers | 2.7 |
| Projectiles | 2.5 |
| Carcasses | 2.5 |
| Other Pawns | 2.0 (Spectators, Titans, Bunny Rabbits, etc) |
| Effects | 2.0 (sparks, blood, smoke) |
| Simpler effects | 1.4 (shell casings) |
| Inventory | 1.4 or 2.5 |
| Everything else | 1.0 |

This shows that the server considers a Player's (be it bot or playerpawn) information to be much more important than something like bullet casings from a gun. And it makes sense. A guided redeemer also has unpredictability to it, since it is controlled by a player, and thus needs a higher NetPriority. A projectile follows the simple Physics given to it, eg: PHYS_Falling, PHYS_Pojectile, and so they are not very unpredictable. Thus, its NetPriority is not as high as a player's. However, projectiles are still quite important to gameplay (doesn't it suck to get hit by that rocket you never saw? :), and they get a high NetPriority. It's the same with the carcasses, since their animations as they fly through the air or fall to the ground are quite important (not to gameplay, but to that touchy-feel-good feeling when you kill someone.) You should use these examples as a guide when creating your own actors that need a different priority.

One additional thing to note is that you can change the NetPriority on the fly. This is done with Inventory, for example. It exists as 1.4 for most of its life, but when it is thrown from a player's inventory, it now is a moving actor with more importance attached to it. It is then given a higher NetPriority. Once it hits the ground, it gets a NetPriority of 1.4 again, since it does not need to be updated as often when it is sitting stationary on the ground.

In general, all of the relevant actors for a particular client are put in an array each tick, and then sorted by their priorities. The server then starts replication the actors, starting with the highest NetPrioritized actor, working it's way down. UT then stops replicating actors when the bandwidth becomes saturated. That's why it's important to prioritize your actors correctly, so that when a lot of actors are on

screen, or on the lowly 28.8K connections, UT will know how to perform triage, and who to starve of bandwidth first. If an actor is not replicated, the variables it was supposed to tell the client about are not forgotten. They will be put on the list during the next tick, to have another go at getting replicated. Note that doubling the size of the actor's NetPriority does not actually change anything in regards to giving it more bandwidth, since a NetPriority only has meaning in relation to other NetPriorities. Since 3.0 is the highest on the list, there is no real difference between giving your actor a NetPriority of 4.0 and 4000.0, since they will both be placed at the top of the list.

## Roles and RemoteRoles Introduction

Now is a good a time as any to explain the purposes of the various roles in Unreal. Roles simply guide UT in determining what role this particular actor plays in the game of replication. For example, a player needs to be replicated quite differently from an ammo box sitting on the ground, or a projectile moving according to the rules of physics. On a server, the Role will be ROLE_Authority, which makes sense since that actor is currently being evaluated on the authoritative server. The RemoteRole of that actor will be set to any of a variety of roles, defined below. On the client, the Role and RemoteRole will be reversed. On the client, the RemoteRole will be ROLE_Authority, and the Role will be one of the other role types. Let's go through the various roles (which will be set in the RemoteRole default properties), in order of increasing 'control.'

**ROLE_None**
: This role is probably the simplest. It simply tells the server not to make this actor relevant to the client. No information will be sent about it, and it will exist where it was created only. If it was created on both the client and the server (via a simulated function, described later), and given a Role of ROLE_None, then it would exist both on the server and on the client, and they would both operate independently of each other.

**ROLE_DumbProxy**
: This role is used for objects that don't move much, yet still need information about them updated to the client. The server does this by sending periodic updates about this actor's location and velocity to the client. If the actor has any sort of logic to it, with Physics, for example, it's much better to use the following role.

**ROLE_SimulatedProxy**
: This is used for anything that should be simulated over the network, via prediction. Take a rocket, for example. It always travels in a straight line, and is an ideal candidate for SimulatedProxy. By the same token, UT can predict the falling physics clientside, and so grenades, and falling people are also good candidates for this role. Anything that can be predicted clientside because of Physics or clientside physics should use this Role. All Bots and PlayerPawns use this role as well, since when a player is running, he's more than likely to continue to run. It's the best prediction method that can be used for unpredictable players. The only exception to this is the playerpawn you yourself are playing as.

**ROLE_AutonomousProxy**
: This is used for you. When you play online, you yourself should be treated differently from all other playerpawns. You don't want your own self being predicted based upon the server's expectations. Rather, you want to control yourself and tell the server where you are moving. With this Role, that's easily possible. Many things that should only be done for your own playerpawn, like server-corrections of your location when you get lagged out, are only done with this role.

**ROLE_Authority**
: This role is different from the others in that it is always the Role on the server. On the server, all Roles will be ROLE_Authority, and the RemoteRole will be one of the above. On the client, the reverse will be true. This role doesn't have any real significance, beyond it's use in replication statements, described below.

## General Replication

For each actor that is deemed to be relevant to a particular client, the server needs to determine what subset of its variables will be sent to the client. The most obvious rule is to only send variables if they change. Unfortunately, that rule alone is not enough to fit data over a modem. Instead, the server has to further whittle down its list of variables, and it does this through replication statements. These replication statements are executed for each client, for each relevant actor to that client. When a subset of the variables of a subset of the actors are sent over the network, only then can it be fit on a regular modem. But how can a client operate without all of the information in the game? Once you stop and think about it, it's not quite that unthinkable. Do all of a Bot's AI variables need to be replicated to the client? Only the result of the AI needs to be sent, which basically consists of the same thing for any other Pawn in the game. Also, depending upon the RemoteRole, different things are replicated. Some cause the Location to be updated periodically as a poor man's simulation of its motion, while another role causes the physics to be replicated, and the location not to be, creating true client-side simulation of the actor's motion.

### No Recursive Relevancy

Please note one important point that can easily be overlooked. Relevancy is not recursive with replicated references to variables. Let's say that Actor A contains a reference to Actor B in Property C, and Actor A replicates Property C (currently set to Actor B). Let's also say that Actor A is relevant, and Actor B is not. (Don't worry, there's only three letters in this example.) In this example, Actor A is relevant, and so it's Property C variable will be replicated to the client. Now what does the client know? The client knows about Actor B through Actor A's reference, but Actor B is not relevant. All of Actor B's variables are not checked for replication. The client only knows about Actor B, which really is of no use whatsoever, since the variables are not replicated. I initially thought that replicated variables to actors caused those actors to be relevant and parsed themselves for replicated variables, but that view is false. An actor can only be relevant according to the rules stated above, which make no provision for a reliable reference to them from a relevant actor.

### Replication Checks

When you think about the number of clients in a server, and the number of relevant actors for each client, and the number of replication checks on variables for each actor, there's a large number of such checks. Special care should be taken when implementing replication

statements. As each conditional is evaluated, you should be considerate of the server's limited processing ability. Don't call functions from within replication statements, as function calls are much too slow for replication statements. Along the same lines, you should try to keep your checks as simple as possible, only evaluating a condition if its necessary. Finally, note that Epic had a problem with replication checks taking too long for some important actors like Actor or Inventory, both of which replicate many variables. To alleviate the situation on servers, Epic moved the checks to native code. That's why you will see the 'nativereplication' on a few of the Engine classes. Any variables being evaluated in those classes will be covered by the nativereplication code, making any UnrealScript replication statements useless in that class, (except as a form of documentation for UnrealScript programmers.) Any variables that are not defined in a nativereplication class, (defined in a non-nativereplication subclass or superclass,) will be replicated via the unrealscript definitions. Despite the fact that the nativereplication checks are in C++, the UnrealScript checks that remain in those classes are still accurate, and a good guide to go by in determining problems with replication, and ways around it.

## Special Replication Variables

When writing replication conditionals, there are a few things you should be aware of. During replication, a few helpful variables are set for your use in evaluating whether it should be replicated. These variables are defined in actor, and are:

**bNetInitial**
: true if this is the first time this actor is being replicated across the network. Useful for variables that differ from the defaultproperties, yet will not change over the life of the actor.

**bNetOwner**
: true if the player we are replicating to owns this actor directly.

**bSimulatedPawn**
: true if the actor is a pawn (or a pawn subclass) with Role == ROLE_SimulatedProxy. In other words, true if it's a pawn that's not the client.

**bDemoRecording**
**bClientDemoRecording**
**bClientDemoNetFunc**
: used for demo recording purposes.

**bNetRelevant**
**bNetSee**
**bNetHear**
**bNetFeel**
: Not used anymore, never even set.

## NetUpdateFrequency

NetUpdateFrequency is another variable useful in replication. It effectively causes an otherwise relevant (not useful when the actor is not relevant) actor to be relevant NetUpdateFrequency times per second. This is useful when the data changes frequently but you only want periodic updates. For example, PlayerReplicationInfo contains a lot of data about the client. Stuff like the various player's pings may change quite frequently, and having the client keep track of such information would be quite a network hog. But since PlayerReplicationInfo has a NetUpdateFrequency of 2, it is only updated twice a second, which is much nicer for that player's bandwidth. The list of the current NetUpdateFrequency's for all UT classes are listed below:

| | |
|---|---|
| Actor | 100 Hz (times per second) |
| ZoneInfo | 4 Hz |
| GameReplicationInfo | 4 Hz |
| PlayerReplicationInfo | 2 Hz |
| Inventory | 8 Hz |

## Reliable versus Unreliable

Another optimization used in networking is the difference between Reliable and Unreliable data. In the case of variables, all data is reliable. It is all guaranteed to reach the client, even if it arrives in the wrong order. (Pawns however, have code that prevent them from appearing jumpy, although the effects of this can still be seen at times when the player bounces back and forth in a somewhat laggy game.....location updates are being received out of order.) Functions however, exist in both reliable and unreliable forms. A reliable function, like reliable data, is guaranteed to reach the client. An unreliable function however, is sent to the client, but it is not guaranteed of reaching the client. Something that sends messages to the client, like BroadcastMessage or BroadcastLocalizedMessage, are reliable to ensure that the client gets the message. Something like a client's update to the server about its position need not be reliable, however. Since the client sends these update requests to the server every single tick (in ServerMove, if you're wondering), it does not matter if they all reach the server. And since each update is timestamped, the server knows which ones to regard because they are out of order or lagged. And so, the server can be assured of getting updates from the client, while not requiring that they ALL be sent. A reliable function is repeatedly sent over the network until the client acknowledges it. This can lead to a one second delay or more if it has to try to send the message once or twice. Since you don't want to bog down the network connection with reliable updates about where the client is looking or moving, unreliable is best in this situation since the server only needs to get "most" of them to work great.

## Writing Replication Statements

There are a few useful things to know about when writing your own replication statements. First, you cannot override replication statements. If a replication statement exists in a parent in regards to Location, Velocity, some function, or another variable you want to change the conditions for...you can't. Sorry. Instead, you have to mold your actor into fitting the requirements so that it's property or

function will be replicated. Usually you can accomplish this by setting a variable or two (you can't set the "const" ones which are set natively of course), or by fiddling with the RemoteRole until it matches up. However, don't go around doing that blindly, or you're sure never to succeed. Instead, keep reading, and you will discover (hopefully, if I do it right :), exactly what each of the Roles are for, and what side effects they cause. Next, keep them simple. We already discussed the importance of not doing anything overly complicated, and that rule should be adhered to. Second, use the bNet* variables if possible. These variables, like bNetInitial, bNetOwner, bSimulatedPawn are quite useful, and are gratis. There is no additional CPU requirement to use these variables, as they are set for you automatically. Another very important thing to hinge your variables on is the Role and RemoteRole. "if (Role == ROLE_Authority)" would cause that variable/function to be replicated from the server (where the Role equals ROLE_Authority) to the client. Something like: "if (Role < ROLE_Authority)" would cause the variable to be replicated from the client (where the Role is less powerful than ROLE_Authority) to the server. Checks can also be performed to do various things depending upon if the Role/RemoteRole is a SimulatedProxy or a DumbProxy. The best place to find examples of these more intricate checks is in the Engine classes, particularly Actor, Pawn, and PlayerPawn. It should also go without saying that you should not change anything in a replication statement. Doing i++ in a replication statement causes i to change unpredictably as it is being replicated, which is not something you really want to do. And finally, as the last word advice, the server checks all incoming data to be sure it is legitimate. If the client replicates something (a variable or a function) to the server, the server will exchange the Role and RemoteRole variables temporarily, and evaluate the replication condition to see if the other end had a legitimate reason for sending that data. If the check fails, then the data (variable or function call) is discarded. That means that when you are writing replication statements, make sure that the server also has all the data needed to perform the replication check. Otherwise, it will never be processed or accepted by the server.

## Variable Replication

Variables are one type of data that can be sent through replication statements. The other is function calls (RPC, if that helps). Here, we'll be discussing how data is replicated from the server to the client, and vice versa. There's a lot of little things that server to complicate it more than the replication described above. As I already stated above, all variables are considered reliable, so the unreliable versus reliable discussion is irrelevant here.

### Newly Relevant Actors (Spawning)

When an actor becomes relevant to a client, it is spawned on the client, with all the defaultproperties of the actor. If the server set variables immediately after the spawning of the actor, (before the relevancy checks), then those variables will be sent to the client as an 'addendum' of sorts, assuming those variables meet the replication statements. The server will send any variables that differ from the defaultproperties. Things can get hairy though, when the server has a different idea of an actor's defaultproperties than the client does. Initially, the defaultproperties on the server should match up with what the client has. If a defaultproperty is changed on the server, that change is not replicated to the client. You must change it in a simulated function in order for that to occur. However, what happens if the actor becomes irrelevant? In that case, bad things happen. When the actor becomes relevant again, the default spawned actor on the client will have different defaultproperties from the server. When the server then proceeds to perform the checks on what to replicate, it sees that the default var is the same as that var, and so that var is not replicated. The client then never gets data, and so has the REAL default variable, of a fresh actor. That variable can be incorrect, and cause strange behavior to occur.

As an example, for a particular class-based mod, a programmer (not me, this time ;-) had set both the Mesh and Default.Mesh to the playerclass that the player had chosen. Players however, go in and out of scope, to save on network bandwidth. Normally they are only relevant when you can see them, and for about three seconds once them disappear from view. Let's say the player changed their playerclass at that point. Or say they changed their class at the start of the game while the round has not started and they are invisible, and thus not relevant. When they did become relevant to the client, because they turned a corner, or because the round just started and they are spawned and made visible, the Mesh == Default.Mesh, and so the Mesh is never replicated. The skin however, was still replicated normally. To clients in the game, they saw the player as the original mesh, with a inappropriate skin that was intended for another model. Like I said, strange behavior, that can be hard to trace down. The way to handle this, is to not set the Default variables unless you are really know what you are doing. Just setting the Default.Mesh because it feels right will only cause problems. In this case, the Default.Mesh was not needed for everything, and removing that assignment caused everything to work fine.

### A Time for Replication

Replicated variables are only replicated at the end of each tick. That means that if you change the variable repeatedly during a loop, or some other block of code, only the final value will be replicated. If you change a variable many times during a tick, and change it back to its original value when the tick completes, then no variable change will have been seen, and no network bandwidth will have been spent.

### Client-Server Replication

Another quirk to be aware of is that client-server replication exists ONLY for your own playerpawn. That means that if you want a variable replicated from the client to the server, it needs to be done in your current PlayerPawn (or one of its parent classes). (You can use function replication to get around this limitation however.) Other playerpawns in the level are not the one that you currently 'possess', so their variables will not be replicated to the server. Just be aware of that when trying to replicate your variables and they aren't working. :) Thinking about it, it makes sense. What right does the client have to send any information to the server about another playerpawn, or some inventory item? The client is only concerned with variable changes that the player himself (or herself) makes, and thus those are the only ones that are sent to the server.

### Replication Only When Necessary

Further optimizations exist for variable replications as well. A variable will only be replicated if the server thinks the client has the incorrect

version compared to the server's latest version. This usually happens when the server changes a variable, and it knows the clients do not know about such a change. (They would if the function that changed them was simulated and was called clientside, meaning that you technically would not need to replicate that variable. This is explained later.. :) This means your code can change stuff clientside, and it will not be written over by server-side data until the server's data changes. As an example, I used this technique to implement a simple little static LOD back in the Unreal days before Epic introduced their realtime LOD. The client determined how far away the player was, and switched in variously detailed models depending on the distance. This allowed more models onscreen at once. And in multiplayer games, the client could change the mesh to the various LOD meshes, and it would work fine. Since the server never saw a change in the server's mesh (assuming it was a dedicated server), the client could change the LOD mesh without any problems. When the server did change the mesh however, the client would get a new mesh variable sent to it, and so it would have to then change it again. As a bragging aside, it even modified the intensity at which is applied LOD depending upon the framerate. ;)

## Replication Data Approximation

Another optimization is that of how Unreal sends vectors, rotators, and arrays across the internet. The rules for this can best be described from the Unreal Tech page's Networking document, and I've copied it here to keep it in context:

- Vectors and Rotators: To improve bandwidth efficiency, Unreal reduces the accuracy of vectors and rotators slightly, a process known as quantizing. 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 (Pitch8)&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.

## Variable Replication Examples

Now would probably be a good time to go through some important examples of replication and how it is used with variables, to give you a better idea of how things work, and to help give examples for your own adventures in coding.

Let's take a few examples, and describe exactly what they do. We'll start off with the easy ones, and gradually work our way up in complexity. The following examples are all taken from Actor, unless specified otherwise.

In PlayerPawn:

```
reliable if ( Role < ROLE_Authority )
                Password, bReadyToPlay;
```

This here shows where your Password is sent to the server, when trying to join passworded servers, or to log in as an admin to that server. The server needs your password in order to validate it, and this code sends it to the server. Let's evaluate this check both on the server and on the client. Server: On the server, Role == ROLE_Authority. Looking at the conditional, it can easily be shown to be false, and so the Password is not replicated server-side. This means that the server does not send the Password to the other end of the connection. Client: Since on the client, the Role is not ROLE_Authority (remember it's only Authority on the server itself), this replication check evaluations to true on the client. And since variable replication only works in your current PlayerPawn, it will be sent to the server for yourself alone. (It makes sense not to send other's passwords, doesn't it? ;)
This also shows how bReadyToPlay, which is used in Tournament style games, where everyone has to click to start the game. Note that this is only sent for yourself, since the client can only replicate variables to the server if they are part of it's own playerpawn actor.

```
unreliable if( Role == ROLE_Authority )
                Owner, Role, RemoteRole;
```

This ensures that the owner of an object is always replicated to the client. Remember that the owner's variables themselves are not replicated, (because there is no recursive replication,) but the actor itself is. This means that simple comparisons "if (PlayerPawn == Owner)" will work fine because the reference to the Owner is replicated to the client. Remember that the unreliable versus reliable makes no difference in UT, as they are both treated equivilently. They may have had an effect at sone point during the Unreal 1 days, but they no longer play a factor in development. Don't let that confuse you.
Replicating Role and RemoteRole may seem strange at first, since they should be reversed on the client. However, there's native code that causes that switch to be made. On the client, it will ensure that those two variables are reversed, so that RemoteRole is ROLE_Authority. You may wonder why they are even replicated in the first place. While the client never has any direct use for these variables in the code, there is one easy-to-overlook place where they are vitally important: Replication Statements. When the client is evaluating whether it needs to replicate a certain function call to the server, or a playerpawn's variable to the server, the client needs accurate copies of what Role and RemoteRole are. Without those, it would be unable to make the necessary decisions on what to send to the server. And since the server checks the validity of the replication data sent to the server with Role/RemoteRole, the client needs the latest copy in order to successfully replicate that data.

```
unreliable if( bNetOwner && Role == ROLE_Authority )
                bNetOwner, Inventory;
```

Inventory is the head of the linked list that lists the entire Inventory for the Actor (usually only utilized with Pawn and PlayerPawn). We want the inventory to be replicated to the client, so that he knows what Inventory he has, to display on his HUD, and stuff

(makes sense, right?). Simply replicating the head of a linked list (where each object points to the next in the list) is not enough however. Each actor must itself be relevant (satisfied by the criterion that it must be owned by the current playerpawn), and each link itself must be replicated. And since Inventory subclass Actor somewhere in the chain, Inventory also has a replicated Inventory variable. And that's how you are able to see your Inventory client-side. And since I don't need to know the full Inventory list of what every other player in the game has (the current weapon and the shieldbelt effect are transferred via other means), the Inventory will only be replicated if the client is the owner.
The other variable in the above replication statement is bNetOwner. Since this is set natively on both the client and server, this variable does not need to be replicated. It could be argued that it adds a tiny amount to bandwidth, but its effect is probably negligible. Perhaps it will be removed in a future patch.

```
unreliable if( DrawType == DT_Mesh && Role == ROLE_Authority )
                Mesh, PrePivot, bMeshEnviroMap, Skin, MultiSkins, Fatness, AmbientGlow, ScaleGlow, bUnlit;
```

Here we see that all these mesh-specific variables are ONLY replicated if this actor is currently being displayed as a mesh. If it's a sprite, or a brush, or even no drawtype at all, there is no reason to send these variables.
Now let's get into some of the more complex variable replication statements...

```
unreliable if( RemoteRole == ROLE_SimulatedProxy )
                Base;
```

Here we see a slightly more complex replication statement, that does not involve Role == ROLE_Authority. Here we see that the current Base (set via SetBase) is only replicated if the actor is set to be a simulated proxy. If you set the actor to a DumbProxy, you'll see no Base change clientside, and instead will get the jerky Location updates that the server sends to you, (more on this later). So if you are using SetBase, make sure it's a SimulatedProxy, or if it's not a SimulatedProxy, use something other than SetBase. :) Or if you need both SetBase and a SimulatedProxy, the Base will not be replicated for you. You'll have to work out some other mechanism to do that, probably a simulated function that sets the base, so it is run on the client, (more on this later, too.) You might still have troubles when the actor gets its simulated function called while the actor was not relevant. This results in the simulated function never being called clientside, and then when the actor does finally become relevant, it's as if that function was never called (and the base isn't set), which will result in those things affectionately referred to as 'bugs.' You can make the actor being attached (and what it's being attached to) bAlwaysRelevant so that the simulated functions will always be called, but that might be a bit too harsh on network bandwidth. :)

```
unreliable if( RemoteRole == ROLE_SimulatedProxy && Physics == PHYS_Rotating && bNetInitial )
                bFixedRotationDir, bRotateToDesired, RotationRate, DesiredRotation;
```

This one is still relatively easy, but it's getting more complex. Here we have similar logic to what was seen above, with the server only sending these variables to the client if the client is a SimulatedProxy. However, since all these variables apply ONLY to PHYS_Rotating, there is no need to replicate these variables to the client if the client is not using PHYS_Rotating. And finally, a very important clause at the end is the bNetInitial one. This states that these variables will only be replicated to the client when it is being replicated for the first time.

```
unreliable if( bSimFall || (RemoteRole == ROLE_SimulatedProxy && bNetInitial && !bSimulatedPawn) )
                Physics, Acceleration, bBounce;
```

Here we have some of the interesting variables replicated. We see that if the bSimFall is set to true, it will replicate the Physics to the client. This is used when tossing weapons from your inventory. When they are tossed, they require a Physics change from PHYS_None while sitting in your Inventory to PHYS_Falling as they fly through the air. The then get set back to PHYS_None when they land on the ground. All these Physics changes are accomplished by setting bSimFall at the key points during the TournamentWeapon's life. It is set to true when it is thrown from the inventory, to capture the change, and left on until it hits the ground, where it changes Physics again. After it finally hits and has its physics reset, bSimFall is set to false so no further Physics changes occur. Looking at the second half of the statement, we see that the Physics is replicated ONLY if it's a SimulatedProxy, it's the first time this actor is being replicated over the internet, AND it's not a simulated pawn. The bSimulatedPawn variable is set to true if it's a Pawn with a RemoteRole of ROLE_SimulatedProxy (who would have thought? :) This means any changes to the Physics of a SimulatedProxy non-pawn actor before it is replicated will be replicated to the client. The Physics are never replicated for pawns. Rather, the code for their physics is hard-wired into the code to cause them to be pushed to the ground. Basically, when a pawn jumps, the server sets his vertical velocity so that he travels upwards. That Velocity is then replicated to the client (described below). The client then checks if the pawn is a player (eg: a bot or a playerpawn), that they are currently unable to fly (set via the Pawn's bCanFly variable), and that they are not in a water zone (since that involves different gravities and physics). So in fact, the Physics for a pawn is never replicated to the client, it only appears to do so. If you are attempting to create alternate physics with the playerpawn, you will need to ensure that you set bCanFly to true so that the native code does not enforce the falling Physics upon the player when he's on the wall or in the air. You must then implement the alternate means of transportation yourself.

```
unreliable if( !bCarriedItem && (bNetInitial || bSimulatedPawn || RemoteRole < ROLE_SimulatedProxy) && Role == ROLE_Autho
                Location;
```

Here we see another important variable, Location, and it's replication. We see that it is not replicated for carried items. This is useful in the case of Inventory. When a player is carrying it, there's no reason its location should be replicated, since it isn't used for anything. replicating a player's entire inventory would be quite the strain on network bandwidth. Let's look at the next section of the statement. The location is replicated if this is the first time the actor is being replicated (assuming the other parts of the conditional are true). This is very useful, since all pawns will be spawned in different locations when you enter the game, and rockets will need their start locations set when they spawn out of a rocket launcher, etc. The location is also sent if this is a bSimulatedPawn. This helps correct any errors that may occur in replicating pawns. Since a pawn can change direction, and a client

may not always know about it (due to lag, etc), this location resetting is the only way in which the locational updates can be 'corrected'. Note that this is not used to correct your own location when you experience lag. When that happens, you are an AutonomousProxy, and functions specific to PlayerPawn handle that (namely ClientAdjustPosition). For example, PHYS_Walking is not like PHYS_Projectile, where the position can be predicted with great accuracy. PHYS_Walking just means "keep them attached to the ground", basically, The client's only updates (without location) would have been their velocity, which can easily lead to errors in the player's movements over any significant amount of time. This location is used as a correcting factor, making sure the client's view does not straw too far from where he expects the player to be, while at the same time keeping the velocity's replication so he can be predicted inbetween server updates. The other option for allowing the Location to replicate is that of where the RemoteRole is less than a ROLE_SimulatedProxy, namely, a ROLE_DumbProxy, since the ROLE_None prevents relevancy in the first place. DumbProxies get periodic updates from the server every few ticks, and would create a jumpy effect in netplay. It was the cause of a jerky puck in the hockey mod I attempted awhile ago, although I knew nowhere near enough to fix it at the time. :) A SimulatedProxy would not get sent Location updates, since it would be predicted through the use of the Velocity and the current Physics. Since DumbProxies aren't simulated client-side to achieve smoothness, they only get the Location updates. I'm going to clear this all up after the examples of the various replicated variables.

```
unreliable if( !bCarriedItem && (DrawType == DT_Mesh || DrawType == DT_Brush) && (bNetInitial || bSimulatedPawn || Remote
                Rotation;
```

Here we see another important variable, Rotation. This also has the same bCarriedItem clause, for the same reasons as described above. The rotation is also only replicated if this actor is a mesh or a brush. Since sprites always face the player, replicating their rotation is useless. The bNetInitial also causes the rotation to be replicated the first time through, (again, assuming if the other conditions are true,) since rockets need to be facing in the right direction when it has them flying through the air. Their velocity determines their direction of movement, but the direction they are facing which is equally important, is determined by their rotation. The rotation is replicated if they are a simulated pawn (every playerpawn and bot in the level except your own self). This is so you can see what direction the other people are facing. The ViewRotation (which determines where the client is looking), is sent to the server via ServerMove, where it is translated into a rotation. it is this rotation that is then replicated to the clients. And finally, if it is a DumbProxy (the only valid one less than SimulatedProxy), it also gets the rotation updates. Rotation updates do not need any interpolating between updates like location does. Location lag is much more noticeable when the player is moving quickly, but you never really notice Rotation lag. Besides, there is no real way to predict or forecast rotation. It depends entirely upon the other user's mouse.

```
unreliable if( bSimFall || ((RemoteRole == ROLE_SimulatedProxy && (bNetInitial || bSimulatedPawn)) || bIsMover) )
                Velocity;
```

Here's one of the last important variables with a complex replication statement. The velocity is replicated if it has bSimFall set, (used when throwing weapons from your inventory). The PHYS_Falling alone isn't enough. It needs to know exactly what it's initial velocity is when it is launched from the pawn. And no, bNetInitial will not work here, since it's not the first time it's being replicated. It's existed as a weapon for quite some time. It's just for the initial velocity of when it's being thrown from the player that we want updates. Moving along, we see that SimulatedProxies get their Velocities replicated if it the first time on the replication channel, for newly spawned rockets, or for grenades, or shock projectiles, etc. SimulatedProxies also get their velocities replicated if they are a simulated pawn, as all the various pawns need their velocities replicated so that they can be predicated locally. And finally, movers get their velocities replicated, so that the client can accurately predict the mover's movement locally. In the early Unreal 1 days, mover's velocity was not replicated, and so the client got periodic update locations because it was a DumbProxy. This resulted in very jumpy movers in netplay, something that was fixed for Unreal 224+.

```
unreliable if( DrawType == DT_Mesh && ((RemoteRole <= ROLE_SimulatedProxy && (!bNetOwner || !bClientAnim)) || bDemoRecord
                AnimSequence, SimAnim, AnimMinRate, bAnimNotify;
```

There are a few more variables here, all of which relate to animation. These variables are only replicated if it is currently being drawn as a Mesh. If we are recording a demo, then the animations are always sent. Otherwise, it checks the somewhat confusing conditional. If the actor is a DumbProxy, and the player is not the owner of this object, and bClientAnim is not set, then it replicates the animation variables. In the case of weapons, you see those animations clientside, and so the animation variables do not need to be replicated from the server. That is because all TournamentWeapons have the bClientAnim set to true, indicating that their animations are handled clientside. If the animations are not handled clientside, and the actor is a dumb proxy, then it will send them from the server. SimulatedProxies, which include pawns and rockets and other projectiles, are all simulated proxies, and so they do not receive animations from the server. Instead, the client animates them with its client-side prediction. In the case of pawns, this is handled internally in the engine (for another tutorial ;), and you need not worry about it.

## Role and RemoteRole Summary

Hopefully this intricate look into some of the more complex and confusing variables has helped you to grasp the concepts of replication some more, and realize many of the differences between the various roles. However, with the above knowledge in mind, I'd like to do a little re-cap, emphasizing the use of dumb, simulated, and autonomous proxies, and help give you a clearer idea of when each should be used.

ROLE_Authority - Again, this is the Role for all actors on the server, and is the RemoteRole for all actors on the client. It basically means that I'm the authority on this actor. This is the only one variable that Role is set to, server-side. The remaining ones are all used for the RemoteRole when on the server.

**ROLE_None**

This basically says: Never make this actor relevant. This prevents replication conditions from even being run, and so does not factor

into any of the replication statements.

**ROLE_DumbProxy**

This role is used for simple actors that require no real client-side prediction. They don't need any Physics prediction, but they only need to be relevant to the client to get their variables replicated. This is used for all Inventory actors when they reside on the ground, which prevents any higher-bandwidth SimulatedProxy stuff, but yet allows you to see all the stuff you need for the inventory actor. If you do try to move this actor, the client will get periodic updates, usually around every half-second or so on a regular connection. This creates quite a jumpy effect if you are using DumbProxy with a particular Physics type, since it really isn't intended for that. Rather, the Location updates are useful for when moving the actor across the level with a flag respawn, etc. The same is done with rotation updates, with periodic changes coming in to the client every so often. This role also causes animation updates to be sent from the server to the client, unless the client specifically tells it not to. In summary, an actor with this role is force-fed data from the server. It's intended for actor's that don't move (except in jumps like transportation, respawn, etc), yet are still relevant.

**Other quirks:** There's a few quirks that can have a significant effect on your netplay development. DumbProxy actors do not have a variety of events run clientside. Simulated Tick(), Timer(), and state code, along with the physics calculations, are not performed on the client of a DumbProxy actor. They are supposed to be a dumb proxy actor that gets data sent over the net, and so these events are not called. One exception to the above rule is that physics **are** performed clientside for DumbProxy actors if the actor has a falling physics, which allows tossed weapons to fall correctly in netplay. In an early attempt at the weapon in the tutorial at the bottom of this document, I attempted to create a client-side physics that would combine with the location and rotation updates that were inherently sent via the DumbProxy role, but the client would not run my Tick()-based physics calculations, and so that idea had to be thrown out the window. :)

**ROLE_AutonomousProxy**

This is used for things that the client directly controls. In most cases, this will only be used by the playerpawn. This RemoteRole value should only be set for the one particular player, and it should be a SimulatedProxy for every other client in the game. The player for which it is AutonomousProxy is the player possessing that particular PlayerPawn. When it is set to AutonomousProxy, it appears to be a SimulatedProxy for everyone BUT the owner of the proxy or the instigator. This ensures that your PlayerPawn looks like a SimulatedProxy to everyone but yourself. Note that this technique is also used with the guided redeemer, to ensure that it works the same way. Having the server tell you where your guided redeemer is flying would not be appropriate. Rather, you want to tell the server where your guided redeemer is. While AutonomousProxy doesn't make this just 'work' for you, it does differentiate you from the other people viewing your redeemer, which is what is needed to have it act differently for you, compared to everyone else.

**ROLE_SimulatedProxy**

There are two different things that can happen with this particular type. It depends on whether the actor with this role is a pawn or not a pawn. I'll discuss the latter first.

**Non-Pawn ROLE_SimulatedProxy**

This one is basically the opposite of DumbProxy. It expects full client-side prediction of the actor, given initial values. It's like extrapolating out a curve given some data, where the current Physics is the curve fit, and you have initial data available to extrapolate from. ROLE_SimulatedProxy gives you an initial Location, Rotation, and Physics. It then keeps you updated with the Velocities, should they change. These variables should enable the client to perform full client-side prediction on this object, assuming it continues along the physics prediction. In the case of PHYS_Falling and PHYS_Projectile, this is the case. There can be no aberration form these behaviors, (unless you play around with the Physics variables bBounce and such, which are very useful in prediction.) Perhaps I'll explain Physics in more detail in another tutorial. Perhaps. ;) Finally, for this role, you also get animation updates, assuming the client is not the owner and it's not set for client-side animation. In summary, this Role is for actors that smoothly predict their data on the client.

**Pawn ROLE_SimulatedProxy**

When ROLE_SimulatedProxy is combined with Pawns, a different sort of simulated proxy is created, that act differently in netplay. Pawns are one of the most complex actors in terms of replication, because of their need for clientside prediction, yet inherent unpredictability in their movement. There is no way to duplicate this behavior except by subclassing pawn, so don't expect to achieve it in your projectile subclass. A SimulatedPawn gets the best of dumbproxy and simulatedproxy. It gets velocity updates for the clientside prediction, while at the same time getting location updates from the server to 'readjust' the player's locations. You might then wonder why the pawn does not jump when a new location update comes in. Again, there is native code that causes the actor to smoothly head towards the location given by the server if the location position is too far off from the player's real position. This again ensures seamless movement of player's while keeping them accurate to the server's version. All non-players will get by with velocity-only client-side prediction. No physics will be taken into account for these pawns. This is because some non-players can be PHYS_spider, or a variety of other physics that don't work with the following always-walking/falling prediction. Only players (bots and playerpawns) get the special client-side prediction to have falling/walking physics. This physics is an assumed physics, which is accomplished by applying the zone's gravity if the player is not on the ground. This logic will be bypassed if the pawn's bCanFly variable is set, which frees it of the clientside physics, but must be accounted for in your code, since there is no physics replication in pawns. This logic is also bypassed if the pawn is in a water zone, which also differs from the default walking/falling physics. In water zones, actors are predicted using velocity alone, which is sufficient with the low amount of gravity that exists in those zones. The actor's get their locations 'adjusted' anyway, so it doesn't make any real difference in the long run. Finally, pawns get rotation updates. Since rotation (which is directly based off the ViewRotation, or the direction the player is looking) can change quite radically, no prediction is done with it. Instead, the player's rotation is adjusted as new player rotation information comes in from the client. Animation is handled the same as any other non-pawn simulated proxy.

## Function Replication, and Simulation

In a network game, functions can be called on the server, on the client, or on both. Sometimes you will want one situation, and sometimes you will want another. Function replication and simulation both achieve different pieces of these goals. Despite what may at first seem like two different topics of discussion altogether, they are actually quite related at their root. I will discuss them together, and then finish with a clarification of their differences so that each can be understood clearly.

Internally, the Unreal Engine uses a very simple system for its function stuff. It calls a function to determine if it can call an UnrealScript function on that machine. if it can, it does, if it cannot, then it does not. That's all that simulation and replication entails, and the only

additions to this system are the origin of UnrealScript simulation, events, (which will be discussed later), and the function replication itself (much of which you know already.)

## Rep/Sim Function Rules

I will first present the basic simple rules to determine if a client or server is able to execute the function locally. The following conditions will determine whether that happens. Again, if a condition matches, then the checking stops there, and it executes locally depending upon the resulting clause of that condition.

1. If it is a static function, it can be called on that machine.
2. If we're on a standalone server, it can be called on that machine.
3. If it's not a replicated function, then go to step 10.

(The rest apply ONLY to replicated functions, per condition three.)

4. If we are the server, and there is no parent PlayerPawn to this actor, (at any point in the parent hierarchy,) or if the player does not have a connection (listen servers for the player on the server,) then it is executed locally.
5. If the replication condition is not met for the current machine, (meaning that it was replicated to us, or that it is not a valid replication statement), then go to step 10.

(The rest apply ONLY to replicated functions where the replication condition is met, and it is a valid function to replicate.)

6. If it is an unreliable function, and the connection (server-client or client-server) is saturated, then 'pretend' it was executed remotely, (eg: do not execute it locally.)
7. Replicate the function, and then do not execute it locally.

This last step is only used if it is called upon by the above steps.

a. If we are on the server, then it can be called.
b. If we are on the client as an AutonomousProxy, it can be called locally.
c. If we are on the client as a DumbProxy or SimulatedProxy (NOT an AutonomousProxy,) and we are a simulated function, then it can be called locally.

While this may seem complex, it accurately describes in every detail how Unreal does it's function handling, and RPC (remote procedure call) nature. Hopefully, in the next paragraph or two, I'll be able to simplify it into a few simple rules.

## Rep/Sim Function Rules Restatement

In order for a function to be replicated to the other end of the connection, it must be all of the following:

1. Owned by a network PlayerPawn at some point it it's parent hierarchy.
2. It must be a reliable function, or an unreliable function with available bandwidth.

In order for that replicated function to actually be executed on the other end of the machine, it must pass rule number 10. Additionally, if we are on the client, it must meet any of the following rules.

1. A static function.
2. A simulated function.
3. In an actor with an AutonomousProxy Role (a local PlayerPawn or local GuidedWarhead on the client.)

The above rules need to be passed for ANY function that is executed on the client, whether it is through simulation or replication.

## Rep/Sim Function Summary

With those rules in mind, I'll try to summarize the whole situation below.

On a server, every function is executed, unless it is a (client-server) replicated function and meets the checks above. So you can be assured of all your functions running on the server, unless you specifically tell them not to.

On the client, every function has to originate from a 'source.' A source can be a server-client replicated function, an exec function, or an event. Events, while not covered previously, are quite simple. An event is any function that is called from native code. Events happen both on the server and on the client. The server does not directly tell the client to execute the event, but the client knows to execute the event based upon the game state (colliding actors, hitting a wall, a timer event, ticking of the game state, etc). If that event is simulated, then it can also serve as a 'source' of clientside function calls. A source can call any function it wishes, assuming it meets the client-side execution conditions, which can then call any...and so on.

What happens if a function is called, but doesn't meet the conditions for execution, either because it was replicated to the other end, or because it does not meet the conditions for running client-side. In that case, it is as if the function never ran in the first place. The return value of such a function is 0, "", or None, depending upon the context. Any out parameters defined by the function are passed back exactly as they were entered. Relying on the result of a function that is never executed is the source of many accessed nones that appear only on the client. Exec functions (functions that can be bound to keystrokes) naturally evaluate on the client. Again, remember

to make it simulated, (unless you are in an AutonomousProxy, like a PlayerPawn subclass.) In some cases, you may want to have it replicated to the server, as in the case of a Server-Administrator command (there are plenty of these defined in PlayerPawn.) Exec functions can be replicated to the server, the same as any other function can.

## Spawning Within Simulated Functions

There are other interesting hidden behaviors that may not be immediately apparent. For example, because of the requirements that an actor be owned by a playerpawn at some point in its parent hierarchy, a function can only be replicated to a single player in the game. (No, making a player owned by another player will not work. ;) Multicast function replication does not exist. You can try to get this behavior using simulated functions, but no data can be passed between the server and the client, (since they both originate on their local machine, and exist only on that local machine.) Or you can use replicated variables, which can and are replicated to all players in the game, assuming they are relevant to them.

If a spawn is performed in a simulated function (simulated functions are explained later), then multiple copies of the actor are spawned: the server, and each of the clients that the simulated function was run on. The server's version is normally then replicated to the client, so the client would see two versions. This would create strange behavior as the two became out of sync, and the client-spawned one could easily be mistaken for the 'official one' that for some reason, isn't having it's variables replicated.

If you need to do a spawn in simulated code (there are many valid reasons, like spawning effects clientside, etc), then you must account for this. If you are attempting to make clientside effects, then make sure you set the RemoteRole to ROLE_None immediately afterwards. Since relevancy is only checked between ticks, this will ensure that when that check does come around, it will not be relevant, and thus not replicated. That way, the server and the client will each have their local copy of the effects that run their course and then destroy themself. This is not useful for permanent effects, since simulated functions are not run on ALL clients, only clients where the actor containing the simulated function is relevant. That means that if the simulated function didn't run (because it's actor was not relevant at the time), then the effect will not be visible when you come into view of that effect. That is why it should only be used for temporary effects that run their course and then destroy themselves.

If however, you are spawning something that is intended to be permanent, or needs data replicated from the server, (like a crosshair dot,) then you SHOULD NOT spawn it on the client. Usually, you can accomplish this with a 'If (Level.NetMode = NM_Client)' check before you perform the spawn. This way, the spawned actor will then become relevant (assuming the conditions match), and it's variables will be replicated to the clients. If a client-side actor had been spawned, then there would be two actors, one of which would just sit there without information from the server

## Networked-Function Paramater Optimizations

And finally, when replicating functions, Unreal optimizes the parameters of a function call through a few techniques. When sending a vector, (which is composed of three floats,) the individual components are rounded to integers, and thus lose their fractional values, and any values outside of -32768 to 32767. A common technique is to multiply the values before they are sent to make them scaled integers, and then dividing them on the other end back into their appropriate values, so it is almost completely accurate. With rotators, the values are scaled between 0 and 255, where the 128+ range corresponds to the normally negative range of rotator component values. Then they are rounded to ints, and sent over the network. They are then fixed back up again to their original values (except they are left as positive values, since they are equivalent to their negative counterpart that existed on the other end of the connection. One approach that applies to both vectors and rotators is to send the X/Y/Z or Pitch/Yaw/Roll values of the struct as individual floats so that complete accuracy can be maintained. And finally, for completeness, planes (which are not used much at all,) replicate their components as rounded integers.

## Simulated States

Another, many times overlooked feature is that of simulated states. You probably have already seen state code, code that exists outside of functions, but in states, usually prefixed with labels. This code, when in a regular state, is executed on the server alone. But by prefixing the `state statename` with `simulated`, you can make the state code itself simulated. So the revised state declaration would be `simulated state statename`. Just beware that state changes are NOT replicated or sent to the client by default. However, if a gotostate is executed in a function that is run on the client (either through replication or simulation), then the client will know of the new state. Playerpawn's inherent simulation allows all it's functions to be able to control the state, but other classes would need to have simulated functions that are executed from a simulated source and chain of function calls in order to know of the state change. The current UT code makes no use of simulated states, but I thought I would include this section for the sake of completeness, in case you ever needed it.

# Level NetModes and their Implications

In the Level variable, which references a LevelInfo actor and is accessible from every object, is a NetMode variable. The NetMode variable is used to describe what role or position the current instance of UT plays in the overall scheme. The possible values, and their uses are:

**NM_Standalone**
: This is for when the game is played alone, like botmatches, the single player tournament ladder, etc

**NM_DedicatedServer**
: This is for when the server is running alone, with no one playing directly in that instance of UT. This is the netmode for the majority of servers that are online. It can be started with 'ucc server', or by clicking 'Dedicated' when you select 'Start New Multiplayer Game.'

**NM_ListenServer**

When you click 'Start' in the 'Start New Multiplayer Game' window, you are starting a listen server, which is when you are playing and hosting at the same time. This causes CPU to be spent on rendering the world for you, as well as hosting the game for any connecting players. Thus, dedicated servers are preferred to listen servers.

**NM_Client**

When you are connected to a running UT server, no matter what type of server it is, you are this netmode.

These netmodes are usually not important as Roles and RemoteRoles can handle everything. However, when you want to execute a block of code on the client, but don't want to replicate it, these netmodes can help. For example, code is run on the client through a chain of simulation. If the chain of simulation is upheld, then you can do a 'if (Level.Netmode == NM_Client) {', then that code will be executed on the client only, or on many clients, if that event exists in many places. By combining that with a check on Owner, it is possible to ensure that it is only run on one single client. One thing that makes the whole deal more complex is that you want to ensure your code will run in standalone games, and for the guy running and playing in listen servers. For this reason, you will often see code that does 'if (Level.NetMode = NM_DedicatedServer) {', which probably does exactly what you want. Other examples include only spawning non-relevant gibs if you aren't on a dedicated server (because of the increase in CPU time). Another example is the use of spawning a gibbed head. The gibbed head needs to travel in the same, but random, direction as the server. Giving it a random velocity on the client would cause it to readjust midflight when the network-replicated data came through the pipe. Instead, the carcass head is not spawned clientside, but rather it lets the network data create the head, and give it the server's random velocity, so that it will be correctly in sync. Further uses are left to your imagination. :)

# Special Properties about Special Stuff

## bNetOptional Usage

You've seen bNetOptional talked about a few times. Now's a good a time as any disucss what it was intended for, and when you should use it. As was discussed earlier, any actors that have bNetOptional set get put at the bottom of the list of actors. The effect of this is that they get starved of bandwidth when the connection becomes full. If there's bandwidth available however, then they'll be replicated. They have 150 ms to be replicated, before they are forgotten about completely, and the client never sees them. When should you use these? In most cases, you'll want to use them for temporary effects that can be replicated to the client, and then allowed to complete on their own. For example, a little puff of smoke coming from a wall after you shoot it would be a good example. If your bandwidth is overloaded, either from a big firefight, or from excessive lag, then the little puff of smoke will not be seen. You probably won't notice it in either case. bNetOptional actors will not be replicated after their initial replication, either. They're conditions are evaluated only once, the first time they are executed. They are expected to complete their actions without any interaction of the server. This also saves a lot on bandwith. It wouldn't make sense to have the server continually evaluate replication checks for these actors, since that would only waste bandwidth. bNetOptional is not for everything though. An explosion sprite, like a rocket's animated explosion, is necessary for everyone in view. It's easy to miss a sprite puff of smoke in the heat of battle, but a large explosion is not something easily missed. You'll want to use the bNetOptional only for small, non-vital actors that aren't necessary. In Unreal Tournament, they are used for decals, fragments (those little chunks of 'wall' that fly off), shell casings, smoke puffs, the wall hits (the little yellow 'spark' when you hit the wall), and the respawn effects. Note that if the default lifespan is set to 0, then a bNetOptional actor will be treated just like any other actor. If the bNetOptional actor does have a default lifespan set, and it extends beyond the first 150 ms of it's lifetime, then it will never be replicated. Any request to destroy an actor will be replicated to the client regardless of at what point it exists in it's lifespan, as UT will always make sure the client destroys what the server destroys.

## bNetTemporary Usage

While we're at it, we might as well cover bNetTemporary here. bNetTemporary actors are for 'tear-off replication', as someone once put it. They are replicated when they are first spawned, and that is it. There is never any further connection between the server and client in regards to that bNetTemporary actor. A bNetTemporary actor is not necessarily temporary, just that the replication of it's variables is temporary, (only once.) This is used in many places. In UT, all decals and projectiles are bNetTemporary by default. A decal is replicated to every client that it is relevant to, and then the connection is broken. this saves significantly on bandwidth. This means that if you walk into view of a scene where there was a battle that you did not witness, then you will not see any of the decals in that room. They were temporarily replicated, and so you will never see them if you weren't present for it's one-time replication. In Half-Life, decals were not 'bNetTemporary', and so were sent to every client when they came into view. This contributed to part of the lag that many experienced because of the many decals and spraypaint decals that were spawned in a game. bNetTemporary actors need to be destroyed in clientside simulated code, since a server destruction of the actor **will not** replicate to the client, because of it's bNetTemporary status. This is done when decals grow stale and need to delete themselves based upon client information, since the server cannot tell the client about that. Most projectiles utilize bNetTemporary. Take a razorblade, for example. It gives the client the initial velocity, and the client can work everything out from there. Since the razorblade is entirely predictable and deterministic, it can be predicted on the client with 100% accuracy. The wall it bounces off on the server.....it also bounces off the equivalent wall on the client. Because it is entirely predictable, the server only needs to send it once, and the client can figure it out. For that reason, projectiles are bNetTemporary. Problems can happen when you enter a big room that has razorblades flying around. Because the razorblades are bNetTemporary, you will not know about them. So you can walk into what seems like a perfectly empty room, and then get your head chopped off with a razorblade. However, you'll since most people do not use 360 FOV, you'll probably attribute it to just not seeing it coming. So no one ever notices that. Other projectiles like the the guided redeemer missles, or even regular redeemer missles (which can be shot down midflight) cannot be bNetTemporary because the server needs to send status updates to the client. The green globs from the goop gun need to be visible on the floor when you run into a room, (because they are sorta-long-lasting land mines,) and so they too cannot be bNetTemporary. There are other examples of where projectiles are not bNetTemporary, but I'm sure you get the idea.

## PlaySound's and PlayOwnedSound's Special Behaviors

This one issue was brought to my attention when someone was trying to figure out how come a sound wasn't playing correctly in his mod,

even though it worked fine in UT. After looking into it, I couldn't see how the sound was working in UT. The replication and simulation behavior was not set up for it. Turns out that it had hidden behavior, depending upon where and how it was called. In theory, PlaySound is used to have an actor play a sound. The game will determine how loud to play it, depending upon occlusion and distance, etc. UT tries to determine the 'right' and intelligent way to propogate the sound, depending upon how it was called. First, we'll cover PlaySound. It can be called in two contexts:

From a simulated function, or from a replicated-to-client function

In this scenario, the sound is being played in many places. The simulated nature means that any player that can see the actor, is getting the simulation events. So each client will play the sound on it's own, due to the simulation. In the case of a replicated function, (where it's being run on the client,) UT assumes you know what you're doing, and that you want the sound to be played on that client only. You may wonder why there is a check for a replicated-to-client function when there's already a simulated function check, (since all replicated-to-client functions need to be simulated,) but if you remember, a PlayerPawn that is operating under ROLE_AutonomousProxy can execute replicated functions that do not have the simulated prefix. This case covers those. In general, this play-locally behavior is what you want, when you call PlaySound from within a simulated function.

From a non-simulated, non-replicated-to-client function

When you call PlaySound in this context, UT does more work for you. In this scenario, the PlaySound is being called server-side. That does absolutely no good in a real game, since that sound would then not be heard by anyone. You might think it only has to be replicated, but if you'll remember, function replication is not multicast, so the sound would only be heard by the owner of the actor PlaySound is being called upon. Simulation doesn't help, since that would only happen if the sound was being called on the client. Simulation does not actually send anything over the network. So, UT performs some special magic instead. In Pawn, there is a ClientHearSound, whose definition is:

```
native simulated event ClientHearSound (
        actor Actor,
        int Id,
        sound S,
        vector SoundLocation,
        vector Parameters
);
```

This ClientHearSound is replicated from the server to the client, per the replication definition in Pawn. Remembering that it only goes to it's owner, this works fine, since calling Pawn.ClientHearSound(someactor...) on the server will cause that one pawn to hear the sound on the client, as if the sound came from someactor's direction. The special magic performed, is that UT effectively turns the call to PlaySound into a multicast function replication on ClientHearSound. When PlaySound is called, it iterates through all nearby players that should hear the sound, and replicates a call to ClientHearSound to each player. In this fashion, each player that should hear it, does hear it. And the original identity of the actor playing the sound is maintained through the first parameter to ClientHearSound.

This is actually quite the nifty behavior for PlaySound, as it causes the function to work the way it 'should,' in most cases. In the cases where it doesn't, you'll need to figure in it's dual nature, and figure out how to call it, (or call some other home-grown sound replication feature,) to make it work.

The other interesting behavior to talk about is that of PlayOwnedSound. The purpose of this function is to play the sound for everyone but the owner. This may seem strange at first, but there are a few uses for it. In UT, it used in most of the weapons. When a client fires a gun, they should get immediate feedback about the shot. That is why you see the weapon animations plya clientside, and hear the sounds clientside. However, the client cannot force sounds to be played elsewhere. What is done, is that the weapon plays a sound clientside immediately when the shot is fired. The client then tells the server it fired the gun, which then calls PlayOwnedSound with the sound, so that the sound can be heard by all the other clients. You wouldn't want the sound to be played on the player firing the gun, since they already played the sound. Hopefully that example is clear, but you can check the examples provided in UT's weapons if it does not satisfy you. One final note: when UT determines who is the owner, it only checks the actor itself (to see if it is the pawn or the playerpawn), and the immediate owner of the actor. It does not follow the hierarchy up like some other functions do.

## ClientPlaySound and ClientReliablePlaySound

Here's a simple example, but still noteworthy nonetheless. ClientPlaySound is an unreliable function that plays a sound on the client. Calling PlayerPawn.ClientPlaySound(...) causes that player to play the sound locally on their machine. It is a location-less sound, and the sound is played as if it came from within their head, (as opposed to next to them, or in front of them, etc.) When called on the client, it is guaranteed to run, since no replication is involved, (it's a simulated function being called from a simulated function...) When it is called from the server, it is not guaranteed to reach the other end, however. What happens if you want to ensure that the sound is played? You call ClientReliablePlaySound, of course. This is a reliable function, which ensures that the sound reaches the client and is played. I'll just rip the whole ClientReliablePlaySound function, so you can see what is going on:

```
simulated function ClientReliablePlaySound(sound ASound, optional bool bInterrupt, optional bool bVolumeControl )
{
        ClientPlaySound(ASound, bInterrupt, bVolumeControl);
}
```

You can see that the reliable definition which is farther above in PlayerPawn ensures that ClientReliablePlaySound is run. Once that happens, the body of ClientReliablePlaySound is run on the client. There, it calls ClientPlaySound which ensures that the sound is played.

## When Things Go Awry

What should you do when things go awry? There's two main techniques that I use, when trying to figure out what's going on. Technically, everything can be figured out with the first technique alone, but the second helps a lot in figuring out the applying technique.

## Figure Out What's Going On

You need to isolate your problem. Know exactly what's happening. What stuff is being replicated, and what isn't. Then realize that those are only your assumptions, and that no matter what, you cannot be positive about them. With all the native stuff happening with replication, it's so easy to blame your problems on what's going on under the hood. In fact, after readinng this guide, you should be able to solve your problems yourself. I think I've covered just about every native replication aspect there is in this document. It's why it's so big. When I was having replication troubles, I remember that I wished there was documentation. I'd (and I'm sure others) would rather have too much than too little. Anyways, now that I've had access to the source, and have been able to know exactly what's been going on, I've been able to figure out why problems are happening in netplay, even ones that aren't my own. I say this only to point out that just knowing and understanding what's going on under the hood can make any problem solvable. You don't need extensivle trial and error to figure out where things are screwing up. Just thinking the problem through, and checking the unrealscript code, can provide you with the answer.
First, figure out what you believe is not replicating. Is it a function? A variable? What's going on exactly? Sometimes, because of intricate relationships in your code, you'll be unable to tell exactly what is 'missing' on the client. In that case, you'll need to use logging to figure out what's there and what's not. Logging is the second technique I mentioned. I'll discuss more on how to use it properly in debugging networking problems in a bit.

## Check the Replication Statements

Anyways, once you figure out what's not being replicated, check the replication statement. 98% of the time, you can figure out why something is not replicating, by evaluating the replication statement yourself. I went through a great many replication statements in actor up above in the variable replication section. You might want to check there for some of the more common ones. But thinking about each part of the statement in turn will help you figure it out. Many times, stuff has a bNetInitial clause, which only happens the first time the actor is sent over the network when becoming relevant. This causes problems when your data isn't getting replicated midstream, or is at odd points, like when it goes out of and then comes back into relevancy. You can follow the function replication and simulation guidelines listed above to see if the function meets the criteria for being sent accross the network, or evaluated client-side.
Another programmer I've talked to had problems with his velocity not replicating when the grenades were thrown, but it was because they were only replicated on bNetInitial, and so that's why he never saw his velocity updates. Fiddling with the variables, he found that a RemoteRole of ROLE_DumbProxy seemed to work, as the grenades did seem to be moving. It may be hard to tell that's what is going on when you're on a LAN or even connecting to your own server, because of the small ping, and high bandwidth. But you can be sure that you're players playing over their modems will notice it. :) Remember the various properties of ROLE_DumbProxy, and ROLE_SimulatedProxy, and what side effects each causes, in terms of what is replicated. I've seen a few cases where that is the reason for their strange behavior. The problem with his approach was that he was getting the spoon-fed location updates from the server, and so in a real environment online, that would never work, since players cannot get location updates over the net and expect it to work right. In that case, the grenades should never be thrown from the inventory. ROLE_SimulatedProxy was a perfect fit for what he was trying to do, except for the fact that when he threw it, it was not bNetInitial. Instead, he just had to spawn a new actor, and give it the 'thrown' properties. That satisfied the criteria needed for the variable replication for ROLE_SimulatedProxy, and allowed the grenades to be thrown correctly, be simulated correctly, and it to all just work.

## Logging is Your Friend

If that doesn't help you, try sticking logging statements in there. When you have two copies of UT running, a server and a client, two different logs are generated. One is a server log, and one is a client log. (doh :) If you aren't sure if a function is executing clientside, place a log in it. Then, check to see if that log appears in the clientside log. If it does, then the function is running on the client, and your strange behavior must be coming from elsewhere. If there is no log happening, then the function is not being executed. You may think it must be, because it is the only explanation for behavior you are seeing. But the logs do not lie. What is more often the case is that the function is being run on the server, and modifying variables, which are being replicated to the client, giving the appearance that the function ran on the client. As i talked about before, it can be easy to overlook this in LAN settings, but it will surely be found in real world conditions. I'll discuss how to test things in 'real-world' conditions in just a bit. Logging is your friend, and you should become quite familiar with it. Most Unreal Tournament programmers are already familiar with it, and if they've gotten bogged down in replication problems already, they're just about gunaranteed to have logged stuff in trying to figure things out. However, combine logging with a knowledge of what's actually going on, and you increase your chances of success manifold. I'll be covering a few 'case studies' of sorts at the end of the document. You'll be able to see exactly what's going on as each part is done, and hopefully the real-world examples of replication will make things clearer.

# Testing Real-World Conditions

## Testing a Mod in Netplay without a Network

It's always important to test your mod in netplay. Sometimes, it can be hard, when you don't have a bunch of people available to test with. However, there is nothing preventing you from running a server and a client on the same machine. This allows you to test netplay on a single machine.

### Windows 9x

On Windows 9x, the UT client will consume 100% of the CPU by default. This is so that extraneous programs like ICQ, mirc, or whatever else you might have open do not steal CPU resources from your UT client, which should get full priority. This means that if you run a client and server on the same machine, the server will be starved of CPU cycles, and your UT client will experience **extreme** lag and packet loss. While you may want to simulate these at some point, this is not the way to go about doing it. Luckily, someone wrote a simple

program that helps solve this dilemma of testing a UT server and client together. It was originally intended to solve this same problem for QuakeWorld servers, but it is perfectly applicable to our situation.
Let's disucss how to use it. First, download priority and extract it somewhere safe. I have mine in my UnrealTournament\System directory. Next start your server. Your best bet is to do this with the 'ucc server' commandlet. The general format for testing your mods should be:

```
ucc server mapname.unr?game=packagename.classname?packagename1.classname1,packagename2.classname2
```

Some example commandlines follow:

- ucc server DM-Morbias][
- ucc server DM-Morpheus?game=RocketArena.RocketArenaGame
- ucc server CTF-Face?mutators=Botpack.InstaGibDM,Botpack.LowGrav

Next, you have to tell priority to give more priority to our UT server. You can read the readme associated with priority if you're interested in exactly what commandline parameters do what. But for our UT client/server testing purposes, just type the following in another window or command prompt:

```
ucc -title ucc
```

Now, our UT server will have the necessary CPU time it needs to run, even if a UT client is running. Now run the UT client, connecting to your local machine, with:

```
unrealtournament 127.0.0.1
```

Test your mod, and make sure it all works as you have planned. Of course, some things will require the use of more than one person to test, but at least you should be able to test and fix a large portion of your problems yourself, without requiring someone's valuable time, or requiring you to waste your own organizing a beta team and stuff.

## Windows NT/2000

On Windows NT/2000, this is not a problem, since NT supports true multi-tasking, and so the use of priority.exe is not an issue. ////////////////////////////

## Simulating Lag and PacketLoss

Testing multiplayer as described above only solves one problem. It helps you see if your code will execute in multiplayer correctly. However, the successful completion of that test does not mean that your mod will work in netplay. When testing netplay on your machine, your computer connects directly to itself, accross an incredibly fast and large connection. Testing on a LAN has the same 'problem', in that you will not be simulating the effect of people playing over modems on servers halfway around the world. Luckily, UT includes commandline options to help you simulate this type of behavior. They are: PktLoss, PktOrder, PktLag, and PktDup. These variables only affect data as it is being sent, so you will need to place them on the server and client to accomplish true lag. But first, if you check your UnrealTournament.ini, you'll notice references to SimPacketLoss and SimLatency. These worked once, in a land far, far away in a time long, long ago. They do absolutely nothing now, so you might as well remove them from your ini.
The true way to simulate lag is to use the commandline parameters when you start UT. We'll cover and document all of these variables here. To use these variables, you need to start UT like this: `unrealtournament.exe server.ip.addr?PktLoss=4?PktLag=400`. This will cause your UT client to experience this amount of lag when sending data to the server. Data sent from the server to the client till still travel at full speed, however. By the same token, you can also pass these variables to ucc with `ucc server mapname.unr?PktLoss=10?PktLag=600`. This will create a very 'bad' server with terrible conditions...probably too extreme even for real-life conditions. (If you play under 600 ping and 10% packetloss, then I feel bad for you. :) Let's get into the documentation of the individual variables. PktLoss will work all the time. The remaining three, PktOrder, PktLag, and PktDup are mutually exclusive, and if multiple options are passed on the commandline, the order just stated is the order they are checked. For example, if you pass PktLoss, PktLag, and PktOrder, then only PktOrder and PktLoss will be used. PktOrder has a higher priority than PktLag, and PktLoss always works.

### PktLoss

The first, PktLoss. is set a percentage of packets sent. It determines the percentage of packets lost over the connection. Passing 10 will mean that 10% of the packets sent to the client will be lost to the bandwidth graveyard. Any packetloss usually will hurt a connection in real world connection, as many people complain about anything over 0% packetloss. So when testing, this value doesn't need to be that high, and probably should only range between 0 and 10. PktLoss set with: `PktLoss=5`

### PktOrder

The next, PktOrder, determines whether packets will be sent out of order. This is simply a boolean flag, meaning that the only option for it is on/off, not a variable as in PktLoss. If this flag is set to on, it preclues the use of PktLag and PktDup, described below. PktOrder works like this: Every time a packet is going to be sent, it gets put at the end of a 'to-send' list. Then UT goes through the 'to-send' list, and sends half of the variables over the connection. While this is not truly random, it doesn't matter. It succeeds in it's goal in sending packets out of order, since they can be kept on the list for an indefinite length of time. PktOrder set with: `PktOrder=1`

### PktLag

If you want to delay packets and simulate lag, then PktLag is the variable you want. Precluding the use of PktDup, and only working if PktOrder is not set, PktLag is set as a value of milliseconds. All packets being sent over the connection will be delayed by PktLag milliseconds. PktLag set with: `PktLag=450`

### PktDup

If you want to simulate the effects of sending packets twice over the network, PktDup fits the bill. Only working if both PktOrder and PktLag are not set, PktDup is set as a percentage of packets that are duplicated. Every packet that is sent has a PktDup chance of being sent twice. All packets will be sent once. PktDup set with `PktDup=20`

### Using the Variables

Now that you know how the variables work, here's a little lesson in how to use them. Remember that they only work for packets being sent. So if you set PktLag to 500 on the server, the client will experience 500 ms lag between what the server sent and what the client sees. Client packets sent to the server will go at their regular speed, however, which is usually around 40 ms each way on my machine. To truly test lag, you will need to use PktLag on both the client and the server. Ping, if you remember, is the time it takes for a packet to go round trip from client to server and back to client. To simulate a 500 ping, the best way would be to give a 250 PktLag on both the server and the client. All clients to connect to the server would then experience 250 ms additional lag, regardless of their local settings. The client would then add 250 ms additional lag itself through PktLag, resulting in a 500 ms effective ping.
Of the variables above, you will find yourself using PktLoss and PktLag most often. The others were more intended for Epic to test the results of it's own network code, and test it under extreme conditions that it could not duplicate. I doubt you will have need of PktDup or PktOrder unless you are unable to reproduce the player-reported problems about your mod being unresponsive or ineffective.

## Conclusion

That's it, folks. Hopefully you now understand replication a lot more than you did before. Granted, it all may seem COMPLETELY overwhelming and confusing at first, and it is. But through application and testing of these principles, you will become more familiar with what is going on, and will better grasp network coding as time passes. You may need to read this document through a few times, before it all sinks in. If you find a section that is confusing even after you've read it through, send me an email at mongo@planetunreal.com. Don't expect me to help you out personally however, since I imagine I will probably get countless emails from new people that don't meet the basic assumptions I made and stated at the beginning of the document. It is always inevitable. Of course, those people probably haven't even read this far, so they'll be emailing me anyway, but I'm just letting you faithful readers know. :) I'm planning to read all your emails, and if they have good or valid suggestions for things to cover, or describe ambiguous or confusing aspects of my tutorial, then I'll make corrections and updates to this document in the future.
When something doesn't work, don't assume it's UT's native code in regards to replication. I believe I have successfully delineated all such hidden tricks in this document. The error has to be in your code, however much you don't like hearing that. Trust me, this is from experience. ;) Check your replication statements to make sure their logic is correct, and run through your code as both the server and the client, following the flow of logic to figure out what's going on. Make sure what you're expecting to be replicated or simulated IS ACTUALLY replicated or simulated, by following the checklists I've given in the document above. And as I stated before, LOG LOG LOG LOG. Fill your code chock-full of logs, to make sure that things are happening the way you want. After following all the guidelines, the only true test to make sure things are happening the way you expect is to put logs in, and see what is happening where. With all this in mind, I'm sure you will be successful in your future networking endeavors.
Good luck!
Mongo

---

**Still to cover:**

## Special Networking Tricks (Case Studies)

Let's take a look at a few examples and problems that people have had, and an approach to solving them. Hopefully, this will help you solve your own problem, or at the very least, help you see how others have approached their problems, and fixed them.

---

thoroughly read through above sections on special properties of special stuff, realworld conditions, things go awry serverpackages packageflags native final functions are simulated automatically (?) state changes only seen for playerpawns, unless changed in simulated functions (then they cause problems for newly relevant actors) window stuff on client functions usually replicated before variables, cannot be sure Case Studies:

- HUDMutators (everyone wants them :/ )
- DrSiN's Window Replication Info
- How to get custom pris: RAUT's PRI extra stuff (avoid spawnnotifies)
- Laser-Guided-Ripper Tutorial get a better laserdot
- UT's method for sending data to the server for playerpawns
- How they did the GuidedWarhead in netplay

Weapon Replication/Networking overview (where?)
Other Neat Tricks

- Faking replication of parent-variables that wouldn't normally replicate
- Faking multicast replication with specially-owned objects by each player, that get functions called on them from a general function.
- Use physics to your advantage Use of PHYS_Trailer for Halo, Shieldbelt
- Use of Simulated ticks to accomplish clientside manual physics just a discussion, as complex code: Taliesin's code for inverse IK (hey, it's cool, too!)
- Other ways of doing non-standard variable replication with functions, like Taliesin's code for seekers