DarkRift

Embedded Server Guide

Introduction

For a lot of projects server side physics and pathfinding, amongst others, are very important to the game and are needed to prevent cheating and hackers. Previously DarkRift had no support for these but with V1.3 DarkRift can be run as a child to another application and therefore can use Unity's physics and navmesh engines as well as providing visual feedback from the server and much more.

Running DarkRift under Unity changes things fairly significantly both internally and externally, hence there's a whole instruction manual dedicated to it. It is recommended that you read the Server Manual first and have some prior experience with DarkRift before reading this.

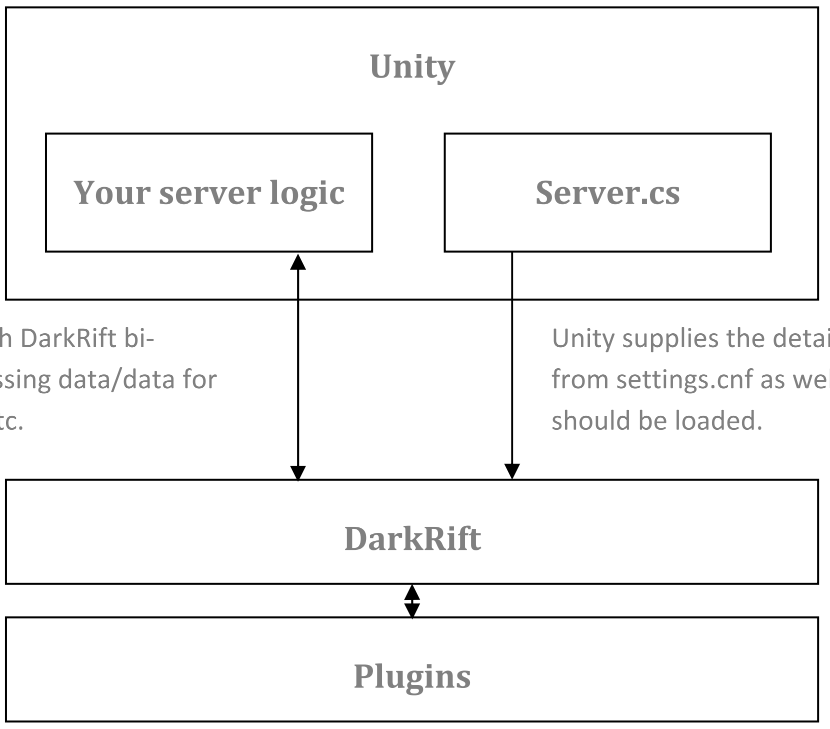
Major Differences

There’s a fairly large incompatibility between DarkRift servers and Unity, here’s a list:

1. **DarkRift is very heavily multithreaded – in Unity everything must be called from the main thread.** If you’re not that familiar with threading just imagine a 10 lane motorway turning (very suddenly) into a single track lane – all the different threads DarkRift uses must suddenly merge their processing into a single thread to execute anything on Unity. There’s code for this but you can still keep everything multithreaded if you want.
2. **DarkRift is designed to work as its own standalone application – Unity is also designed to work as its own standalone application.** DarkRift is now completely rewritten so that it always acts as a child to another process, when you run the console application it’s actually now a simple application that hosts a DarkRift server as Unity does.
3. **DarkRift is designed to have directories to store things in – I had no clue where those directories were going to end up in each different Unity build.** When you put DarkRift in embedded mode it disables most of the modules that require directories e.g. the plugin system, config files etc.
4. **DarkRift already has plugins people might want to use and we just disabled the plugin system.** To get around this so you can still use the login plugins/room plugins and other plugins you may have you can now specify plugins to be loaded, by type, when the server is loaded. So you can still load plugins, just not from a Plugins directory.

Getting Started

To best understand the embedded DarkRift it is best to illustrate it:

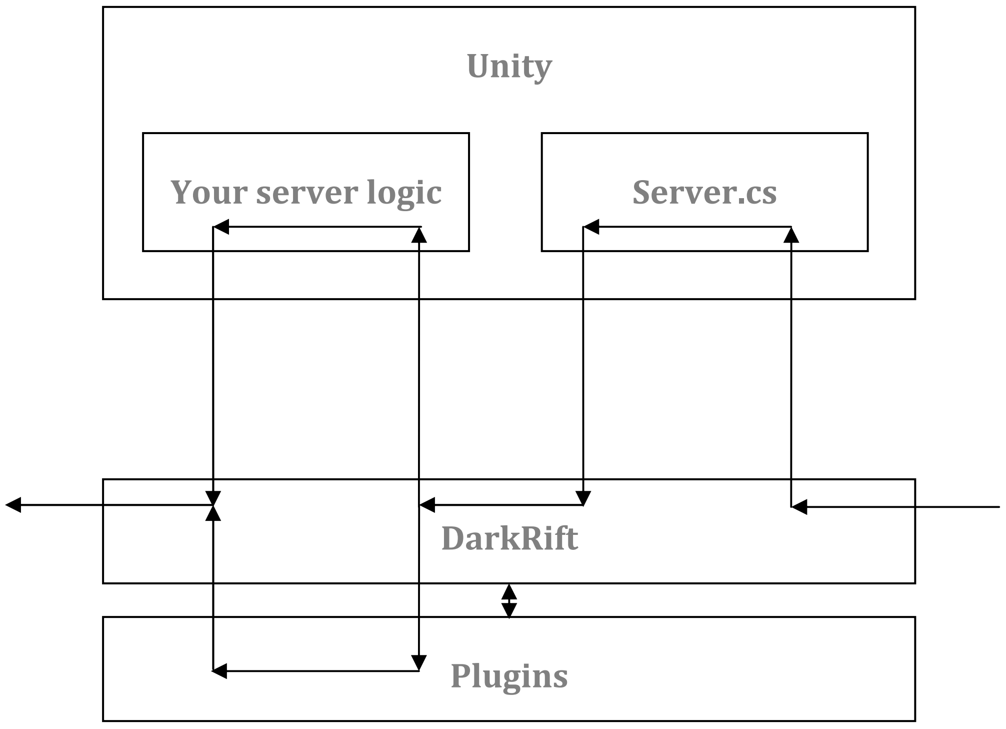


Your logic talks through DarkRift bi-directionally about passing data/data for plugins/control data etc.

Unity supplies the details usually picked up from settings.cnf as well as any plugins that should be loaded.

However that is a large simplification of what actually happens. The diagram above illustrates perfectly how the server would work if we didn’t worry about multithreading but, as we identified earlier, that’s not easy with Unity.

Instead the following diagram illustrates where a packet of data actually goes in the server to overcome the multithreading difficulties.



Data is received by the server and sent to Server.cs to wait. Once an Update/FixedUpdate/LateUpdate (you choose) occurs data is sent to DarkRift to be processed. DarkRift then executes as normal distributing the data to the events subscribed to by plugins or your server logic and finally transmits the data.

Obviously this isn’t ideal and slows down DarkRift quite drastically from the speeds it could achieve without Unity, but it is currently the best method and is possibly has the minimal effect on speed (particually when there is no rendering occuring).

As a side note there is nothing stopping you altering the Server.cs file so that DarkRift operates normally and fires events from multiple threads concurrently but you should be aware that you will then need to transfer the data to the main thread for use in Unity on your own. The other thing to bare in mind is that the process laid out here is potentially going to change in the future to improve speed etc. so you may need to make other alterations later if you change Server.cs.

Server.cs

This information is subject to change at any point without backwards compatibility, don't change Server.cs if you don't want to have to update everything you changed!

Bootstrapping

Bootstrapping in DarkRift is the process of starting the server and is usually handled by hidden code eg. DarkRiftServer.exe or Server.cs. When you are using embedded servers, however, it can be useful to have a little understanding of what goes on.

In the Awake routine of Unity the bootstrap process is started so that the server is running by the time the Start routine executes. The parameters are as follows:

- The mode to start the server in (Embedded/Standalone)
- The port number to listen on
- The maximum number of connection to host
- Whether to log data to the console by default
- The method to output Interface.Log information by
- The method to output Interface.LogWarning information by.
- The method to output Interface.LogError information by.
- The method to output Interface.LogFatal information by.
- The IManualDataProcessor, if any, to use to execute events etc.
- The plugins to force load.

Force Loaded Plugins

In standalone DarkRift servers plugins are loaded from the Plugins directory however in embedded servers there is no directories. The workaround for this is to supply plugins that are loaded by type rather than by placing them in the Plugins directory. To load a plugins you can add it to the forceLoadPlugins field in the Server.cs using typeof(T) and it will by loaded as usual. For an example see the Server.cs file.

IManualDataProcessor

The IManualDataProcessor supplied to the server is the system that ensures data that could be used in Unity is sent from unity (specifying this parameter in the Bootstrap method as null causes DarkRift to act multithreaded as usual).

When DarkRift goes to process an item it will pass it to the IManualDataProcessor (in this case it is actually the Server class) to store and, if necessary, return a WaitHandle to DarkRift. When Unity fires the event selected in the callEventsFrom parameter in Server.cs all events queued for execution by DarkRift will be exeuted and, if present, the WaitHandle will be set so that DarkRift can continue what it was originally doing. This means that most events are fired from Unity’s main thread and thus reduces the headache caused by the multithreading.