

# JOCosim: Using Java, OPNET, and C to Enable Mobile Co-Simulation

Christopher J. Augeri

Air Force Institute of Technology  
Wright-Patterson AFB, OH 45433  
E-mail: chris.augeri@afit.edu

Kevin M. Morris

Air Force Communications Agency  
Scott AFB, IL 62225  
E-mail: kevin.morris-03@scott.af.mil

Barry E. Mullins

Air Force Institute of Technology  
Wright-Patterson AFB, OH 45433  
E-mail: barry.mullins@afit.edu

## Abstract

Military services have an active interest in the use of unmanned aerial vehicles (UAVs) for applications including surveillance and unmanned combat. In the future, UAV swarms will be able to gather intelligence cooperatively without human intervention. Since the physical employment of UAV swarm technology is not yet economical, simulation is often used to explore their design.

This paper discusses a co-simulation prototype, JOCosim, for UAV swarms that leverages Java, OPNET, and other simulation engines. The fundamental contribution presented herein is that Java is used to provide command and control for a co-simulation involving external application-layer protocols and OPNET.

## Introduction

A key area of research at our institution is the performance and capabilities of unmanned aerial vehicles (UAVs). Our particular research investigates various aspects of UAV swarms, to include swarming algorithms and networking performance. We model distributed mobile sensor networks involving several thousand sensor nodes. For various reasons, some of our algorithms are developed external to OPNET, thus, integration is desirable.

A network often studied at our institution are unmanned vehicle swarms we refer to as a Host of Armed Reconnaissance Vehicles Enabling Surveillance and Targeting (HARVEST) [17]. Within this paper, all nodes in a HARVEST are presumably unmanned aerial vehicles (UAVs). A HARVEST example that was used to study the AODV protocol is shown in Figure 1.

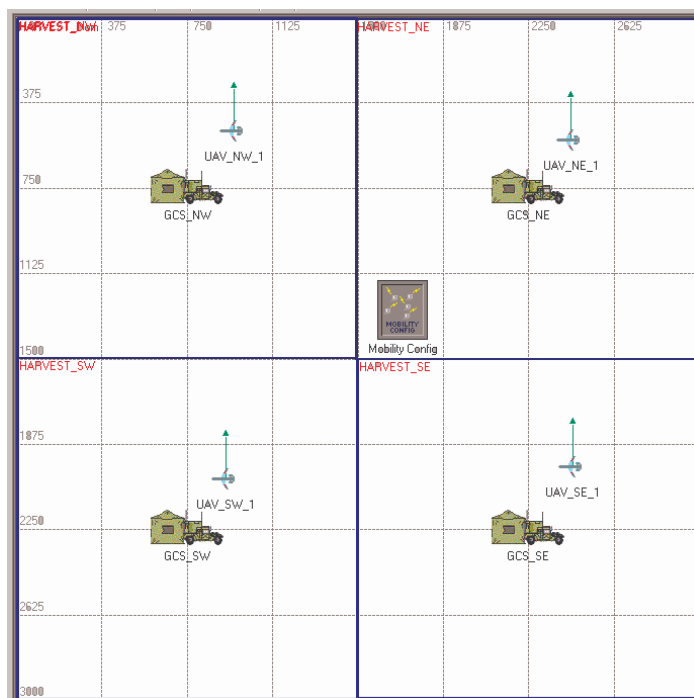


Figure 1: HARVEST Deployment

The HARVEST application layer we are studying is shown in Figure 2. Our research seeks to model various aspects of this application layer in a simulated network. A significant amount of our research is conducted within OPNET. However, there are certain functions in Figure 2 that have limited capabilities or are not available in OPNET. Two of these areas are flight simulation and application-layer services. Mobility exists in OPNET, but it does not support wind modeling or swarming algorithms.

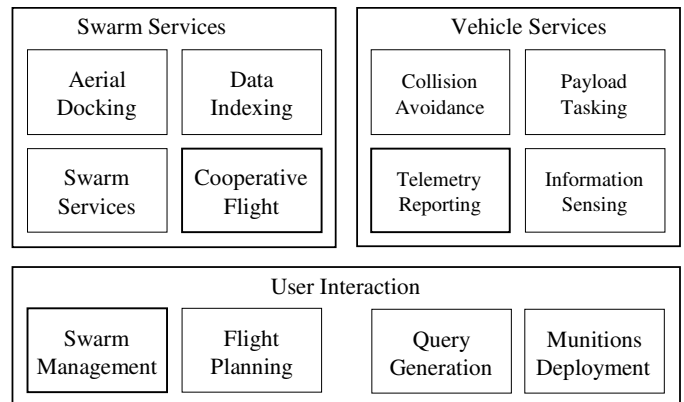


Figure 2: HARVEST Application Layer

Our approach is to develop a co-simulation that leverages the strengths of simulation and computation engines in use by the U.S. Department of Defense (DoD). This approach enables researchers to independently study elements of the application layer. The co-simulation environment we have developed is a Java, OPNET, and C-based co-simulation, dubbed JOCosim II. Additional OPNET co-simulations are described in [18, 19].

The JOCosim version described herein extends the one in [17]. This version demonstrates Java can control OPNET simulations with respect to event execution and data injection. Key reasons for choosing Java are platform independence, threading, and networking support [2]. The added functionality in JOCosim II is listed Figure 3. Items marked with a '\*' are partially complete.

- Move co-simulation control to Java from C
- Develop management cloud for ESYS access
- Create models via external model access (EMA)
- Enable broadcast & routed transmissions
- Support node failure and recovery
- \*Active statistic collections via probes
- \*Port a node mobility algorithm to Java
- \*Integrate a distributed data indexing application
- \*Migrate from OPNET 10.5 to 11.5

Figure 3: JOCosim II Extensions

The views expressed in this paper are the authors and do not reflect the official policy or position of the United States Air Force (USAF), Department of Defense (DoD), or the U.S. Government.

## HARVEST Scenario

To provide a context for the technical discussion that follows, a HARVEST simulation in JOCosim is shown in Figure 4, as displayed by the OPNET animation viewer. Although JOCosim I provided animation, JOCosim II only uses OPNET’s animation viewer, given the emphasis on having control be via Java. The core parameters for our simulations are the physical dimensions of the network, the number of search and target nodes, search cell size, routing protocol {AODV, DSR, TORA}, transmission power, simulation execution time, and a random seed. The application layer protocols may have additional parameters.

The network in Figure 4 is 100 m<sup>2</sup>, sub-divided into search cells that are 10 m<sup>2</sup>. Fifteen UAVs are randomly dispersed within the search grid and explore the network via a cooperative search algorithm [20]. The cleared cells (those without a dark center square) are considered “explored”. A cell is considered explored when a UAV passes within a cell’s center quarter. If a target is discovered, this information is shared via a flooding protocol. A large ‘X’ marks the location a UAV was destroyed. A UAV that does not have radio waves around it is being held in reserve. All other UAVs shown are considered active — these are the only UAVs able to participate in routing and application-layer protocols. More importantly, these are the only UAVs that will have statistics collected about their performance.

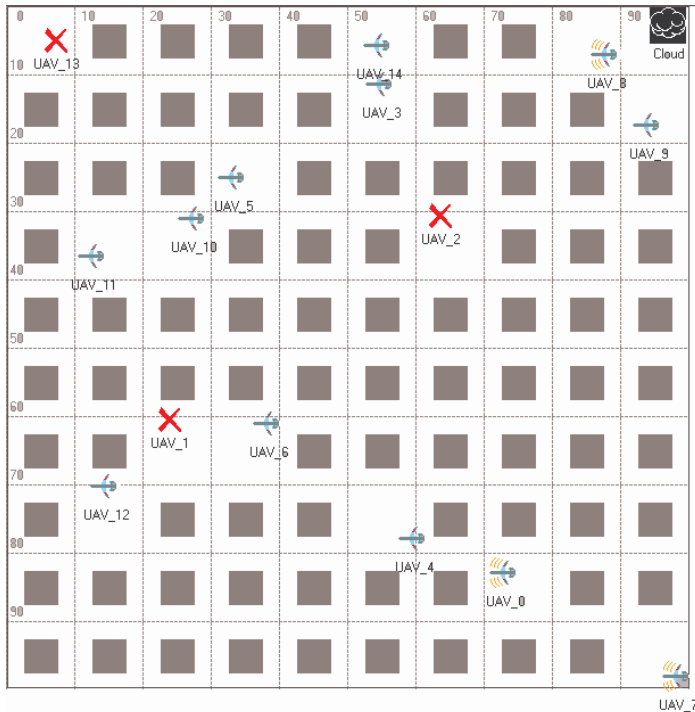


Figure 4: JOCosim Simulation (15 nodes)

In the upper right is the OPNET cloud node that provides all co-simulation support for this model. The cloud node consists of a single external system module process model. Conceptually, a similar cloud exists in the external Java controller. The UAVs are advanced MANET station process models that have been minimally modified to provide the necessary statistic probes. Additionally, the packet generators of the UAVs have been removed, as all packet generation occurs via application-layer algorithms that exist in Java. It is thus a relatively simple matter to change these nodes to a different node model if desired.

## JOCosim Design

The fundamental contribution of this work is given in Figure 5. We have created an environment that enables the control of an OPNET simulation via Java. This contribution is conceptual, shown by the design choices described herein, and technical, given the complexity of JOCosim that demands expertise in Java, OPNET, and C, as well as operating system configurations. OPNET’s software documentation is the predominant source of information for this research [7–16].

The Java controller provides a user interface and simulation command and control. The Java controller uses the Java Native Interface (JNI) to access a C wrapper library that calls OPNET external system access (ESA) methods. The C library and OPNET exchange data using an external systems interface (ESI). To ensure simulations are repeatable, synchronization occurs via registered callbacks between C and OPNET and C and Java. These include callbacks to process ESI calls and text output from OPNET. Currently, there is no user interaction after a simulation begins, thus no input callbacks are registered with OPNET. The Java controller also integrates with an external model access (EMA) network model constructor, mobility code, and with suitable extensions, to other simulation environments.

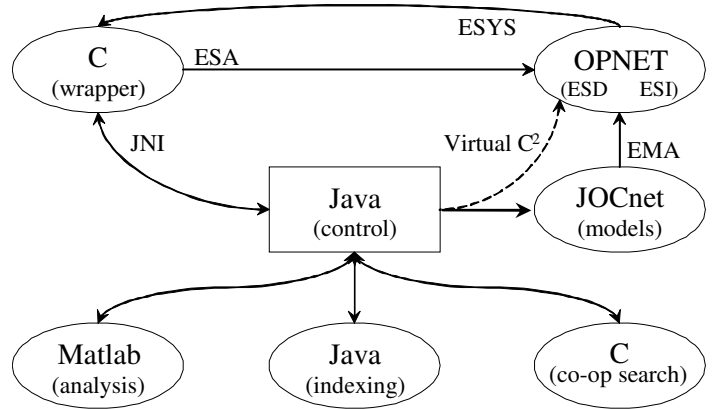


Figure 5: JOCosim II Top-Level Design

To demonstrate that this design is achievable, we have only implemented a simply timing mechanism where control is returned to Java after execution of a single OPNET event. If Java determines that no action is necessary via the registered callback, it calls the C wrapper and has OPNET execute another event. If a packet must be processed, the packet is sent to the counterpart node in Java, that is, each UAV in OPNET has an application-layer node in Java that conceptually corresponds with the HARVEST application layer shown in Figure 2.

This design provides the foundation for the basic goals: flight simulator integration, co-simulation, and application-layer services. With respect to the vehicle simulator integration, we are exploring both ground vehicle simulators and flight simulators. Once suitable systems are identified, they will replace the mobility component of the cooperative search algorithm. This extension will hopefully include the incorporation of DTED data for improved radio simulation. Second, although we are currently using the generic external system architecture, JOCosim should be capable of using the High-Level Architecture (HLA) as specified by the Defense Modeling and Simulation Office (DMSO) [1].

A simulation generally occurs as follows. The user is prompted for a set of model parameters, as shown in Figure 6. The current version uses the default power and routing setting of the nodes. The number of nodes, network/cell dimensions, simulation time, randomization seed, animation display, and statistic probe file can be specified via this interface. The advanced tab provides for options such as debug versus optimized mode. The model and scenario names are auto-generated to minimize user overwrites.

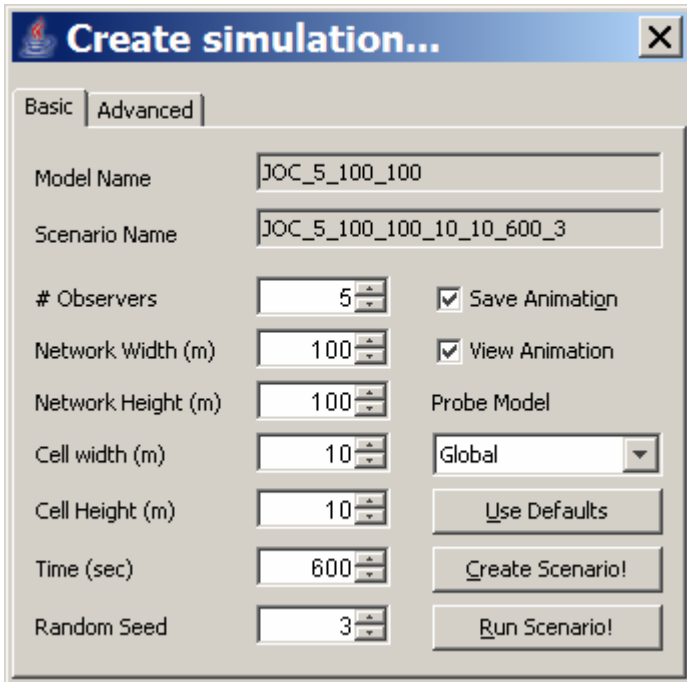


Figure 6: Simulation Parameter Entry Window

The default animation probe file is shown in Figure 7. Two additional probe files are also available, one that will collect global statistics, the other to collect statistics at the node level of our application layer traffic.

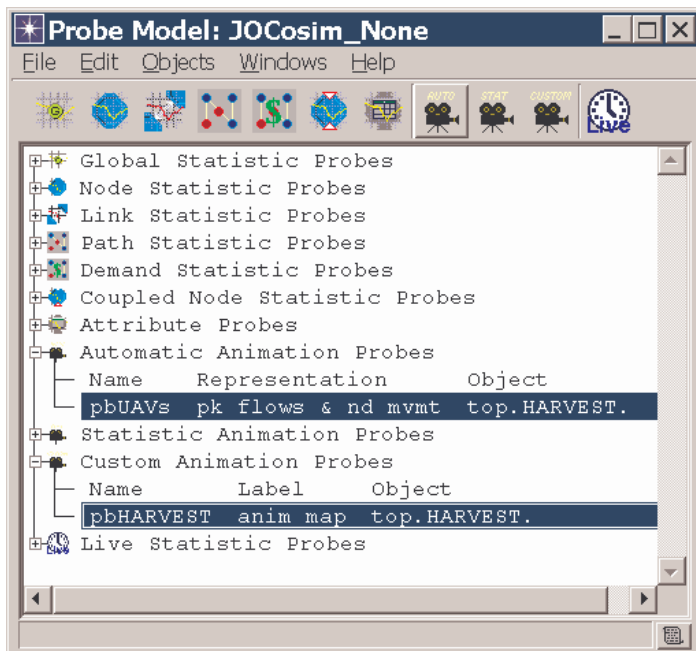


Figure 7: Animation Probes

A network model is constructed when the “Create Scenario!” or “Run Scenario!” buttons are selected. The Java controller passes the parameters to JOCnet, our automated EMA-based model builder. JOCnet then creates both a network model and ESD file based on the supplied parameters. The Java controller creates an environment file (EF) from the model parameters. The model parameters are read by the OPNET cloud (Figure 4) and simplify the ESD and ESIs that JOCnet creates. The network model that JOCnet creates can be imported in OPNET if desired.

After the “Run Scenario!” button is clicked (Figure 6), the Java controller launches a simulation control window (Figure 8). The control window is relatively simple, as this version of JOCosim focuses on demonstrating we can control OPNET via Java. The progress bar reflects the current simulation time as reported by OPNET. This is actual simulation time, not raw wall clock time with respect to simulation execution.

The OPNET text output stream is piped via a registered callback in the C wrapper to a Java queue and is displayed in the “Output Messages” text box. The error messages are similarly handled, but only display internal messages from the C wrapper to assist development. These queues are processed in an independent thread from the user interface and the Java controller. This enables a simulation to proceed without pausing for text output.

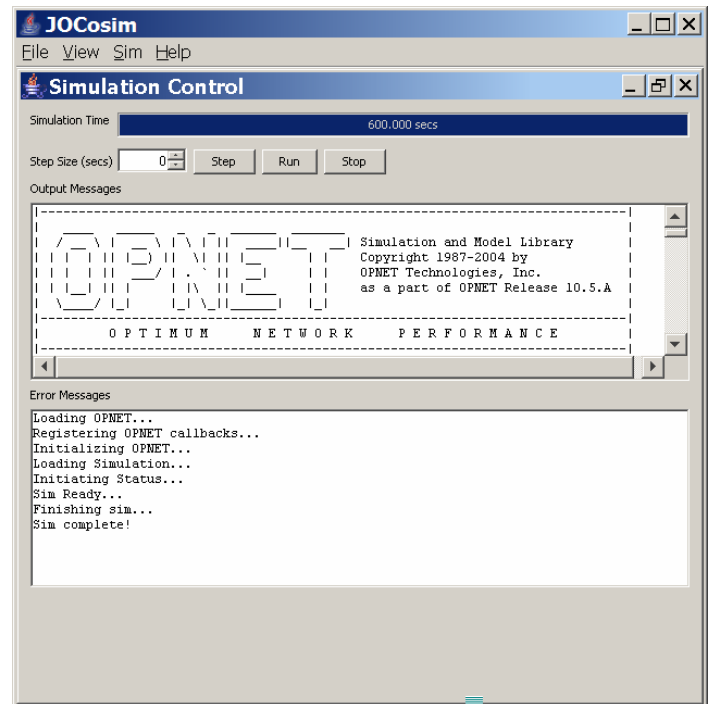


Figure 8: Simulation Execution Window

Once OPNET is initialized, it will activate an animation stream (Figure 4), assuming the OPNET animation viewer is activated. If the Java controller determines the next event time is at or after the time of the next queued packet, it will pass the packet via the C wrapper. The controller thread then executes a single OPNET event. If the event involves receiving a packet at a node, the OPNET cloud will update the statistics and deliver the packet to the Java controller via a C wrapper and the ESIs. The cloud will also process any received cell coverage, node status, and node position updates at this time.

From a timing perspective, a JOCosim simulation executes the sequence shown in Figure 9. Each transition marked by a dashed line occurs only once. The main loop is in the Java controller, as noted by the first node in the loop, to instruct OPNET to execute a single event. This sequence only captures the main Java control thread. The user interface threads are not shown.

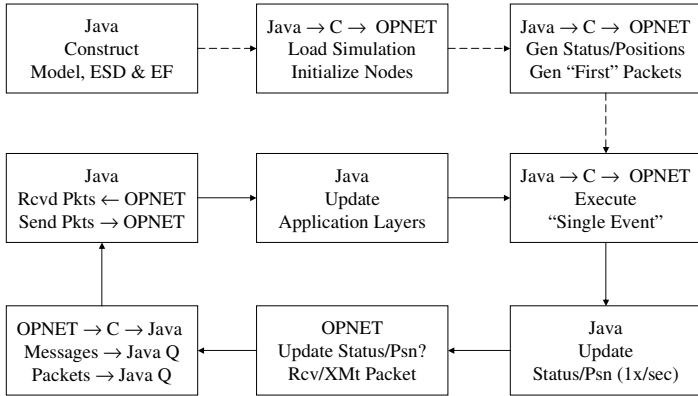


Figure 9: Simulation Execution Flow

Once the simulation has loaded successfully, communication occurs via the ESD. An example ESD is listed in Table 1. The first three fields, {esd\_status, esd\_psnX, esd\_psnY}, hold a single value for each node. Although this ESD was used for a 15-node simulation, the first position of each of these fields is reserved for the control cloud described previously; hence, each field has 16 elements. These three fields are updated at 1-second intervals by the Java controller. The data for the fields is passed in bulk format via the C wrapper library.

Field Name	Data Type	Data Flow	Size
esd_status	integer	Java → OPNET	16
esd_psnX	double	Java → OPNET	16
esd_psnY	double	Java → OPNET	16
esd_inCtrl	integer	Java ↔ OPNET	0
esd_inPkt	character	Java → OPNET	4000
esd_outCtrl	integer	Java ↔ OPNET	0
esd_outPkt	character	Java ← OPNET	4000

Table 1: External System Definition (ESD) – 15 nodes

The {esd\_inCtrl, esd\_outCtrl} fields provide a basic signaling mechanism between Java and OPNET for packet exchange. If Java has a packet for OPNET, it reads the “esd\_inCtrl” field. If the field is less than zero, it sets the field to a positive value, thus reserving the “esd\_inPkt” channel. Java then writes a packet to “esd\_inPkt”. After the packet is written, Java schedules an ESYS interrupt in OPNET. The character format of the {esd\_inPkt, esd\_outPkt} field enables them to be processed as raw bytes.

Once OPNET receives the interrupt and reads the data packet, it sets the “esd\_inCtrl” field to a negative value, signaling the channel is available. A similar process occurs in the reverse direction for “esd\_outCtrl” and “esd\_outPkt”. A minor abuse of the “esd\_inCtrl” channel occurs during simulation initialization by Java controller to notify OPNET to begin the simulation. The control and packet channels demonstrate the ability to have asynchronous communication, in contrast to the synchronized approach with the status and position ESIs. Both modes were implemented to assess both ease of use and performance.

The packets exchanged via ESI use an additional signaling mechanism to maintain synchronization between the Java and OPNET clouds. The modes in use are specified via the network source and sink fields as listed in Table 2. A nodes sends packets via its respective Java/OPNET management cloud, i.e., a node exists on both the Jana and OPNET sides of the co-simulation.

Source	Destination	Communication Mode
+ID1	+ID2	route (ID1 → ID2), e.g., AODV
+ID1	0	broadcast (1x) ID1 → neighbors
+ID1	-ID2	broadcast (1x) ID1 → ID2
+ID1	-∞	flood — reserved for future use
-ID1	0	IPC (ID1 → cloud)
-ID1	-ID1	IPC (ID1 → ID1)
0	-ID1	IPC (cloud → ID1)
0	0	IPC (cloud → cloud)

Table 2: Simulation Communication Modes

Each communication mode operates as follows. The first mode, (+ID1, +ID2), is used when a Java application layer operating on ID1 needs a packet routed to ID2 using the currently specified routing protocol such as AODV, TORA, or DSR. A node uses mode (+ID1, 0) to request a single broadcast of a packet. Mode (+ID1, -ID2) is similar, except that neighbor nodes other than ID2 drop the packet and do not deliver it to their corresponding Java layer. This determination occurs in the OPNET cloud after the packet has been broadcast received. Since there is currently no wireless network layer flooding protocol in OPNET, we use mode (+ID1, 0) in coordination with the application layer to achieve flooding. Mode (+ID1, -∞) is reserved in the event we implement a more robust flooding protocol.

The remaining modes allow inter-process communication across co-simulations between autonomous nodes. In the event a node needs to advise the cloud of information, such as its state, then mode (-ID1, 0) is used. The reverse mode, although it is not generally used, is denoted as (0, -ID1). If a Java node needs to correspond with its OPNET counterpart (or vice versa), then mode (-ID1, -ID1) is available. Finally, in the event that the clouds need to exchange information beyond node status and position, mode (0,0) is available — this enables extensibility via new packet definitions versus ESD and ESI modifications. There are no backdoor inter-process communications modes such as (-ID1, -ID2) — nodes communicate through a network mode.

Thus, if the cooperative search application layer in Java determines that it has just explored a cell, it constructs a “cell\_explored” packet and notifies the Java cloud that needs to transmit the new packet in mode (+ID1, 0). The Java cloud then sends the packet to the OPNET cloud. The OPNET cloud then inserts the packet at the network layer stream of the transmitting node in broadcast mode. The receiving neighbors will then deliver the packet back to their corresponding Java nodes in mode (-ID1, -ID1). If the cooperative search protocol determines that this is new information, it will send the packet back through the Java cloud in mode (+ID1, 0).

The goal of these modes is to provide a mechanism that enabled any application layer algorithm to be implemented externally, yet still use OPNET to model actual packet communication.

The packet format used for the “esd\_inPkt” and “esd\_outPkt” fields is shown in Table 3. Each packet contains 4,000 bytes; the numbers in parentheses indicate the field size (in bytes). The “net source” and “net sink” are used as specified in Table 2. The “app\_source” and “app\_sink” fields are reserved for future use in application-layer protocols. The “payload length” field gives the number of bytes in the payload field that contain useful data. The “payload type” is a designator to provide for different payload packets within JOCosim. We have thus far defined only one “payload” type, the “cell\_explored” payload. This payload contains a 4-byte integer that lists the cell identifier a UAV has explored and is used as part of the cooperative search algorithm.

Net Source (4)	Net Sink (4)	
App Source (4)	App Sink (4)	
Payload Length (4)	Payload Type (1)	Reserved (3)
Payload (3976)		

Table 3: JOCosim Cloud Exchange Packet

### OPNET Modifications

The primary node model in JOCosim is the “manet\_station\_adv” model, shown in Figure 10. The only changes made to this node model were in the “traf\_src” process model, shown in its original form in Figure 11. This same node model was used in [17] due to mobility, packet generation, and wireless requirements.

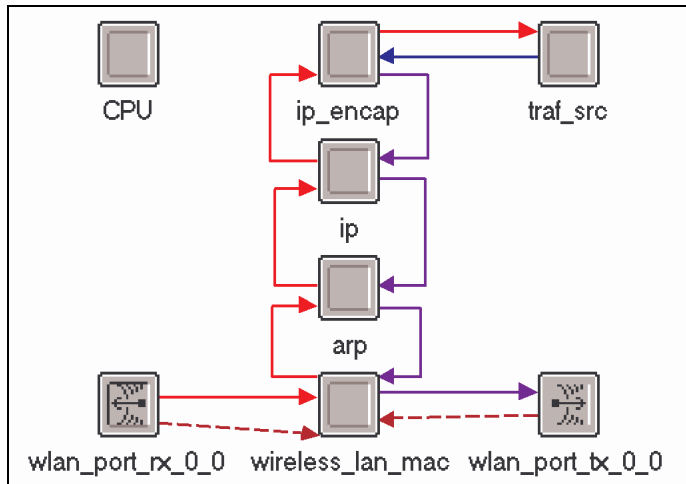


Figure 10: Node Model – Advanced MANET Station

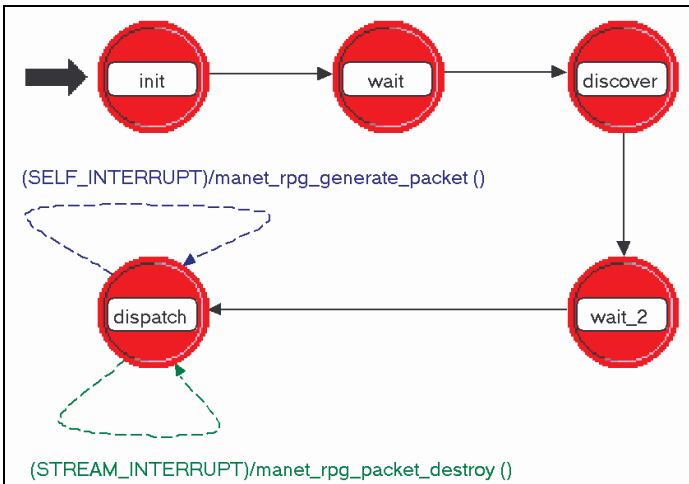


Figure 11: Process Model – Traffic Source

The changes that were made to the traffic source model are listed in Figure 12. The numbers in parentheses give the approximate lines of code required to implement the modification. The packet generator is disabled via deletion of the self-interrupt transition that calls the “manet\_rpg\_generate\_packet”. Packet destruction is replaced with OPNET cloud delivery — a 1-line replacement in the “manet\_rpg\_packet\_destroy” function. Statistics were added (at our discretion) to track packet traffic and are not strictly necessary. The statistics modifications only define the probes — statistics updates are strictly performed in the cloud to minimize node-level modifications. This enables us to specify the statistic handles via standard probe model (Figure 7).

- Remove packet generator (1)
- Disable packet destruction (1)
- Add packet delivery to cloud (1)
- Add packet format statistics handles (80)

Figure 12: Process Model Changes — Traffic Source

To further limit the changes required to the traffic source process model, when a node is held in reserve or destroyed, the OPNET cloud lowers the UAV’s receiver sensitivity and sets its transmit power to zero. This effectively removes a node from a running simulation without requiring the implementation of FAILURE or RECOVERY interrupts. The cloud drops any packets that are queued for the node during the time it is inactive. This approach simplifies statistics collection when a reserve node is activated.

The process model of the cloud node is shown in Figure 8. The “init” state obtains ESI interfaces, node addresses, and generates the cell coverage squares. The “wait\_Cosim” state waits until it receives a signal on the “esd\_inPkt” ESI (Table 1). The “init2” state reads the status and position information and updates the animation window. The cloud then transitions to the “idle” state.

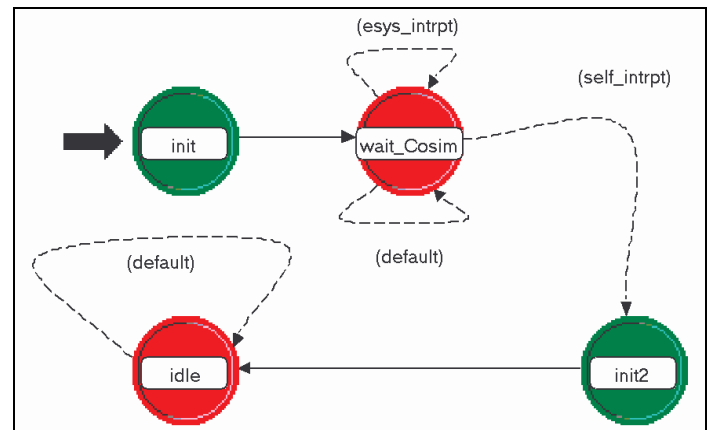


Figure 13: Process Model – OPNET Cloud

The “idle” state is a slight misnomer, as all node management within OPNET occurs in this state, to include inserting packets, communicating with the Java cloud via the ESIs, and animation updates. All interrupts are handled via selection statements based on the ESI channel and/or the communication mode. If a packet is received from Java that must be transmitted via the network, the cloud inserts it at the “ip\_encap” stream (Figure 10) of the transmitting node. When a node receives a packet, it forwards the packet to the OPNET cloud for delivery via the C wrapper to its corresponding application-layer node in Java.

## JNI Development

From OPNET’s perspective, it is unaware that it is participating in a co-simulation. Although the cloud node is communicating via the ESI, the OPNET network model does not explicitly link with external libraries. This is a significant enhancement, as the version in [17] explicitly required OPNET to link to an external library. In this version, the Java controller, via the C wrapper library constructs, loads, injects data, and executes a simulation.

This approach is reflected in the system definition (SD) file that is specified in the ESD, informing OPNET that it is participating in a co-simulation (Listing 1), but that it does not have to be aware of the library it is coordinating with, as all communication is via the ESI. To link with OPNET, the C wrapper library links with the OPNET-provided *esa.h* header file in the “C:\Program Files\OPNET\10.5.A\sys\include” directory (or its equivalent). This header file defines the various constants and methods that an external co-simulation can use to interact with OPNET.

```
start_definition
  platform: windows
#  bind_objs: jocosim_CoSimOPNET.obj -- not needed
  use_esa_main: no
end_definition
```

**Listing 1: System Definition (SD) File**

The enabling technology for our co-simulation is the Java Native Interface (JNI). The JNI specification exists to enable linking with libraries developed in other languages, most notably C. Although JNI provides a means to link with external libraries, the specification is complex and its use is non-trivial. JNI’s use is recommended only when the cost to implement functionality within Java is prohibitive, for instance, a complete Java port of OPNET! A brief overview of JNI is provided here; additional material is in [3–6].

The user must install the Java Runtime Environment (JRE) and the Java Development Kit (JDK), and then make the path variable changes in Table 1. In addition, a C compiler will also be required — we used Microsoft Visual C.

Program	“PATH” Environment Variable Additions
Java	C:\Program Files\Java\jdk1.5.0_06\bin C:\Program Files\Java\jdk1.5.0_06\include C:\Program Files\Java\jdk1.5.0_06\include\win32 C:\Program Files\Java\jre1.5.0_06\bin\client
MS Visual C	C:\Program Files\Microsoft Visual Studio\VC98\Bin
OPNET	C:\Program Files\OPNET\10.5.A\sys\pc_intel_win32\bin

**Table 4: Path Environment Variable Modifications**

Since the JNI obfuscation specification is difficult to implement manually, we strongly recommend developers use the Java-provided *javap* and *javah* utilities to determine the Java and C method signatures, as described here. For example, although a “class.method” notation is used in Java, JNI extensions replace a ‘.’ with a ‘\’ or ‘\_’ in some contexts, increasing its complexity. Additionally, each Java method name is extended and a unique signature such as “(I)V” is created. The ‘I’ specifies an integer parameter and the ‘V’ (void) signifies that no data is being returned. The most challenging aspect lies in exchanging strings due to Java’s use of Unicode relative to native C; this particular aspect is beyond the scope of this paper.

JOCosim requires that Java be able to call the C wrapper library to control OPNET. To retain control, it is necessary for the C library to register callbacks To call Java from C (Listing 2), a method is first defined in Java and the Java source code is compiled with *javac*. The *javap* utility extracts the obfuscated JNI method signatures; we store them in a file named *class\_name.s*. The “jEnv” pointer marks a Java environment. Every call from Java to C passes an environment, with the caveat that each Java thread has its own environment, so it is simplest for Java to register with C. The method ID is obtained using the “GetStaticMethodID” function. Finally, the Java method is called from C via “CallStaticVoidMethod”.

```
Java: Define Method
public static void xSimSleep(int ms)...

Java: Compile & Extract Signatures
javac callbacks
javap -s callbacks > callbacks.s
type callbacks.s
  public static void xSimSleep(int);
  Signature: (I)V

C: Find Method (use signature from callbacks.s)
jMIDs[JVM_SLEEP] = (*jEnv)-> GetStaticMethodID
(jEnv, jCls, "xSimSleep", "(I)V");

C: Call Method
(*jEnv)-> CallStaticVoidMethod
(jEnv, jCls, jMIDs[JVM_SLEEP], 5000);
```

**Listing 2: Calling Java from C**

Listing 3 shows the other required functionality, calling C from Java. First, the Java method stubs are defined with the “native” keyword. After the Java code is compiled with *javac*, the *javah* utility is used to extract an obfuscated C header, *class\_name.h*. This header is included in the C source and self-includes the required (Java-provided) *jni.h* file. The C source code is then compiled to a dynamic link library (DLL). The obfuscated method name “Java\_Callbacks\_cSetSeed” refers to the original Java method, “Callbacks.cSetSeed”.

```
Java: Define Method Stub
private static native void cSetSeed(int seed);

Java: Call Method
private static native void cSetSeed(int s);

Java: Compile & Extract C Header
javac callbacks
javah callbacks
type callbacks.h
JNIEXPORT void JNICALL Java_Callbacks_cSetSeed
(JNIEnv *, jclass, jint);

C: Define Method (from callbacks.h)
#include <callbacks.h>
JNIEXPORT void JNICALL Java_Callbacks_cSetSeed
(JNIEnv *, jclass, jint);

C: Compile Dynamic Link Library (DLL) — all one line
CL /LD /MD /I.
/I"C:\Progra~1\opnet\10.5.A\sys\include"
/I"C:\Progra~1\Java\jdk1.5.0_06\include"
/I"C:\Progra~1\Java\jdk1.5.0_06\include\win32"
jocosim_CoSimOPNET.c opsim.lib /link
/LIBPATH:C:\Progra~1\OPNET\10.5.A\sys\pc_intel_win32\lib
```

**Listing 3: Calling C from Java**

To start a simulation, the Java controller executes the commands in Listing 4 via the C library. Although this information is in the OPNET’s documentation, the sequence is not given explicitly. The “argc” and “argv” fields in the “Esa\_Main” and “Esa\_Init” calls are the same as if launching an OPNET simulation from the command line — they specify a model and environment file. The STDOUT callback enables the Java output message pump. The other callback registration enables the C library to notify Java that data has been received from OPNET via an ESI.

```

Java Cloud
// Call SimInit in C wrapper
public static native int SimInit
    (int argc, String[] argv);

C Wrapper
// Load ESA_MAIN
Esa_Main (argc, argv, ESAC_OPTS_NONE);

// Register STDOUT Callbacks
Esa_Text_Output_Callback_Register (opnetEsaState,
    Callback_Feed_StdOut, NULL, -1);

// Load ESA_INIT
Esa_Init (argc, argv,
    ESAC_OPTS_NONE, &opnetEsaState);

// Load ESA_LOAD
Esa_Load (opnetEsaState, ESAC_OPTS_NONE);

// Register ESI Callbacks
Esa_Interface_Callback_Register (opnetEsaState,
    opnetStatus, esdIDs[5], Call_Recv_Ctrl,
    Call_Recv_Ctrl_Array, NULL);

```

Listing 4: Simulation Initialization

As is common in multi-threaded applications, synchronization issues were encountered. One issue is that the ESA methods are not re-entrant in OPNET, as even a terminate method called multiple times crashed the simulation. We used a mutex in these situations. To further simplify development, the C wrapper calls Java’s sleep method while waiting for to acquire a mutex.

```

C Wrapper
// Create MUTEX
static volatile int isBusy;

// Acquire MUTEX
while (isBusy) {Callback_Sleep(200);}
isBusy = TRUE;

// Get JNI status (Figure 10)
JNI_GetStatus(&env, &cl);

// Create Java string in UTF
jStr = (*env)-> NewStringUTF(env, s);

// Get Java method ID
jMID = (*env)-> GetStaticMethodID(env, cl,
    "callbackFeedStdOut", "(Ljava/lang/String;)V");

// Call Java method to push text
(*env)-> CallStaticVoidMethod(env, cl, jMID, jStr);

// Release Java string to avoid memory leak
(*env)-> ReleaseStringUTFChars(env, jStr, s);

// Release MUTEX
isBusy = FALSE;

```

Listing 5: OPNET Thread Synchronization

For various reasons including synchronization, endian order, and OPNET design, exchanging packets between Java and OPNET was our most significant challenge (Listing 6). First, the packet is constructed in Java, the easiest step. The packet is inserted in the outbound packet queue (not shown) and scheduled for future delivery in the C wrapper. The OPNET cloud examines the “net\_source” and “net\_sink” fields to check the communication mode upon receipt. In this example, a packet is inserted at the “ip\_endcap” inbound stream via “op\_pk\_deliver\_forced”.

```

Java Cloud
// set fields
pktJOCosim.netSource = 0;

// allocate ByteBuffer
java.nio.ByteBuffer buf =
    java.nio.ByteBuffer.wrap(new byte[PKT_SZ]);

// set endian order
buf.order(order);

// stuff fields in packet
buf.putInt(this.netSource);
...

C Wrapper
const int len = (*env)-> GetArrayLength(env, pk);
char * pkB = malloc(sizeof(char) * len);

(*env)-> GetByteArrayRegion(env, pk, 0, len, pkB);

// send data
Esa_Interface_Array_Set(opEsaState, &opStatus
    esdIDs[4], ESAC_NOTIFY_NEVER, (void *)pkB, 0);

// send notification
Esa_Interface_Value_Set(opEsaState, &opStatus
    esdIDs[3], ESAC_NOTIFY_IMMEDIATELY, 1);

OPNET Cloud
// Get packet array
op_esys_interface_array_get(esi_pktIn, pktIn, 0);

// Read info from packet
intPtr = (void*)&pktIn[offset];
src_dex = *intPtr;
...

sent_stats(src_dex-1, -1, pksize, pkt_type);

// Send packet to node
for(i=0; i<pksize; i++) {
    vv2 = op_vvec_create(OPC_TYPE_BIT);

    op_vvec_from_value(&vv2, JOC_ENDIAN_TYPE,
        OPC_VVEC_BITFORMAT_BINARY, 8,
        OPC_VVEC_NATIVE_UCHAR, pktIn[i]);

    op_vvec_insert(vv, OPC_VVEC_VECTOR_END, vv2);

    op_vvec_destroy(vv2);
}

// Create packet, put vvec inside & check size
pkpPtr = op_pk_create_vvec();
op_pk_vvec_set (pkpPtr, vvec);
op_pk_total_size_set(pkpPtr, 8*pksize);

// Get src/dest IP addresses, Set ICI (not shown)

// Deliver packet
op_pk_deliver_forced (pkpPtr, send_ids[src_dex], 1);

```

Listing 6: Packet Exchange Source Code

Returning to the multi-threading issue, we used the source code in Listing 7 to guarantee that we were referencing the correct Java environment from within the C library. We operated under the assumption that any number of threads may be calling the C library. We always had at least two threads that might be calling the wrapper library — the controller thread and the user interface thread to deliver asynchronous early simulation termination.

```
JNI_GetStatus (JNIEnv ** env, jclass * cl) {
    // Get JNI environment (JNIEnv)
    (*jVM)-> GetEnv(jVM, (void **)env,
        JNI_VERSION_1_2);

    // Get Java Class (jclass)
    *cl = (**env)-> FindClass(*env,
        "jocosim/CoSimOPNET");
}
```

Listing 7: Obtaining JNI Thread Environment State

### Conclusion

This implementation of JOCosim demonstrates that Java can be successfully integrated with OPNET in a meaningful way via the JNI specification. Although this version is only a prototype, we believe its development will enable useful co-simulations to be constructed. In particular, it enables students at our institution to develop application-layer algorithms in an external library and assess their performance within a network simulation.

There have been several difficulties that we have encountered thus far. At the time of this writing, our most significant issue is the migration from OPNET 10.5 to OPNET 11.5. In our assessment, this is attributable to changes in the wireless model and a design decision. As discussed previously, we choose to adjust node power and receive levels versus implementing FAILURE and RECOVERY interrupts to minimize source code length. However, the precise attributes and interfaces that this technique to work have changed.

Additional steps that are awaiting completion include the support for additional packet payloads and a more refined timing loop in the Java controller. Although we have extracted the cooperative search algorithm from OPNET as a stand-alone C library, we are currently only providing random positions versus using it within JOCosim. A further step is the actual integration of a tool such as MATLAB or a stand-alone flight simulator as shown in the JOCosim design framework of Figure 5.

### Acknowledgement

The authors thank the Air Force Communications Agency for partial funding of this research.

C. Augeri and K. Morris thank Scott Graham for significant latitude on the assigned “individual” project.

### References

- [1] *High-Level Architecture (HLA)*. [Online] <https://www.dmsomil/public/transition/hla/>.
- [2] *Java 5.0*. Sun Microsystems. [Online]. <http://java.sun.com/>.
- [3] *Java Native Interface (JNI)*. Sun Microsystems. [Online] <http://java.sun.com/docs/books/jni/>.
- [4] *JNI Calling Java Methods*. Sun Microsystems. [Online] <http://java.sun.com/docs/books/tutorial/native1.1/implementing/method.html>.
- [5] *JNI Invocation Interface*. Sun Microsystems. [Online]. Available: <http://java.sun.com/docs/books/jni/html/invoke.html>.
- [6] *JNI Loading the JVM*. Sun Microsystems. [Online]. Available: <http://java.sun.com/docs/books/tutorial/native1.1/invoing/example-1dot1/invoke.c>.
- [7] *OPNET Discrete Event Simulation*. Chaps. 2–9,15,24. Ver. 10.5+.
- [8] *OPNET Editors Reference*. Chap. 6. Ver. 10.5+.
- [9] *OPNET Model File Access API Reference*. Ver. 10.5+.
- [10] *OPNET Modeling Concepts Reference*. Chaps. 1,5,8. Ver. 10.5+.
- [11] *OPNET Simulation Control API Reference*. Ver. 10.5+.
- [12] *OPNET Wireless Module User Guide*. Chap. 2. Ver. 10.5+.
- [13] Session 1532. “Interfacing Multiple Simulators Using the OPNET Co-Simulation API,” *OPNETWORK*. 2005.
- [14] Session 1901. “Co-Simulation with OPNET Using High-Level Architecture (HLA),” *OPNETWORK*. 2004.
- [15] Session 1933. “Hardware-in-the-Loop (HIL) Co-simulation with OPNET,” *OPNETWORK*. 2005.
- [16] Session 1938. “Effectively Planning and Implementing HLA Federations using OPNET,” *OPNETWORK*. 2005.
- [17] C. Augeri, K. Morris, and B. Mullins. “HARVEST: A Framework and Co-Simulation Environment for Analyzing Unmanned Aerial Vehicle Swarms”, (accepted), *Military Communications Conference (MILCOM) 2006*. Washington, D.C.
- [18] V. Dham. *Link Establishment in Ad Hoc Networks Using Smart Antennas*. M.S. Thesis, Virginia Polytechnic Inst. & State Univ., Jan 03.
- [19] K. Fujita. *Extending OPNET Modeler with Client Profiles for Selecting Data Sources in WAN*. University of Pittsburgh. [Online]. Avail.: <http://db.sis.pitt.edu/projects/Nebula/public/opnet/report2.html>.
- [20] K.M. Morris, B. Mullins, D. Pack, G. York, and R. Baldwin, “Impact of Limited Communications on a Cooperative Search Algorithm for Multiple UAVs”, In *Proc. of the IEEE Intl. Conference on Networking, Sensing and Control*, Fort Lauderdale, FL, April 23-25, 2006.