

## Openskies HLA Guide

This document is written to demonstrate to programmers, by example, how to integrate the Openskies functionality into their own application using the HLA API. There are three parts to this document: *Installation and Overview*, which describes installation and the distinct components of the CybernetRTI system, *Client/Player Additions*, which describes what you need to do to your application to network enable it, and *Server Additions*, which describes how you can modify the server code to increase network performance and/or implement game-specific functionality.

### Installation and Overview

The CyberRTI SDK is composed of two components. These are the Client-side API and the Server-side API.

The Client-side API is an HLA (High Level Architecture) compliant library. That is, it adheres to a US Dept. of Defense mandated specification. If you've dealt with HLA before, then you will already be familiar with most of the Client-side API. If not, don't worry. This guide is written for those who don't know anything about HLA. The Client-side API is written for Windows95/98/NT/2K/ME and includes some header files, library files, and associated dynamic linked libraries (DLLs). The CybernetRTI SDK is delivered using InstallShield and places itself in your Windows directory structure.

The Server-side API installs to a Unix-based machine (currently tested under Redhat Linux 7.0). It comes as a standard Linux RPM installation and places itself in your Unix directory tree. Besides residing on a different operating system, one main difference between the Client and Server API's is that on the client side, your application is the main executable and it calls into the CybernetRTI libraries. On the server side, there are two main Executables (*FedHost* and *Lobbymanager*), and you, as the developer, write culling and authentication modules that are called by these two programs.

### *How does it work?*

So how does this all work? During the course of a single multiplayer game there will be a number of executables running and communicating back and forth. These are:

- Application – your game, on a Win32 machine, compiled and linked against the CybernetRTI Client-side API to communicate out to the network. Hopefully there will be **many** of these running FedHost - a CybernetRTI Server-side Executable running on a Linux machine. There will be a number of these running at the same time in order to distribute the network load so that not all of the traffic is going

through any one machine at a given time. Each FedHost is responsible for handling the traffic from a number of Applications. Each Application is assigned to **one** FedHost and all traffic to and from that Application goes through its assigned FedHost. The majority of the Server-side API is designed to allow you, the developer, the ability to custom design culling routines to efficiently utilize network bandwidth and take advantage of game-specific culling possibilities.

- Lobbymanager – a CybernetRTI Server-side Executable running on a Linux machine. This application's job is to assign Applications to FedHosts. There will be one Lobbymanager for any given game, but there may be more than one game managed by a given Lobbymanager.

What is the sequence of events? Well, the first thing that needs to happen is that there has to be a number of FedHost programs running (one or more, but enough to handle the number of players you will have). Once the system administrator has these programs up and running, he can then execute the Lobbymanager program. The Lobbymanager reads in an initialization file at startup that contains a list of the FedHosts that it will manage. When the Lobbymanager is started, it looks for these FedHosts and quits if they're not all there.

Now that the Lobbymanager and all its Fedhosts are running, there's still no network game going. There are function calls in the Client-side API that allow an application to request a new federation (that's HLA-speak for a *network game*). The application is now designated the Game-host since it caused the network game to start. The Gamehost then simply joins the game it just created. Finally, other applications/players are free to join into that same federation/game. That's all.

### ***What about a Centralized Game Server?***

This may sound a bit like a peer-to-peer network solution and you might be thinking "but I to do a bunch of stuff on a central game server". The way that a developer can create a Game server is by following these simple steps:

1. Create an Application (as described in the previous bullet list) that implements your game server capability. e.g. simulates enemies, simulates physics, whatever. There may be more than one of these running in the game should you want to distribute the game server load as well! It may be desirable for your first Game Server to in fact be the Game Host (thereby creating the federation).
2. Design Culling rules using the Server-side API that routes traffic differently for Game Servers than for regular player clients. As the developer of the culling rules for the FedHosts, you have ultimate control over where all the data goes!

### ***The FED File***

The FED file is simply a file that describes the traffic you're going to be sending back and forth between the Clients. An example of this fed file is in the HelloCRTI project and is called *HelloCRTI.fed*. Here is a copy of that file (with some line numbers added):

```

1.  (FED
2.    (Federation HelloCRTI)
3.    (FEDversion v1.3)
4.    (culling-rules
5.      (module main)
6.      (module HelloCulling.so)
7.    )
8.    (spaces
9.    )
10.  (objects
11.    (class ObjectRoot
12.      (attribute privilegeToDelete reliable timestamp)
13.    (class RTIprivate)
14.    (class eoeVehicleFast
15.      (attribute Altitude best_effort receive)
16.      (attribute Latitude best_effort receive)
17.      (attribute Longitude best_effort receive)
18.      (culling-module HelloCulling.so)
19.    )
20.  )
21. )
22. (interactions
23.   (class InteractionRoot reliable timestamp)
24.   (class RTIprivate reliable timestamp)
25.   (class MissionMonitor best_effort receive
26.     (parameter Message)
27.     (culling-module HelloCulling.so)
28.   )
29. )
30. )
31. )

```

The first important part of this file is the culling rules section (lines 4-7). In this section, we list the culling modules we are going to use. Notice that culling modules are Unix Shared Object files (.so). They are dynamically linked to the FedHost when it is executed. Our HelloCRTI example implements only one culling module for simplicity's sake. The code for this culling module is described below and is provided in the CybernetRTI SDK as an example.

As you can see, lines 14-19 define the *eoeVehicleFast* object, which has three attributes that describe geographic location. Line 18 associates the HelloCulling.so culling module with the *eoeVehicleFast* object type. You'll see how the names in this file are associated with actual variables in the client and in the culling modules in the following sections. Each object is meant to describe the state of an object in your game/application (typically a player's "gamepiece"). E.g. the *eoeVehicleFast* object may be associated with an airplane, or maybe a soldier running around on the ground. Typically, each player/application will send his object updates out periodically (maybe every 5<sup>th</sup> of a second), and these will be forwarded through the FedHost(s) to the other players/applications based on the culling rules.

Unlike the Objects, the Interactions do not have a persistent state associated with them. They are generated from a player/application when some event happens (E.g. somebody fires a weapon, two players collide). Interactions are typically not streamed out like object data is. In this example the missionmonitor interaction uses the same culling module as does the *oeVehicleFast* object. The benefit of allowing an interaction to share culling modules with an object is that it allows the culling module to associate the interaction with an object. For example, you could add geographic location to your interaction so that the culling module could forward the interaction to only those players whose objects are nearby. The mechanism by which interaction culling gets access to the object culling data on the server is by having those two culling modules be one in the same!

## Client/Player Additions

Now that we know about the FED file, we can write code that can send and receive the data described in it. The Openskies SDK includes a HelloWorld example project written to demonstrate the Openskies HLA API. Here is a brief description of the different files and what you'll find in this example. The files themselves have descriptions throughout and since most programmers learn faster by doing, we suggested that you look at this example code and then incorporate it into your own application to add network functionality.

The files are located in the `CybenretRTI_Static\Samples\HLA HelloWorld` directory.

- **hello\_CRTI.cpp (.h)** – herein lies *\_tmain* (since this is a windows console application). This file contains the calls to initialize the network and contains the main loop. It creates only one local *oeFastObject* but allows you to use key commands to view the other players.
- **netconnect.cpp (.h)** – contains the *CNetCon* class that provides functions for connecting to the Lobbymanager and creating and entering federations. You can import this file right into your own project.
- **Vehicle.cpp (.h)** – contains the *CVehicle* Class which corresponds to the *oeVehicleFast* object in the *HelloCRTI.fed* file. Interactions are also handled here. If you have persistent objects in your game, you can model them after this class.
- **HwFederateAmbassador.cpp (.h)** – implements all of the callback functions for incoming network data. You can import this file right into your own project.

## Server Additions

On the Server-side, it is up to you, the application/game developer to implement culling rules that are used to route network traffic. You can choose not to use culling at all, but

then every client gets every other client's data, in which case network traffic increases with  $N^2$  (where N is the number of players). In the SDK we have included the source code for an example culling module. The module is called *HelloCulling.so* (note that this is the culling module referenced in the *HelloCRTI.fed* file). Probably the easiest way for you to get started creating your own culling module is to take this one and modify it to your needs. The files are briefly described here and are more thoroughly described internally.

- **HelloCulling.cpp (.h)** – This is the main entry point for the culling module. It contains a class that is constructed when the FedHost reads the HelloCRTI.fed file.
- **oeVehicleFastClassEx.cpp (.h)** – This file implements a class that maintains information about the oeVehicleFast object type (not particular instances, just general information). In here is where we maintain a list of the objects for culling purposes. This class also exports a function for generating new oeVehicleFast instances.
- **oeVehicleFast.cpp (.h)** – The oeVehicleFast is implemented here. It is a class that contains information about an instance of an object (so there will be one of these created for each player). The culling logic is done in here.
- **MissionMonitorClassEx.cpp (.h)** – This file implements a class that maintains information about the missionmonitor interaction type (not particular instances, there is no persistent instance of an interaction). This file contains the logic for culling... which is simply to use oeVehicleFast's culling function (see the code).