A Tcl Binding for HLA/RTI

William H. Duquette
Jet Propulsion Laboratory, California Institute of Technology
William.H.Duquette@jpl.nasa.gov

ABSTRACT

HLA/RTI is a "High Level Architecture" with accompanying "Run-Time Infrastructure" designed by the U.S. Department of Defense to support the creation of networks (called *federations*) of cooperating simulation applications (called *federates*). The RTI software layer provides basic inter-task messaging, as well as management of simulated time across federates. RTI APIs are commercially available for C++ and Java. The paper presents two Tcl APIs, a low-level binding to the C++ API which adheres to HLA/RTI naming conventions, and a higher-level API built on top of it which is intended for actual use. The paper includes an introduction to HLA/RTI, but the main focus is on the design and implementation of the two APIs, which were developed as part of an HLA/RTI federate called the Joint Non-kinetic Effects Model (JNEM).

1. Joint Non-kinetic Effects Model

The Joint Non-kinetic Effects Model (JNEM) is a military training simulation which participates in a network of simulations used to train military commanders. The network, or *federation*, is called the Joint Land Component Constructive Training Capability (JLCCTC) Multi-Resolution Federation (MRF). JNEM's role as a *federate* in the federation is to model the responses of the civilian population, thus adding *non-kinetic* effects to the *kinetic* effects modelled by the battlefield simulation. JNEM is written primarily in Tcl/Tk 8.4 with a small amount of code in C/C++.

2. HLA/RTI

In 1993, the Department of Defense proposed the definition of a "High Level Architecture" (HLA) for simulation programs which would facilitate linking multiple simulations interoperably into a single federation; the work was a follow-on to the previously developed Aggregate Level Simulation Protocol (ALSP) and Distributed Interactive Simulation (DIS) architectures, and was intended to replace them both. Eventually a standard communications API, the Run-Time Infrastructure for HLA (RTI) was defined. The RTI allows federations to be created, programs to join a federation and thus to become federates, intercommunication between the federates, and synchronization of the federates in simulated time.

The JLCCTC MRF federation uses RTI as its communication infrastructure; specifically, it uses RTI NG Pro V3.x, sold by Virtual Technology Corporation. [1]

2.1 Interaction Classes

The most basic type of communication between federates is by means of *interactions*. An interaction is a dictionary of values which is broadcast by one federate and received by subscribing federates. Every interaction that is sent belongs to a particular interaction class; an interaction's class determines the names and data types of the interaction's *parameters*. Parameters may be scalar-valued, array-valued, structure-valued, and so forth. Note that the RTI API treats all parameter values as byte-arrays; each federation must agree on an encoding scheme, and it is the responsibility of each federate to encode and decode parameter values accordingly.

Each federation's interaction classes are defined in a file called the Federation Object Model, or FOM. Interaction classes are defined in a single-inheritance hierarchy, which is reflected in the class names; interaction classes REPORT.TEMPERATURE and REPORT.HUMIDITY are both subclasses of interaction class REPORT and inherit its parameter definitions. Note that interaction classes are not classes in the OOP sense, as they define data only, not behavior.

RTI has its own peculiar nomenclature, which is reflected in all of the API calls. A federate which wishes to send interactions of a particular class must declare its intent by *publishing* the interaction class. A federate which wishes to receive interactions of this class must similarly declare its intent by *subscribing* to the class. Then, when the first federate *sends* the interaction, all subscribed federates *receive* it.

The sending federate may choose to send all of the interaction's parameters or only a subset. Note that the RTI API has no notion of sending an interaction to a specific federate; in principle, every federate can receive every interaction. In practice, any given federate is usually interested in a small subset of the available interactions and subscribes only to those.

Interactions have two general uses. First, an interaction may indicate that some event has just occurred in the sending federate's simulation; the interaction's class will indicate the nature of the event, and the parameter values the details. Second, interactions are used to implement *orders*, messages that direct another federate or federates to take a particular action. In this case the interaction class indicates the nature of the action and the parameters any required details. One gathers that the former use was the one primarily intended by the designers of RTI, as interactions are poorly suited to positive closed-loop control; unfortunately, RTI provides no better mechanism for this purpose.

A suscriber may subscribe to any class in the hierarchy, and will get all interactions belonging to that class or its subclasses. Continuing the above example, if a federate subscribes to class REPORT it will receive all interactions of class REPORT, REPORT.TEMPERATURE, and REPORT.HUMIDITY. However, for interactions of the latter two classes it will only receive the subset of their parameters defined in class REPORT, which is probably not what was wanted. In the MRF, federates tend to subscribe to the leaf classes.

2.2 Object Classes

In addition to sending interactions, a federate may also publish data relating to simulation objects, e.g., military units. From the

RTI point of view, an object, like an interaction, is a dictionary of values. The difference is that objects have an identity and a lifespan where interactions are anonymous and transient. Objects belong to classes, just as interactions do, and as with interactions the FOM defines a single inheritance hierarchy of object classes. The naming convention is the same; GROUND.MANEUVER and GROUND.CONVOY are two subclasses of the base class GROUND. Note that objects have no more behavior, so far as RTI is concerned, than interactions do.

Where an interaction has parameters, an object has *attributes*. Attribute values, like parameter values, may be arbitrarily complex and are sent as byte arrays, with encoding and decoding left to the application. A federate may publish or subscribe to all or a subset of an object class's attributes.

The nomenclature for objects is as follows. First, a federate declares its intent to publish objects of a certain class by *publishing* the class; the operation specifies the class name and the names of some or all of the class's attributes. Note that *publish*, in RTI, refers only to this declaration of intent, not to the publication of any particular instance of the class (using "publication" in the normal sense). Other federates may *subscribe* to specific attributes of the class.

Having published the class, the federate may *register* instances of the class with the RTI. (The term reflects the notion that the objects already exist in the federate.) When an object is registered it becomes known to the federation, and the subscribing federates are said to *discover* it.

Having registered an instance the federate may *update* the values of any or all of those attributes whose names it originally published for the instance's class. After each update, the subscribing federates *reflect* the new values.

Finally, the federate may decide to destroy the instance; it so notifies the federation by *deleting* it. All subscribing federates are then notified to *remove* the instance.

The RTI's role in object management is to deliver *discover*, *reflect*, and *remove* messages to subscribing federates. The RTI does not retain any attribute values; it is the responsibility of each federate to preserve the values of all attributes that it updates or reflects. In particular, when a federate joins the federation it will immediate discover all registered object instances—but it will have no knowledge of their attribute values until the owning federates choose to update them.

Note that ownership of an object's attributes may be shared by two or more federates. Federate A can register an object and update one set of attributes; on discovering the object, federate B may then update additional attributes for the same object (provided, of course, that it has declared its intent to publish those attributes for the object's class).

As with interactions, a suscriber may subscribe to any class in the hierarchy, and will get all objects belonging to that class or its subclasses. And as with interactions, this capability isn't generally that useful (at least in the MRF).

2.3 Time Management

Some RTI federations run in real-time. In those federations RTI is used purely as a messaging infrastructure, and all federates rely on a real time clock for synchronization. This is especially the case when hardware simulators, e.g., flight simulators or tank simulators, are included in the mix.

On the other hand, many simulations, including JNEM, run on simulated time, which might or might not be geared to the passage of wallclock time. JNEM, for example, implements a "game ratio" which controls the passage of simulated time in a ratio with real time. When running in federation, simulated time must be coordinated across federates; this is the aim of RTI's time management features.

In a time-managed federation, a federate may be time-regulating, time-constrained, or both. A time-regulating federate timestamps its messages; the timestamps serve to regulate the time for other federates. A time-constrained federate is constrained to run at a simulated time consistent with the time-regulating federates in the federation. A federate that is both time-constrained and time-regulated is said to be time-managed.

A time-managed federate operates in the following loop:

```
Initialize simulation
Forever, Do
    Request advance to next simulation time of interest.
    Process incoming messages up to that simulated time.
    Receive time advance grant to time of interest.
    Advance time and do related work.
```

The RTI sees to it that a federate never gets a time advance grant until it has received all messages timestamped prior to the requested simulation time.

3. The C++ API

The RTI is specified as a C++ API. There are two primary classes: the RTIambassador and the FederateAmbassador.

3.1 The RTIambassador

The RTIambassador class encapsulates the RTI API proper: all API calls are member functions of the class. To join multiple federations simultaneously, a federate must create an instance of RTIambassador for each.

The member function names are extremely verbose; for example,

- `publishInteractionClass`
- `subscribeInteractionClass`
- `sendInteraction`
- `publishObjectClass`
- `subscribeObjectClassAttributes`
- `registerObjectInstance`
- `updateAttributeValues`
- `deleteObjectInstance`

One has the sense that the entire API was spec'd out in English in great detail prior to implementation—and then the section headings in the spec were used as the names of the member functions.

3.2 The FederateAmbassador

All messages received by the federate are passed to the federate as callbacks to member functions of a FederateAmbassador object. FederateAmbassador is an abstract base class, and provides no implementation at all; the federate must define a subclass and provide an implementation (possibly trivial) for every member function. FederateAmbassador member function names are just as verbose as RTIambassador names:

- `receiveInteraction`
- `discoverObjectInstance`
- `reflectAttributeValues`
- `removeObjectInstance`

The federate passes a pointer to its FederateAmbassador object to its RTIambassador when joining the federate. If the federate participates in multiple federations, it should create a FederateAmbassador for each.

3.3 Polling for Input

The RTI specification does not define the mechanism by which the federate polls for input; in particular, it does not presume that the federate is using an event loop. RTI NG Pro provides a "tick" RTIambassador method which polls for incoming messages and calls FederateAmbassador methods accordingly. When calling "tick" the caller may specify the minimum and maximum amount of time to spend waiting for incoming messages.

An unpleasant feature of this implementation is that most recursive calls into the RTIambassador are forbidden. For example, a federate cannot register an object instance or update an object's attributes within a FederateAmbassador callback; such actions must be postponed until after "tick" returns.

4. rti(n): The Tcl binding

JNEM's binding to the C++ API is called rti(n). rti(n) is intended to be a one-to-one mapping from a subset of the C++ API into Tcl; all operations have the same names and semantics as they do in C++. (Section 6 lists the supported operations.) Consequently, the rti(n) documentation need not duplicate the C++ API documentation; rather, it explains how the C++ operations are expressed in Tcl. The intent is that an experienced RTI programmer should be able to read Tcl code that uses rti(n) and understand it without reference to the rti(n) documentation. Nevertheless, rti(n) does include a number of significant enhancements to make the API both more Tcl-like and more convenient (which amounts to the same thing).

rti(n) is implemented as a part of a custom Tcl interpreter (called, for historical reasons, shark(1)), rather than as a loadable extension, because the C++ API is multi-threaded, and multi-threaded extensions can't be loaded into a single-threaded tclsh(1). Since we'd be building a multi-threaded tclsh(1) anyway, we chose to link rti(n) statically at build time.

4.1 ::rti::RTIambassador

rti(n) defines a single command, ::rti::RTIambassador, which is used to create RTIambassador objects following the standard Tcl object model:

```
$ ::rti::RTIambassador ra
::ra
$
```

::rti::RTIambassador is implemented in C++. It creates an instance of the C++ RTIambassador object and ties it to a new Tcl command; subcommands of the new command map to member functions of the C++ object in the obvious way.

As stated above, the binding API is intended to be a straightforward translation of the C++ API. Consider the following call, which is used to create a federation given a federation name (an arbitrary string) and the name of the *fed file*, which contains the FOM information needed at run-time:

```
#include <RTI.hh>
void
RTI::RTIambassador::
    createFederationExecution (
        const char* executionName,
        const char* FED
    ) throw ( /* Omitted, but lengthy */);
```

The signature of the equivalent rti(n) call is as follows:

```
$ra createFederationExecution executionName FED
```

The similarity is evident, as is the convenience Tcl adds.

4.1.1 Exceptions

RTI defines a plethora of exceptions, all of which are subclasses of RTI::Exception. RTI::Exception conveniently has methods which return the exception's name and an error message; rti(n) consequently converts all RTI exceptions into Tcl errors, with the errorCode set to the exception's name.

As a side note, the example code that comes with RTI NG Pro is simply encrusted with try/catch constructs that deal with the vast number of RTI exceptions; the catch blocks usually respond to an error by shutting down the application gracefully if one is thrown. Few of the exceptions are likely to occur in a properly written program unless an incorrect FOM is provided at run-time (in which case nothing is going to work anyway), so it would be desirable to handle most of these exceptions in a top-level try/catch block in main()...but because the exceptions are documented explicitly in each RTIambassador call this is difficult to do without cluttering the application code with exception propagation clauses.

JNEM applications using rti(n) and the Tcl event loop leave handling RTI exceptions up to bgerror; and as thrown exceptions need not be declared explicitly in Tcl, the related problems don't arise and the code looks much cleaner.

4.1.2 Enumerations

Enumerated types are handled in the usual way by translating the name of the symbolic constant into the enumerated integer before passing it to the C++ code:

```
$ra resignFederationExecution theAction
```
theAction must be one of the following strings:

- RELEASE_ATTRIBUTES
- DELETE_OBJECTS
- DELETE_OBJECTS_AND_RELEASE_ATTRIBUTES
- NO_ACTION

4.1.3 Overloaded Methods

Several of the RTIambassador member functions are overloaded. Usually there's a base method and an additional method with extra arguments. rti(n) implements these using optional arguments or option/value syntax:

```
$ra requestFederationSave label ?theTime?
```

In the RTI specification, there are two versions of this method, one with *theTime* and one without. In rti(n), *theTime* is simply an optional argument.

```
$ra deleteObjectInstance objectHandle ?options...?
```

In the RTI specification, there are two versions of this method, one that allows the caller to specify a timestamp and one that does not. Each also has a string argument, *theTag*, which is used to send an arbitrary and possibly empty string along with the *remove* message that results from deleting the object. Since the timestamp is effectively optional, and *theTag* has an obvious default, they both become options:

```
-time fedTime
```
fedTime is the federation time as a floating-point value.

```
-tag theTag
```
theTag is the tag string.

4.1.4 Handles vs. Names

Object class names, object attribute names, interaction class names, and interaction parameter names are all defined in the FOM file. At run-time these names are associated with handles. The application is expected to look up the handles for each name it uses, and then use these handles in all subsequent calls. For example, the C++ code to declare intent to send interactions of a particular class looks like this:

```
RTI::InteractionClassHandle h =
    ra.getInteractionClassHandle("MYCLASS");

ra.publishInteractionClass(h);
```

rti(n) supports this idiom; you can write the following:

```
set h [ra getInteractionClassHandle MYCLASS]

ra publishInteractionClass $h
```

The various kinds of handles are all represented as 32-bit unsigned integers, which rti(n) handles as wide integers in order to preserve the full unsigned range.

However, using handles for what are effectively predefined constants is extremely un-Tcl-like; so any command that expects a class, attribute, or parameter handle will also accept the actual name:

```
ra publishInteractionClass MYCLASS
```

4.1.5 C++ Sets vs. Lists and Dictionaries

The RTI API includes a number of collection objects called "sets". For example, when publishing an object class the application must specify the particular set of attributes it plans to update in an RTI::AttributeHandleSet. When this is combined with the need to convert attribute names to integer handles, the C++ code becomes spectacularly ugly. Here, for example, we declare that our application wants to publish the NAME and COLOR attributes of instances of the THING class:

```
RTI::ObjectClassHandle hThing =
    ra.getObjectClassHandle("THING");

RTI::AttributeHandle hName =
    ra.getAttributeHandle("NAME", hThing);
RTI::AttributeHandle hColor =
    ra.getAttributeHandle("COLOR", hThing);

RTI::AttributeHandleSet attr =
    RTI::AttributeHandleSetFactory::create(2);
attrs->add(hThing);
attrs->add(hColor);

ra.publishObjectClass(hThing, *attrs);
```

In rti(n), of course, the RTI::AttributeHandleSet is simply a Tcl list of attribute handles—or, in practice, a list of attribute names:

```
ra publishObjectClass THING {NAME COLOR}
```

Updating an object's attributes is even uglier; the attribute handles and their values must all be plugged into a data structure called an RTI::AttributeHandleValuePairSet. With rti(n), a dictionary is naturally used instead (note that the Tcl 8.5 dict(n) command is not required). We will leave imagining the C++ code as an exercise for the reader; the rti(n) code is as follows:

```
set hObject [ra registerObjectInstance THING]

ra updateAttributeValues $hObject \
    {NAME "This Thing" COLOR "Green"} \
    -time $t
```

4.2 The FederateAmbassador

Whenever rti(n) creates a C++ RTIambassador object, it goes on to create a matching C++ FederateAmbassador object as well. This object is invisible to the Tcl programmer, but it is responsible for translating all supported FederateAmbassador callbacks into equivalent Tcl callbacks. The details of this translation are described in the following subsections.

4.2.1 FederateAmbassador Ensembles

An application joins an RTI federation by creating an RTIambassador and calling its joinFederationExecution method. One of the arguments to this method in the C++ API is a pointer to an instance of a FederateAmbassador subclass. In rti(n), that argument is replaced by the name of a Tcl command; the

command is treated as the name of an ensemble command whose subcommands are the FederateAmbassador callback methods. This ensemble command can be implemented however the user prefers; for example, here's a FederateAmbassador that simply logs the callbacks it receives:

```
proc fedAmb {subcmd args} {
    puts "FederateAmbassador: $subcmd $args"
}

::rti::RTIambassador ra
ra joinFederationExecution $myname $fedname fedAmb
```

Writing more realistic ensemble commands from scratch in pure Tcl is a nuisance, of course, so JNEM uses Snit [2] to define a FederateAmbassador type, as described in Section 5.

4.2.2 Callback Enhancements

When translating C++ FederateAmbassador callbacks into calls to subcommands of the Tcl FederateAmbassador ensemble, rti(n) makes the same kind of convenience adjustments as it does for the RTIambassador commands: names are passed instead of handles, overloaded methods are handled using optional arguments and option/value syntax, and C++ collections are passed as Tcl lists and dictionaries.

For example, when rti(n) receives an object attribute update, the reflectAttributeValues method is called with a dictionary of attribute names and values. This is a significant convenience; C++ code must usually compare each attribute handle in the update with each of a set of variables containing attribute handles to determine which specific attribute each is.

4.2.3 Queuing Callbacks

One of the nuisances of the RTI API, described in detail in Section 3.3, is that the C++ RTIambassador is not re-entrant—and therefore, since the FederateAmbassador's callbacks are always called from within the RTIambassador's "tick" method, FederateAmbassador callbacks cannot call the RTIambassador's methods. As a result, RTI applications generally queue up the inputs received via the FederateAmbassador, and handle them after "tick" returns.

rti(n) handles this queuing automatically. Each incoming FederateAmbassador callback received during a call to "tick" is translated into a Tcl command in the form of a list of Tcl_Obj* structures, to be executed using Tcl_EvalObjv(). [3] Instead of evaluating the command immediately, however, the command is queued; and immediately after "tick" returns, rti(n) evaluates the queued commands in sequence.

As a result, rti(n) programs are free to call RTIambassador methods from within FederateAmbassador callbacks. The decision of whether or not to queue inputs can then be based solely on the needs of the application.

4.2.4 Error Handling

The Tcl FederateAmbassador callbacks invoked by rti(n) may of course contain bugs and throw Tcl errors. rti(n)'s C++ Federate-Ambassador object catches those errors and implicitly invokes "bgerror" to handle them, just as Tk does.

4.3 rti(n): Results

RTI NG Pro ships with a sample application called "FoodFight", in which an arbitrary number of identical federates simulate a food fight in a school cafeteria. (The fight continues until all students have exhausted their supply of cash.) While developing rti(n), I implemented a Tcl version of the FoodFight federate which could interoperate with the C++ version. The Tcl version is both more concise and more readable; but it is recognizably the same program, and should be easily understood by an experienced RTI developer after a short introduction to Tcl syntax.

Thus, rti(n) can be used to write RTI federates in a familiar style, but with the convenience and expressiveness of Tcl.

5. rtiproxy(n): Convenience + Policy

rti(n) adds considerable value over the C++ API, but because it is a straightforward binding to the C++ API there are many details that it must necessarily leave to the application. Some of these details can be handled in a general way for all rti(n) clients. Others depend on the policies of the specific federation, but can be handled in a general way for all federates in the federation.

Operationally, JNEM includes only one RTI federate; but for testing and development it is naturally useful to implement others. As a result, it's worthwhile to implement these general mechanisms and policies in a reuseable object. JNEM does so, using a higher-level API called rtiproxy(n). rtiproxy(n)'s API is equally expressive yet more concise. It also implements a number of policies that are specific to the JLCCTC MRF federation and to JNEM itself. Consequently, rtiproxy(n) is specific to JNEM's environment and is not directly reuseable by other projects. Its design and interface are nevertheless instructive.

5.1 Architecture

rtiproxy(n) implements a single command, a snit::type called, unremarkably, rtiproxy. Each instance of rtiproxy encapsulates an instance of ::rti::RTIambassador, and in addition serves as its own FederateAmbassador. Here is the skeleton of an rtiproxy-like type:

```
snit::type rtiproxy {
    component ra   ;# RTIambassador

    constructor {args} {
        # Create RTIambassador
        install ra using \
            ::rti::RTIambassador ${selfns}::ra

        # Process options, etc.
    }

    # Log undefined FederateAmbassador callbacks

    delegate method * using {%s Unknown %m}

    method Unknown {method args} {
        puts "+++ FedAmb: $self $method $args"
    }

    # Define required FederateAmbassador callbacks
    # here....
}
```

Note that it isn't necessary to define a method for every FederateAmbassador callback, but only for those callbacks one actually needs; the "Unknown" method will handle the rest.

Note also the name used for the ::rti::RTIambassador object. Every instance of rtiproxy requires its own instance of ::rti::RTIambassador, each of which must have a unique name. If ::rti::RTIambassador were a Snit object, it would be usual to specify the name "%AUTO%", which would direct Snit to automatically generate a unique name. Some time ago, however, Andreas Kupries [4] observed that component objects could be created within the instance's private namespace, $selfns—and that if they were, they would be automatically destroyed when the parent was destroyed, with no explicit destructor required. This has since become my preferred way of creating and naming component objects.

5.2 Mechanisms

The following subsections describe a number of mechanisms which every RTI application must implement, and which rtiproxy(n) handles automatically.

5.2.1 Event Loop

Although RTI does not require an event loop, most Tcl programmers prefer to use one. Upon joining a federation, rtiproxy(n) schedules a "ticker" timeout which is called approximately every 10 milliseconds. This timeout calls the RTIambassador's "tick" method, asking it to return after all available messages have been processed. The result is that all FederateAmbassador callbacks occur in the context of the event loop, and the application can be developed using the usual event-driven paradigm.

5.2.2 Federation Saves and Restores

RTI provides a mechanism by means of which an entire federation can save its current state to disk such that it can be restored at a later time. Supporting federation saves and restores requires a federate to implement a complicated protocol. For example, here's how a federation save proceeds:

- First, some federate calls requestFederationSave.
- Our federate receives an initiateFederateSave callback.
- Our federate must then:
    - Call federateSaveBegun
    - Save its state however it pleases
    - Call federateSaveComplete
    - Or, if it failed, call federateSaveNotComplete
- If all federates save successfully, our federate will receive a federationSaved callback.
- Otherwise, it will receive a federationNotSaved callback.
- Note that all other activities are suspended during the save process.

The protocol for federation restores is nearly identical.

When creating the rtiproxy(n) object, the application specifies save and restore callbacks using the `-savecmd` and `-restorecmd` options; the callbacks need only save or restore the application's state, and throw an error if they fail. rtiproxy(n) handles the rest of the protocol automatically.

5.2.3 Attribute Value Management

As stated in Section 2.2, the RTI retains the identity of all registered objects, but does not retain any record of the current attribute values. If a federate joins the federation after objects have been created, it will immediately receive a discoverObject-Instance callback for each registered object...but it will receive attribute updates only as the other federates decide to update them.

In such a case, the late-joiner can call

```
$ra requestClassAttributeValueUpdate \
    theClass attributeList
```

giving the name of the class and a list of the names of the attributes in which it is interested. The federate or federates that own objects of this class will receive

```
$fa provideAttributeValueUpdate \
    theObject theAttributes
```

for each object of that class that they own. They must then republish all of the specified attributes for that object by calling

```
$ra updateAttributeValues \
    theObject theAttributes ?options...?
```

The ultimate effect is to refresh the late-joiner's set of object data.

rtiproxy(n) helps with all of this by keeping a record of every attribute value published by the application. Consequently, it can handle the provideAttributeValueUpdate callback completely automatically; no application code need be involved.

This feature could be problematic for federates that publish a vast quantity of data, as it means that the attribute data is effectively stored twice, once by the application, and once by rtiproxy(n). For smaller federates like JNEM, however, it's a reasonable convenience.

5.2.4 Synchronization Points

RTI synchronization points are essentially a generic equivalent of the federation save and restore protocols; they allow federations to synchronize their federates for other arbitrary kinds of housekeeping tasks. JLCCTC MRF defines a synchronization point called "REGISTER_OBJECTS"; federates are not supposed to register objects or update their attributes until they receive this synchronization point. In this way object registration can be delayed until all federates have joined the federation, thus removing the need for federates to request an attribute-value update, as described in the previous section.

Responding to a synchronization point involves a dance similar to that shown for federation saves; and as with federation saves, rtiproxy(n) handles most of the details. The application provides a synchronization point callback via the `-synccmd` option. The callback receives the synchronization point's name; the application is then required to call the rtiproxy(n) object's "synced" method to indicate that synchronization is complete.

5.3 Policies

The following subsections describe a number of federation-specific and application-specific policies which are implemented by rtiproxy(n).

5.3.1 Encoding and Decoding

As indicated in Section 2, the RTI treats all attribute and parameter values as byte arrays. It's up to the application to encode and decode them as required for interoperability. If a federation consists only of rti(n) clients, encoding and decoding isn't an issue (provided that all clients use the same unicode encoding), as all values are sent as Tcl strings. If an rti(n) client is to interoperate with federates implemented in other languages, or resident on hosts with differing hardware architectures, then encoding the data is a serious issue.

Many federations use "xdr" to encode and decode data; JLCCTC relies on an *ad hoc* scheme documented in the federation agreement. JNEM includes a module called schema(n) which lists the object classes, attributes, interaction classes, and parameters used by JNEM (a small subset of the FOM) along with sufficient type information that another module called codec(n) can perform the JLCCTC-specific encoding and decoding. rtiproxy(n) relies on schema(n)/codec(n) to encode and decode data automatically on send and receive. schema(n)/codec(n)'s implementation could reasonably be swapped out for another without affecting the rtiproxy(n) code.

5.3.2 Time Management

JNEM is a time-managed federate; that is, it is both time-regulated and time-constrained. Consequently, all incoming messages are timestamped by the sending federate), and all outgoing messages must be timestamped by JNEM.

RTI NG Pro expresses simulation time as a double-precision floating point value starting at 0.0; in JLCCTC, this value is interpreted as decimal hours. JNEM is a time-step simulation with a step size of one minute; hence, JNEM keeps time in integer minutes starting at 0. JNEM's simulation time is managed by an object of type simclock(n), which does all translation between differing time formats; one of rtiproxy(n)'s responsibilities is advancing the simclock when a time advance is granted. The algorithm described in Section 2.3 is therefore implemented something like this in rtiproxy(n):

On joining the federation:
    Become time-managed, and get the current federation simulation time T as a value in integer minutes, updating the simclock accordingly.
While in the federation,
    When enough wallclock time has passed, request a time advance to time T + 1 minute.
    Process incoming messages, which will be timestamped between T and T + 1. Call application callbacks as appropriate. This will continue until RTI determines that all messages with timestamps less than T + 1 have been received.
    Receive time advance grant to time T + 1, and advance the simclock accordingly.
    Call `-advancecmd` callback, notifying the application of the time advance.

The application registers callbacks with rtiproxy(n) for the various interaction and object classes, and for the time advances.

The Game Ratio. Note the phrase "When enough wallclock time has passed..." in the above pseudocode. Analytical simulations usually run as fast as possible, which in a federation is at the speed of the slowest federate. JLCCTC is a training federation, with human beings very much in the loop, and consequently is usually run operationally at approximately 1 simulated minute per wall-clock minute. During development, though, and at times during operations, it's convenient to run at other (often much higher) rates. Consequently, rtiproxy(n) implements a *game ratio*, which is the desired number of simulated minutes per wallclock minute. If the game ratio is 0.0, time does not advance at all. If it is 1.0, the simulation runs at approximately real time. If it is 20.0, time can advance quite rapidly. The game ratio can also be set to the string "auto", in which case rtiproxy(n) will advance time as fast as the rest of the federation—and the host processor—will allow.

The game ratio is implemented by the same "ticker" timeout which is used to periodically call "tick"; see Section 5.2.1.

Timestamps and Lookahead. RTI incorporates a scheme whereby it is guaranteed that time can advance in the federation: the timestamp on an outgoing message must be slightly later than the federate's current simulation time. This small interval is called the federate's lookahead. When sending a message, a federate simply adds the lookahead to the current game time and uses that to timestamp outgoing messages. Since rtiproxy(n) expects a time step of one minute it uses a lookahead of one minute as well. Further, because rtiproxy(n) knows both the current simulation time and the lookahead, it automatically timestamps all outgoing messages.

5.4 rtiproxy(n) API

This section describes a portion of rtiproxy(n)'s API, and contrasts it with the lower-level rti(n) API. In the example code in the following subsections, `$ra` denotes the name of an rti(n) ::rti::RTIambassador object and `$rp` denotes the name of an rtiproxy(n) object.

5.4.1 Implicit Argument Values

There are a number of ::rti::RTIambassador subcommands that require an argument which rtiproxy(n) implements as one of its options instead. The joinFederationExecution subcommand, for example, requires the name of the federation that the federate should join. This makes it straightforward to join different federations at different times; however, it's more usual to join and then resign from the same federation many times in succession during the course of a training exercise. Consequently, rtiproxy(n) provides a `-federation` option, thus taking on the burden of remembering the federation name, as well as allowing the client to join and resign from the federation without explicitly specifying the federation name each time. A number of inputs are replaced by options in this way.

Similarly, as a time-managed federate Snit must provide a timestamp for all outgoing messages. Because rtiproxy(n) knows both the current simulation time and the lookahead interval, and because JNEM is a timestep simulation that never sends messages

stamped later than the current time plus the lookahead, rtiproxy(n) can timestamp all of the messages automatically.

5.4.2 Hierarchical Command Set

The Tk text widget presents a vast, rich API concisely by using sub-ensembles to group related subcommands. For example, text tags are manipulated using subcommands of the widget's "tag" subcommand, embedded widgets are manipulated using subcommands of the widget's "window" subcommand, and so forth. The full set of text widget operations is thus represented hierarchically, rather than as a flat list of operations. This hierarchical presentation is of great psychological importance when reading either application code or the text widget's documentation: it's much easier to ignore whole branches of the tree than it is a selection of operations in a flat list. If there's no need to use an embedded window in a particular text widget, there's no need to read—or even notice—the related documentation.

rtiproxy(n) uses Snit's hierarchical method names to do the same thing. The following code, for example, defines a method "fed" with two submethods, "join" and "resign":

```
method {fed join} {} {
    # Join the federation ...
}

method {fed resign} {} {
    # Resign from the federation ...
}
```

rtiproxy(n) uses this feature to define three sub-ensembles, "fed", "obj", and "int", to group subcommands related to the federation as a whole, to objects and their attributes, and to interactions. The following table relates some of their subcommands with their rti(n) equivalents (omitting argument lists):

```
$ra createFederationExecution   $rp fed create
$ra destroyFederationExecution  $rp fed destroy
$ra joinFederationExecution     $rp fed join
$ra resignFederationExecution   $rp fed resign

$ra publishInteractionClass     $rp int publish
$ra subscribeInteractionClass   $rp int subscribe
$ra sendInteraction             $rp int send

$ra publishObjectClass          $rp obj publish
$ra registerObjectInstance      $rp obj register
$ra updateAttributeValues       $rp obj update
$ra deleteObjectInstance        $rp obj delete
```

A developer wanting information on object management knows to go directly to the section of the man page on the "obj" method. Plus, the resulting API is both more concise and more Tcl-like.

5.4.3 Finer-grained Callbacks

The FederateAmbassador provides one callback for each kind of incoming message. All interactions, for example, are passed to the federate ambassador's receiveInteraction callback. This leads naturally to long ugly switch statements, and makes it difficult to have different interactions handled by different modules. The same consideration applies to object attribute updates. Consequently, rtiproxy(n) allows the client to register callbacks by interaction or object class.

Receiving them requires subscribing to the class and defining the FederateAmbassador receiveInteraction callback:

```
# In FederateAmbassador type definition

method receiveInteraction {class parmdict args} {
    switch -exact -- $class {
        REPORT.TEMPERATURE {
            .
            .
            .
        }

        .
        .
        .
    }
}
```

An application using rtiproxy(n) would do the following instead:

```
$rp int receiveWith REPORT.TEMPERATURE \
    receiveTEMPERATURE

proc receiveTEMPERATURE {class parmdict tag} {
    # Process REPORT.TEMPERATURE interaction
}
```

The callback is defined as a command prefix in the usual way; the actual command may be defined anywhere in the application, and may be implemented as a proc, a Snit method, etc. To cancel the callback, set it to the empty string.

A more advanced mechanism would allow multiple callbacks to be registered for each interaction class; to date, however, that hasn't been needed in JNEM.

5.5 rtiproxy(n): Results

The rtiproxy(n) module has proven to be a solid and robust interface for JNEM. Its primary flaw is that it mingles essential mechanism (which is useful to many federates regardless of federation) with policies specific to the JLCCTC MRF federation or to JNEM itself. This was a mistake.

For example, there is a strong desire within the U.S. Army's training community to use JNEM in a number of other training federations. One of these, the JLCCTC Entity Resolution Federation (ERF), does not use RTI time management; instead, each federate is regulated purely by wall-clock time. rtiproxy(n) does not currently support this mode of operation. With essential mechanism and MRF-specific policy mixed together, it becomes harder to reuse the code that applies to both environments. Similarly, rtiproxy(n) uses a specific encoding/decoding scheme rather than allowing different schemes to be selected based on context. Such policies should either be implemented in a separate object, or should be selectable by option. As JNEM moves into additional federations, I'll need to fix these defects.

6. Supported RTI Features

rti(n) does not implement the full set of RTIambassador methods and FederateAmbassador callbacks. Advanced features, such as management of the ownership of object attributes and Data Distribution Management (DDM) have been omitted.

6.1 Supported RTIambassador Methods

- createFederationExecution
- deleteObjectInstance
- destroyFederationExecution
- disableTimeConstrained
- disableTimeRegulation
- enableAttributeRelevancyAdvisorySwitch
- enableTimeConstrained
- enableTimeRegulation
- federateRestoreComplete
- federateRestoreNotComplete
- federateSaveBegun
- federateSaveComplete
- federateSaveNotComplete
- getAttributeHandle
- getAttributeName
- getInteractionClassHandle
- getInteractionClassName
- getObjectClass
- getObjectClassHandle
- getObjectClassName
- getObjectInstanceHandle
- getObjectInstanceName
- getParameterHandle
- getParameterName
- joinFederationExecution
- publishInteractionClass
- publishObjectClass
- queryLBTS
- queryLookahead
- registerFederationSynchronizationPoint
- registerObjectInstance
- requestClassAttributeValueUpdate
- requestFederationRestore
- requestFederationSave
- resignFederationExecution
- sendInteraction
- subscribeInteractionClass
- subscribeObjectClassAttributes
- synchronizationPointAchieved
- tick
- timeAdvanceRequest
- unpublishInteractionClass
- unpublishObjectClass
- unsubscribeInteractionClass
- unsubscribeObjectClass
- updateAttributeValues

6.2 FederateAmbassador Callbacks

rti(n) necessarily implements all of the FederateAmbassador callbacks, as the code will not successfully compile otherwise. The following callbacks are actually passed through to the Tcl application (the remainder are stubbed out via the NullFederate-Ambassador class provided with RTI NG Pro V3.0):

- announceSynchronizationPoint
- discoverObjectInstance
- federationNotRestored
- federationNotSaved
- federationRestoreBegun
- federationRestored
- federationSaved
- federationSynchronized
- initiateFederateRestore
- initiateFederateSave
- provideAttributeValueUpdate
- receiveInteraction
- reflectAttributeValues
- removeObjectInstance
- requestFederationRestoreFailed
- requestFederationRestoreSucceeded
- startRegistrationForObjectClass
- stopRegistrationForObjectClass
- synchronizationPointRegistrationFailed
- synchronizationPointRegistrationSucceeded
- turnInteractionsOff
- turnInteractionsOn
- turnUpdatesOffForObjectInstance
- turnUpdatesOnForObjectInstance

7. REFERENCES

[1] *RTI NG Pro Programmer's Guide V3.0*, by Virtual Technology Corporation, http://www.virtc.com/rtingpro-full.jsp.

[2] Duquette, William H., "Snit's Not Incr Tcl", http://www.wjduquette.com/snit.

[3] Tcl Manual, http://www.tcl.tk/man/tcl8.4/TclLib/Eval.htm.

[4] Andreas Kupries, personal e-mail.

8. ACKNOWLEDGEMENTS

This research was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration, during the development of the Joint Non-kinetic Effects Model (JNEM) for the U.S. Army's National Simulation Center at Fort Leavenworth, Kansas.