

# From HLA to MMOG and Back Again

Charles J. Cohen, Ph.D.

Rob C. Buse

Douglas Haanpaa

Charles J. Jacobus, Ph.D.

Cybernet Systems Corporation

727 Airport Boulevard

Ann Arbor, Michigan 48108

734-668-2567

[ccohen@cybernet.com](mailto:ccohen@cybernet.com), [rbuse@cybernet.com](mailto:rbuse@cybernet.com), [dhaanpaa@cybernet.com](mailto:dhaanpaa@cybernet.com), [cjacobus@cybernet.com](mailto:cjacobus@cybernet.com)

Keywords:

HLA, Massive Multiplayer, MMOG

**ABSTRACT:** *In order to simplify development of HLA compliant networked applications, we have created alternate, higher-level APIs to access the networking functionality. In order to realize a networking infrastructure that enables Massive Multiplayer Online Gaming (MMOG), not only must the infrastructure support the number, but the game developers must be willing to adopt it. Since the learning curve for HLA is rather steep, these simpler, higher-level APIs were required to support the computer game development community. Based on the lessons learned in the computer gaming world, this system has been brought back into the DoD space by implementing an HLA compliant RTI in order to achieve certification against the HLA 1.3NG and IEEE1516 specifications.*

*While the gaming and military communities have similarities in terms of simulation and networking needs, their implementation requirements and ultimate simulation goals are greatly different. This paper details the differences between the different interfaces in terms of requirements, complexity, and flexibility. We will compare and contrast gaming APIs and the HLA standards in order to analyze the similarities and differences between developers and their applications in those two areas. Furthermore, we will take a look at the specific functionalities in each API type to highlight the requirements that are typical for a DoD application versus a game.*

*Finally, to illustrate how functionality can be built to serve both groups' needs, we will detail how we have created a routing system that is able to address both cultures in terms of development ease, scalability, and robustness.*

## 1. Background

HLA is middleware between the simulation application and the network/operating system infrastructure. Like other middleware, for instance CORBA,<sup>1</sup> HLA defines a common Federation Object Model or FOM across multiple independently executing simulation applications (the FOM is a common object definition structure used by all federation participants. In a CORBA system this model would be coded in the IDL). Unlike CORBA, HLA is about many-to-many messaging or updates whereas middleware, like CORBA, is about point-to-point messaging or object invocation.

High Level Architecture (or HLA) is implemented as a standard API-level specification for distributed simulation networked applications.<sup>2,3</sup> It is essentially a language that allows the developer to generate

messages about objects and interactions as well as their component parts: attributes and parameters respectively. This language has a send side, which is an API that allows one client (Federate) to create objects, and modify the status (e.g. ownership, transportation type, region) and value of these attributes. It also has a receive side, which is a callback API that allows clients to receive a controlled subset of this information, based on publication, subscription, and region status.

The HLA specification is standardized through the SISO/IEEE under standard number 1516 and an HLA compliant RTI (Runtime Infrastructure) implementation must support the following sets of capabilities, some of which are relevant in the gaming space, and some of which are not:

- Object Management
- Federation Management

- Time Management
- Data Distribution Management (DDM)
- Management Object Model (MOM)
- Ownership Management

The HLA specification is a fairly complex API specification that supports real-time simulation as well as constructive simulation that enforces the controlled time management of the networked application. The real-time aspect of HLA is very similar to capabilities found in commercial game development, although it is defined in a more generalized form. In fact, HLA's requirement for generalization and interoperability between disparate applications is almost certainly the source of not only its complexity, but also its value. The time-managed capability described in the HLA specification is seldom, if ever, used in game development.

Cybernet's first game-oriented HLA implementation in 1998 required only a very small subset of the functionality specified in the various APIs. We had implemented a procedural flight simulator that required network-enabled objects to be discovered and updated during the course of the simulation. After having some difficulties using Windows platform versions of DMSO RTI, we decided to roll our own mini-RTI. Time management was not required for gaming nor was DDM, Save/Restore, or the MOM (we implemented a more efficient mechanism for Lobby Management). We therefore began with a small subset of the HLA specification that allowed for publication, subscription, discovery, and reliable and best-effort communication, but that was all.

During the development of the flight trainer software, we also wrote a layer above HLA that further simplified application development. This eventually became the ImportExportTable interface familiar to the game development community and described later in this paper.

Some needed items to support commercial game development were missing from the basic HLA definition and include:

1. Play Time Billing – connecting player time to the monthly use charge.
2. Supporting web sites (which include things like patch downloads, game story boards, bulletin boards for player exchange, and an all time winner score boards).
3. Player Authentication.

4. Permanent persistence database so that players would restart their experience from where they left off in the last play session.
5. Patching and database updates.

Items 1 & 2 were implemented as a set of demonstration web site functions leveraging technology commonly used in open source web stores and web sites. 3 was implemented as a custom API that uses data stored in 4 (and initially set through purchase events implemented by 1 & 2). 4 was implemented as an added API that allows the client game application to download or upload persistent data from/to an open source SQL database that operates at the website internet site. 5 is implemented through the game patching mechanism implemented through the web site (2).

Our implementation added a key set of capabilities that supports massive multiplayer (>100,000 players). This capability was designed to convert the peer-to-peer communications approach implemented in other HLAs which is order  $nxn$  (Figure 1) to an approach more similar to the client-server approach adopted by commercial game servers which is order  $n$  (Figure 2). After reviewing the evolution of DoD HLA we decided that this massive multiplier capability would be of value if brought back to the SISO community through certifying Cybernet's HLA. To this end, we audited Cybernet HLA against DMSO 1.3 and IEEE 1516 standards, adding missing time management, DDM, Save/Restore, and MOM APIs and have submitted the HLA for both certifications (they are self-certified as of this writing).<sup>4</sup>

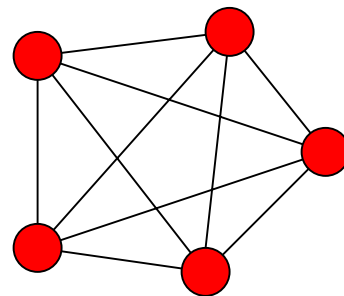


Figure 1. Peer-to-Peer DIS/HLA Systems – communications from  $n$  peers generates  $nxn$  communications paths.

The great advantage HLA provides the developer is that it defines a shared distributed object definition set that standardizes all communications within a shared distributed interactive simulation (DIS). This allows multiple independently created simulations to interact within this shared standardized object definition

framework. Each simulation participant need only be aware of the objects and their definitions that are relevant to that particular simulation's role in the federation. Each simulation can be independently developed and tested without affecting the operation of any other, making development over an extended time frame and by multiple simulation vendors (i.e. development teams) feasible.

It has historically been limited in its scalability because of its peer-to-peer model of communications, but the HLA architecture allows for implementations like Cybernet's that remedies this problem. Distribution of patches, billing, persistence, and web collateral are still beyond the definition of current HLA and must be added outside of the current specification.

### 1.1 Typical Game Messaging Systems

In its pure form, HLA and DIS simulations are peer-to-peer yielding  $n \times n$  communications as shown in Figure 1. This allows for easy simulation environment enlargement over high-speed connections and LANs because none of the simulations need be designed with the whole system in mind – integration is through a shared Federation Object Model. However, communications over WANs of varying performance can become a significant bottleneck.

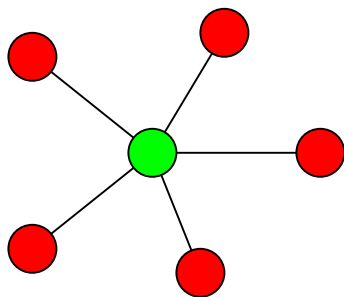


Figure 2. Client-Server – communications from  $n$  clients generates  $n$  communications paths to the game server.

Commercial multiplayer games, driven by the dual needs for security and bandwidth management, have adopted the client server model. The clients implement “game terminals” which generate visuals keyed by data from the game server (and some internal client logic) and accept user inputs, which are relayed back to the game server (assuming that the data effect game logic). The remote game server provides game logic and game security (by removing hacker access to central game logic), and can manage overall game messaging traffic from a central point. This makes the communications

order  $n$  and can scale communications linearly with the number of players that connect (Figure 2).

In the following sections we will review two alternative commercial networking approaches.

### 1.2 Unreal Network Architecture

The Unreal game engine is used by many commercial combat simulation games, including Unreal Tournament, America's Army, Duke Nukem, Navy Seals, Aeon Flux, and others. The networking architecture for the Unreal engine<sup>5</sup> is an advanced commercial game messaging system. The Unreal engine, like HLA, defines objects. These include:

1. A **variable** is an association between a fixed name and a modifiable value – equivalent to an HLA attribute.
2. An **object** is a self-contained data structure consisting of a fixed set of **variables** – equivalent to an HLA object.
3. An **actor** is an object capable of independently moving around in a **level** and interacting with other actors in that **level**.
4. A **level** is an object that contains a set of **actors**.
5. A **tick** is an operation that updates the entire **game state** given that a variable amount of time called DeltaTime has passed.
6. The **game state** of a **level** refers to the complete set of all **actors** that exist in that **level** and the current values of all of their **variables** at a time when a **tick** operation is not currently in progress.
7. A client is a running instance of **the Unreal engine** that maintains an approximate subset of the **game state** suitable for client operation.
8. A **game server** is a running instance of the Unreal engine that is responsible for **ticking** a single **level** and communicating the **game state** authoritatively to all of the **clients**.

The **game server** communicates current **game state** to all clients each **tick** (i.e. each DeltaTime) and each **client** sends movement requests to the **game server**, accepts **game state** updates, and renders the current worldview. To manage communication into the available bandwidth, the Unreal **game server** takes advantage of it's knowledge of objects and their variable changes to send abridged updates to each **client** so that each maintains a reasonable approximation of the **game state**. For instance, by **actor** type (objects that move) the **game server** provides updates based on fairness and **actor** priority. On a scale from 1-10 each actor type gets the following priorities: bots get 8, movers get 7, projectiles get 6,

pawns get 4, decorative objects (moving background) get 2 and pure decorations get 0.5.

Like in HLA, important objects (**actors**) can be created in the **game server** and replicated (discovered) by the client. Values for these objects are replicated (updated) automatically when the **game server** changes them. The replication scripting capability allows the game developer to specify who sees what. It is somewhat similar to the Data Distribution Management (DDM) component of the HLA specification, but it is more akin to the low level Cybernet RTI API that implements the DDM. This API allows the developer to implement object properties-based message delivery algorithms, which we call culling modules. These module allow the developer to control quality of message delivery at a fine grain based on object-by-object properties. Culling routines can control update rates, delivery-or-dump, and multilevel security (i.e. rules about what data particular security level uses are allowed to access or access without re-write rules).

The Unreal engine also puts emphasis on the need to perform physics and game logic on centralized servers (to prevent cheating). A brute-force server based approach will naturally introduce significant latency that naturally arise when the physics engine is separated from the input terminal by a typical Internet lag of about 200ms or more. To avoid this lag the Unreal engine introduces prediction mechanisms into the clients. There approach goes beyond a simple dead-reckoning scheme by essentially replicating some of the physics implementation on each replicated (remote) client.

### 1.3 DirectPlay Network Architecture

Microsoft DirectPlay characterizes an alternative approach used in some commercial games. DirectPlay is the network component of DirectX, Microsoft’s game development environment on Windows and the Xbox.<sup>6</sup> DirectPlay is not object oriented (like the Unreal engine described previously) at all.

The DirectPlay API does not dictate a peer-to-peer or client-server architecture, but rather implements a higher-level set of “socket”-like connection functions over the normal Windows IP stack for the convenience of the game developer. DirectPlay provides APIs for:

- Lobby Management (joining leaving a multiplayer game – connecting to the shared game server)
- Sending messages to a client or to the game server
- Receiving messages from a client or game server
- Implementing callbacks on messages received

The DirectPlay lobby architecture (Figure 3) is a means for connecting (third party) games and lobby applications (game servers) through the lobby application. The lobby application is basically a phone book that allows these connections to be made without direct knowledge of IP addresses needed for ordinary socket connections.

DirectPlay *DOES NOT* define an object oriented structure within the game application that can be share between clients and game servers (as is done by both he Unreal engine and HLA). Messaging in Direct Play is done strictly from player to player or from player to game server in whatever datagram format that is defined by the game developer – it is basically nothing more than a higher-level TCP/UDP based communications API.

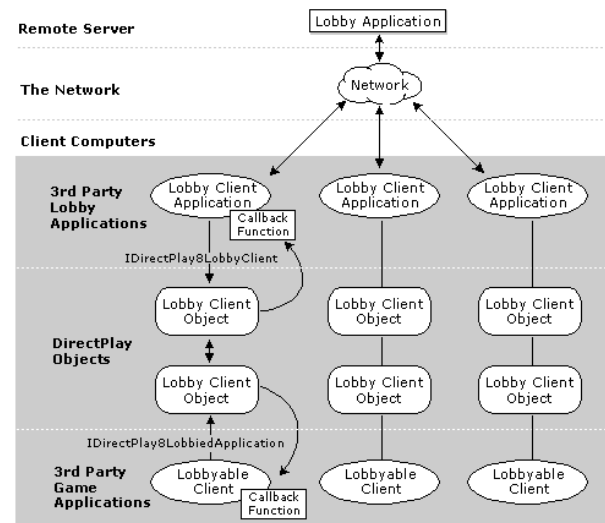


Figure 3. DirectPlay Lobby Architecture

## 2. Bridging the Divide

Application developers in the gaming world are similar to developers in the military space in that they are interested in getting many players online in a common virtual environment, while minimizing the required bandwidth and latency. However there are differences in the relative importance of various aspects of networking in games vs. military simulation. The key to bridging the gap between these two genres is to allow each group to use their own API and to then implement a scheme to translate between them.

Towards this goal, HLA is a specification that can and is implemented in a variety of ways through alternative RTIs. Because there is no such standard adopted in the gaming space, HLA-based implementations that can conform to the game developer’s desired attributes for

a networking engine have a unique opportunity to fuse the two communities together.

### 2.1 API Considerations

Developers on both the military and gaming sides desire easy-to-use development tools in order to implement their applications. In this regard, developers in the government space are more willing to sacrifice ease-of-use for standardization among disparate applications developed by multiple vendors, perhaps at multiple times. HLA provides such standardization, but at a price. The HLA programmer interface is somewhat difficult to learn and is more complex than the average game developer wants.

### 2.2 Cheating/Security/Authentication

This is an issue that plagues the gaming community. Hackers are very good at figuring out how to bring ruin to online communities, or how to give themselves a leg up in competition by enhancing their character, army, vehicle, etc. Because of this, the overwhelming consensus in the game developer community is that a significant amount of computation, at least that which determines hits, score, health, etc., must be performed on game servers which are physically located at a secure location. Peer-to-peer communication cannot be used without checks because it puts the intelligence of the simulation (i.e. the game) in the hands of the enemy (i.e. the hacker) who can then alter its behavior to their own advantage.

Military simulation, on the other hand, has historically used peer-to-peer communication. The reason for this is that it supports easier “plug-and-play” distributed simulations and military personnel are deemed trustworthy. They are “playing” to train and are often supervised and/or in physical proximity to their leaders and/or peers while engaging in virtual training. However, as pointed out earlier, this approach does not support scalability to massive play and will have to eventually be discarded as joint military training simulations grow even to divisional-sized exercises.

### 2.3 Authentication

This is important for both Military and Gaming applications. It is important for military application for reasons of assessing access rights based on national security policies. It is important in the gaming space to support correct billing and maintenance of the privacy of personal information.

## 3. An MMOG 1516 Compliant RTI

As described earlier in section 1.1.1, Cybernet has implemented a 1516 compliant HLA that blurs the boundary between the military training simulation

developer’s and game developer’s needs. The HLA incorporates a fully 1516 API for Windows, WinCE, and Linux/Unix users, including lobby management, DDM, time management, MOM, object, ownership, and federation management, but has been implemented to support key extensions for massive multiplayer and game development requirements: These include:

- A distributed fault tolerant dynamically scalable Federation management and message routing fabric. This fabric is like a distributed game server that handles all multicast distribution of object creations, updates, discovers, and destroys. Each simulation client makes a two-way connection to a single federation host (FedHost – one process in the distributed federation cloud) so communications are order  $n$  like in commercial game server architectures (Figure 4).
- Attached message filters that are executed in the routing fabric that allow control of update and message flow between clients and objects within each client based on current federation state and client/object properties. These are called culling rules or modules.
- A layer of game developer interface over HLA that implements point-to-point messaging (called the P2PS or Point-to-Point Switch) and a simplified object oriented interface called the Import/Export Table (IET).
- Companion software that implements a shared persistent SQL database function that can be accessed by game servers or clients to save game state data from playtime to playtime, a web site template that supports shared display of game status information, current high scores, bulletin board functions, etc., and an authentication/billing function so that players can be identified and optimal billing operations can be controlled.

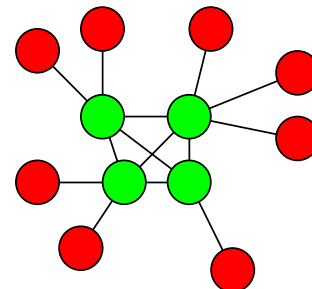


Figure 4. OpenSkies HLA Client/Distributed Routing Fabric Architecture – communications from  $n$  clients generates  $n$  communications paths into the server fabric.

The **Lobby Manager** can be placed on a "broker server" computer to manage a large network of federations over the Internet. A game application can use the "Login" member function to authenticate to the Lobby Manager, and the "Logoff" member function to log off. The Lobby Manager keeps track of a list of "repeater servers." Each "repeater server" is running one or more copies of FedHost to be discussed later.

If a user requests creation of a new federation, a new FedHost process will be launched on each repeater server. If a user joins an existing federation, the least busy repeater server with the lowest ping time is assigned to him. All FedHost processes for the same federation on various repeater server machines communicate with each other through multicast and each has information about the entire federation.

**FedHost** processes are spawned on each repeater server by command of the Lobby Manager. The FedHosts perform all the host functions in the existing RTI code. They communicate with each other via TCP/IP connections, IP multicast, and UDP datagrams. They maintain a complete list of federates, but each will communicate directly with a limited number of these clients based on dynamic load assignment. Each client connects into the **FedHost** cloud through only one **FedHost**, which relays all messages and therefore can form an "image" of the current game state.

Because clients do not communicate directly with each other, network traffic is reduced to order  $n$  into the **FedHost** cloud. **FedHosts** also perform culling to further reduce network traffic. The Culling functions, reside on the repeater server, defined as "member functions" for the attribute sets in the Federation Object Model (FOM) file. Application developers can use culling modules already in the system by default or can develop their own game-specific culling routines in DLL's, or "so" files under Linux to implement security or quality of service algorithms. Each function can be turned on and off at run-time. The standard HLA DDM facility is implemented as default culling modules controlled at the Client side by the HLA DDM APIs.

As an example, if we start with "CybernetBaseEntity", which consists of double Altitude, double Latitude, and double Longitude, we can override the "Cull" member function, and define it in such a way that it returns

TRUE if (a2 >= Altitude-Altitude0 >= a1) and  
 (b2>=Latitude-Latitude0>=b1) and  
 (c2>=Longitude-Longitude0>=c1)

FALSE if otherwise.

(Altitude0, Latitude0, Longitude0) is a "CybernetBaseEntity" that belongs to the potentially receiving federate.

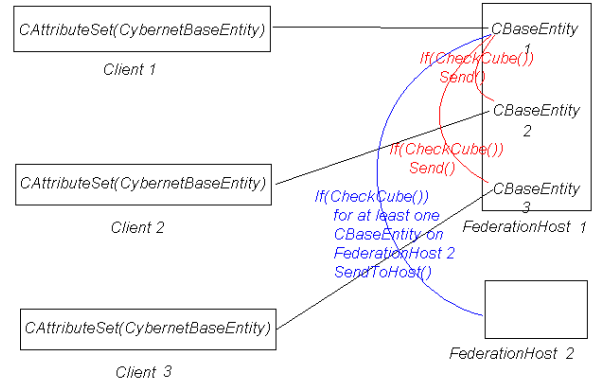


Figure 5. Culling Rules attached to each FedHost control message relay to clients based on programmed criteria. In this way distributed simulation designers control quality of service and/or security access.

Because each **FedHost** has a copy of the current game state, logic is easily included that rolls clients off of any failed **FedHost** on to the remaining ones (i.e. without informing the players so fault tolerance is transparent) and new **FedHosts** can be brought on line and inserted into the Lobby Manager's use list without stopping the game (so that the routing fabric is enlarged dynamically if the user community exceeds fabric capacity).

The **Client** is the game run by the game user. Clients are also known as **Federates**. The Cybernet RTI currently supports federates residing on WinCE, Windows, or Linux/Unix machines.

A federation (*network game*) is created by the first **Host Federate** request to the Lobby Manager. The Lobby Manager responds to this request by commanding its FedHosts to create a federation that this and other federates can join. Player federates (Clients) can be given **Host Federate** capability or not, depending on command structure, business and/or game paradigms. The **Host Federate** may be the share Game Server Federate. If there is no game server (clients are logically peer-to-peer), the first player client can be a **Host Federate**.

The **Game Server Federate(s)** is an optional application in the federation that implements the control aspects of the game not implemented in the clients. Typically, these are things that are common to the game universe or aspects of play that are to be

isolated from the players for security or persistence reasons (e.g. book keeping for score, weapons status, etc.). Persistence is implemented in the **game server** by using local persistent storage (disk space) or by using an SQL database server access API that accesses the database server(s). The **game server** accesses the routing fabric to create, update, and destroy objects using the same APIs as player clients.

The **Database server** (also known as **Authentication server**) is the system that runs a database application such as MySQL or Oracle. Some applications may not have any databases and others may have multiple databases. The communications to and from the module that handles passing information to and from the database(s) can be either encrypted via secure socket layer (SSL) or not.

Purposes of the database include:

- Login (user and password) restriction/verification
- Demographic User information
- Logged into system time (useful for time based billing)
- Number and time of login/logout (useful for billing and other stats)
- Game or application specific scoring/information
- Game Server statistics
- FedHost statistics
- Client defined information

Basically, any statistic value that the game developer wants to keep can be placed into the database(s) for retrieval and analysis. This retrieval and analysis can be done via .html or .shtml pages from within an internet browser (i.e. in a web site) or by custom software using the database server access API.

## 4. The Game Developer Friendly API

In order to provide game developers with tools that expose the core capabilities of HLA, but that hide its complexity, we have created two interfaces, one object-based interface and one message-based, which are layered on top of the HLA interface. We call these interfaces the Import/Export Table (IET) and Point-to-Point Switch (P2PS) interfaces respectively.

### 4.1 ImportExportTable Interface

To explain the IET interface, let us assume that we are developing an application that simulates the state of an object. This object could be an aircraft flying around in simulated world, an articulated character, or a piece on a chessboard. It does not matter.

Here is the declaration of the *CAirVehicle* class.

```
class CAirVehicle {
public:
```

```
CAirVehicle (const char *namein) {
    strcpy(m_name, namein);
}
~CAirVehicle (void);

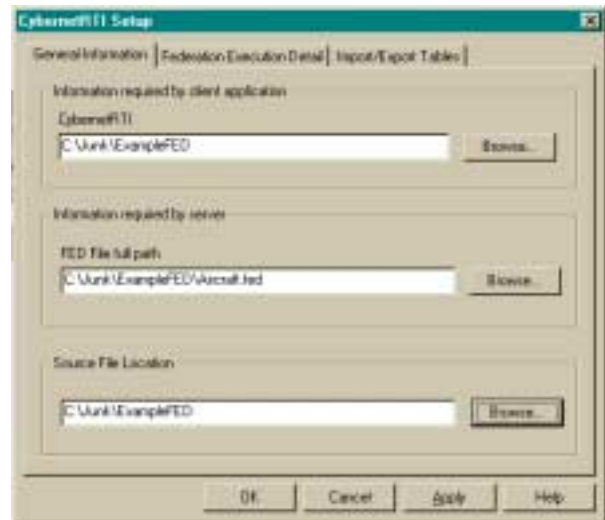
char name[256];
double m_latitude;
double m_longitude;
double m_altitude;
double m_roll;
double m_pitch;
double m_yaw;
};
```

Every time a new vehicle is created in the application a new instance of this class is created:

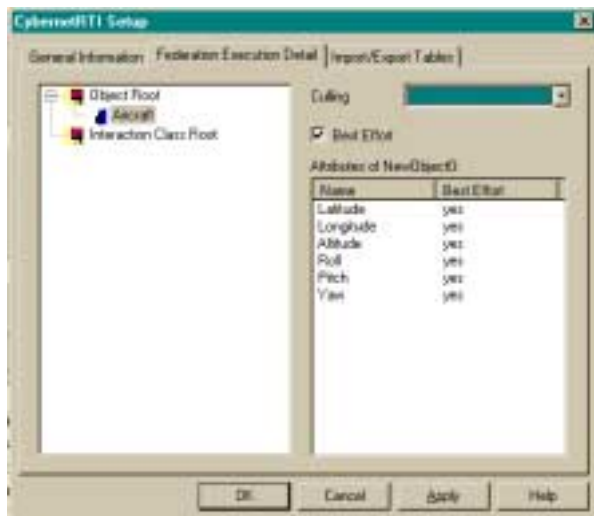
```
CAirVehicle *newAC = new CAirVehicle
("Maverick");
```

This would create a new aircraft with the name "Maverick". The simulation code in the application would move Maverick around by altering the 6 position/orientation member variables in the class. Our task now is to network this information so that other applications can see where Maverick is.

The first step is to use the graphical configuration tool.



The Configure tool has three tabbed sub-documents of information that you need to complete in order to create networked objects. Once completed, the Configure program will automatically generate the configuration and source files needed for a multiplayer application. The first page simply describes the folder location of where these files will be placed.



The second tab is called “*Federation Execution Detail*”. In this page, we first define a new federation object *Aircraft*. This object defines the packet of information constituting one update for the state of the aircraft. The fields of this object are to be sent out over the network for every update. In HLA the object is known as an “*attribute set*”.

Finally, in the *Import/Export Tables* page, we create the C++ class itself, where each member variable is mapped to a corresponding attribute in the *Aircraft* attribute set.



#### 4.2 Point-to-Point Switch

Although these interfaces are somewhat more restrictive than the HLA function calls, they are easier to use. The *ImportExportTable* interface reduces the complexity of an update to one function call whereas an equivalent HLA requires that the developer construct the update attribute by attribute. The functionality of the P2PS interface, the ability to send messages to

named objects that may not even yet exist, is something that is not in the HLA specification, but was important in at least one game development effort: *StarFleet Command II*.

#### 4.3 Authentication

To facilitate authentication, the join functions in the IET API required an authentication key. With authentication enabled, the network servers (*FedHosts*) verify these authentication keys in order to allow or disallow federates from joining the federation.

In order to allow the developer to use in their own authentication scheme, the public servers accept plug-in modules that will receive the data sent by the client/federate. The developer simply provides exactly two plug-in modules so that the public server can process this authentication data. The *Lobby Manager* loads one of these modules and one is loaded for each *FedHost*. These modules are “.so” files for Linux, i.e., shared object (similar to Windows DLL) files. Each module must export two functions:

```
extern "C" CSecurityInfo
*CopySecurityInfoClass(const
CSecurityInfo *pInfo);
```

```
extern "C" CSecurityInfo
*CreateSecurityInfoClass(DWORD
dwSecurityDataSize, const void
*pvSecurityData);
```

where *CSecurityInfo* is defined in *SecurityInfo.h*. These functions should authenticate the incoming class, and they should return NULL if authentication fails.

#### 5. Retrofitting Games that Have Multiplay

The typical military training simulations can go from multiplay to massive multiplay by relinking with Cybernet’s *OpenSkies* implementation of DMSO HLA version 1.3 or the IEEE 1516 compliant version (which are transparently interoperable together within the same federation).

To bridge between the military training HLA world and typical commercial games is more difficult. As described in Section 1, even games that use a standard API like *DirectPlay* do not provide a game or federation wide middleware object definition like the HLA FOM. Therefore, there is no automatic game independent way to achieve an HLA FOM to commercial game object mapping – i.e. automatic network message interoperability.

For a practical matter, we have found from working with *Tesseract* (*Enigma*, *Rising Tide*), *Taldren* (*StarFleet Command*), and other developers of games that it is normally very easy to substitute the IET, P2PS and where necessary other HLA APIs for alternative

commercial game network code when game source access is available.

Some games like those based on the Unreal engine (which includes America's Army and other efforts undertaken by America's Army Government Applications office)<sup>7</sup> or Microsoft's Flight Simulator<sup>8</sup> provide published network API documentation and internally represent objects in a manner similar enough to HLA that transliteration to IET and P2PS is very straightforward.

## 6. Conclusion

The automatic universal translator to make commercial games transparently interoperable with military training federations is not there and probably will only come about by adoption of common set of standards like HLA by the commercial as well as the military simulation world. However some important special cases like Unreal engine interoperability is possible now.

## 7. References

- [1] Object Management Group: CORBA home site, <http://www.corba.org/>.
- [2] Defense Modeling and Simulation Office: High Level Architecture, <https://www.dmsomil/public/transition/hla/>.
- [3] Simulation Interoperability Standards Organization, High Level Architecture Standards, <http://www.sisostds.org/stdsdev/hla/>.
- [4] Cybernet: OpenSkies 1516 compliant HLA, home site with documentation and demonstration applications, [www.openskies.net](http://www.openskies.net).
- [5] Sweeney, Tim: Unreal Networking Architecture, -- <http://unreal.epicgames.com/Network.htm>.
- [6] Di Benedetto, Robert: Microsoft DirectPlay 8 Overview, <http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnplay/html/dp8ovrview.asp>.
- [7] Gaudiosi, John: Army Sets Up Video-Game Studio, <http://www.wired.com/news/games/0,2101,63911,00.html>, Wired News, Jun. 21, 2004.
- [8] Microsoft, FS2004 SDK Documentation, [http://www.microsoft.com/games/flightsimulator/fs2004\\_downloads\\_sdk.asp](http://www.microsoft.com/games/flightsimulator/fs2004_downloads_sdk.asp).

## Author Biographies

**DR. CHARLES COHEN** is Vice President of Technology for Cybernet and was the original developer of Cybernet Gesture technology, derived from his PhD work completed at the University of Michigan under Professor Lynn Conway. Dr. Cohen received his BSEE from Drexel, and his MSEE and PhD from Michigan. Dr. Cohen has performed research and project management in robotics, computer vision, gesture recognition, and artificial intelligence for 15 years.

**MR. ROB BUSE** joined Cybernet Systems to further develop Cybernet's OpenSkies networking platform and help bring it into HLA compliancy. He has a rigorous understanding of Linux/Unix development with experience in C/C++, LISP, Assembly, OpenGL, Gtk, X-windows, Gnome, and the POSIX system libraries. He also has experience with MS Windows development using their Win32 API, MFC libraries, and DirectDraw/DirectSound SDKs. Mr. Buse received his BSCS from Michigan Technological University on 2002.

**MR. DOUG HAANPAA** is the technical team manager for Cybernet's Virtual Reality group. He is responsible for the management of a general-purpose, open-standard simulation architecture called OpenSkies. He also contributed to much of the design of this product as well as implementation of flight dynamics, collision, and force-feedback algorithms. He also acted as a key designer for many of the OpenSkies subsystems including the terrain parsing/rendering/LOD system, scenegraph, weather model, and parallel thread/timer system. Mr. Haanpaa received his MSCS and BS, Applied Physics from Michigan Technological University in 1993 and 1991 respectively and has been with Cybernet since that time.

**DR. CHARLES JACOBUS** is a founder of Cybernet ([www.cybernet.com](http://www.cybernet.com)) and currently is its CEO. He has performed research and product development in robotics, computer vision, networking, medical applications, and electronics for over 30 years. He received his BSEE, MSEE, and PhD from the University of Illinois, Urbana-Champaign and worked at Texas Instruments and the Environmental Research Institute of Michigan prior to co-founding Cybernet.