Implementing a DIS-like Federation within the HLA

Len Granowetter
MÄK Technologies
10 Fawcett Street, Cambridge, MA 02138
email: lengrano@mak.com

KEYWORDS
DIS-like Federation, HLA, Implementation, Common Software

ABSTRACT

This paper discusses some of the issues involved in implementing a DIS-like federation within HLA. The process entails defining a Federation Object Model for the federation, developing software that implements the protocol described by the FOM, and modifying simulator code to use this software in order to interact with the other federates.

FOM development must be done only once per federation, and the FOM shared by all of the federates. Implementing protocol-specific software could be done independently by the author of each federate, however since this software is common to all federates, time and money can be saved by implementing it once in a library which is used by all applications.

Based on MAK's experience porting its VR-Link Networking Toolkit from DIS to HLA, we describe an implementation that attempts to be as resilient as possible to FOM changes, and highlight the way that many DIS concepts translate to HLA.

INTRODUCTION

Over the next several months and years, many DIS-style simulators (where the objects being simulated are individual battlefield entities, and an application's simulation time is not constrained by the actions or clocks of other applications) will need to be ported to HLA.

The process involves at least three steps:

- Defining a Federation Object Model (FOM) for the Federation. This must be done only once for the federation, as the FOM is shared by all federates.
- Developing software that implements the protocol specified in the FOM.
- Modifying simulator code to use this software layer in order to interact with the other federates.

Technically, the second and third steps don't need to be distinct. FOM knowledge can be built directly into simulator code. However, there are several reasons that creating a separate layer with FOM knowledge is a good idea, including:

- Confining FOM-specific knowledge to a single software module insulates the application to the greatest degree possible from changes in the FOM.
- Since all applications in the federation are using the same FOM, software that implements the objects and interactions described in the FOM can be placed in a library and shared by all federates. This layer of software has often been called Middleware, or Common Federation Software.

DEFINING A DIS-LIKE FOM

HLA's power lies in the fact that rather than providing a protocol, it provides a framework for developing and using protocols. Any federation is free to create its own protocol, tailoring the kinds of objects and attributes to its own problem domain.

However, this does not mean that everyone who is implementing a federation needs to write his own protocol. Not only would this mean large amounts of duplicated work, but it would mean that applications could only participate in the federation for which it was originally developed, compromising interoperability.

Since a great many DIS-style federations will have similar protocol needs (after all, they've shared the DIS protocol for years), it stands to reason that the community will come to some agreement on one or more reference FOMs based on the DIS standard.

Creating a FOM based on the IEEE 1278.1-1995 DIS Specification is not as straightforward as it would appear, however. As evidence, consider that three efforts to do just that - STOW, JPSD, and the Plug and Play FOM group - have taken three different tacks. Although we stress that we don't think that most federation implementers will be developing FOMs, we would like to point out some of the issues that FOM developers will have to consider as they create DIS-like FOMs.

HLA has replaced the concept of PDUs, with two other concepts - interaction classes and object classes.

Probably the easiest way to define a DIS-like FOM, is to just say:

1. For each DIS PDU that represented the update of an object's state, create an Object Class in the FOM. Give the Object Class one attribute, whose type is a structure exactly mirroring the PDU.

2. For each DIS PDU that represented an interaction between objects, create an Interaction Class in the FOM. Give the Interaction Class one parameter, whose type is a structure exactly mirroring the PDU.

This scheme has the advantage that it will probably require the least modification of code for most existing DIS applications, but has the disadvantage that it requires that just like in DIS, when one attribute needs to be updated, we must send out new values for *all* attributes.

A second scheme would be to continue to map each PDU to either an Object Class or an Interaction Class, but say that each field of the PDU becomes its own attribute (or parameter).

A variation on this theme, is for each PDU, to make judgments about which fields should be grouped together into a single attribute. For example, for Entity Objects, we might want to say that Location, Velocity and Acceleration must always be updated together.

These choices allow the federation to take advantage of the HLA's bandwidth saving capability to only update attributes when necessary.

FOM developers could choose to get more advanced, and lose the one-to-one mapping between DIS PDUs and FOM classes. For example, instead of just mapping Entity State PDUs to Entity Objects, we could create subclasses of the Entity Object class. A Tank object class might have attributes to describe the orientation of the turret, and elevation of the gun, eliminating the need for such huge generality in Articulated Part Records.

For interactions, perhaps there could be a subclass of the Detonation Interaction Class, called EntityHit, eliminating the need to send a target entity ID for detonations which did not occur at an entity.

Rather than make a judgment about which approach to DIS-like FOM development is best, we recognize that different choices have different merits, and there may be several different FOMs that become standards to be used in different situations. Because of this, it is important for software developers to structure their federates such that they may switch among different FOMs with a minimum of effort.

IMPLEMENTATION ISSUES

This section is based on our experiences porting MAK's VR-Link Networking Toolkit to the HLA paradigm. VR-Link has served as "common federation software" for DIS for several years, and it now serves that purpose as well for HLA. In our design, we had an additional requirement that we continue to support DIS, but we do not discuss the issues introduced by that requirement here.

Throughout, we assume the use of an RTI built to version 1.0 of the HLA Interface Specification, such as RTI F.0. The code we present is C++, but the approach is not limited to applications written in C++. In addition, we do not focus here on software architecture, rather, we present the implementation issues that would arise regardless of architecture.

Type safety and platform independence

From an application's perspective, we believe that common federation software should provide at least the following things:

A typesafe, platform-independent interface to sending and receiving interactions.
A typesafe, platform-independent interface to the current state of all remote objects.
A typesafe, platform-independent interface for setting the current state of all locally simulated

object, and the ability to publish updates when necessary.

The RTI's interface to inspecting and publishing data is, by design, very generic. When interaction data arrives, it is passed to the receiveInteraction callback in the form of an ParameterHandleValuePairSet (PHVPS). We can request the values of particular parameters by ParameterHandle. However, since the RTI assigns a handle to each parameter at run-time, it is not recommended that an application use hard coded ParameterHandles. Instead, we can hard-code parameter names, and use RTI::getParameterHandle to find out the handle being used for each attribute at run-time.

It is not a good idea for application code to depend even on parameter names for two reasons: First, application code would need to be modified if parameter names in the FOM change. Besides gratuitous name changes, this includes the possibility that a complex parameter is broken up into two or more component parameters, or conversely, that several attributes are combined into a single structure. Second, if an application author makes a mistake in the name of a particular parameter, he would not be notified until run-time, and only then if he catches the right exceptions.

Figure 1 contains sample code for a function called processFireInteraction that converts the values in a PHVPS to a form that is more convenient to application developers, and stores them in a structure:

Note that as compared to DIS, HLA requires an extra copy in getting an attribute value from the network into a slot in native-representation structure. In DIS, a PDU's network representation gets copied from kernel space to user space when we use a system call such as *recvfrom*. We can then convert each field from network to native representation and store the result in a structure or class that provides an interface to an application.

In HLA, data is copied from kernel space to user space when the RTI makes a system call to read data from the network and stores it in an AttributeHandleValuePairSet or ParameterHandleValuePairSet. Now, as RTI users, we can not immediately convert each parameter (or attribute) to the desired representation and store the result in its final destination, as the RTI API does not provide us with pointers to the values of the parameters. Instead, RTI::ParameterHandleValuePairSet::getValue requires us to pass in a buffer to which the value will be copied (in network representation, of course.) Only then can we convert to native representation, and store the result where we would like.

There need not necessarily be a one-to-one relationship between the parameters of the Fire Interaction Class described in the FOM, and the members of the FireInteraction structure that we use as a typesafe interface to provide to applications. For example, the FOM might indicate that a fire interaction only has one parameter, which is a structure that looks exactly like a DIS Fire PDU. In this case, our FireInteraction structure would not have to change, nor would the application that uses it. Only the implementation of processFireInteraction would need to be modified to reflect the fact that only one attribute must be pulled out of or pushed into the PHVPS.

The sending side is very analogous. Figure 2 contains sample code for a function called generateFireInteraction, that will convert from native representation to a PHVPS.

(By the way, better implementations of these two functions are certainly possible. Specifically, implementing processFireInteraction using a table mapping parameter handles to processing functions would be more efficient than the if-then-else construction described.)

Turning from interactions to object state, we have a similar job to accomplish. On the receiving side, processEntityStateUpdate would look very similar in structure to processFireInteraction - for each parameter in the AttributeHandleValuePairSet (AHVPS), perform the necessary network to host representation conversions, and store the result someplace accessible to an application. On the outgoing side, however, there is a new wrinkle.

For interactions, we know that before sending, we need to set values for all parameters. In contrast, attribute updates need not contain all of the attributes for a particular object. Because of this, a function along the lines of generateEntityStateUpdate needs to know which attributes to include in the AHVPS. This can be achieved by requiring that a list of AttributeHandles be passed to the function.

Figure 1: processFireInteraction

```
typedef struct
{
  EntityId attackerId;
  EntityId targetId;
  ...
  Vector location;
  ...
} FireInteraction;

void processFireInteraction(const RTI::ParameterHandleValuePairSet
                                    &params, FireInteraction *nativeRep)
{
  static char buffer[MAX_PARAM_LENGTH];
  static unsigned long length;
  for (int i = 0; i < params.size(); i++)
  {
    RTI::ParameterHandle h = params.getHandle(i);
    RTI::ParameterName pName = RtiAmb().getParameterName(h, fireClassHandle);

    /* Alternatively, we can cache the mappings between
      parameter names and handles, and switch on handle instead of name here */

    if (!strcmp(pName, "targetId"))
    {

      /* Pull out the value of targetId, in network representation */

      params.getValue(i, buffer, &length);
      Int16 *intArr = (Int16 *) buffer;

      /* TOHOST would, for example, do byte swapping if we're on a little endian machine.
        Assume targetId is a typed native-representation object */

      nativeRep->targetId.site = TOHOST(intArr[0]);
      nativeRep->targetId.host = TOHOST(intArr[1]);
      nativeRep->targetId.entity = TOHOST(intArr[2]);
    }
    else if (!strcmp(pName, "location"))
    {
      /* convert the data to host format and store it in nativeRep->location. */
    }
    /* and so on for the other parameters */
    ...
  }
}
```

Although this means that FOM knowledge is no longer confined to generateEntityStateUpdate, the only user of that function will be the layer that performs thresholding and decides what attributes need to be updated. This layer would also be part of common federation software, meaning that application writers would still be shielded from this FOM knowledge. If one truly wanted to confine FOM knowledge to generateEntityStateUpdate, a scheme could be devised for mapping native-representation, typed attributes to HLA attributes - allowing the list of attributes to be specified in a FOM-independent way.

Figure 2: generateFireInteraction

```
RTI::ParameterHandleValuePairSet * generateFireInteraction(FireInteraction *nativeRep)
{
  RTI::ParameterHandleValuePairSet *returnVal = RTI::ParameterSetFactory::create(NUM_PARAMS);
  static char buffer[MAX_PARAM_LENGTH];
  Int16 *intArr = (Int16 *) buffer;

  /* TONET is a macro that does byte swapping if we're on a little endian machine. */

  intArr[0] = TONET(nativeRep->targetId.site);
  intArr[1] = TONET(nativeRep->targetId.host);
  intArr[2] = TONET(nativeRep->targetId.ent);
  returnVal.add(RtiAmb().getParameterHandle("targetId", fireClassHandle), buffer, 3 * sizeof(Int16));
  /* and so on for the rest of the parameters */
  ...
  return returnVal;
}
```

Heartbeat

In DIS, entities were required to publish an Entity State PDU periodically even if no attributes had changed, in large part to make sure late arriving applications received information about all of the existing entities in the exercise in a timely fashion.

This need can be satisfied in a different fashion in the context of HLA. Federates may make a call to RTI::requestClassAttributeValueUpdate after joining a federation execution, requesting that all existing objects of a particular class publish all needed attributes.

Timeouts

Timeouts are a stickier issue. The problem is keeping the application's database of objects consistent with the RTI's, in the face of the following: The RTI lacks the concept of objects timing out, while it is a concept that is necessary for most DIS-style simulations. We don't want to keep drawing a tank moving right off the edge of our terrain, when it stopped sending updates a long time ago without notifying the RTI.

When an object times out, (from the application's perspective), we can not just delete all information relating to it, as we might have done in DIS. If the object starts publishing updates again, a new call to discoverObject will not be generated by the RTI, since from the RTI's perspective, the object never left.

This means that we must at least remember to which object class the object ID belonged. We might also need to remember its last state, as there's no guarantee that the next update will contain all necessary attributes. Alternatively, we can use requestObjectAttributeValueUpdate to query for the necessary attributes when we realize that some of the "forgotten" data was not present in the update we received.

Dead Reckoning

In DIS, in order to compute up to date dead-reckoned values for location, velocity, and orientation, one only needed to know the time of validity of the last PDU for an entity, and the current time. With HLA, depending on the layout of a FOM, the parameters that enter a dead-reckoning equation may have been last updated at different times.

While we expect that most DIS-like FOMs will require that dead-reckoning parameters be updated together, software that can handle the general case will be more robust.

It is not sufficient to merely keep track of the time of validity of each of the parameters individually. This following scenario points out the problem: At $t = 0$, we receive an update that says location is $\{0,0,0\}$, and velocity is $\{1,0,0\}$. Then at $t = 1$, we receive an update that says that velocity is now $\{2,0,0\}$. When we attempt to compute the dead-reckoned position at $t = 2$, we would say: $Loc = \{0,0,0\} + (2-0) * \{2,0,0\} = \{4,0,0\}$. In actuality, the correct position should be $\{3,0,0\}$, as the entity has traveled at the velocity of $\{1,0,0\}$ for one second, and the velocity of $\{2,0,0\}$ for one second.

What we will need to do is to update our stored value for location with the current dead-reckoned value, every time a new value for velocity is received. Similarly, we will need to update our stored values for both position and velocity when we receive a new value for acceleration. This allows us to store a single time of validity for all three parameters. We can now use the normal dead-reckoning equation to compute subsequent positions.

Distributed Objects

DIS does not make it very easy to distribute the simulation of different attributes of a single entity across several applications. (One way would have been for the applications to communicate with each other through some non-DIS mechanism, delegating the responsibility to one of them to publish Entity State PDUs to the exercise). In HLA, distributed object ownership is a fundamental concept, however the RTI's interface introduces the following issues:

Declaration of the intent to publish attributes is done on a per class basis, using publishObjectClass, rather than on a per-object basis. Therefore a software module that is capable of publishing certain attributes of an object can not be responsible for declaring the intent to publish those attribute. There might be a second module capable of publishing other attributes, and if they both declare the intent to publish attributes of the same object class, only the last declaration is valid. The first module will cause an exception to be thrown when an attempt is made to actually publish an attribute.

Even if both modules make their declarations through a wrapper function that ensures that the RTI always knows about the union of all modules' declarations, problems arise. Because each registered object begins with ownership of all attributes being published by the application, a module can not know the set of attributes of which he should divest ownership, without knowing the set of attributes that the application as a whole has registered to publish.

SUMMARY

The conversion of simulators from DIS to HLA involves many difficult issues, both in FOM development and software development.

We have outlined some of the choices FOM developers can make, pointed out how common federation software hides these choices from the application, and shown how these choices affect the implementation of common federation software.

In addition, we have discussed some new approaches that are necessary for dealing with late arrivals, timeouts, dead-reckoning and distributed objects.

REFERENCES

Department of Defense High Level Architecture Run-Time Infrastructure Programmer's Guide Version F.0, December 16, 1996.