**NetcodeUnitTest**

• Commandline parameters

• UI Elements

• Unit Test Classes

• Unit Test Functions

• Unit Test Variables

• Unit Test Flags

• Unit Test Result Types

**Pending Documentation:**

* Documentation of **NUTActor** helper
* Documentation of various debugger helper classes
* Documentation for UE4 UScript/BP VM reflection (to be done after new helper class)
* Expanded reference section
* Full documentation of the **ClientUnitTest** API (probably reorganizing the core API into a 'ClientUnitTestBase' class), and how to implement/use each part of it
* Log output colour legend
* Recommended steps for debugging a broken unit test
* Console command reference

**Introduction**

The NetcodeUnitTest package introduces a unit testing framework aimed at testing game client/server netcode, using live instances of servers and minimal client test code (run within unit tests, which create 'fake'/minimal clients internally).

**Note:**

The user-facing/UI portion of this tool is aimed at developers writing/debugging unit tests and for QA teams testing/tracking issues, but the coding framework for unit tests is aimed at advanced developers, who are very comfortable with extensive debugging/testing of (often undocumented) code, without external support.

Unit tests may break as game codebases evolve, base UE4 updates modify the engine, and as issues are fixed - sometimes these changes will create hard to track issues, which may require deep-diving into the UE4 net/engine code.

**Getting started**

Steps for running the unit test tool:

* Compile the unit test plugins for the current game
* Open UnrealEd for the current game, and enable the unit test plugins:
* Create a shortcut for the game, with the commandline:
* UE4Editor.exe ShooterGame **-run=NetcodeUnitTest.UnitTestCommandlet**
* Specify a specific unit test, to avoid running all unit tests: **-UnitTest=FTextCrash**
* **NOTE**: To see a list of all unit tests, type in the console: **UnitTest**
* When run, it displays a '**Status Window**', and one '**Unit Test Window**' for each unit test:
* Unit Test Status: This displays the overall status/progress of all unit tests, and when all unit tests are finished, will print a final summary of the results of all unit tests.
* Unit Test Windows: Each unit test will launch its own window, displaying more detailed progress for each individual unit test.
* When unit tests complete, there are four result types:
* **VerifiedNotFixed**: Unit test has verified that the issue being tested is not fixed.
* **VerifiedFixed**: Unit test verified that the issue has been fixed or isn't reproducible.
* **VerifiedNeedsUpdate**: The unit test is not functioning, and needs to be manually checked and updated by a programmer. This could mean that the issue is fixed but needs verification by a programmer, or could mean that the unit test is broken.
* **VerifiedUnreliable**: The unit test is verified as executing unreliably, with neither a positive or negative result. Some unit tests, e.g. those that depend on race conditions, don't return reliable results.
* **IMPORTANT**: Unit tests require special manual verification by a programmer, before they can be marked as 'VerifiedFixed' - see the 'Verifying fixes' section.
* Internally, unit tests are marked as expecting one of the above four results (through the '**ExpectedResult**' variable). If a unit test is known to be fixed, the programmer wil mark it as expecting **VerifiedFixed**, if it is known to not be fixed, the programmer marks it as expecting **VerifiedNotFixed**.
* If the unit test returns a result different to '**ExpectedResult**', a yellow warning will be shown in the status window:
* Unit tests with the expected result, report the result in red/green with no warning:
* In general, **RED** = Not Fixed, **GREEN** = Fixed, and **YELLOW** = Needs attention.

**Verifying fixes**

All unit tests have an '**ExpectedResult**' value, which is for specifying whether the programmer expects an issue to be fixed or not. Unit tests can return a result that is different to this value, e.g. can return '**VerifiedFixed**' if an issue can no longer be triggered - this does not mean an issue is actually fixed though, and this will trigger a warning requesting programmer attention.

Unit tests require a special/strict process for manually verifying that issues have been fixed, before any unit tests can be marked as expecting **VerifiedFixed** - this process is slightly laborious, but due to the way the unit test tool works, it is the only supported way for verifying that issues are fixed.

The process - from writing a unit test, to updating it to mark an issue as fixed - is as follows:

1. When writing a unit test, the implementation should focus on successfully triggering a bug/issue (it should NOT focus on testing for the absence of an issue, as this is not a reliable way of tracking issues or verifying that they are fixed), and by default unit tests should be marked as expecting '**VerifiedUnfixed**' as its result.
2. Unit tests should be written to detect successful repro of an issue, and then return '**VerifiedUnfixed**' for that, and when repro fails (i.e. can no longer trigger the issue), should return '**VerifiedFixed**' (this does not yet mean the issue is actually fixed though).
3. If a unit test has an '**ExpectedResult**' of '**VerifiedUnfixed**', but returns '**VerifiedFixed**' (i.e. can't repro the issue), then a programmer needs to manually check the UE4/game code where the issue is, and should engage in exhaustive attempts at modifying the unit test, to repro that issue again (if it looks possible; don't assume an issue is fixed, test it).
4. If the programmer is able to manually verify that the issue is fixed, '**ExpectedResult**' for the unit test can then be marked as '**VerifiedFixed**'; the issue is then fixed, but the unit test will continue to run in the future, to watch for regressions and broken unit tests.
5. The programmer should also carefully examine/diff the changes that fixed the issue, and look for any potential new issues caused by the changes.

**Writing Unit Tests**

Currently, the unit test tool is solely focused on writing unit tests which launch a server, and then connect a stripped-down/minimal/'fake' client, so these are the only unit tests that are supported right now.

This is done through subclassing the monolithic **ClientUnitTest** support class, which provides a base for configuring how the minimal/fake client will function, provides many useful notification functions (many of which need enabling using '**UnitTestFlags**'), and tries to do all of the heavy-lifting for aiding the rapid development of unit tests, and minimization of actual unit test code.

The fake/minimal client is aimed at stripping down the netcode, and particularly what is allowed to be loaded/executed clientside, in order to minimize resource usage and avoid execution of clientside RPC's that risk crashing the unit test commandlet process.

Due to this, unit tests should strictly minimize the clientside features that are enabled, and should only allow net channels (particularly actor channels) and clientside RPC's, on a whitelist basis.

Depending upon the issue being tested, some unit tests may require highly advanced knowledge of the engine - especially netcode - in order to implement a proper test.

The unit test tool has extensive helper and debug classes (many of them tied into the monolithic '**ClientUnitTest**' class), aimed at encapsulating many of these more advanced techniques, in easier to use classes.

**NOTE**: It is recommended that existng unit tests, such as in the '**NUTUnrealEngine4**' package, be looked at for examples of how to implement unit test code.

**NOTE**: The **VAXSnippets** folder provides users of Visual Assist X, a few handy code snippets for declaring and implementing unit test classes.

**Plugin Package**

The first step to implementing any unit tests, is to create a plugin package - this can be based on the '**NUTUnrealEngine4**' package - minus the contents of the 'UnitTests' folders.

If you are creating unit tests for your own game, then you also need to create an **FUnitTestEnvironment** class for your game, which will be referenced by your unit tests; base this on the game environments defined in '**UnrealEngine4Environment.h**'.

The unit test plugin package should not depend upon any game packages, and any referencing of game packages that is needed, should be done dynamically through reflection with the UE4 Virtual Machine (e.g. using **FindField** to reference class functions/properties and the like); there are examples of how to do this in existing unit tests, and there is some helper code to aid this as well.

There can be exceptions to this rule (e.g. some things may not be accessible through the VM), but it is highly recommended to keep things this way for maintainability - unit tests are designed to be kept around permanently (long after issues are fixed) for regression testing, and so many of them are expected to break as the game code changes - they should be able to break gracefully, without blocking use of the entire plugin.

**Adding Unit Tests**

Put unit test headers in '**YourPackage\Classes\UnitTests\**' and source files in '**YourPackage\Private\UnitTests\**', and you can use the VAX macro's to fill the header and source file with the template for a unit test, which will look like this:

• **Header**:

// Copyright 1998-2015 Epic Games, Inc. All Rights Reserved.  
   
#pragma once  
   
#include "ClientUnitTest.h"  
   
#include "YourUnitTest.generated.h"  
   
/\*\*  
 \* YourUnitTest  
 \*/  
UCLASS()  
class UYourUnitTest : public UClientUnitTest  
{  
 GENERATED\_UCLASS\_BODY()  
   
public:  
 virtual void InitializeEnvironmentSettings() override;  
   
 virtual void ExecuteClientUnitTest() override;  
   
 virtual void NotifyProcessLog(TWeakPtr<FUnitTestProcess> InProcess, const TArray<FString>& InLogLines) override;  
};

• **Source**:

// Copyright 1998-2015 Epic Games, Inc. All Rights Reserved.  
   
#include "NetcodeUnitTestPCH.h"  
   
#include "UnitTests/YourUnitTest.h"  
   
#include "UnitTestEnvironment.h"  
   
   
/\*\*  
 \* UYourUnitTest  
 \*/  
   
UYourUnitTest::UYourUnitTest(const FObjectInitializer& ObjectInitializer)  
 : Super(ObjectInitializer)  
{  
 UnitTestName = TEXT("YourUnitTest");  
 UnitTestType = TEXT("Test");  
   
 UnitTestDate = FDateTime(201?, ?, ?);  
   
 bWorkInProgress = true;  
   
 // @todo: Add Bugtracking/changelist tracking here  
   
 ExpectedResult.Add(TEXT("YourGame"), EUnitTestVerification::VerifiedNotFixed);  
   
 UnitTestTimeout = 60;  
   
   
 UnitTestFlags |= (EUnitTestFlags::LaunchServer | /\*\* @todo: Add required flags here \*/);  
}  
   
void UYourUnitTest::InitializeEnvironmentSettings()  
{  
 TArray<FString> SupportedGames = GetSupportedGames();  
   
 BaseServerURL = UnitEnv->GetDefaultMap(SupportedGames, UnitTestFlags);  
 BaseServerParameters = UnitEnv->GetDefaultServerParameters(SupportedGames);  
}  
   
void UYourUnitTest::ExecuteClientUnitTest()  
{  
 // @todo: Implement the unit test here  
}  
   
void UYourUnitTest::NotifyProcessLog(TWeakPtr<FUnitTestProcess> InProcess, const TArray<FString>& InLogLines)  
{  
 Super::NotifyProcessLog(InProcess, InLogLines);  
   
 if (InProcess.HasSameObject(ServerHandle.Pin().Get()))  
 {  
 for (auto CurLine : InLogLines)  
 {  
 // @todo: Implement fixed/notfixed detection here  
 }  
 }  
}

**Unit Test Flags**

Once the template for the unit test is setup, you need to configure what is required for the unit test through the '**UnitTestFlags**' value, the settings for which are described in the reference section.

These settings are heavily interdependent, with the '**UClientUnitTest::ValidateUnitTestSettings**' function outlining the dependencies; the unit test will crash on launch if any dependencies aren't met.

For example, if you want to send RPC's using the special **NUTActor** helper class, and are testing a server crash, you need the following flags:

UnitTestFlags |= (EUnitTestFlags::LaunchServer | EUnitTestFlags::AcceptActors | EUnitTestFlags::AcceptPlayerController | EUnitTestFlags::SendRPCs | EUnitTestFlags::NotifyAllowNetActor | EUnitTestFlags::NotifyNetActors | EUnitTestFlags::RequireNUTActor | EUnitTestFlags::ExpectServerCrash);

The flags need to be explicitly specified by the programmer in one place in this way, instead of the unit test tool automatically enabling dependent flags, so that the programmer can always see directly what is/isn't enabled.

**Execution/State Preparation**

Before the actual issue being tested is triggered, the server and local minimal/fake client need to be prepared and put in the right state before it is actually possible to trigger the issue.

In many common cases, such as needing a valid client PlayerController or Pawn, there is an **EUnitTestFlags** value - prefixed with 'Require\*' - which handles this behind the scenes.

For cases where you need to implement your own state preparation, you should use the **RequireCustom** flag, and either trigger **ExecuteClientUnitTest** manually when ready (not recommended), or override the **HasAllCustomRequirements** function to signify when ready (preferred).

**Unit Test Execution**

The '**ExecuteClientUnitTest**' function is where the main unit test code is implemented, which triggers the issue being tested, with most of the support/preparation steps leading up to this, happening prior to this function being called.

This usually takes the form of triggering an RPC on the server, but can cover a wide variety of things - triggering a control channel message as defined in DataChannel.h, manually constructing packets/bunches to test low level netcode, among more.

For sending RPC's, there is the **RPCSend** VAX macro:

struct ServerRPC\_Params  
{  
 // @todo: Specify RPC parameters here  
};  
   
ServerRPC\_Params RPCParams;

// @todo: Assign all RPC parameters (RPCParams isn't memzeroed, so assign all)  
   
SendRPCChecked(TargetObj, TEXT("ServerRPC"), &RPCParams);

This uses the special **SendRPCChecked** helper function, which uses reflection to automatically find and trigger an RPC specified by a string, and also verifies behind the scenes, that the RPC was actually sent (triggering a unit test failure if it was not).

**Success/Failure Detection**

The '**VerficiationState**' variable is used to mark a unit test as complete, and whether or not it has succeeded in reproducing the issue, or has failed, using an **EVerificationFlags** value.

Success/Failure is typically detected using the log output of the server, which passes through the **NotifyProcessLog** function - here is an example from the **FTextCrash** unit test, detecting a server crash:  
Super::NotifyProcessLog(**InProcess**, **InLogLines**);  
  
if (**InProcess**.HasSameObject(ServerHandle.Pin().Get()))  
{  
 const TCHAR\* **AssertLog** = TEXT("Unhandled Exception: EXCEPTION\_ACCESS\_VIOLATION reading address");  
  
 for (auto **CurLine** : **InLogLines**)  
 {  
 if (**CurLine**.Contains(**AssertLog**))  
 {  
 VerificationState = EUnitTestVerification::VerifiedNotFixed;  
 break;  
 }  
 else if (**CurLine**.Contains(ExploitFailLog))  
 {  
 VerificationState = EUnitTestVerification::VerifiedFixed;  
 break;  
 }  
 }  
}

Note that the 'break' is important after assigning **VerificationState**, so that you don't set VerificationState multiple times.

Check the above **Verifying Fixes** section for further important details, on implementing unit test success/failure detection correctly.

**Reference**

**Main Components:**

* Unit Test Commandlet: Central process for kicking off and managing unit tests
* Fake/Simulated Client: Barebones client connection for performing the unit test
* Game Server Instance: A live server instance which the unit test is performed on
* Game Client Instance: Optionally, a full client instance, for unit tests which require it

**UI Components:**

* **Status Window**: Primary window listing the status of all running unit tests.
* **Unit Test Window**: Window for viewing status of and controlling individual unit tests.



1. **Filter Tab**: Applies a filter to log output (2), each filter having its own tab/log-output.
2. **Log Output**: Log output from various sources, coloured depending on source/context.
3. **Search Bar**: Used to search the log output in the current filter tab (Ctrl+F).
4. **Search Button**: Opens the search bar for the current filter tab.
5. **Suspend/Resume**: Suspends the server process - typically, to attach a debugger.
6. **AutoClose**: Toggle whether the unit test window closes upon unit test completion.
7. **AutoScroll**: Toggle whether the log output scrolls to the bottom, on new log entries.
8. **Developer**: Prevents unit tests ending and servers/clients closing, for testing/debugging.
9. **Console Context**: Whether to execute commands locally, on the server, or on the client.
10. **Console**: Executes a console command, in the specified context.

**Commandline**

The base commandline for launching the unit test commandlet, is:

* UE4Editor.exe ShooterGame **-run=NetcodeUnitTest.UnitTestCommandlet**

Parameters:

* **-UnitTest=UnitTestName**
* Launches only the specified unit test (by default, all unit tests are launched)
* **-UnitTestNoAutoClose**
* Sets the default option for the 'AutoClose' button to False
* **-UnitTestServerParms="CommandlineParameters"**
* Adds additional commandline parameters to unit test server instances (useful for e.g. unsuppressing specific logs)
* **-UnitTestClientParms="CommandlineParameters"**
* Adds additional commandline parameters to unit test client instances

**Primary Class List:**

* UnitTestCommandlet
* UnitTestManager
* UnitTestBase
* UnitTest
* ClientUnitTest
* NUTActor

**Settings Class List:**

* FUnitTestEnvironment

**Unit Test Verification Flags**:

* **Unverified**: Unit test is not yet verified (i.e. not yet run, or still running).
* **VerifiedNotFixed**: Issue is verified as not fixed.
* **VerifiedFixed**: Issue is verified as fixed (may not actually be fixed, unless '**ExpectedResult**' matches this value)
* **VerifiedNeedsUpdate**: Unit test is no longer functioning, and needs a manual update.
* **VerifiedUnreliable**: Unit test execution gave an unreliable result

**Unit Test Flags**: (excludes debugging flags)

* **LaunchServer**: Automatically launch a game server for the unit test.
* **LaunchClient**: Automatically launch a full client for the unit test.
* **AcceptActors**: Have the fake/minimal client accept actor channels, which are then allowed on a whitelist-only basis, through '**NotifyAllowNetActor**'.
* **AcceptPlayerController**: Have the fake client accept creation of a local PlayerController.
* **AcceptRPCs**: Have the fake client accept execution of client RPC's (blocked by default).
* **SendRPCs**: Mark the fake client as needing to send server RPC's.
* **SkipControlJoin**: Stop the login process prior to sending NMT\_Join or NMT\_BeaconJoin.
* **BeaconConnect**: Make the fake client connect to the servers beacon (massively limits fake client capabilities).
* **RequirePlayerController**: Delays '**ExecuteClientUnitTest**' until the fake clients PlayerController is set.
* **RequirePawn**: Delays '**ExecuteClientUnitTest**' until the fake client has a pawn.
* **RequirePing**: Sends a ping on join, and delays '**ExecuteClientUnitTest**' until it returns.
* **RequireNUTActor**: Delays '**ExecuteClientUnitTest**' until the server **NUTActor** replicates.
* **RequireBeacon**: Delays '**ExecuteClientUnitTest**' until the server beacon replicates.
* **RequireCustom**: Delays '**ExecuteClientUnitTest**' until custom conditions are met, or until executed manually.
* **ExpectServerCrash**: Mark the unit test as expecting a server crash, to disable automated crash detection, which would mark the unit test as broken/failed.
* **NotifyAllowNetActor**: Enables the '**NotifyAllowNetActor**' event, for actor channel whitelisting.
* **NotifyNetActors**: Enables notification of new actor channels, but only after a valid actor has been replicated and assigned to the channel.