CYBERNET

OpenSkies

Networking Engine

ImportExportTable (IET) API Guide

www.openskies.net

MMPOG

MASSIVE MULTIPLAYER ONLINE GAMING

Openskies IET Guide

This document provides a brief description by example, of the Openskies IET (*Import/Export Table*) application programmer interface (API). This example describes the process of network-enabling your application. A functional VC++ 6.0 project that implements this example can be found in the *samples* subdirectory under *ExampleFED*

Application

First, let us assume that the application 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.

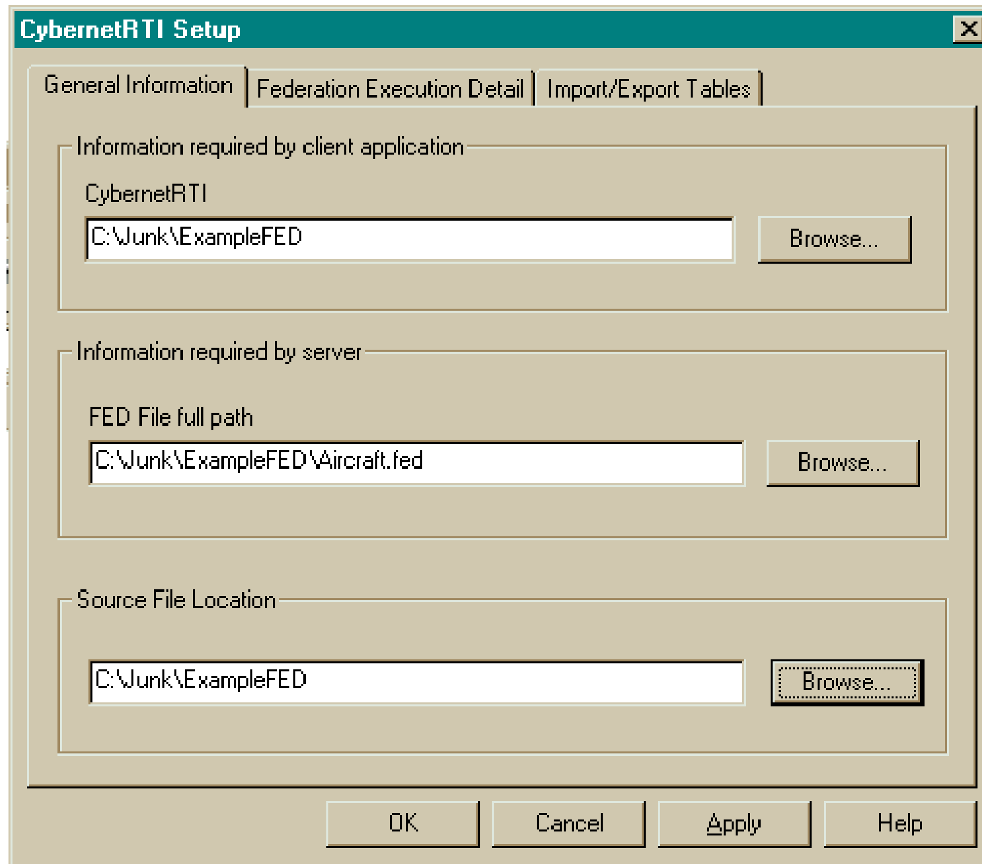
```
class CAirVehicle {
public:

	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;
};
```

(For simplicity's sake, I have made everything public instead of using access functions as well as hard-coded the length of the name). Every time a new vehicle is created in the application a new instance of this class is created:

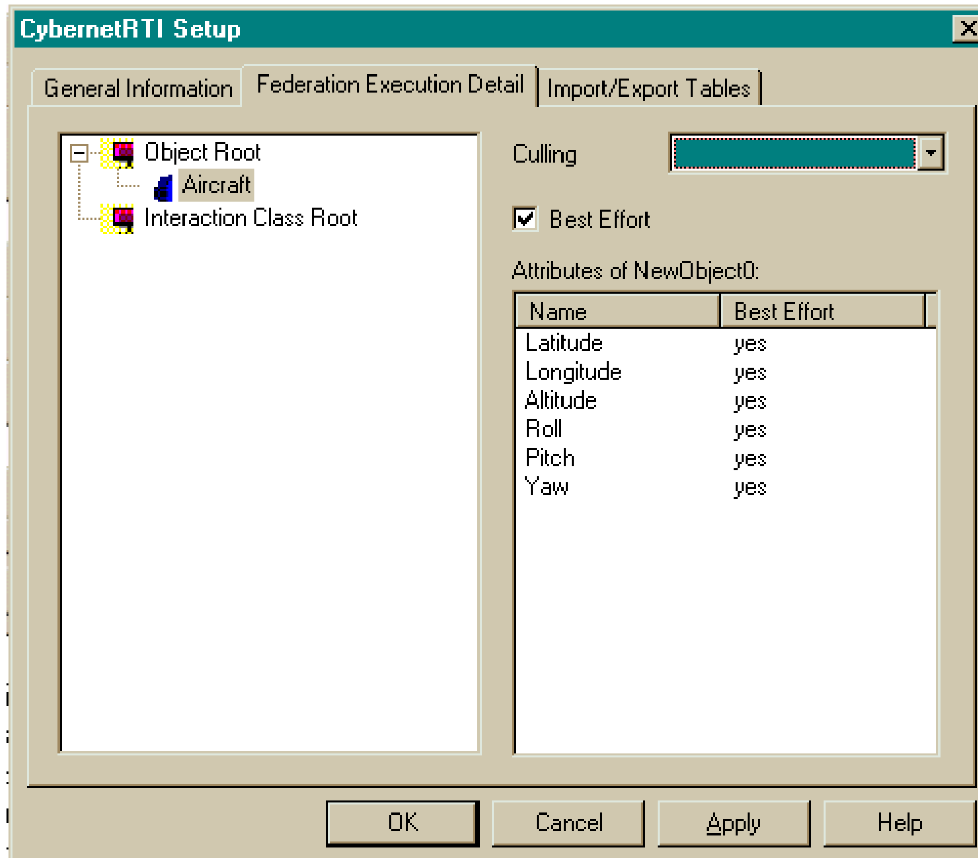
```
CAirVehicle *newAC = new CAirVehicle ("Bob");
```

This would create a new aircraft with the name "Bob". The simulation code in the application would move Bob 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 Bob is.

Step 1: Use the Openskies Configure Tool



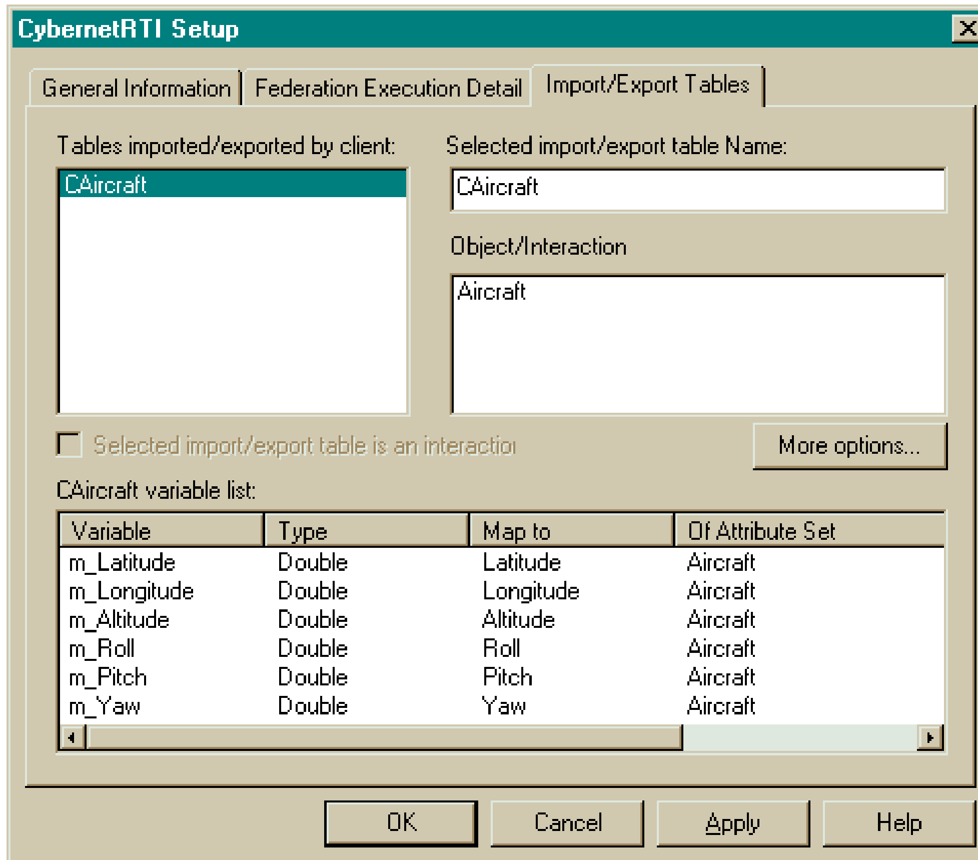
The Configure tool has three tabbed sub-documents of information that you need to complete in order to create networked objects. Once completed, the Configure program will automatically generate the configuration and source files needed for your multiplayer application. The first page simply describes the location of where these files will be placed. The "*Information required by client application*" is a configuration file that is automatically named "CybernetRTI.ini" and will be placed in the location specified. This file must eventually reside in the same directory as the application executable does. The "*Information required by the server*" is another configuration file called the FED file (must have the extension *.fed*) and is fully specified in the second edit box. This file must eventually reside on each of the network servers or, in peer-to-peer mode, must, like the CybernetRTI.ini file, reside in the same directory as the main application. Finally, the "*Source File Location*" is where all the auto-generated source and header (*.cpp and *.h) files will be placed.



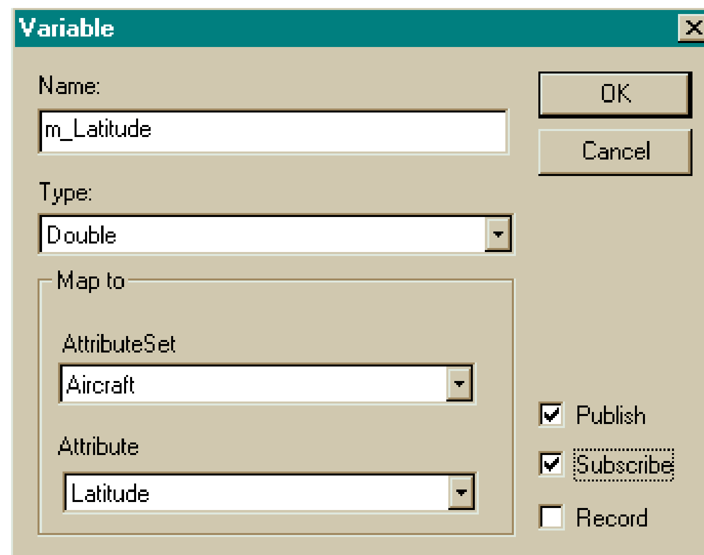
The second tab is called "*Federation Execution Detail*". In this page, we first define a new federation object *Aircraft.* This object defines the packet of information constituting one update for the state of the aircraft. The fields of this object are to be sent out over the network for every update.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

Adding an attribute is done by right-clicking on the *Object Root* and selecting *add*. We can then similarly add *attributes* for this *attribute set* in the list box on the right, by right-clicking in it. We select *Best Effort* because this information will be sent repeatedly, and so no *one* update is critical. Best Effort is sent using UDP/IP whereas *Reliable* attributes are sent using TCP/IP.

Now, in the *Import/Export Tables* page, we create the C++ class itself, which is tied to the attribute set *Aircraft*:



In this interface, we make sure to use the same variable names (e.g. *m_Latitude*) as the application. The variables listed in the "*CAircraft variables list*" will be sent out whenever *CImportExportTable::UpdateAttributeSets* is called. Similarly, for remote *CAircraft* instances, these variables are updated automatically whenever data comes in over the network. The variables above are added by right-clicking in the variable list box and selecting *add*, which will bring up the following dialog box.



Use this interface to define the types for each of the variables and to assign them to the corresponding FED value.

Step 2: Configure Generated Source Code

The Configure tool will automatically create the *CybernetRTI.ini* file and the *Aircraft.fed* file. This will also create a header file and some class definition for the *CAircraft* class. Here is the auto-generated *CAircraft.h* header file:

```
#ifndef INCLUDE_AIRCRAFT_H
#define INCLUDE_AIRCRAFT_H

#include "IET.h"
const DWORD dwAircraft_IET_DoubleCount = 6;

class CAircraft : public CImportExportTable
{
      double m_Latitude;
      double m_Longitude;
      double m_Altitude;
      double m_Roll;
      double m_Pitch;
      double m_Yaw;

      CDoubleValue m_aDoubleValues[dwAircraft_IET_DoubleCount];

public:
      CAircraft();
      virtual ~CAircraft();

      virtual double *GetIETDouble(LPCTSTR pszName);

      virtual void GetIETID(CIETID &ID);
      virtual void ImportCallbackProc(void);
};

#endif //INCLUDE_AIRCRAFT_H
```

The Following is the auto-generated Aircraft.cpp source file:

```
#include "Aircraft.h"

CAircraft::CAircraft(LPCTSTR pszName, LPCTSTR pszClassName,
      LPCTSTR pszURL)
{
      m_Latitude = 0.0;
      m_aDoubleValues[0].m_szName = "m_Latitude";
```

```
        m_aDoubleValues[0].m_pValue = &m_Latitude;
        m_Longitude = 0.0;
        m_aDoubleValues[1].m_szName = "m_Longitude";
        m_aDoubleValues[1].m_pValue = &m_Longitude;
        m_Altitude = 0.0;
        m_aDoubleValues[2].m_szName = "m_Altitude";
        m_aDoubleValues[2].m_pValue = &m_Altitude;
        m_Roll = 0.0;
        m_aDoubleValues[3].m_szName = "m_Roll";
        m_aDoubleValues[3].m_pValue = &m_Roll;
        m_Pitch = 0.0;
        m_aDoubleValues[4].m_szName = "m_Pitch";
        m_aDoubleValues[4].m_pValue = &m_Pitch;
        m_Yaw = 0.0;
        m_aDoubleValues[5].m_szName = "m_Yaw";
        m_aDoubleValues[5].m_pValue = &m_Yaw;
}

CAircraft::~CAircraft ()
{
}

double *CAircraft::GetIETDouble(LPCTSTR pszVariableName)
{
        for(DWORD i = 0; i < dwAircraft_IET_DoubleCount; i++)
        {
                if(!_tcscmp(pszVariableName,
m_aDoubleValues[i].m_szName))
                        return m_aDoubleValues[i].m_pValue;
        }
        return NULL;
}

void CAircraft::GetIETID(CIETID &ID)
{
        _tcsncpy(ID.m_szClassName, _T("CAircraft"),
IET_MAX_STRING);// First level category, e.g. cars, boats
        ID.m_szClassName[IET_MAX_STRING -1] = 0;

        ID.m_szAbstractName[0] = 0;      // e.g. Corvette, Viper
        ID.m_szInstanceName[0] = 0;      // e.g.
The_one_that_Doug_is_driving
        ID.m_szUserName[0] = 0;                  // e.g. Passenger1,
driver, etc.
}
void CAircraft::ImportCallbackProc(void)
{
        // Add your code here
}
```

Your application must override the function *GetIETID* to fill in *ID.m_szAbstractName, ID.m_szInstanceName, and ID.m_szUserName.*

Step 3: Integrate Generated Source Code

In the main application, we now derive the *CAirVehicle* class from this newly created *CAircraft* class. In this example, we also take out the *CAirVehicle* variables since they are now in the *CAircraft* class. Another approach would be to create new names for the *CAircraft* variables and then write some code to copy them to and from the *CAirVehicle* class. This would be useful for dead-reckoning, for example.

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

	virtual void GetIETID(CIETID &ID);

	char name[256];
};

void CAirVehicle::GetIETID(CIETID &ID)
{
	// Let the base class version fill in class name
	CAircraft::GetIETID(ID);

	_tcsncpy(ID.m_szInstanceName, m_name, IET_MAX_STRING);
	ID.m_szInstanceName[IET_MAX_STRING -1] = 0;
}
```

Before we can send out an update for the class instance, we need to call its *CreateAttributeSets* function, and then its *RegisterAttributeSets* function.
At this point, the existence of the new attribute set that is associated with your class instance is known over the network, with ID as specified in *GetIETID*. A remote client will receive a call through *CIETCallbacks::CreateIET* (to be explained later) to indicate the existence of this attribute set, with the given ID. This remove client will need to create and return a new class instance for this attribute set in its *CIETCallbacks::CreateIET* function. Assume that this instance can be referred to by a pointer `CAirVehicle *pAircraftOnRemoveClient`.

Now, in order to send out an update for the class instance, we simply call its *UpdateAttributeSets* function. This will automatically look at the 6 variables (e.g. *m_Latitude*) and send them out. On the remote client, variables of `*pAircraftOnRemoveClient` will be updated, and its *ImportCallbackProc* will be called.

When we are done with this class instance, we call its *UnregisterAttributeSets* function, and then its *DestroyAttributeSets* function.

Step 4: Create Some Callback Functions

Two source files, IETCallbacks.h and IETCallbacks.cpp, are provided and need to be complied and linked into your main application. These files define a class called *CIETCallback*. Please do not alter IETCallbacks.h. The file IETCallbacks.cpp, however, is a skeleton file, and must be altered. For example, Openskies will inform you of new attribute sets discovered over the network through *CIETCallback::CreateIET* defined in IETCallbacks.cpp. We now explain these callback functions one by one:

`CIETCallbacks::GetAppName.` This function needs to return a text string that can be used by Openskies to create registry entries for your application. For example:

```
LPCTSTR __fastcall CIETCallbacks::GetAppName(void)
{
      return _"MyApplication";
}
```

Important: If your application uses MFC (Microsoft Foundation Class), you MUST call `AFX_MANAGE_STATE(AfxGetAppModuleState())` before using MFC functions such as `AfxGetApp`:

```
LPCTSTR __fastcall CIETCallbacks::GetAppName(void)
{
      AFX_MANAGE_STATE(AfxGetAppModuleState());

      return AfxGetApp()->m_pszAppName;
}
```

`CImportExportTable * __fastcall CIETCallbacks::CreateIET(const CIETID &ID).` This function needs to return a pointer to a new instance of your class. For example:

```
CImportExportTable * __fastcall
CIETCallbacks::CreateIET(const CIETID &ID)
{
      // We can only understand CAircraft
      if(tcscmp(pID->m_szClassName, "CAircraft"))
            return NULL;

      CAirVehicle *pAircraftOnRemoveClient = new
            CAirVehicle(pID->m_szClassName);

      // Add pAircraftOnRemoveClient to a container
      // class of some kind here. Please remember that
      // this callback is asynchronous.
```

```
        return pAircraftOnRemoveClient;
    }
```

`void __fastcall CIETCallbacks:: DestroyIET(CImportExportTable *pImportExportTable).` This function needs to delete what is pointed to by `pImportExportTable`. For example:

```
void __fastcall DestroyIET(CImportExportTable
*pImportExportTable)
{
    // Find pImportExportTable in your container
    // class and do whatever cleanup necessary here
    // Please remember that
    // this callback is asynchronous.

    return pImportExportTable;
}
```

`CImportExportTable * __fastcall CIETCallbacks::FindIET(const CIETID &ID).` This function needs to search your container class to find the one instance identified by `ID`, and return a pointer to it. Please remember that this callback is asynchronous.

Step 5: Configure Openskies

Before your application can start or join a network federation, you need to configure Openskies by calling *CNetInterface::Configure*:

```
// First call GlobalInitCybernetRTI - this function
    inializes the Openskies library and must be called
    before anything else.
CImportExportTable::GlobalInitCybernetRTI();

// Now get the netinterface pointer
CNetInterface *pNetInterface =
    CImportExportTable::GetNetInterface();

// the RegistrySettings class contains all the information
    specifying how this application will communicate to
    the outside world
CRegistrySettings RegistrySettings;

// LobbyManager
RegistrySettings.m_eRunLobbyManager =
    CRegistrySettings::eRunLobbyManagerOnlyWhenHosting;
RegistrySettings.m_szLobbyManagerAddress = "192.168.1.1";
```

```
RegistrySettings.m_eLobbyManagerSearch =
      CRegistrySettings::
      eSearchForLobbyManagerWhenNotHosting;
      RegistrySettings.m_fUsePublicServer = false;
      RegistrySettings.m_dwLobbyManagerPort = 2000;

// Multicast
RegistrySettings.m_fUseMulticast = true;
RegistrySettings.m_szMCastAddress = "224.9.9.1";
RegistrySettings.m_dwMCastTTL = 5;
RegistrySettings.m_dwMCastPort = 22500;
RegistrySettings.m_dwMaxMCastPort = 22530;

// FedHost
RegistrySettings.m_dwUDPPort = 0;
RegistrySettings.m_szFedFileName = "ExampleFED.fed";

// Retry settings
RegistrySettings.m_dwTimeOut = 3000;
RegistrySettings.m_dwRetryCount = 10;

// tell the netinterface about these new settings
pNetInterface->Configure(&RegistrySettings);

// close down the library
CImportExportTable::GlobalResetCybernetRTI();
```

CRegistrySettings is defined in *NetInterface.h*. The following are explanations of the member variables of *CRegistrySettings*, one by one:

- *m_eRunLobbyManager*: This is a enum variable of type *ERunLobbyManagerFlags*. It can take on three different values, namely *eNeverRunLobbyManager*, *eRunLobbyManagerOnlyWhenHosting*, and *eAlwaysRunLobbyManager*. LobbyManager is what maintains a list of running federations. There can be more than one LobbyManagers running on a network, each maintains a different list of running federations, provided that they are not using the same multicast settings if they do use it.
- *m_szLobbyManagerAddress*. This is a text string for LobbyManager address. If you run LobbyManager locally, or if you use a public server provided by a third party, you need to specify this.
- *m_eLobbyManagerSearch*. This is a enum variable of type
- *ElobbyManagerSearchFlags*. It can take on three different values, i.e., *eAlwaysSearchForLobbyManager*, *eSearchForLobbyManagerWhenNotHosting*, and *eNeverSearchForLobbyManager.* When you do not run LobbyManager locally, this flag specifies if you want to search for a LobbyManager, and when.
- *m_fUsePublicServer*. This is a boolean variable that indicates if you want to use LobbyManager and FedHost servers provided by a third party, possibly outside of you LAN. When this is true, *m_eRunLobbyManager* and *m_eLobbyManagerSearch* are ignored.

- *m_dwLobbyManagerPort*. This is a 4-byte unsigned integer. If you run LobbyManager locally, or if you use a public server provided by a third party, you need to specify this.
- *m_fUseMulticast*. This is a boolean variable that indicates if you want to use multicast. If you want to run LobbyManager locally and want others to be able to find you without using specific IP address, or if you want to search for LobbyManagers when you are not running it locally, you need to set this variable to true. Enabling multicast can reduce network traffic in a federation and improve performance.
- *m_szMCastAddress*. This is a text string that specifies the multicast IP address.
- *m_dwMCastTTL*. This is a 4-byte unsigned integer in the range 0 through 255 that specifies multicast packet TTL. Each router decrements the TTL by one. When the value reaches a predefined lower limit, the router throws the packet away.
- *m_dwMCastPort.* This is the port that is used by LobbyManager. It is a 4-byte unsigned interger less that 65536.
- *m_dwMaxMCastPort*. This is a 4-byte unsigned integer value. It is used only when running LobbyManager locally. Each federation listed with the LobbyManager is assigned a multicast port between *m_dwMCastPort* and *m_dwMaxMCastPort.*
- *m_dwUDPPort*. This is a 4-byte unsigned integer value. This is used for best-effort traffic when multicast is disabled. Setting it to 0 to allow the operating system to choose one automatically.
- *m_szFedFileName*. This is a text string for the FED file name. The FED file is generated in "Step 1". If you are not using a public server, and you are hosting a federation, this file must be in the same directory as the *ImportExportTable* DLL.
- *m_dwTimeOut*. This is a 4-byte unsigned integer value in milliseconds. For each attempt communicating with LobbyManager or FedHost, this is how long Openskies will wait for a reply when a reply is needed.
- *m_dwRetryCount*. This is a 4-byte unsigned integer value. It specifies how many times Openskies will try to communicate with LobbyManager or FedHost when a reply is needed.

Step 6: At Startup and Shutdown

When your application is ready to start a network federation, you need to make the following calls:

```
CImportExportTable::GlobalInitCybernetRTI();

CNetInterface *pNetInterface =
   CImportExportTable::GetNetInterface();

// Describe the federation that we are going to start
CFederationInfo FederationInfo;
```

```
// description of the federation
FederationInfo.m_szDescription = "This is a sample
   application";

// Host federate information. This identifies yourself over
// the network as the one
// who started the federation. We will use your computer name
// for this
DWORD dwBufferSize = 256;
GetUserName(FederationInfo.m_szHostInfo.GetBuffer(
         dwBufferSize), &dwBufferSize);
   FederationInfo.m_szHostInfo.ReleaseBuffer();

// List of strings for each federate
// We are the only one in the federation right now. But add
// it anyway so that others who join later can see it
// CIETString is included in the CybenetRTI SDK and
// implements a subset of the MFC CString capability
CIETString *pID = new CIETString(FederationInfo.m_szHostInfo);
FederationInfo.m_FederateInfoArray.Add(pID);

// Identify this new federation on the network
dwBufferSize = 256;
GetComputerName(FederationInfo.m_szFederationName.GetBuffer(
         dwBufferSize), &dwBufferSize);

FederationInfo.m_szFederationName.ReleaseBuffer();

// A flag to be passed into the startup function. It
// will change during the startup; thus indicate what
// the network code is actually doing. But since we are
// not doing multithreading, we cannot check it here
LONG lFlag;
if(!pNetInterface->StartHostingFederation(FederationInfo,
   lFlag))
{
   CImportExportTable::GlobalResetCybernetRTI();
   return;
}
```

If your application is ready to join an existing network federation instead, you need to make the following calls:

```
CImportExportTable::GlobalInitCybernetRTI();

CNetInterface *pNetInterface =
   CImportExportTable::GetNetInterface();

// We need to specify a federation by name to join.
```

```
// Get the list of current hosts from the network
CIETStringPtrArray HostArray;
if(!m_pNetInterface->QueryFederationHosts(HostArray))
{
   return;
}

// We will just join the very first one for now
if(!pNetInterface->JoinFederation(HostArray[0]))
{
   CImportExportTable::GlobalResetCybernetRTI();
   return;
}
```

After your either started a new network federation or joined an existing network federation, you may perform network operations such as creating and registering attribute set for a *CAirVehichle* class.

Upon termination, your application should call

```
CNetInterface *pNetInterface =
   CImportExportTable::GetNetInterface();
pNetInterface->StopHostingFederation();
CImportExportTable::GlobalResetCybernetRTI();
```

if you started the federation, or

```
CNetInterface *pNetInterface =
   CImportExportTable::GetNetInterface();
pNetInterface->ExitFederation();
CImportExportTable::GlobalResetCybernetRTI();
```

if you joined the federation.

Advanced Topics:

Asynchronous Processing

All callbacks, which include all members of *CIETCallbacks* as well as *CAircraft::ImportCallbackProc* are called asynchronous to your main application code. This simplifies code structure in that you do not need to worry about adding polling loops, calculating polling time, etc.
But this also requires some other consideration, as all multithread programs do. One way is to use mutex:

```
CImportExportTable * __fastcall CIETCallbacks::CreateIET(const
   CIETID &ID)
```

```
{

   // We can only understand CAircraft
   if(tcscmp(pID->m_szClassName, "CAircraft"))
      return NULL;

   CAirVehicle *pAircraftOnRemoveClient = new
   CAirVehicle(pID->m_szClassName);

   // Add pAircraftOnRemoveClient to a container
   // class of our own, “MyContainer”, using a mutex
   // handle “hMutexOfMyContainer”:
   WaitForSingleObject(hMutexOfMyContainer, INFINITE);
   MyContainer.Add(pAircraftOnRemoveClient);
   ReleaseMutex(hMutexOfMyContainer);

   return pAircraftOnRemoveClient;
}
```

In this code we protect the function call

```
MyContainer.Add(pAircraftOnRemoveClient);
```

with a mutex so that "`MyContainer`" will not be used while we modify it by adding our new class instance, "`pAircraftOnRemoveClient`". Of course, this means that when we use "`MyContainer`" elsewhere in our main application, we need to protect that code with the same mutex handle as well.

There is another way, which might be much easier in the sense that you will not need to add mutex everywhere you use "`MyContainer`" in your main application:

```
CImportExportTable * __fastcall CIETCallbacks::CreateIET(const
   CIETID &ID)
{
   // We can only understand CAircraft
   if(tcscmp(pID->m_szClassName, "CAircraft"))
      return NULL;

   CAirVehicle *pAircraftOnRemoveClient = new
   CAirVehicle(pID->m_szClassName);

   // Add pAircraftOnRemoveClient to a container
   // class of our own, “MyContainer”, using a
   // windows message WM_ADD_AIRCRAFT that we define as
   // WM_USER + <some number>:
   SendMessage(hMyWindow, WM_ADD_AIRCRAFT, 0,
   (LPARAM)pAircraftOnRemoveClient);

   return pAircraftOnRemoveClient;
}
```

where “pAircraftOnRemoveClient” is the handle of a window in your main application. In your main application, in the window proc of “hMyWindow”, you will need to add a case like

```
case WM_ADD_AIRCRAFT:
   MyContainer.Add((CAirVehicle *)lParam);
   break;
```

or add a function to your window class if your application is a MFC application:

```
LRESULT CMainFrame::OnAddAircraft(WPARAM wParam, LPARAM
   lParam)
{
   MyContainer.Add((CAirVehicle *)lParam);
   return 0;
}
```

Of course, in the latter case you will need to add a line in the same source file

```
ON_MESSAGE(WM_ADD_AIRCRAFT,  OnAddAircraft)
```

like so

```
BEGIN_MESSAGE_MAP(CWnd, CMyWindow)
//{{AFX_MSG_MAP(CMainFrame)
   …
ON_MESSAGE(WM_ADD_AIRCRAFT,  OnAddAircraft)
//}}AFX_MSG_MAP
END_MESSAGE_MAP()
```

and the prototype

```
afx_msg LRESULT OnAddAircraft(WPARAM wParam, LPARAM
   lParam);
```

the header file as follows:

```
//{{AFX_MSG(CMyWindow)
   …
afx_msg LRESULT OnAddAircraft(WPARAM wParam, LPARAM
   lParam);
//}}AFX_MSG
```

User Authentication / Security

In Step 6, “Start up and Shutdown”, we used several member functions of the class *CNetInterface*. Look at the prototypes of these functions in *NetInterface.h*, one would notice that these functions have some additional parameters that we did not use.

Indeed, if you do not use a public server as described in Step 5, “Configure CybernetRTI”, you never need to worry about these extra parameters. CybernetRTI behaves like a peer-to-peer network. Anyone who has access to the underlying network connections will be able to join in. If you do use a public server, however, these parameters allow you to setup user authentication.

We do not need to go through these functions individually because each accepts the same authentication parameters:

```
void *pSecurityData
DWORD dwSecurityDataSize
```

pSecurityData points to the data buffer that contains data for user authentication, and *dwSecurityDataSize* specifies how large the buffer is. CybernetRTI simply passes the content of the buffer to the public server, and makes no modification to it. Your application must match the format of the data to what the public servers (LobbyManager and FedHost) use. The server side of the authentication process is described in another document called *Openskies Security Guide*. If the authentication process fails, the *CNetInterface* call will fail and return 0 (false).

The Openskies SDK also includes a central database facility and secure communication facility for user management and account information. It is described in the *DB_Connection Guide* document. Using the CDB_Connection class, you can access a central database from which you can get an encryption key to be used in the *pSecurityData* buffer described above.

Culling

A culling module is a plug-in module for FedHost running on a public server. This module is an “.so” files for LINUX, i.e., shared object (similar to a Windows DLL) file. This culling module must be written by you; the game developer.
which is referenced in as described in Step 5, *Configure Openskies*, you never need to worry about culling. CybernetRTI behaves like a peer-to-peer network. Anyone who has access to the underlying network connections will receive all information in classes that he subscribes to. If you do use a public server, however, culling modules may become necessary to avoid network traffic congestions.

Culling modules are connected to attribute set types or classes, which are described in “Step 1. Use the CybernetRTI Configure Tool”. A culling module may contain culling

code for multiple attribute set classes. In such case, the culling module is a library of many code segments that are not directly related to each other.

Each culling module must export a function prototyped as follows:
extern "C" CAttributeSetClassEx *GetAttributeSetClassEx(const CAttributeSetClass *pClass),
where *CAttributeSetClass* and *CAttributeSetClassEx* are classes defined in *AttributeSetClass.h*. Suppose that there are two attribute set classes, AttributeSetClass1 and AttributeSetClass2 defined in your FED file, and you are building a culling module for both, the following code sample can be used for your version of this function:

```
extern "C" CAttributeSetClassEx *GetAttributeSetClassEx(
           const CAttributeSetClass *pClass)
{
     /* The pClass->GetName() function holds the name of the
           object or interaction where this culling module was
           found */
     if(!strcmp(pClass->GetName(), "AttributeSetClass1"))
     {
           try
           {
                /* If we found the class to be
                AttributeSetClass1, we create a class extension
                for it. Only one of these will be created, since
                the fed file is only parsed once. The class
                definition is in AttributeSetClass1Ex.cpp */

                return new CAttributeSetClassEx(pClass);
           }
           catch(...)
           {
                return NULL;
           }
     }


     if(!strcmp(pClass->GetName(), "AttributeSetClass2"))
     {
           try
           {
                /* If we found the class to be
                AttributeSetClass2, we create
                a class extension for it. Only one of these will
                be created, since the fed file is only parsed
                once. The class definition is in
                AttributeSetClass2Ex.cpp */

                return new CAttributeSetClass2Ex;
           }
           catch(...)
```

```
        {
            return NULL;
        }
    }

    // Unknown class
    return NULL;
}
```

where you must define the classes *CAttributeSetClass1Ex* and *CAttributeSetClass2Ex*, and derive each class from *CAttributeSetClassEx*. In general, culling modules must define such attribute set class extensions, and derive them from *CAttributeSetClassEx*. From this point on, each time FedHost creates a new attribute set of CAttributeSetClass1 type, for example, it will call *CAttributeSetClass1Ex::CreateAttributeSet.*

Besides defining *GetAttributeSetClassEx* and attribute set class extensions, you must also define derived classes from *CAttributeSet* defined in *AttributeSet.h*, and override the following member functions:

virtual bool Cull(const CAttributeSet *pAttributeSet). Override this function if the culling is not based on any other attribute set class. Return true if the owner of *pAttributeSet* needs to be updated.

virtual bool Cull(const CAttributeSet *pAttributeSet, const CAttributeSet *pCullingAttributeSet1, const CAttributeSet *pCullingAttributeSet2). Override this function if the culling is based on another attribute set. pCullingAttributeSet1 is associated with this class, and pCullingAttributeSet2 is associated with pAttributeSet. Return true if the owner of *pAttributeSet* needs to be updated.

virtual bool GetPreferredRecipientIDArray(CSortedDWORDArray &RecipientIDArray). Override this function if the culling is not based on any other attribute set class. This function should copy into RecipientIDArray a list of attribute set handles whose owners should discover this attribute set. Note that discovery does not guarantee that these owners will receive updates unless the Cull function returns true also. Return false if all subscribers should discover this attribute set. Be very careful when returning false since it disables culling.

virtual bool GetPreferredRecipientIDArray(CSortedDWORDArray &RecipientIDArray, CAttributeSet *pCullingAttributeSet). Override this function if the culling is based on another attribute set class. pCullingAttributeSet is associated with this class. This function should copy into RecipientIDArray a list of attribute set handles whose owners should discover this attribute set. Note that discovery does not guarantee that these owners will receive updates unless the Cull function returns true also. Return false if all subscribers should discover this attribute set. Be very careful when returning false since it disables culling.

virtual void SetCulleeHandle(DWORD dwCulleeClassHandle, DWORD dwCulleeHandle). Override this function if your culling module requires this information.

virtual bool UpdateValues(const CNetDataBuffer &NetDataBuffer). Your culling module will receive value updates via this function. Return true means that this has caused a change in the “preferred recipient ID array” for this attribute set or some other attribute sets.