Table of Contents

PART I:  FDK ARCHITECTURE ................................................................. 5

PART II:  RTI INTERFACES ............................................................ 7
  INTRODUCTION ........................................................................... 7
  BASIC RTI .................................................................................. 8
    Overview ................................................................................... 8
    Initialization Services .............................................................. 8
    Declaration Management Services ....................................... 9
    Object Management Services ............................................... 11
    Time Management Services .................................................. 12
    Miscellaneous Services .......................................................... 13
    Example ................................................................................... 14
  HLA RTI ...................................................................................... 14
    Overview ................................................................................... 14
    Internals .................................................................................... 14
    The I/F Specification ............................................................... 16
    Auxiliary Classes and Code ................................................... 25
    Adding Additional Services to the DRTI ................................ 29
    Using the DRTI .......................................................................... 29

PART III:  RTI-KIT ......................................................................... 30
  FM API ....................................................................................... 30
    Rationale ................................................................................... 30
    Implementation ......................................................................... 31
  MCAST ......................................................................................... 33
    Rationale ................................................................................... 33
    Implementation ......................................................................... 33
  DDM-KIT ..................................................................................... 38
    Rationale ................................................................................... 38
    Implementation ......................................................................... 39
  RM KIT ......................................................................................... 45
    Rationale ................................................................................... 45
    Implementation ......................................................................... 45
  TM-KIT ......................................................................................... 51
    Rationale ................................................................................... 51
    Implementation ......................................................................... 52
  RTI-CORE ..................................................................................... 56
    Rationale ................................................................................... 56
    Implementation ......................................................................... 56
  SESSION MANAGER ................................................................. 66
    Rationale ................................................................................... 66
    Implementation ......................................................................... 66
  OTHER UTILITIES ................................................................. 69
    Kit .............................................................................................. 69
    RANDB ....................................................................................... 71
    Heap .......................................................................................... 71

PART IV:  APPENDICES ................................................. 73
  PERFORMANCE ........................................................................... 73
  INSTALLATION ........................................................................... 75
  RELEASE NOTES ........................................................................... 78

PART V:  REFERENCES .......................................................... 80
PREFACE

The High Level Architecture (HLA) represents one of the most widespread exploitations of parallel and distributed simulation technology in real-world systems. Yet, much of the research in the parallel and distributed simulation community (e.g., time management, load distribution, and performance optimization) does not attempt to apply that work in the context of the HLA. As such, this research is currently not well positioned to provide technological advances to drive future generations of the HLA.

A major impediment to researchers attempting to orient their work around the HLA is the high start-up cost in performing HLA-relevant research. A significant intellectual effort is required to become familiar with the HLA, and existing educational materials are focused more toward simulation practitioners (e.g., how to write HLA compliant simulations) rather than the technology driving the HLA development (e.g., time management or data distribution algorithms). Further, a substantial software development effort is required to realize an HLA RTI, essential for experimental research. While public versions of RTI software exist, researchers require source code, documentation, and (to a certain degree) support for the software to effectively use it in their research. These facilities do not exist in the current research environment.

The principal goal of Federated Simulations Development Kit (FDK) software package is to provide a vehicle for researchers in academia, industry, and government laboratories to rapidly become familiar with and contribute novel research that will both advance distributed simulation technology, and can be quickly transitioned for use in shaping future directions for the HLA. The RTI-Kit software, a principal component of FDK, was developed for this purpose. RTI-Kit provides libraries to support realization of Runtime Infrastructure (RTI) software that realizes services to support parallel and distributed simulations.

The first version of the RTI-Kit software was realized in 1997. Several revisions and enhancements have since been developed to improve performance, maintainability, and to increase the range of platforms over which it would run. The initial version was developed as a by-product of research funded by the Defense Advanced Research Projects Agency (DARPA) and the Ministry of Defence in the United Kingdom. Much of the subsequent development and distribution of the software was funded by the Defense Modeling and Simulation Office (DMSO) to provide HLA software to the community to support research and development efforts in distributed simulation systems. Research at Georgia Tech in parallel and distributed systems utilizing the RTI-Kit software has been funded by several sources, including the Ballistic Missile Defense Office (BMDO) under management by the U.S. Army Space and Missile Defense Command, the Common High Performance Computing Software Support Initiative (CHSSI) Program, DARPA, and the National Science Foundation (NSF). RTI-Kit software is currently used worldwide to support research in both defense and non-defense applications.

This document is intended for users of RTI-Kit (i.e., simulation developers) and RTI developers. Following an introduction to FDK, the guide is organized in two parts. For simulation developers, Part II contains information on the RTI-Kit interfaces as well as guidance on using the services. In Part III RTI developers will find implementation information on each of the RTI-Kit modules. Installation and execution procedures are contained in the Appendix.
PART I: FDK Architecture

FDK stands for Federated Simulations Development Kit, which is a software system developed at Georgia Tech by the PADS research group led by Professor Richard Fujimoto. It contains composable modules for building run-time infrastructures (RTIs) and a simulator-independent, client/server-based graphical interface and scripting tool called Jane.

RTI-Kit is a collection of libraries designed to support development of Run-Time Infrastructures (RTIs) for parallel and distributed simulation systems, especially federated simulation systems running on high performance computing platforms. Each library is designed so it can be used separately, or together with other RTI-Kit libraries, depending on the functionality required by the user. Because each library is designed as a stand-alone component, RTIs that are constructed using RTI-Kit are highly modular, with clear, well-defined (and documented) interfaces. These libraries can be embedded into existing RTIs, e.g., to add new functionality or to enhance performance by exploiting the capabilities of a high performance interconnect. For example, RTI-Kit software was successfully embedded into an HLA RTI developed in the United Kingdom (Hoare, Magee et al. 1997) (Fujimoto and Hoare 1998). Alternatively, the libraries can be used in the development of new RTIs.

The "library-of-libraries" approach to RTI development offers several important advantages. First, it enhances the modularity of the RTI software because each library within RTI-Kit is designed as a stand-alone component that can be used in isolation of other modules. Modularity enhances maintainability of the software, and facilitates optimization of specific components (e.g., time management algorithms) while minimizing the impact of these changes on other parts of the RTI. This design approach facilitates technology transfer to other RTI development projects because utilizing RTI-Kit software is not an "all or nothing" proposition; one can extract modules such as the time management component while ignoring other libraries. The FDK architecture is shown in Figure 1.

![Figure 1. FDK Architecture](image-url)
The FDK architecture is organized in three parts: Communications, RTI, and Applications. From the Communication developer’s perspective, multiple implementations of the RTI-Kit software can be realized targeting different platforms. Specifically, RTI-Kit can be configured to execute over shared memory multiprocessors such as the SGI Origin, cluster computers such as workstations interconnected via a low latency Myrinet switch (Myricom 1996), to workstations interconnected over local or wide area networks using standard network protocols such as IP.

The lowest level described in this document is the communication layer that provides basic message passing primitives. Communication services are defined in a module called FM-Lib. This communication layer software acts as a multiplexer to route messages to the appropriate module. The current implementation of FM-Lib implements reliable point-to-point communication. It uses an API based on the Illinois Fast Messages (FM) software (Pakin, Lauria et al. 1996) for its basic communication services, and provides only slightly enhanced services beyond those of FM.

Above the communication layer are modules that implement key functions required by the RTI. This is the RTI developer’s domain. These modules form the heart of the RTI-Kit software. Specifically, TM-Kit is a library that implements distributed algorithms for realizing time management services. Similarly, DDM-Kit implements functionality required for data distribution management services. MCAST is a library that implements reliable group communication services. RM-Kit is used to perform an asynchronous distributed reduction. Other utilities provide software for buffer and queue management, a random number generator library, and a wrapper module to initialize and tick the different RTI-Kit modules.

RTI Core is intended to pool together the resources of the other libraries within FDK to present a unified interface to the RTI developer. RTI Core’s main focus is to provide basic time management functionality that is useful to a wide range of RTIs.

Finally, the Application developer utilizes the RTI Interface layer to access a specific Application Program Interface (API) such as the HLA Interface Specification. The current RTI-Kit distribution includes two APIs. The Basic RTI (BRTI) is a simple RTI, implementing a minimal number of HLA-like services for declaration management, object management, and time management. It uses a simple C language interface, rather than the more complex Class/Object paradigm specified by the HLA interface specification. The Detailed RTI (DRTI) uses the 1.3R6 interface specification. Although not all services are implemented, the DRTI includes the entire interface (RTI and Federate Ambassador) in its definition. This allows programs to be linked interoperably with DRTI or any other compliant RTI. The DRTI is also referred to as the TRTI (Telecomm RTI) in certain documentation concerning use of the HLA to interconnect network simulations.

Jane is a simulator-independent, client/server-based graphical interface and scripting tool for interactive parallel and distributed simulations. Jane uses a session manager to shield federates from variations in the environmental setup for RTI-Kit modules on different computing and communication platforms. The session manager is described in this document, however Jane is not. Please see (Perumalla and Fujimoto 1999; Perumalla and Fujimoto 2001) for Jane documentation. FDK also includes several well-known benchmarks widely used in the parallel discrete event simulation community. Ping is a simple time managed program illustrating use of essential HLA services. Phold is a synthetic workload program, and Qnet is a queueing network simulator (Fujimoto 1990).
PART II: RTI Interfaces

This chapter is oriented towards the application developer, and focuses on the relevant interfaces to realize distributed simulations over FDK RTIs. It is assumed the reader has some familiarity with the High Level Architecture, and services specified in the Interface Specification. HLA terminology is adopted throughout this document.

INTRODUCTION

An RTI implementation can be thought of as an interface to RTI-Kit functionality. An RTI implementation presents services to the federate according to a specific paradigm for simulation execution management and exchange of data (e.g. HLA). Each RTI implementation must manage whatever global and local state information is required for its paradigm.

Typically, an RTI will have state variables which include time management information (such as local time, result of the most recent LBTS computation, lookahead, the state of any federate requests to advance time), communication information (such as the multicast groups, group membership, and the mapping of groups to message types) and the state of execution management processes (such as pause/resume, save/restore, join/resign). The RTI must also have a means for delivering messages and other information to the federate. In the case of an HLA federate, this is done using callback functions. Therefore, the RTI must have a means of registering callback functions.

One important feature of a modular RTI design is the ability to explore design trade-offs. The overhead of a particular interface design may lead one to choose a modified or partial implementation. This may produce a more efficient execution for the target federation. This is a reasonable trade-off, even in an HLA execution environment, considering that freely available compliant RTIs exist, and the principle reason for choosing a different implementation would either be for 1) performance or 2) federation specific architectural considerations.

An example of this type of trade-off is evident when considering the flexibility in configuring object attribute updates. The HLA IF specification allows for the ownership, transport and ordering of every attribute of every object to be individually set. While this could be a powerful tool for customizing the communications configuration of a federation execution, there is a significant overhead associated with checking each attribute in an attribute handle-value pair set (AHVPS). In federations where ownership is static, and transport is never altered from the default, a significant simplification is possible. This fact was exploited in the design of the FDK’s AHVPS class. The design assumes that a new AHVPS (or Parameter HVPS) will eventually be sent as an object attribute update or an interaction message. The AHVPS constructor allocates memory for the entire message, marshalling the AHVPS data into the appropriate slot. This eliminates the need to copy any data during an UpdateObjectAttributeValue() or SendInteraction() call. Such an implementation would not be efficient if attribute updates cannot be assumed to be atomic.

A few conventions and procedures are used for all RTI-Kit libraries. In the current version, RTI-Kit assumes the number of processors used during the execution does not change, and all are available during initialization and throughout the entire execution. RTI-Kit automatically assigns processor numbers. If there are N processors, they are numbered 0, 1, … N-1. For the most part, RTI-Kit applications need not be concerned with the processor numbers that are assigned because they are used primarily for the internal operation of the library, and applications may use their own processor identification scheme independent of what RTI-Kit uses. The processor numbers used by RTI-Kit are available to RTI-Kit users through the following definitions:
RTIKIT_numnodes: number of processors in system

RTIKIT_nodeid: number of the local processor

Designing and building an RTI is a straightforward matter of mapping user services to RTI-Kit functionality. RTICore, as explained in Part IV handles most of the work in initializing and dealing with RTI-Kit modules. The RTI developer must decide on an applications programmers interface (API) for the RTI, and, of course, develop additional code to handle functions outside the scope of the RTI-Kit. The two included RTIs show a very basic approach to an API, and sophisticated (partial) implementation of the HLA I/F Specification.

**BASIC RTI (BRTI)**

The basic RTI is useful for simple applications, or for users who are learning the basic ways that federates in a distributed simulation must interact. The basic RTI provides a minimal set of services. These services are sufficient to develop and deploy relatively simple distributed simulations. The services are similar to those defined in the Department of Defense High Level Architecture, but are accessed using a simplified Application Programming Interface (API).

**OVERVIEW**

The basic RTI is useful for simple applications, or for users who are learning the basic ways that a distributed simulations must interact between the federates. The basic RTI provides a minimal set of services, and these services are sufficient to develop and deploy relatively simple distributed simulations. The services are similar to those defined in the Department of Defense High Level Architecture, but are accessed using a different Application Programming Interface (API).

The services provided by the basic RTI fall into four basic categories, listed below.

- **INITIALIZATION** services allow the RTI-Kit to set up the execution of the federation, including reading the federation object description file (the FedFILE), contacting other federates in the distributed simulation, and creating the basic two-way communications paths between all federates.

- **DECLARATION MANAGEMENT** services declare objects and set up communications between federates using a newsgroup-like publish/subscribe paradigm.

- **OBJECT MANAGEMENT** services allow federates to update attributes (i.e., send messages) and receive updates to attributes (receive messages) produced by other federates.

- **TIME MANAGEMENT** services control the advancement of simulation time within each federate, and prevent federates from receiving messages in their simulated past.

- **MISCELLANEOUS** services to control the overall execution of the simulation are also provided.

**INITIALIZATION SERVICES**

The RTI-Kit is initialized by calling the InitRTI subroutine, described below. No other RTI function should be invoked prior to making this call. The initialization procedure is defined as:
Where argc and argv are the command line parameter count and the command line command list passed to the main C function. NOTE: As of this writing, these parameters are not used, so the federate can safely pass 0 and NULL for these values. This function reads the federation object description file (the FedFile) mentioned earlier. This file must be present with the name “fedfile” in the same directory as the federate executable. The format for the FedFile is the same as that described in the Detailed RTI (DRTI) section of this document.

The lookahead value to be used for the distributed simulation is set and examined by the two functions described below. Lookahead refers to the minimum simulation time in the future that a simulation can schedule a new event (i.e., send an update message, see Object Management below). The TM_Time type referenced below is defined as a C “double” in the basic RTI.

Called by the federate to set its lookahead to lookaheadvalue.

Used to return the current lookahead of the federate.

DECLARATION MANAGEMENT SERVICES

These services are used to specify the object classes and instances of objects used in the federation, and to specify which federates should receive messages when instances of objects are modified.

Each class defined and used by the federate has a unique ASCII string name. The initial set of object class names are listed in the FedFile as previously discussed. In general, one may specify more names in the fedfile than are used, however, performance may be reduced if an excessive number of unused names are specified.

Rather than, or in addition to, the class names listed in the fedfile, the application can create additional object classes. Specifically:

Defines a new class with name GroupName. Applications using this procedure to define class names must call RTI_CreateClass() after it has called InitRTI(). Further, after creating all of the class names, it is advisable to perform a barrier synchronization by calling RTIKIT_Barrier() to ensure all of the classes (multicast groups) have been established before any processor begins the simulation.

In addition to the ASCII string name, each object class also has an internal name (a handle), of type RTI_ObjClassDesignator. The RTI_CreateClass function returns this handle. The handle corresponding to a given class name can also be retrieved as follows:
Locates the handle (type `RTI_ObjClassDesignator`) for a given object class name.

It is important to note that only one federate in the distributed simulation should create any given object class. Once the class is created by any federate, all other federates have access to this object class name.

Creating an object class name does not create any objects. The RTI must be notified of the instantiation of object instances by the federate registering instances of object class names. Object instances are referred to by a handle, known as the Object Instance Designator (type `RTI_ObjInstanceDesignator`). This handle is returned by the `RTI_RegisterObjInstance` function, as follows:

```
RTI_ObjInstanceDesignator RTI_RegisterObjInstance(
    RTI_ObjClassDesignator ClassHandle)
```

Creates an instance of an object of class C, and returns a handle for this object instance.

Once object classes and object class instances have been created, the federate must inform the RTI of the intent to publish these instances. In this context, the publishing of an object class instance indicates that at some point, the federate will change some or all of the data in the object class instance, and will inform other federates of the changes. The act of notifying other federates of new values for an object instance is known as updating the object. For any object class for which updates will occur, the federate must inform the RTI of the intention to perform these updates. This is done by publishing the class:

```
RTI_PublishObjClass (RTI_ObjClassDesignator ClassHandle)
```

This service is invoked by a federate to tell the RTI it may be publishing updates to object of class `ClassHandle`.

Next, a federate must inform the RTI of its interest in receiving updates for any given object class. This is a two-step process, first by initializing the subscription information for the object class, and secondly by subscribing to the object class. Since the initialization should be performed only once by a single federate, a subroutine is provided to test for an existing subscription initialization. The three functions that manage object class subscriptions are:

```
Int RTI_IsClassSubscriptionInitialized (RTI_ObjClassDesignator C)
```

This procedure tests whether the object class C has been initialized for subscriptions, and returns TRUE if it has, and FALSE if it has not. `RTI_InitObjClassSubscription` (see below) should only be called if this returns FALSE.

```
RTI_InitObjClassSubscription (RTI_ObjClassDesignator C,
    MCAST_WhereProc P, void* Context)
```

This procedure initializes an `ObjectClassDesignator` for subscriptions, and must be called once before subscribing to C via the `RTI_SubscribeObjClassAttributes` procedure. The P specifies a "whereproc" procedure that is called whenever a message of this class arrives, and is responsible for allocating memory for the incoming message (see the MCAST documentation for a more detailed definition of WhereProc procedures).
The Context parameter is passed to each WhereProc call to allow the WhereProc to
differentiate between messages of different object classes.

**RTI_SubscribeObjClassAttributes** *(RTI_ObjClassDesignator C)*

Notifies the RTI-Kit that this federate wishes to receive new values for all instances of any
object of object class C. The methods for updating objects and receiving object updates
are described below.

**OBJECT MANAGEMENT SERVICES**

Object Management services are used to transmit messages between federates. Sending a
message is known as an Update, and receiving a message is known as a Reflect. To send a
message, use the **RTI_UpdateAttributeValues** function as follows:

```c
EventRetractionHandle RTI_UpdateAttributeValues(
    RTI_ObjInstanceDesignator O,
    struct MsgS*  M, long  MsgSize, long  MsgType)
```

This sends the message specified by M to all federates that have subscribed to the object
class of the object instance O. The total length of the message M is specified by
MsgSize, and the(MsgType) value is passed unchanged to the reflect function described
below. All messages must start with the MsgS structure, defined as follows:

```c
TM_Time     TimeStamp
TM_TagType   TMStuff
EventRetractionHandle erh
unsigned char    Payload[1]
```

The TimeStamp must be filled in by the caller prior to calling
**RTI_UpdateAttributeValues**, and indicates the simulation time of the update event.
This time must be at least as far in the simulated future as the lookahead value
previously discussed. The **TMStuff** field is used by the RTI-Kit and must not be
changed by the federate. The **erh** value is a 32-bit handle that may later be passed to
the **RequestRetraction** subroutine, requesting that the specified event be retracted.
The **Payload** field is the content of the message. The predefined size of the payload is
only one byte, but federates of course can create messages of arbitrary size, passing the
actual size to the update via the **MsgSize** parameter.

Receiving updates is not as straightforward as sending them. The reason is that a given federate
knows precisely when an update is to occur (by calling the Update function discussed above), but
messages arrive asynchronously at arbitrary times. Thus, there is no function in the RTI to call to
receive an update. Rather, the updates are supplied directly to the federate using a **callback**
function. For the BRTI, the callback function must be named **ReflectAttributeValues**, and
must have the signature shown below:

```c
ReflectAttributeValues( TM_Time T, struct MsgS*  pMsg,
    long MsgSize, long  MsgType);
```
Parameter $T$ is the timestamp of the update event, $pMsg$ is a pointer to the actual update message, $MsgSize$ is the size of the message as specified on the update call, and $MsgType$ is the message type specified on the update. The buffer pointed to by $pMsg$ is that provided by the WhereProc. If the WhereProc uses a dynamically allocated buffer, the reflect subroutine is responsible for insuring this buffer is released when it is no longer needed.

**TIME MANAGEMENT SERVICES**

The time management services ensure that messages are delivered to each federate in time stamp order, and that no federate receives a message in its past, i.e., a message with time stamp less than the current simulation time. In the current implementation, all events (attribute reflects) are delivered to the federate in time stamp order.

Time management is implemented by services where the federate requests that its simulation time be advanced, and the RTI responds by issuing a grant when it can guarantee that it is safe to advance to the specified grant time. In this context, safe indicates that there is no chance of later receiving a reflect with an earlier timestamp. Specifically, there are two services to request simulation time advances. The grant is implemented via a callback to the federate in a fashion similar to the reflect discussed above.

Use of these services always result in the following scenario:

1) the federate requests a time advance,

2) the RTI delivers zero or more messages to the federate via \texttt{ReflectAttributeValues} callbacks described above,

3) the RTI notifies the federate its simulation time has been advanced via a \texttt{TimeAdvanceGrant} callback. No additional \texttt{ReflectAttributeValues} callbacks will be made until the next time an advance in simulation time is requested.

The time management services are:

\begin{verbatim}
RTI_NextEventRequest (TM_Time TNE)
\end{verbatim}

This service will normally be used by event driven simulations. In this case, the $TNE$ parameter will normally be the time stamp of the next local event stored within the federate. The federate must know if it is safe to process this event, or if another event carrying a smaller time stamp will be delivered by the RTI. After invoking this service, the RTI will respond in one of two ways:

1) If there are one or more events that will be delivered to the federate with time stamp less than or equal to $TNE$, the event among these containing the smallest time stamp (even including events that will be generated by other federates in the future) will be delivered, as well as all others events containing the same time stamp. The RTI will then call \texttt{TimeAdvanceGrant} with a time parameter equal to the time of this (these) event(s), indicating the simulation time can be advanced to this value.

2) If there is no event with time stamp less than $TNE$ and none will be produced later in the execution, the RTI simply calls \texttt{TimeAdvanceGrant} with parameter $TNE$ to indicate the simulation time can be advanced to $TNE$. At this point, the federate can
now safely process its local event (with time stamp TNE). Upon invoking this service, the federate conditionally guarantees that it will not produce any new messages with time stamp less than TNE+lookahead (lookahead is defined above) presuming it does not receive any additional messages from other federates.

**RTI_TimeAdvanceRequest (TM_Time TS)**

This service will normally be used by time-stepped simulations. The federate calls this service to request that its simulation time be advanced to TS. The RTI will deliver all messages with time stamp less than or equal to TS (in time stamp order) and will then call **TimeAdvanceGrant** with parameter TS after it has guaranteed no additional messages are forthcoming. After invoking this service, the federate unconditionally guarantees it will not produce any new messages with time stamp less than TS + lookahead (lookahead is defined above).

**TimeAdvanceGrant (TM_Time T)**

This is a callback procedure that is called to indicate the federate’s simulation time may be advanced to T. Exactly one call will be produced for each **RTI_NextEventRequest** and **RTI_TimeAdvanceRequest** call. The RTI guarantees that the federate has delivered all events that will ever be produced with time stamp less than or equal to T.

**MISCELLANEOUS SERVICES**

The basic RTI provides the following services to control the overall execution of the simulation and to retract previous updates.

**BRTI_Tick (void)**

The **BRTI_Tick** function must be called periodically to allow the RTI some CPU time to complete the necessary tasks. Specifically, no message reflects or time advance grants will occur unless the **BRTI_Tick** function is called.

**RTI_FlushQueueRequest (TM_Time T)**

The **RTI_FlushQueueRequest** function can be used by the federate to request delivery of all receive order and time stamp order messages as soon as possible. Thus using this service implies that messages are not guaranteed to be delivered in time stamp order to the federate. **RTI_FlushQueueRequest** also requests to the RTI that time be advanced to time T or advance time as far as possible but potentially not at all. This function should not be called more than once between the **BRTI_Tick** call, and should not be called when a time advance request is already pending.

**RTI_Retract (EventRetractionHandle erh)**

The **RTI_Retract** function allows a federate to request the retraction (cancellation) of the event update message specified by erh. An update message can only be successfully retracted if the time stamp of the original message is greater than the value of the last Time Advance Grant plus lookahead.
The **RequestRetraction** is a callback function, which must be defined by the federate using the exact signature shown above. **RequestRetraction** is called by the RTI in response to receiving a retract message and the message to be retracted has been delivered to the federate or the message has not arrived. If the message is in the RTI's internal TSO queue the message will annihilate with the retract message and any memory used will be reclaimed. Note, that **RequestRetraction** will be called if the message to be retracted is delayed in transit and does not arrive before the its retract. It is the federate's responsibility to handle this situation correctly.

**EXAMPLE**

There are several example programs (e.g., tm_ping, phold, qnet) distributed with FDK to illustrate how the BRTI API is used. Tm_ping is perhaps the simplest. It is recommended that this program be examined in detail to illustrate typical usage of the BRTI primitives.

**HLA RTI (DRTI)**

**OVERVIEW**

If you have taken the time to read the previous section, you are probably equipped to begin exploring the HLA. The best source for understanding the HLA Interface is, of course, the specification itself. This is available at [http://hla.dmso.mil](http://hla.dmso.mil). The DRTI uses the 1.3R6 interface specification. Although not all services are implemented, the DRTI includes the entire interface (RTI and Federate Ambassador) in its definition. This allows programs be linked interoperably with DRTI or any other compliant RTI.

**INTERNALS**

The DRTI is, as stated earlier, is an interface to the RTI-Kit. Logically, one might think of the DRTI as having four parts: 1) RTICore required functions, 2) the federate state data, 3) the RTI Ambassador implementation, and 4) auxiliary classes, needed to fit into the HLA paradigm. The DRTI Internal architecture is depicted in Figure 2 DRTI Architecture.

In the figure, one can see how the implementation uses the RTI-Kit. In addition to its internal functionality, the RTI must implement a set of functions required by RTICore. The RTICore required functions are:

```c
char * MyWhereProc (long, void *, long);
void MyHandler (MCAST_Addr Msg, long MsgSize, MCAST_Addr Context, long MsgType);
void MyLBTSDone (TM_Time, long);
fdkErrorCode DeliverOneMessage (TM_Time TimeStamp, char *cMsg, long MsgSize, long MsgType);
fdkErrorCode DeliverROEvents ();
fdkErrorCode DeliverTSOEvents (TM_Time TARTime);
void TimeAdvanceGrantWrapper (TM_Time theTime);
TM_Time DRTI_TSOMin (void);
char * DRTI_TSOPop (TM_Time *minTS, long *MsgSize, long *MsgType);
```
The requirements for these functions are similar for any RTI, and one can gather a good understanding by reviewing the code. Two functions, MyHandler, and DeliverOneMessage, deserve special explanation. MyHandler is the function that is called whenever a message comes in off the wire that is not handled internal to the RTI-Kit itself. In the DRTI, we must handle Receive-Order and Time-Stamp-Order messages, so we have two methods for queing these messages. Of course, in some implementations it might be appropriate to take additional actions and deal with the message upon arrival. In the case of the DRTI, messages are always either handled in RO or TSO.

DeliverOneMessage is the function that is called when the RTI determines that is appropriate to dispose of a queued message. This function must be able to determine the type of message, (from the message format) and how to dispose of it. In the case of incoming updates, or interactions, the function creates the appropriate data container (a Handle-Value Pair Set) and calls a Federate Ambassador method.

The RTI/Federate state is contained in a single struct called RTIambPrivateData, shown below. (This is one reason that the RTI cannot currently support more than one federation). The struct is instantiated in two places. The majority of the initialization is conducted in the RTI Ambassador constructor. Federation specific information is initialized in joinFederation(). Notice that nothing is required in createFederationExecution, because only one federation can exist at a time, and because of RTI-Kit, federate 0 is always responsible for creating the federation execution. Much of this functionality is expected to change over time.
The I/F Specification

The HLA Interface Specification defines sets of services to support realization of distributed simulations. The Runtime Infrastructure (RTI) in HLA is software that implements those services. The DRTI software implements services in five of the categories defined in the HLA Interface Specification:

**FEDERATION MANAGEMENT** services initialize the execution of the federation.

**DECLARATION MANAGEMENT** services define object and interaction classes and set up communications between federates using a newsgroup-like publish/subscribe paradigm.

**OBJECT MANAGEMENT** services allow federates to declare object instances, update attributes, send interactions, receive updates to attributes, and receive interactions produced by other federates.

**TIME MANAGEMENT** services control the advancement of simulation time within each federate, and prevent federates from receiving messages in their past (i.e., time stamp less than the federate's current simulation time).

**SUPPORT** services provide mapping between string representations of names and integer handles used in the other services, and provide miscellaneous utilities which do not neatly fit into other categories.

There are two other sets of services defined in the HLA. The **OWNERSHIP MANAGEMENT** services allow one federate to transfer ownership of object instance attributes to another federate (at which point the second federate would be responsible for updating those attributes). These are not implemented in the current version of DRTI. The **DATA DISTRIBUTION MANAGEMENT** services...
allow you to attach "regions" of interest to publications and subscriptions (to allow minimization of unnecessary network traffic). An initial version of a library for realizing the DDM services, called DDM-Kit, is included. All of the data types used in the RTI interface are declared within a class named RTI which acts as a namespace (requiring the scope qualifier RTI:: before each type name).

The C++ class RTI::RTIambassador contains methods for each of the services implemented in the RTI. The abstract C++ class RTI::FederateAmbassador declares the methods that must be defined by the federate (these are the callback functions used to notify a federate when a message has been received).

**Coverage**

Currently, the DRTI implements over half of the 1.3R6 Interface Specification. The implementation goals have been driven by research priorities. Changes, improvements, contributions are, of course, welcome, and will be considered for inclusion in the general FDK distribution with appropriate citation of the source.

<table>
<thead>
<tr>
<th>I/Fspec Implementation</th>
<th>Support</th>
<th>A</th>
<th>A*</th>
<th>N</th>
<th>Grand Total</th>
<th>Percent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Service Class</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4. Federation Management Services</td>
<td></td>
<td>1</td>
<td>3</td>
<td>7</td>
<td>11</td>
<td>36%</td>
</tr>
<tr>
<td>5. Declaration Management Services</td>
<td></td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>8</td>
<td>50%</td>
</tr>
<tr>
<td>6. Object Management Services</td>
<td></td>
<td>3</td>
<td>5</td>
<td>8</td>
<td></td>
<td>38%</td>
</tr>
<tr>
<td>7. Ownership Management Services</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0%</td>
</tr>
<tr>
<td>8. Time Management Services</td>
<td></td>
<td>10</td>
<td>2</td>
<td>7</td>
<td>19</td>
<td>63%</td>
</tr>
<tr>
<td>9. Data Distribution Management</td>
<td></td>
<td>3</td>
<td>9</td>
<td>12</td>
<td></td>
<td>25%</td>
</tr>
<tr>
<td>10. RTI Support Services</td>
<td></td>
<td>21</td>
<td>8</td>
<td>29</td>
<td></td>
<td>72%</td>
</tr>
<tr>
<td>12. Misc.</td>
<td></td>
<td>5</td>
<td></td>
<td>5</td>
<td></td>
<td>100%</td>
</tr>
<tr>
<td>Grand Total</td>
<td></td>
<td>45</td>
<td>7</td>
<td>49</td>
<td>101</td>
<td>51%</td>
</tr>
</tbody>
</table>

* A Service Available, functional
* A* Service Available, partially functional
* N Call available, not functional

Table 1 - DRTI I/F Spec Implementation

**Federation Management Services**

The joinFederationExecution service is used to initialize the RTI and to define the object classes and interaction classes that are valid for the federation. No other RTI function should be invoked prior to making this call! Constants indicating the number of federates (RTIKIT_numbnodes) and the ID of this federate (RTIKIT_nodeid) are undefined until this procedure is called. The initialization procedure is:

```cpp
FederateHandle RTIambassador::joinFederationExecution(const char *FedFileName, FederateAmbassador *pFedAmb);
```

Where FedFileName is the name of the file defining the object and interaction class names and the inheritance relationship between them, and pFedAmb is a pointer to an C++ object that is inherited from FederateAmbassador and defines all of the callbacks needed by the RTI to inform the federate when messages have been received (the required callbacks are described in the object management and time management service groups). The handle returned is the federate's unique identifier in the federation.
(this implementation assigns federate handles 0, 1, ... n-1, where n is the number of federates in the federation).

The format of the fedfile is as defined for the 1.3 I/F specification. However, the current DRTI does not accept comments in the fed file.

Note that all federates joining the federation must supply identical fed files. Otherwise, the handles assigned to go with the names will not match, and the federation will not work correctly.

DECLARATION MANAGEMENT SERVICES

These services are used to specify those object and interaction classes for which a federate intends to send messages, and specifies those classes for which a federate desires to receive messages.

Some means is required to specify which federates are to be notified when an interaction is sent, an object instance is created, or an attribute of an object instance is updated. A publish/subscription mechanism not unlike Internet newsgroups is used for this purpose. Specifically, the HLA uses something called "class-based filtering." This means a federate can subscribe to receive all updates to all instances of objects of a certain class. For example, to get a message whenever any tank moves, a federate can subscribe to the Tank class. Note there is no mechanism to only get updates for a specific object instance, only for all objects of some class. DDM mechanisms are defined in the HLA for this type of data filtering.

The relevant RTI services for publishing and subscribing are:

```
RTIambassador::publishObjectClass(ObjectClassHandle theClass, const AttributeHandleSet& attributeList)
```

This service is invoked by a federate to tell the RTI it may register object instances of class `theClass`, and it may update the attributes in `attributeList`.

```
RTIambassador::subscribeObjectClassAttributes(ObjectClassHandle theClass, const AttributeHandleSet& attributeList)
```

This service is invoked by a federate to specify it is to receive a notification (via object management services) whenever any instance to any object of class `theClass` is registered and when any of the attributes in `attributeList` are updated.

```
RTIambassador::publishInteractionClass(InteractionClassHandle theClass)
```

This service is invoked by a federate to tell the RTI it may send interactions of class `theClass`.

```
RTIambassador::subscribeInteractionClass(InteractionClassHandle theClass)
```

This service is invoked by a federate to specify it is to receive a notification (via object management services) whenever an interaction of class `theClass` is sent.
OBJECT MANAGEMENT SERVICES

These services are used to transmit messages between federates. A message will be sent whenever a federate creates (registers) an instance of an object class, updates the attributes of an object instance, or sends an interaction. Callbacks are used to receive messages. When the RTI is ready to deliver a message to the federate, it calls one of the methods of the FederateAmbassador class. FederateAmbassador is an abstract class. This means the federate developer has to derive a class from FederateAmbassador and provide definitions for each of the methods.

When a federate creates an instance of an object (of some class specified in the FedFile), it must notify the RTI of this fact by REGISTERING the instance. When an instance is registered, the RTI returns a handle for the object instance that is used in future references to it, e.g., to update attributes of the object. The type ObjectHandle denotes a reference to an object instance.

Specifically, the following object management services are defined:

```
ObjectHandle
RTIambassador::registerObjectInstance(ObjectClassHandle theClass)
```

Register an instance of an object of class `theClass`, and return a handle for this object instance.

```
FederateAmbassador::discoverObjectInstance(ObjectHandle theObject, ObjectClassHandle theObjectClass, const char *theObjectName)
```

The RTI will call this method when some other federate has registered an instance of some object class that you subscribed. Or, if a federate registers an instance of an object class that is a subclass of one that you subscribed, you will discover that object as an instance of the class you subscribed. Here is an example:

```
fed file:
(objects
 (class A
  (class B
   (class C )
  )
 )
)
Federate1    Federate2
--------    --------
publish A    subscribe B
publish B
publish C
register A
  (nothing happens - Federate2 won’t discover As)
register B    discover B
register C    discover B (because this is the closest subscribed superclass)
```
RTIambassador::updateAttributeValues (ObjectHandle theObject,
    const AttributeHandleValuePairSet& theAttributes,
    const FedTime& theTime, const char *theTag)

Update the attributes of object instance theObject, using time-stamp order; the update occurs at theTime. Attribute handle/value pairs should be added to theAttributes before making this call. theTag is not currently used by DRTI.

FederateAmbassador::reflectAttributeValues(ObjectHandle
    theObject, const AttributeHandleValuePairSet&
    theAttributes, const FedTime& theTime, const char
    *theTag)

The RTI will call this method when the federate who owns theObject updates its attributes (using the version that includes a time stamp). The update occurs at theTime, so the RTI will deliver this message in time-stamp order, in accordance with time advances and grants. The RTI will call this method only after it has called discoverObjectInstance (so it will only use this method for object instances that you discover because you have subscribed to their class, or a superclass as described above). theTag is currently just a dummy value (eventually it would be the tag value supplied by the owner when it made the update).

RTIambassador::updateAttributeValues (ObjectHandle theObject,
    const AttributeHandleValuePairSet& theAttributes,
    const char *theTag)

Update the attributes of object instance theObject, using receive order. Attribute handle/value pairs should be added to theAttributes before making this call. theTag is not currently used by DRTI.

FederateAmbassador::reflectAttributeValues(ObjectHandle
    theObject, const AttributeHandleValuePairSet&
    theAttributes, const char *theTag)

The RTI will call this method when the federate who owns theObject updates its attributes (using the version that does not include a time stamp). There is no time associated with this update; it is delivered in receive order (i.e., whenever it is received -- time advance requests and grants do not affect the delivery of this message). theTag is currently just a dummy value.

RTIambassador::sendInteraction (InteractionClassHandle
    theInteraction, const ParameterHandleValuePairSet&
    theParameters, const FedTime& theTime, const char
    *theTag)

Send a message of class theInteraction, consisting of theParameters, using time-stamp order. The message occurs at theTime. Parameter handle/value pairs should be added to theParameters before making this call. theTag is not currently used by DRTI.
FederateAmbassador::receiveInteraction (InteractionClassHandle theInteraction, const ParameterHandleValuePairSet& theParameters, const FedTime &theTime, const char *theTag)

The RTI will call this method when a federate sends an interaction of class theInteraction (using the version that includes a time stamp), and you are subscribed to theInteraction or to a superclass of theInteraction. The rules for receiving interactions are like those for discovering instances. The message occurs at theTime, so the RTI will deliver it in time-stamp order, in accordance with time advances and grants. theTag is currently just a dummy value (eventually it would be the tag value supplied by the federate who sent the message).

RTIambassador::sendInteraction (InteractionClassHandle theInteraction, const ParameterHandleValuePairSet& theParameters, const char *theTag)

Send a message of class theInteraction, consisting of theParameters, using receive order. Parameter handle/value pairs should be added to theParameters before making this call. theTag is not currently used by DRTI.

FederateAmbassador::receiveInteraction (InteractionClassHandle theInteraction, const ParameterHandleValuePairSet& theParameters, const char *theTag)

The RTI will call this method when a federate sends an interaction of class theInteraction (using the version that does not include a time stamp), and you are subscribed to theInteraction or to a superclass of theInteraction. There is no time associated with this update; it is delivered in receive order (i.e., whenever it is received -- time advance requests and grants do not affect the delivery of this message). theTag is currently just a dummy value.

Here is an example of send and receive using a class hierarchy:

```
fed file:
  (interactions
   (class A
    (class B
     (class C
      )
    )
   )
  )
```

Federate1    Federate2
---------    ---------
publish A    subscribe B
publish B
publish C
send A        (nothing happens -- Federate2 won’t receive As)
send B        receive B
send C        receive B (because this is the closest subscribed superclass)
OWNERSHIP MANAGEMENT SERVICES
At present, these services are not implemented.

TIME MANAGEMENT SERVICES
The time management services ensure that messages sent with a time stamp are delivered to each federate in time-stamp order, and that no federate receives a time-stamped message in its past, i.e., a message with time stamp less than the federate's current simulation time. In the current implementation, updates and sends that include a time stamp are delivered in time-stamp order. All other messages are delivered in receive order.

Time management is implemented by services where the federate requests that its simulation time be advanced, and the RTI responds by issuing a GRANT when it can guarantee the time advance will not later result in a message in the federate's past. Specifically, there are two services to request simulation time advances. The grant is implemented via a callback to the federate.

Use of these services always results in the following scenario:

1) the federate requests a time advance and then calls tick,

2) the RTI delivers zero or more messages to the federate via the reflectAttributeValues and/or receiveInteraction callbacks in FederateAmbassador,

3) the RTI notifies the federate its simulation time has been advanced via a timeAdvanceGrant callback. No additional reflectAttributeValues and/or receiveInteraction callbacks *with time stamps* will be made until the next time an advance in simulation time is requested. (If messages without time stamps are received, they will be delivered the next time the federate calls tick, regardless of whether an advance in simulation time has been requested.)

The time management services are:

```cpp
RTIambassador::timeAdvanceRequest(const FedTime& theTime)
```
This service will normally be used by time-stepped simulations. The federate calls this service to request that its simulation time be advanced to theTime. The RTI will deliver all messages with time stamp less than or equal to theTime (in time stamp order) and will then call FederateAmbassador::timeAdvanceGrant with parameter theTime after it has guaranteed no additional messages are forthcoming. After invoking this service, the federate unconditionally guarantees it will not produce any new messages with time stamp less than theTime + lookahead (lookahead is defined below).

```cpp
RTIambassador::nextEventRequest(const FedTime& theTime)
```
This service will normally be used by event-driven simulations. In this case, theTime will normally be the time stamp of the next local event stored within the federate. The federate must know if it is safe to process this event, or if another event carrying a smaller time stamp will be delivered by the RTI. When this service is invoked, the RTI will respond in one of two ways:
1) If there are one or more events that will be delivered to the federate with time stamp less than or equal to theTime, the event among these containing the smallest time stamp (even including events that will be generated by other federates in the future) will be delivered, as well as all other events containing the same time stamp. The RTI will then call FederateAmbassador::timeAdvanceGrant with time parameter equal to the time of this (these) event(s), indicating the federate's simulation time has been advanced to this value.

2) If there are no events with time stamp less than theTime and none will be produced later in the execution, the RTI simply calls FederateAmbassador::timeAdvanceGrant with parameter theTime to indicate the federate's time has been advanced to theTime. At this point, the federate can now safely process its local event (with time stamp theTime). Upon invoking this service, the federate conditionally guarantees that it will not produce any new messages with time stamp less than theTime + lookahead (lookahead is defined below) IF it does not receive any additional messages from other federates.

FederateAmbassador::timeAdvanceGrant(const FedTime& theTime)

This federate-defined procedure is called to indicate the federate's simulation time has been advanced to theTime. Exactly one call will be made for each call to RTIambassador::nextEventRequest and RTIambassador::timeAdvanceRequest. The RTI guarantees that the federate has received all events that will ever be produced with time stamp less than or equal to theTime.

Lookahead is a concept used to improve performance. If a federate's lookahead is L, then the federate guarantees that any new time-stamped messages it sends (via calls to RTIambassador::updateAttributeValues or sendInteraction) will have a time stamp of at least T+L, where T is the federate's current time. In other words, the federate guarantees it will only generate time-stamped messages at least L units of time into the future.

The following services are defined concerning lookahead:

RTIambassador::modifyLookahead(const FedTime& theLookahead)

Called by the federate to set its lookahead to theLookahead.

RTIambassador::queryLookahead(FedTime& theTime)

theTime is an output parameter used to return the current lookahead of the federate.

In the current implementation the lookahead is initialized to zero, and can only be increased during the execution.

DATA DISTRIBUTION MANAGEMENT

Data Distribution Management services are not implemented in the default DRTI compilation. However, an experimental DDM Module can be used to compile a DRTI which supports DDM. Please see the DDM Module documentation for details.
RTI SUPPORT SERVICES

The set of valid object class and interaction class names, along with their attribute and parameter names, are defined using ASCII strings in the fed file. The name of a subclass is determined by concatenating its parent class name with the subclass name using a dot ‘.’ as a separator. So, in the fed file example above the three class names are "A", "A.B", and "A.B.C". Class names are not case sensitive.

Each ASCII string name is associated with an internal name (a handle) of type RTI::ObjectClassHandle, RTI::InteractionClassHandle, RTI::AttributeHandle, or RTI::ParameterHandle. The support services provide a way to get the handle for a certain name and vice versa.

Note that for the getName methods, memory is allocated to return the string. You are responsible for freeing this memory when you are finished with it.

```cpp
ObjectClassHandle RTIambassador::getObjectClassHandle(const char *theName)

Returns the handle for the object class with name theName.
```

```cpp
char * RTIambassador::getObjectClassName(ObjectClassHandle theHandle)

Returns the name for the object class with handle theHandle.
```

```cpp
AttributeHandle RTIambassador::getAttributeHandle(const char *theName, ObjectClassHandle whichClass)

Returns the handle for the attribute with name theName belonging to object class whichClass.
```

```cpp
char * RTIambassador::getAttributeName(AttributeHandle theHandle, ObjectClassHandle whichClass)

Returns the name of the attribute with handle theHandle belonging to object class whichClass.
```

```cpp
InteractionClassHandle RTIambassador::getInteractionClassHandle(const char *theName)

Returns the handle for the interaction class with name theName.
```

```cpp
char * RTIambassador::getInteractionClassName(InteractionClassHandle theHandle)

Returns the name for the interaction class with handle theHandle.
```
ParameterHandle RTIambassador::getParameterHandle(const char *theName, InteractionClassHandle whichClass)

Returns the handle for the parameter with name theName belonging to interaction class whichClass.

char * RTIambassador::getParameterName(ParameterHandle theHandle, InteractionClassHandle whichClass)

Returns the name of the parameter with handle theHandle belonging to interaction class whichClass.

RTIambassador::tick()

Use this service to give the RTI time to process incoming messages. All callbacks to the FederateAmbassador will be made during a call to tick. Thus, any time you make a call to timeAdvanceGrant or nextEventRequest, you must then call tick to be able receive the timeAdvanceGrant.

**ADDITIONS TO THE I/F SPEC**

The DRTI adds one service to the RTIAmbassador. Barrier() is a simple barrier which blocks execution of the federate until all federates have called Barrier(), whereupon, all federates continue. This barrier synchronization is a simple and more straightforward implementation than the named, non-blocking synchronization point paradigm in the HLA I/F Spec.

**AUXILIARY CLASSES AND CODE**

In addition to the rti.cpp, the DRTI makes use of several other compilation modules. This section explains the purpose of the different modules.

**AHVPS**

The HVPSImpl and AnPHVPS, module is the implementation and extension of an abstract handle-value pair set. The HVPS makes use of the DRTI message formats. This is desirable because the main reason to create an HVPS is to send or receive message in the form of updates or interactions. The function of the HVPS, as a container for handle-value pairs, is to marshall them into a correctly formatted message which the DRTI can send quickly, whenever the user invokes the appropriate service.
class HVPS {
private:
  DRTI_ULong MaxCount; // the Maximum number of handle value pairs
  DRTI_ULong MsgSize; // the size of the MsgS to be sent
  MsgS *pMsg; // Pointer to the message
  ItemInfoS *pItemInfo; // point to the beginning of the HV pairs
  ItemS *IndexedItems[MAX_HVPS_ITEMS]; // to allow removal, must map user // indices
  bool packed;
public:
  HVPS();
  HVPS(DRTI_ULong theCount) throw (fdkErrorCode);
  ~ HVPS();
  void reset(MsgS *Msg, DRTI_ULong Size, DRTI_Handle Class, bool HVPStype);
  void release();
  void send(const TM_Time& Time, DRTI_Handle Handle,
            MCAST_Handle h, long MsgType, RTI::EventRetractionHandle erh);
  DRTI_ULong size() const ;
  DRTI_Handle getHandle(DRTI_ULong i) const;
  DRTI_ULong getValueLength(DRTI_ULong i) const;
  void getValue(DRTI_ULong i, char *buff, DRTI_ULong& valueLength) const;
  char* getValuePointer(DRTI_ULong i, DRTI_ULong& valueLength) const;
  void add(DRTI_Handle h, const char *buff, DRTI_ULong valueLength);
  void remove(DRTI_Handle h);
  void empty();
  void pack();
};

RTIMessageFmts.hh
The DRTI defines its message formats in a single file. These messages are simple C structs. When adding additional functionality to the RTI, one would normally start with additions to this file.

define enum MsgTypeE {
  MSG_INSTANCE, // notification that new object instance has been registered
  MSG_UPDATE, // attribute value updates for an object instance
  MSG_INTERACTION, // interaction parameters
  MSG_RETRACTION // message which specifies which event to retract
};
// info provided for each attribute or parameter
struct ItemS {
  DRTI_Handle Handle; // The Attribute or Parameter Handle
  DRTI_ULong ByteCount; // how many bytes of data
};

struct InstanceInfoS {
  DRTI_Handle ClassHandle;
  DRTI_Handle InstanceHandle; // federation-unique
  char InstanceName[MAX_NAME_LENGTH + 1]; // federation unique; not enforced.
    // MAX_NAME_LENGTH is from RTItypes.hh
};

struct ItemInfoS {
  DRTI_Handle Handle; // either instance handle or interaction
    // class handle for the transmitted msg,
  DRTI_ULong message_erh; // not necessarily the subscribed class
  DRTI_ULong ItemCount;
};

struct MsgS {
  TM_Time TimeStamp; // logical time time stamp
  TM_TagType TMStuff; // information used by TM-Kit
  DRTI_Handle sender; // The sender; this is used for erh and
    // filtering locally
  union MsgU {
    InstanceInfoS InstanceInfo; // for a INSTANCE message (discover)
INFO CLASSES

Info classes serve as state vector holders for two types of information. The info classes AttributeInfo, ObjectClassInfo, InteractionClassInfo, ParameterInfo, and SpaceInfo, hold information read from the fed file at initialization time. Some of this information can be modified at runtime, but these info classes are mostly read. The instance info classes are generated at runtime for registered and discovered objects. The interface to the info classes is below.

```cpp
struct AttributeInfo {
    string Name;
    bool isSubscribed;
    bool isPublished;
    DRTI_Handle Handle;
    DRTI_Handle Transport;
    DRTI_Handle Ordering;
    DRTI_Handle Space;
};
struct ObjectClassInfo {
    string Name;
    string FullName;
    bool isSubscribed;
    bool isPublished;
    DRTI_Handle Handle;
    MCAST_Handle MCASTh;
    multitree<DRTI_Handle>::node_iterator ClassSubTree;
    vector<DRTI_Handle> attributes;  // Including inherited attributes
    bool SubscriptionSet[MAX_ITEMS] ;  // a bitset indicating attributes subscribed
    ObjectClassInfo() {}  
    void addAttribute(DRTI_Handle h) { attributes.push_back(h); }
};
struct ParameterInfo {
    string Name;
    DRTI_Handle Handle;
};
struct InteractionClassInfo {
    string Name;
    string FullName;
    bool isSubscribed;
    bool isPublished;
    DRTI_Handle Handle;
    DRTI_Handle Transport;
    DRTI_Handle Ordering;
    DRTI_Handle Space;
    MCAST_Handle MCASTh;
    multitree<DRTI_Handle>::node_iterator ClassSubTree;
    vector<DRTI_Handle> parameters;
    bool SubscriptionSet[MAX_ITEMS] ;  // a bitset indicating parameters belong to // the class
    InteractionClassInfo() {}  
    void addParameter(DRTI_Handle h) { parameters.push_back(h); }
};
struct SpaceInfo {
    string Name;
    DRTI_Handle Handle;
    vector<string> DimensionNames;
};
struct AttributeInstanceInfo{
    string Name;
    bool isRelevant;
};
```
bool isInScope;
bool isOwnedByFederate;
DRTI_Handle Handle;
DRTI_Handle Transport;
DRTI_Handle Ordering;
DRTI_Handle Space;
};

struct ObjectInstanceInfo {
  string Name;
  DRTI_Handle Handle;
  DRTI_Handle Class;  // by virtue of promotion.
  DRTI_Handle RegisteredClass;  // the class that was originally registered
  bool Discovered;  // whether or not this federate knows about this
                     // object
  bool owned;       // since we don't yet support ownership, this
  //  MCAST_Handle MCASTh;  // the default space
  //  vector<AttributeInstanceInfo> publishedAttributes;
  //  void addAttribute(AttributeInstanceInfo a) { publishedAttributes.push_back(a); }
};

### MULTITREE CONTAINER TEMPLATE

Multitree is a C++ template designed to hold information about the inheritance tree of the object and interaction classes.

### FEDPARSER

**FedParser** is a module that does just that: it parses the fed file. Its sole function is to initialize the Info classes at federation creation time. It is a fairly delicate hand crafted parser, and has not been tested rigorously, but has performed well. One limitation for this parser is that it does not yet handle comments (\(\text{;};\)) properly. This is planned fix, but has not been a practical limitation. The two functions in the module are listed below.

```c
const char *getToken(FILE *aFile);
fdkErrorCode ReadFedFile(const char *FedFileName);
```

### FEDTIME.CPP

The code in FedTime is used to provide RTI time types. Since the RTI-Kit must also deal with time, FedTime simply provides a mapping between RTI time types and the time types expected by RTI-Kit. This code implements the **RTIfedTime** class. The code ties to the RTI-Kit **TM Time** and must ensure that the internal (**timedefs.h**) and external (**FedTime**) representations do not make assignment errors or introduce problems into the LBTS computation, etc.

### REGIONINFO

**RegionInfo** and **RegionSet** are classes to contain information about regions created by a federate. A **RegionInfo** class is created upon calling **CreateRegion**, and contains an array of extents. Since this information also may relate to routing (DM and DDM) **ClassInfo** retains information about mcast groups. During join (but should be during create fedex) the parsing of the fed file results in the creation of MCast groups based on the value of **ClassSet.CreateMCast**.
**ADDING ADDITIONAL SERVICES TO THE DRTI**

To add additional functions to the RTI, one would consider several typical activities. The beginning of a design process would include a protocol, if communications between the RTI instances were required. After a workable protocol is designed, one would consider the changes to the RTI state vector. This is normally a change to the `RTIambPrivateData` struct. (don’t forget to initialize!) Another issue to consider is the addition of changes required to the `RTIMessages.hh`. Finally, one would include a message handler, and ensure that it is invoked appropriately (normally in `DeliverOneMessage()`).

**USING THE DRTI**

Following the example in the DMSO RTIs, the header files are in DRTI/lang/C++/include. The libraries are compiled to an architecture dependent directory, for example: DRTI/SRC/OBJ/ARCH/FM/libOBJ.o. After inclusion in a library, they are moved to DRTI/LIB, and finally linked to DRTI/lang/C++/libUnfortunately we don’t have libKit.a and libDRTI.a combined right now. This means that a federate will have to link both libraries (statically). Eventually, they will be combined into a shared object, ala libRTI.so.
PART III: RTI-Kit

This chapter is oriented towards the RTI developer, and describes the interfaces to the libraries used to create an RTI. Persons interested in only using the RTIs "as is" without making additional modifications can skip this chapter.

The RTI-Kit architecture, shown in Figure 3, is designed to minimise the number of software layers that must be traversed by distributed simulation services. For example, TM-Kit does not utilise the MCAST library for communication, but rather directly accesses the low-level primitives provided in FM-Lib. This is important in cluster computing environments because low-level communications are on the order of a few microseconds latency for short messages, compared to hundreds of microseconds or more when using conventional networking software such as TCP/IP. Thus, if not carefully controlled, overheads introduced by RTI software could severely degrade performance in cluster environments, whereas such overheads would be insignificant in traditional networking environments where the time required for basic communication services is very high.

![Figure 3. Architecture for RTI-Kit](image)

**FM API**

**RATIONALE**

The current implementations of TM-Kit and MCAST interface to the communications system through FM-Lib. FM-Lib is essentially identical to the Illinois Fast Messages (FM 2.0) API, as described in (Pakin, Lauria et al. 1996), with modest additional functionality. Key features of FM include:

- FM provides point-to-point, reliable, ordered communications between processors.
- FM is designed for low latency, low overhead communications, especially in cluster computer environments interconnected by MyriNet (Myricom 1996). It is particularly well suited for parallel computation applications that send many small messages.
- Message sends are performed by opening a stream to the destination, sending individual parts of the message, and then closing the stream.
• Message receives are performed through message handlers. Each message that is sent specifies a handler (i.e., a procedure) that is automatically called by FM when the message is received at the destination. The handler is specified by having each processor set up a table during initialization containing procedure pointers. The index into this table is transmitted in the header of the message so the receiver knows which handler to call. The message handler then calls `FM_receive()` to receive the message.

• To service the wire, the application must call a procedure called `FM_extract()`. This procedure will then call message handlers for any incoming messages.

IMPLEMENTATION

When a message comes in off the wire, FM must have some means of knowing which handler to call. This is accomplished as follows.

1) Each processor defines an array that maps integer handle numbers (the index into the array) to a pointer to the message handler function.

2) When a message is sent, the sender specifies an integer handler number that identifies which handler is to be called at the receiving end.

3) When the message is received, FM removes the handle ID and calls the procedure in its handler table.

FM requires the application to fill in the handler array in each processor to map handle IDs to function pointers. The array must be set up in a consistent fashion in each processor to ensure a common, global mapping of handler IDs to handle functions is achieved.

If there are a set of separate modules using FM, as is the case here, there must be some means for each module to map handle IDs to message handlers so there are no conflicts in assigned IDs. To address this problem, FM-Lib augments FM by providing a procedure that allocates handler IDs, and binds them to handler functions by filling in the appropriate FM array. Specifically, to bind a handler function to an ID, the following procedure should be used:

```c
void MCAST_RegisterHandler (unsigned int *HandleID, FM_handler hfunction())
```

This procedure allocates the next available handle number and sets that entry in the handler table to point to the function `hfunction()`. Handler numbers are assigned sequentially, starting from zero. The assigned handle ID is returned to the caller in `HandleID`. This establishes the linkage so that messages of type `HandleID` will result in calls to `hfunction()`. Message sends must specify `HandleID` when invoking the FM function to send a message to cause the appropriate handler to be called at the receiving end. An error message is generated and the program is aborted if the application attempts to register more handlers than there are entries in the FM array.

Each processor must call this function for each handler, and in the same order, in order to ensure identical handler tables are set up across all of the processors. Since this binding of handle numbers to functions normally occurs during initialization, this means each processor using RTI-Kit libraries must initialize the libraries it is using in the same order. Users of RTI-Kit need not be concerned with this because RTI-Kit's `RTIKIT-Init()` automatically takes care of this. This is an important detail, however, for applications that bypass RTI-Kit and use FM-Lib directly.
void FM_initialize(void)

This procedure must be called before any FM services can be called.

Messages are sent using a three-part process. First an FM_stream must be opened by calling FM_begin_message.

FM_stream * FM_begin_message(ULONG recipient_id, ULONG msg_length, ULONG handler_id)

The destination of the message is the FM node identified by recipient_id, the message size is msg_length, and handler_id informs FM which handler to call when the message is received. Once the FM_stream is open, the message is constructed by making one or more calls to FM_send_piece.

void FM_send_piece(FM_stream *sendstream, void *data_buffer, ULONG data_length)

FM_send_piece accepts as arguments the FM_stream to use, a pointer to a data buffer containing the data to be sent, and the size of data in the data buffer. Once the message is constructed calling FM_end_message sends it.

void FM_end_message(FM_stream *sendstream)

Receiving messages is a two-part process. First FM_extract must be called to begin the process of extracting the message off the wire. FM_extract will extract up to maxbytes off the wire. However, if there is less than maxbytes to extract FM_extract will extract what it can and return to the caller. FM_extract will not extract partial messages off the wire. If a message is partially extracted when the maxbytes limit is reached FM_extract will read the entire message off the wire. So there is a possibility that FM_extract will extract more than maxbytes off the wire.

ULONG FM_extract(ULONG maxbytes)

FM_extract first extracts the size of the message and the identifier of the FM handler to use to process the message. Then FM_extract calls the FM handler to finish the process of extracting the message from the wire.

int FM_handler_Name(FM_stream *, unsigned MsgSize)

void FM_receive(void *data_buffer, FM_stream *receivestream, ULONG data_length)

The FM handler calls FM_receive to pull each piece of the message off the wire. The pieces of the message coming off the wire are placed in a data buffer pointed to by data_buffer.
As an example consider a message composed of two integers. The message is constructed and sent by calling \texttt{FM\_begin\_message} to open an \texttt{FM\_stream}, calling \texttt{FM\_send\_piece} for each integer, and finally calling \texttt{FM\_end} message to send the message. On the receiving side, \texttt{FM\_extract} will read in enough of the message to determine its size and which \texttt{FM\_handler} to call then call the \texttt{FM\_handler} to complete the task of reading in the remainder of the message. The \texttt{FM\_handler} will read the remainder of the message, namely the two integers, by calling \texttt{FM\_receive} twice, once for each integer.

### MCAST

#### RATIONALE

MCAST provides reliable group communication services. A \textit{group} is the central abstraction used in the communication library. A group contains a list of \textit{subscribers} to the group. Whenever a message is sent to the group, each subscriber is notified by a call to a procedure (a message handler) defined by the subscriber. A single processor may hold multiple subscriptions to a group, in which case it receives a call back for each one when a message for the group is received. Any processor can send a message to any group; in particular, one need not be subscribed to the group in order to send a message to it. If the processor sending a message to the group is also subscribed to the group, it will still receive a call back for each subscription, i.e., processors can send messages to themselves using this service. Of course, the message handler can simply ignore messages sent by a processor to itself if this is the desired semantics.

Each group has two unique names that are used to identify it: (1) a unique ASCII string name, and (2) an internal handle (of type \texttt{MCAST\_Handle}) that serves as a pointer to the group. The internal name is required to send/receive messages to/from the group. A name server is provided that maps the ASCII names of groups to their internal name. To maximize performance, the ASCII name should only be used to initially obtain a handle to the group, and the handle should be used thereafter.

#### IMPLEMENTATION

### INITIALIZATION

Each processor must call the following procedures when it begins executing in order to ensure proper initialization of the library. These procedures must be called before \texttt{RTIKIT\_Init()} is called.

```c
void RTIKIT\_UsingMCAST (void)
```

This procedure sets a flag in RTI-Kit to indicate the MCAST library will be used.

#### CREATING AND OBTAINING POINTERS TO GROUPS

The following types are defined by MCAST:

- \texttt{MCAST\_Handle}: internal name for a group

33
**MCAST_Addr**: pointer to a memory location (an address)

The procedures defined below are available for creating and obtaining handles to groups:

```c
long MCAST_Create (const char *Name, MCAST_Handle *Handle, long
                   TransportType)
```

*MCAST_Create* creates a new group. It allocates memory for a group handle, fills it in, and returns a pointer to the handle in *Handle*. It is the caller's responsibility to reclaim memory for the handle if it is no longer needed. *Name* is the unique, case sensitive, ASCII string name assigned to the group. It is an error to create two groups with the same ASCII string name. *TransportType* should be *MCAST_RELIABLE* if reliable communication is to be used, or *MCAST_BEST_EFFORT* if best effort delivery is to be used. The value returned by this procedure is one of the following:

- 0 indicates the operation completed successfully
- *MCAST_MemoryError* indicates memory could not be allocated for the handle.
- *MCAST_DuplicateName* indicates another group with the same name already exists.

In the current implementation, performance will be maximized if the processor creating the group is also the one that most frequently sends messages to the group.

```c
long MCAST_GetHandle (char *Name, MCAST_Handle *Handle)
```

*MCAST_GetHandle* is used to obtain the handle for the group who's ascii string name is *Name*. If the group exists, memory for a handle is allocated, and a pointer to the handle is returned to the caller in *Handle*. If the group does not exist or an error occurred, the value returned in *Handle* is not specified. The value returned by this procedure is one of the following:

- 0 indicates the operation completed successfully.
- *MCAST_MemoryError* indicates memory could not be allocated for the handle.
- *MCAST_NoGroup* indicates no group with this name could be found.

**JOINING AND LEAVING GROUPS**

A handle for the group must be obtained before joining that group. Once the handle has been obtained, procedures are provided for joining and leaving the group. A single processor may subscribe to a group multiple times, e.g., there may be multiple entities within the processor wanting to subscribe to the group. As will be described later, a message handler is associated with each subscription.

```c
long MCAST_Subscribe (MCAST_Handle Handle, long *MsgHandler(),
                      MCAST_Addr *Context, long *ID)
```
**MCAST_Subscribe** subscribes the caller to the group referred to by **Handle**. **MsgHandler()** is a procedure that is called each time a message arrives for that group. **Context** is a pointer to application defined information pertaining to this subscription; the value specified here is passed to **MsgHandler()** each time it is called. **ID** returns an identifier for this subscription that is needed to later unsubscribe to the group. The group's **WhereProc()** procedure (discussed below) must be set by the processor before subscribing to the group. The integer returned by **MCAST_Subscribe()** is one of the following values:

- 0 indicates the operation completed successfully
- **MCAST_CommError** indicates communications with the Owner was required to subscribe to the group, but this communication failed.
- **MCAST_MemoryError** indicates memory could not be allocated to accommodate the new subscription.
- **MCAST_GroupFull** indicates the subscription failed because the maximum number of subscriptions to the group has already been reached.

```c
long MCAST_UnSubscribe (MCAST_Handle *Handle, long ID)
```

**MCAST_UnSubscribe()** drops a subscription to the group specified by **Handle**. **ID** indicates the subscription number, returned by **MCAST_Subscribe()**, that is being cancelled. The value returned by **MCAST_UnSubscribe** is one of the following:

- 0 indicates the operation completed successfully
- **MCAST_CommError** indicates communications with the owner was required to cancel the subscription, but this communication failed.

**SENDING AND RECEIVING MESSAGES**

A sequence of calls to application-defined procedures is made when a message arrives that has been sent to some group:

1) an application defined *where-procedure* is called to specify where the incoming message is to be stored; this is invoked *only once* for each incoming message, even if the processor holds multiple subscriptions to the group. Each processor must specify exactly one where-procedure for each group to which it has subscribed.

2) an application defined *message handler* procedure is called for each subscriber when a message arrives. A single message arrival may generate several calls to message handlers, one for each subscriber that resides on the processor receiving the message.

3) an optional application defined *end procedure* is called after all invocations of the message handler for the incoming message have been made.

The where-procedure enables all buffer management functions to be realized outside of MCAST, allowing the application to tailor buffer management to its own needs. It also avoids an extra
copy in loading the incoming message into an MCAST buffer, and then transferring it to where the application really wants it loaded. Rather, MCAST will load the incoming message directly into where the application specifies it should be loaded via the where-procedure. For example, a where-procedure might allocate one of the application's internal memory buffers to hold the incoming message, and return a pointer to this buffer. Alternatively, it may specify incoming messages should always be directly loaded into some fixed memory locations, e.g., in the case the application is only interested in the most recent message to arrive on the group rather than being bothered by every message that is received.

MCAST only creates one copy of each incoming message that is shared among all of the subscribers to the group residing on the processor. Thus it is important that each subscriber not modify the copy of the message passed to it, unless it is certain it is the only subscriber. Again, this is done to limit memory copying to only those situations where it is necessary. If a subscriber needs a private copy of an incoming message because it wishes to modify it, its message handler must generate this copy.

Thus, when a message is received, MCAST will look up the group for the message, and call the where-procedure once to determine where the message should be written. The message is then written to that location, and the message handler is called, once for each subscriber to the group that resides on this processor. Finally, the end procedure is called once.

```c
void MCAST_Send (MCAST_Handle Handle, MCAST_Addr *Msg, long MsgSize, long MsgType, long *Count)
```

This procedure sends a message pointed to by `Msg` that is of `MsgSize` bytes to the group specified by `Handle`. The message is treated as an array of bytes, and is sent, uninterpreted, to the destination. Data marshaling must be performed by the sender/receiver. `MsgType` is an application defined value specifying the type of message being sent, and is passed uninterpreted to subscribers to the group. This procedure returns the number of physical processors to which the message is sent in `Count`. If `Count` is NULL, no value will be returned.

One physical copy of the message will be sent to each processor subscribed to the group, even if a processor has subscribed to the group multiple times (e.g., for multiple entities modeled in the processor).

```c
long MCAST_SetWhereProc(MCAST_Handle Handle, MCAST_Addr *WhereProc(), MCAST_Addr *WContext)
```

This procedure sets `WhereProc()` as the procedure that is called whenever this processor receives a message for the group specified by `Handle`. `WhereProc()` is called exactly once for each incoming message (even if the processor has subscribed to the group multiple times), and must return a value indicating where the incoming message should be stored. There is only one `WhereProc()` procedure defined per group per processor. `WContext` is a pointer to context information that is passed to `WhereProc()` on each invocation.

```c
long MCAST_SetEndProc (MCAST_Handle Handle, MCAST_Addr *EndProc(), MCAST_Addr *EContext)
```

This procedure sets `EndProc()` as the procedure that is called whenever this processor receives a message for the group specified by `Handle`. There is only one `EndProc()`
procedure defined per group per processor. **EContext** is a pointer to context information that is passed to **WhereProc()** on each invocation.

The application must define the following procedures:

```c
char *Appl_WhereProc (long MsgSize, MCAST_Addr *WContext, long MsgType)
```

This procedure is called when a message arrives, and must indicate where the incoming message is to be stored. It returns a pointer to the location of the message. **MsgSize** indicates the size of the incoming message in bytes. **WContext** is a pointer to application information concerning the group (defined by the caller). This information can be used by the **WhereProc()** to decide where to write the incoming message. **MsgType** is also an application-defined parameter, specified by the sender of the message. This procedure returns either a pointer to the location where the message should be written, or zero to indicate the application has nowhere to put the message. In this case, the message is discarded. This procedure is called only once for each incoming message, even if there are multiple subscribers to the group residing on this processor.

```c
void Appl_MsgHandler (MCAST_Addr *Message, long MsgSize, MCAST_Addr *Context, long MsgType)
```

This is the message handler procedure called for each subscriber when a message arrives for a group. **Message** is a pointer to the message (the value returned by the **WhereProc()** procedure called when the message arrived), **MsgSize** is the size of the message in bytes, and **Context** is a pointer to context information concerning the group to which the message was sent. **MsgType** indicates the type of message that was sent, as specified by the sender of the message.

```c
void Appl_EndProc (MCAST_Addr *Message, long MsgSize, MCAST_Addr *EContext, long MsgType)
```

This procedure is called after all of the message handlers for the incoming message have been called. The parameters are identical to those passed in the message handler procedure.

There are certain limitations on what operations can be performed within a handler. In the current implementation, recursive calls to FM’s “tick” procedure (called **FM_extract**) are not allowed. This means a handler cannot invoke primitives that require RTI-Kit to wait until a new message is received. For instance, **MCAST_GetHandle** may require MCAST to wait for a “reply message” to be sent from another processor. This will cause MCAST to lock up. In the current implementation, the following MCAST primitives cannot be called from within a handler: **RTIKIT_Tick**, **MCAST_Create**, **MCAST_GetHandle**, **MCAST_Subscribe**, **MCAST_UnSubscribe**, and **MCAST_BARRIER**. Sending a message within a handler is allowed.

**Barrier Synchronization**

The following barrier synchronization primitive is provided:

```c
void MCAST_BARRIER(void)
```
The calling processor is blocked until all processors have executed `MCAST_Barrier()`. When all have done so, `MCAST_Barrier()` returns, allowing the caller to continue execution.

**DDM-Kit**

**RATIONALE**

The goal of data distribution management (DDM) services is to efficiently manage the routing of data between federates to conserve network bandwidth and to reduce the number of irrelevant messages sent to federates that must be discarded at the destination. To achieve this, federates must describe the information they are interested in receiving as well as the information they produce. DDM-Kit then uses this information to establish communication channels from producers of information to consumers interested in receiving it.

In general, *interest expressions* must be specified by a federate to describe information that federate is interested in receiving. *Description expressions* are used to describe information produced by a federate. Both description and interest expressions are chosen and composed from a common set of values known as the *name space*. Based on the approach used in the HLA, DDM-Kit description and interest expressions are specified as a *(region, class attributes)* pair.

Regions are composed of extents, which in turn are rectangular portions of an N-dimensional routing space. In that way, any geometric shape can be approximated with a collection of extents. For example, a two-dimensional routing space might represent the play box in a virtual environment. An update region can be associated with each update message generated by a federate. Federates express interests via subscription regions. If the update region associated with a message overlaps with a federate's subscription region, the message is routed to that subscribing federate. For example, in Figure 4 there are three regions U, S₁ and S₂, each composed of only one (rectangular) extent. Updates using update region U are routed to federates subscribing to region S₁ but not to federates subscribing to region S₂.

![Figure 4 Two-dimensional routing space with subscription regions S₁ and S₂ and update region U](image)

In addition to a region, each expression also includes attributes from a class hierarchy.
DDM-Kit must map the name space, description and interest expressions to the communication services provided by the underlying network. Here we assume multicast services (implemented in the MCAST library) are used to realize communications among federates. MCAST provides standard group communication services (join, leave, and send messages to groups). Thus, the central problem addressed by DDM-Kit software is mapping description and interest expressions represented as (region, class attributes) pair to groups. Interest expressions must be mapped to groups to which the federate must join. Description expressions associated with a message are mapped to one or more groups to which the message must be sent. For any particular description expression, DDM-Kit can determine a set of multicast groups.

Creating and managing multicast groups is done outside DDM-Kit. However, DDM-Kit specifies how many groups are needed during DDM_init, and uses DDM_modify_groups callback to inform an RTI process when to join or leave groups. Multicast groups are referred to uniquely across all RTI processes by an integer value taken from \([0, \text{total groups required} - 1]\) range.

Finally, users of DDM-Kit have to map their attributes to integers by invoking DDM_get_attribute_handle. This integer representation is used to pass attributes back to users when DDM_filter_and_promote is invoked. In addition, special integer value, not assigned to any of the attributes is passed in DDM_init. It is used for DDM_filter_and_promote to designate an attribute as being filtered out.

### IMPLEMENTATION

### INITIALIZATION

Each RTI process must call the following procedures when it begins to execute in order to ensure proper initialization of the library. These procedures must be called before RIKIT_Init() is called.

```c
void DDM_UsingDDM( void)
```

This procedure sets a flag in RTI-Kit to indicate the DDM-Kit library will be used.

```c
void DDM_init( long p_fed_id, long p_n_feds, long p_n_dimensions,
               const char *p_FedFileName, long
               p__DDM_filtered_out_attr_index_value,
               DDM_modify_groups_proc p_DDM_modify_groups, long
               *r_DDM_n_groups, long *r_DDM_size_tag)
```

This procedure sets parameters to be used by DDM-Kit. RTI process's ID and the number of RTI processes are specified in the `p_fed_id` and `p_n_feds`, respectively. `p_n_dimensions` is the number of dimensions in the routing space. The name of the file containing class – attribute hierarchy information is given in `p_FedFileName`. Users of the DDM-Kit library must use the same file in order to ensure classes and attributes are referred to by the same names. `p__DDM_filtered_out_attr_index_value` is a special value for attribute that has to be filtered out. This value should not be assigned to any attribute. DDM_filter_and_promote service stores this value for appropriate attributes in returned parameter of that service, `r_attrs_index` array, as will be explained later. `p_DDM_modify_groups` is a callback procedure defined by the user which is invoked by DDM-Kit to notify the user of group membership change. The number of multicast groups required is returned in `r_DDM_n_groups`, and DDM-Kit refers to a
multicast group by an integer taken from $[0, r_{DDM\_n\_groups} - 1]$ range. Groups are managed outside DDM-Kit according to DDM_modify_groups callback, which specifies groups an RTI process should join and leave. Finally, DDM-Kit must append information to each message. The size of that information is returned in $r_{DDM\_size\_tag}$.

DATA TYPES AND SUPPORT SERVICES

The following types and support services are provided by DDM-Kit:

- **DDM_expression_handle**: a handle that serves as a pointer to a description or interest expression
- **attribute_handle**: a handle for an attribute
- **class_handle**: a handle for a class

The next two services are used to obtain attribute and class handles.

```c
int DDM_get_class_handle( char *p_class_name, class_handle *r_class)
```

**DDM_get_class_handle** returns a class handle to be used in subsequent DDM-Kit service calls. **p_class_name** is a hierarchical name of the class which uniquely identifies it. **r_class** is the returned handle. The value returned by this function can be one of the following:

- **DDM_success** indicates the operation completed successfully.
- **DDM_no_class_error** indicates no class with this name exists.

```c
int DDM_get_attribute_handle( char *p_attr_name, class_handle p_class, long p_attr_index, attribute_handle *r_attr)
```

**DDM_get_attribute_handle** returns an attribute handle to be used in subsequent DDM-Kit service calls. **p_class** is the class handle and **p_attr_name** is an attribute name. **p_attr_index** is an integer value assigned to attribute by the user. The user must map each attribute to a unique integer value. This integer representation is used to pass attributes back to users when DDM_filter_and_promote is invoked. **r_attr** is the returned handle. The value returned by this function will be one of the following:

- **DDM_success** indicates the operation completed successfully.
- **DDM_no_class_error** indicates no class with this handle exists.
- **DDM_no_attribute_error** indicates no attribute with this name and class exists.

Finally, each region is represented as follows. A region is a sequence of extents. An extent is a sequence of ranges, one for each dimension. A range is a half closed interval $[lower\_bound, upper\_bound)$. Dimensions are numbered from 0 to the N-1 in an N-dimensional routing space. Ranges are specified in order of increasing dimensions to define an extent. An extent can be shared among multiple regions.
The following data structures are used by DDM-Kit users to define regions. As it can be seen, region, referred to as `region_handle`, is an array of extents `p_extents_value` with `p_n_extents` number of elements. Extent, in turn, referred to as `extent` is an array of ranges, one for each dimension, whereas `range` is a half closed interval [lower_bound, upper_bound).

```c
typedef struct range_Struct range;
define struct range_Struct *extent;
struct range_Struct {
  long lower_bound; /* lower bound for a range */
  long upper_bound; /* upper bound for a range */
};
typedef struct region_handle_Struct region_handle;
struct region_handle_Struct {
  extent *p_extents_value; /* sequence of extents */
  int p_n_extents; /* number of extents for this region */
};
```

CREATING DESCRIPTION AND INTEREST EXPRESSIONS

This section describes the DDM-Kit services that are used to create arbitrary description and interest expressions. In addition, users can obtain handles for predefined default interest and description expressions. A default expression is one whose region includes the entire routing space, i.e. a region whose extents are the maximum bounds of the dimensions of the routing space. It is used to implement declaration management services, which are a special case of data distribution management services.

```c
int DDM_create_description_expression( region_handle p_region,
  attribute_handle *p_attrs, int p_n_ads,
  DDM_expression_handle *r_exp)
```

`DDM_create_description_expression` creates a new description expression and returns an expression handle at the location pointed to by `r_exp`. An expression’s initial value, that is the (region, class attributes) pair is given by the following parameters. `p_region` is a region handle. `p_attrs` is an array of attribute handles with `p_n_attrs` representing the array’s size. The value returned by this procedure is one of the following values:

- `DDM_success` indicates the operation completed successfully.
- `DDM_memory_error` indicates memory could not be allocated for the handle.

```c
int DDM_get_default_description_expression( attribute_handle
  *p_attrs, int p_n_ads, DDM_expression_handle
  *r_exp)
```

`DDM_get_default_description_expression`, when first invoked, creates a default description expression and returns an expression handle at the location pointed to by `r_exp`. Attributes parameter has the same meaning as that in the `DDM_create_description_expression`. Subsequent invocations of this service return the same expression handle. The return parameter may have one of the following values:
• **DDM_success** indicates the operation completed successfully.

• **DDM_memory_error** indicates memory could not be allocated for the handle.

```c
int DDM_create_interest_expression( region_handle p_region,
        attribute_handle *p_attrs, int p_n_attrs,
        DDM_expression_handle *r_exp)
```

**DDM_create_interest_expression** creates a new interest expression and returns an expression handle at the location pointed to by `r_exp`. All parameters and the return value have the same meaning as in **DDM_create_description_expression**.

```c
int DDM_get_default_interest_expression( attribute_handle *p_attrs, int p_n_attrs, DDM_expression_handle *r_exp)
```

**DDM_get_default_interest_expression**, when first invoked, creates a default interest expression and returns an expression handle at the location pointed to by `r_exp`. Attributes parameter has the same meaning as that in the **DDM_create_description_expression**. Subsequent invocations of this service return the same expression handle. The return parameter may have one of the following values:

• **DDM_success** indicates the operation completed successfully.

• **DDM_memory_error** indicates memory could not be allocated for the handle.

```c
int DDM_delete_expression( DDM_expression_handle p_exp)
```

This procedure reclaims memory for a description or interest expression handle `p_exp`. The returned parameter may have one of the following values:

• **DDM_success** indicates the operation completed successfully.

• **DDM_no_expression_error** indicates no interest or description expression with this handle could be found.

**REGISTERING AND MODIFYING DESCRIPTION AND INTEREST EXPRESSIONS**

At any point during an execution there is a set of description and interest expressions, each of which may either be registered (i.e. active) or not. Registered expressions are used to determine data distribution connectivity. Two procedures are used to tag an expression as being active or not. These are registering and unregistering an expression. They can only be invoked after obtaining a handle for a description or interest expression.

Besides registering expressions, it is also possible to modify description and interest expressions. For this purpose we define two procedures, **DDM_modify_expression** and **DDM_modify_region**. The former procedure allows changing an expression's value by
atomically changing all arguments of an expression’s (region, class attribute) pair. The latter
procedure is used to change a region’s argument only.

```c
int DDM_register_expression( DDM_expression_handle p_exp)
```

This procedure registers an expression with the handle `p_exp`. The returned parameter
may have one of the following values:

- **DDM_success** indicates the operation completed successfully.
- **DDM_no_expression_error** indicates no interest or description expression with
  this handle could be found.

```c
int DDM_unregister_expression( DDM_expression_handle p_exp)
```

This procedure deactivates an expression with the handle `p_exp`. The return value is the
same as for `DDM_register_expression`.

```c
int DDM_modify_expression( DDM_expression_handle p_exp, region_handle p_region, attribute_handle *p_attrs,
                          int p_n_attrs)
```

Expression `p_exp` is modified by modifying all parts of an expression’s value, that is, its
region `p_region`, attributes `p_attrs` with `p_n_attrs`. The return value is the same as
for `DDM_register_expression`.

```c
int DDM_modify_region( DDM_expression_handle p_exp, region_handle p_region)
```

Modifying the region part of an expression’s value, that is, its region `p_region`, modifies
expression `p_exp`. The return value is the same as for `DDM_register_expression`.

**MODIFYING GROUPS AND DETERMINING SET OF MULTICAST GROUPS FOR A DESCRIPTION EXPRESSION**

DDM-Kit’s `DDM_modify_groups` callback specifies how to manage multicast groups. It is the
user’s responsibility to call the appropriate MCAST operations in order to actually modify group
membership.

```c
void DDM_modify_groups( long *p_join_groups, long
                        p_n_join_groups, long *p_leave_groups, long
                        p_nleave_groups)
```

`DDM_modify_groups` passes group IDs for groups that have to be joined by an RTI
process in `p_join_groups` array. Size of this array is given by `p_n_join_groups`. Similarly,
`p_leave_groups` contains group IDs for groups that RTI process has to
leave. Size of this array is given by `p_n_leave_groups`.

43
Finally, it is possible to obtain a list of multicast groups that specify how to route data associated with description expression. This is done via get \texttt{DDM\_get\_groups} service. In addition, description expression attributes have to be checked separately with \texttt{DDM\_filter\_and\_promote} service upon message arrival on a multicast group handled by DDM-Kit. Any particular attribute may have to be discarded or promoted in class hierarchy.

To demonstrate how the attributes are filtered and promoted, consider an inherited attribute \(A_{Cc}\) of class \(C_c\) and corresponding attribute \(A_{Cp}\) in \(C_c\)'s parent class \(C_p\). Let’s assume that RTI process \(F_1\) defines a description expression with attributes \(A_{Cc}\) and \(B_{Cc}\), while RTI process \(F_2\) defines an interest expression with only attribute \(A_{Cp}\). Furthermore, let the regions of such description and interest expressions overlap. Each message associated with that description expression may or may not contain attributes \(A_{Cc}\) and \(B_{Cc}\). If a message contains attribute \(A_{Cc}\), it needs to be promoted to \(A_{Cp}\) when it is received by RTI process \(F_2\). On the other hand, if a message contains attribute \(B_{Cc}\), it needs to be filtered out at RTI process \(F_2\).

```c
int DDM_get_groups( DDM_expression_handle p_exp, long *r_groups,
                    long *r_n_groups, char *r_DDM_tag)
```

\texttt{DDM\_get\_groups} returns a set of multicast group IDs for a description expression \(p\_exp\). After the service completes, \(r\_groups\) will contain a pointer to an array of multicast group IDs, while \(r\_n\_groups\) will point to the number of groups found. Besides, DDM returns a tag \(r\_DDM\_tag\) in order to be able to filter and promote attributes with the \texttt{DDM\_filter\_and\_promote} service. \(r\_DDM\_tag\) is a pointer to the buffer where tag is written to. The size of tag information is fixed and is determined during the initialization as \(r\_DDM\_size\_tag\). The returned parameter may have one of the following values:

- \texttt{DDM\_success} indicates the operation completed successfully.
- \texttt{DDM\_no\_expression\_error} indicates no description expression with this handle could be found.

```c
int DDM_filter_and_promote( attribute_handle *p_attr, int p_n_attr,
                            char *p_DDM_tag, long *r_attr_index)
```

This service is used for filtering and promoting attributes in a class hierarchy when a message arrives. Attributes are passed in an array \(p\_attrs\), where the number of attributes is given by \(p\_n\_attr\). \(p\_DDM\_tag\) is a DDM-Kit’s tag obtained when \texttt{DDM\_get\_groups} is invoked. Filtered and promoted attributes are returned in a \(r\_attr\_index\) array, with elements corresponding to \(p\_attrs\) array. \(r\_attr\_index\) contains integer attribute representation that was determined with \texttt{DDM\_get\_attribute\_handle} service, or a special value \(p\_DDM\_filtered\_out\_attr\_index\_value\) obtained during \texttt{DDM\_init}. If an attribute in \(p\_attrs\) has to be filtered out, special value \(r\_DDM\_filtered\_out\) returned at the initialization is written into the corresponding \(r\_attr\) element. The returned parameter may have one of the following values:

- \texttt{DDM\_success} indicates the operation completed successfully.
- \texttt{DDM\_invalid\_attribute\_error} indicates an attempt to filter and promote an attribute, which is not part of a description expression corresponding to given tag.
• **DDM_tag_corrupted_error** indicates tag is either corrupted or wrong tag is being used.

### RM Kit

#### RATIONALE

The reduction manager (RM) is used to perform an asynchronous distributed reduction of all values \( v_i \) at processor \( i \):

\[
M = v_1 \circledast v_2 \circledast v_3 \circledast v_4 \circledast ... \circledast v_n
\]

in such a way that every processor obtains the reduced value \( M \) at the end of the reduction operation. Here, \( \circledast \) is any operator that is both commutative and associative. The RM guarantees that the \( \circledast \) operator is applied to every \( v_i \) exactly once in the course of the distributed reduction operation. In other words, every value is reduced once, and no value is reduced more than once. However, the reduction is free to permute the order of \( v_i \). In other words, the order of operands for the operators given in the original expression may not be preserved during execution of the distributed reduction, and in fact, the commutative and associative properties of the operator may be exploited for runtime efficiency.

The RM interface is designed such that multiple distinct reduction sessions can be activated and executed concurrently. These reductions execute asynchronously and independent of each other. Each session is called a “user” of the RM services. A unique handle is associated with each user, which uniquely identifies the user. The user is required to pass this handle to subsequent calls to the RM service routines.

The interface is also designed to support asynchronous execution of each session itself.

The RM assumes reliable network delivery, but does not require FIFO message ordering; it is designed to handle arbitrary network delivery ordering of its messages. Further, it is unaware of the actual network transport itself, since it uses user callback functions to send or receive its messages. The user is free to define its own communication primitives for sending and receiving messages.

#### IMPLEMENTATION

The RM uses START messages to jump start reduction communication among processors, and VALUE messages to exchange partial and completely reduced values during the reduction computation. It is designed to handle arbitrary reordering by the network of all its messages.

#### TYPE DEFINITIONS

Any user-defined type can be specified as the reduction value type. To aid in this, a forward declaration to a type called `RVALUE_TYPE_STRUCT` is made, and all references to that type are made in terms of pointers to that type. Since a pointer to a forward-declared type does not require the type to be completely defined (i.e. doesn’t require the type definition to be visible) as long as the pointer is not de-referenced, this approach is ideal for de-linking the reduction manager software from the application software that uses the reduction services.
The reduction value type is defined as a Meta class called RVALUE_CLASS in a way that roughly mimics the Meta class information of a C++ class (see Figure 5):

- The allocator RVALUE_NEW_FUNC rv_new. This function should return a pointer to a newly allocated space for holding an RVALUE_TYPE. It is used by the RM to create new instances of the reduced value type for its operation.

- The de-allocator RVALUE_DELETE_FUNC rv_delete. This function should free the space previously allocated by rv_new. It is used by the RM to de-allocate any instances it creates for its operation.

- The constructor RVALUE_INIT_FUNC rv_init. This function is used by the RM to initialize or re-initialize reduction values during its operation.

- An assignment operator RVALUE_ASSIGN_FUNC rv_assign. This function is used to assign the value of one RVALUE_TYPE instance to another.

- The reduction operator RVALUE_REDUCE_FUNC rv_reduce. This function takes two arguments that represent operands to the reduction operator. It should reduce the two operands and overwrite the first operand with the reduced value. The first operand is thus a value-return parameter, while the second parameter is read-only and hence a value parameter.

- A print operator RVALUE_PRINT_FUNC rv_print. The RM uses this function to print an instance of RVALUE_TYPE.

The application that uses RM should provide its own definition of RVALUE_TYPE_STRUCT. It must also define the methods for that type (the preceding set of described functions), specify pointers to those functions in the meta class information RVALUE_CLASS, and pass a pointer to that class in the call to rm_register().

```
struct RVALUE_TYPE_STRUCT; /*Forward declaration*/
typedef struct RVALUE_TYPE_STRUCT RVALUE_TYPE;
typedef void (*RVALUE_NEW_FUNC)(void);
typedef void (*RVALUE_DELETE_FUNC)(RVALUE_TYPE *a);
typedef void (*RVALUE_INIT_FUNC)(RVALUE_TYPE *a);
typedef void (*RVALUE_ASSIGN_FUNC)(RVALUE_TYPE *a, RVALUE_TYPE *b);
typedef void (*RVALUE_REDUCE_FUNC)(RVALUE_TYPE *a, RVALUE_TYPE *b);
typedef void (*RVALUE_PRINT_FUNC)(FILE *fp, RVALUE_TYPE *a);

typedef struct
{
    RVALUE_NEW_FUNC rv_new;
    RVALUE_DELETE_FUNC rv_delete;
    RVALUE_INIT_FUNC rv_init;
    RVALUE_ASSIGN_FUNC rv_assign;
    RVALUE_REDUCE_FUNC rv_reduce;
    RVALUE_PRINT_FUNC rv_print;
} RVALUE_CLASS;
```

Figure 5 The RVALUE_CLASS meta class (abstract type) definition for reduction value types

The rm_register() method returns a user handle whose type definition is:

```
typedef void *RMUserHandle;
```
The RM executes in different states, and transitions from one state to the next based on the inputs it receives. The different states of the RM are enumerated as follows:

```c
typedef enum
{
    RM_UNDEFINED,
    RM_REGISTERED,
    RM_ACTIVE,
    RM_PROCESSING,
    RM_DONE
} RMStatus;
```

Since the RM doesn’t perform network communication itself, it relies on the caller to send and receive its reduction messages. The RM does this by receiving callback functions for the same from the RM user. The types of the two callback functions are shown below. The `rm_resume()` function expects callbacks of these types.

```c
typedef void (*RM_SEND_START_MSG)( RMUserHandle usr, void *closure, int from_pe, int to_pe );
typedef void (*RM_SEND_VALUE_MSG)(RMUserHandle usr, void *closure, int from_pe, int to_pe, RVALUE_TYPE*v);
```

**INTERFACE FUNCTIONS**

The RM service methods are as shown below.

```c
RMUserHandle rm_register( int N, int myid, int maxn, RVALUE_CLASS *rv_class );
void rm_init( RMUserHandle usr );
void rm_receive_start( RMUserHandle usr, int from_pe );
void rm_receive_value( RMUserHandle usr, int from_pe, RVALUE_TYPE *recd_value);
int rm_resume( RMUserHandle usr, RVALUE_TYPE *result_so_far, RM_SEND_START_MSG send_start_function, RM_SEND_VALUE_MSG send_value_function, void *closure );
RMStatus rm_get_status( RMUserHandle usr, RVALUE_TYPE *rvalue );
```

The constraints on the order in which they can be invoked are shown in Figure 6. The dependencies are on a per-user basis -- there is no dependency across users.
The state of each user logically starts in the UNDEFINED state. The user must be registered with RM using `rm_register()`, after which the state transitions to REGISTERED. The user must then be initialized using `rm_init()`. The state then transitions to ACTIVE. In the ACTIVE state, the user can inform the RM of reduction messages sent by the RM on other processors, using `rm_receive_start()` for START messages, and `rm_receive_value()` for VALUE messages. While in ACTIVE state, the user can ask the RM to try to advance the reduction state to see if additional progress can be made in the reduction in light of newly received messages. This is done by invoking `rm_resume()`. During `rm_resume()`, the state is changed to PROCESSING and additional START and/or VALUE messages are potentially generated. If the reduction completes, the state transitions to DONE. Otherwise, the reduction is blocked waiting for further messages to be received from other processors. In that case, the state is returned to ACTIVE. Once in the DONE state, the reduction user can be reused by calling `rm_init()` again, which re-initializes it and returns it to ACTIVE state.

**REGISTERING FOR RM SERVICES**

```c
RMUserHandle rm_register(int N, int myid, int maxn, RVALUE_CLASS *rv_class);
```

Every RM user invokes this to register for RM services. The number of processor participating in the reduction is specified as N, and the ID of this processor is specified as `myid` (0 <= myid < N). The maximum number of processors that will ever participate in this reduction is given as maxn. The meta class information about the reduction value type RVALUE_TYPE is specified in `rv_class`. 
A handle to that user is returned, which should be used in all subsequent calls to the RM services for that user. Several users can use the RM concurrently; each user gets a unique handle, and users are independent of each other.

**Initialization**

```c
void rm_init( RMUserHandle usr );
```

The RM user invokes this after registering for the first time, or to reuse this previously registered handle after a reduction is completed for that user.

**Receiving Start Message**

```c
void rm_receive_start( RMUserHandle usr, int from_pe );
```

The RM is being told that this processor has received a START message from the RM on another processor whose ID is given as `from_pe`.

**Receiving Values**

```c
void rm_receive_value( RMUserHandle usr, int from_pe, RVALUE_TYPE *recd_value );
```

The RM is being told that this processor has received a VALUE message from the RM on another processor whose ID is given as `from_pe`. The received (partial or completed) reduction value is given as `recd_value`.

**Resuming a Reduction**

```c
int rm_resume( RMUserHandle usr, RVALUE_TYPE *result_so_far,
               RM_SEND_START_MSG send_start_function,
               RM_SEND_VALUE_MSG send_value_function, void *closure );
```

The RM is told to start or continue processing the current reduction. This returns 1 if reduction is completed, 0 if more processing is left. The partial result computed so far is returned in the parameter. If the reduction is completed, the returned result is the final value.

During processing, RM can send start/value messages to other processors. The corresponding helper functions that are passed as parameters will be used by RM to perform the message sends. RM passes the closure as argument to the helper functions.

**Getting Status of the Reduction**

```c
RMStatus rm_get_status( RMUserHandle usr, RVALUE_TYPE *rvalue );
```
The user invokes this to get the current status of the given reduction. This method additionally returns the current (partial) reduced value. The reduced value is the final value if and only if the status is RM_DONE.

INTERNAL TYPE DEFINITIONS

Internally, the RM chooses from a set of possible inter-processor communication patterns for computing the reduction. The patterns supported include the following, as enumerated in Figure 7:

- **STAR**: In this, a leader processor acts as a central node which receives the values from all the processors, reduces them together, and sends back the reduced value to all the processors. The best case completion time is 2 round trip time (RTT) units, and the worst case completion time is N RTT, and the best and worst case message fan-in and fan-out is \(o(N)\). This is suitable for lower number of processors, or if communication to the central processor is faster than that across any other pair of processors.

- **ALL_TO_ALL**: In this, every processor sends its value to every other processor. Every processor then reduces all the received values. The best case completion time is 0.5 RTT, the worst case completion time is N RTT, and the best and worst case message fan-in and fan-out is \(o(N)\).

- **BUTTERFLY**: In this, a butterfly communication pattern is used. The best case completion time is \(\log N \text{ RTT}\), while the worst case completion time is \(3\log N \text{ RTT}\). The best and worst case message fan-in and fan-out is \(o(\log N)\). This is the most scalable pattern among those patterns supported in the RM.

The default schedule used in the reduction is BUTTERFLY. Setting the environment variable RMSCHEDULE to A2A, STAR or BFLY can change the schedule to all-to-all, star or butterfly patterns respectively.

```c
typedef enum
{
    RM_SCHEDULE_UNDEFINED,
    RM_SCHEDULE_STAR,
    RM_SCHEDULE_ALL_TO_ALL,
    RM_SCHEDULE_BUTTERFLY
} RMScheduleType;
```

**Figure 7** RMScheduleType defining the communication patterns supported by the reduction manager

INTERNAL FUNCTIONS

The RM uses an internal function `compute_schedule()` as shown below to compute various parameters for the chosen communication pattern.

```c
int compute_schedule
```
TM-Kit

RATIONALE

The services in TM-Kit provide the basic primitives necessary for time management in distributed simulations. At the heart of TM-Kit are algorithms for computing LBTS information, i.e., a lower bound on the time stamp of future messages that can be received in the future by each processor. The other procedures, for the most part, are there to provide enough information to compute LBTS, and do not actually perform any real services. For example, the "communication primitives" TM_Out() and TM_In() do not actually send and receive messages. They only inform the TM software that messages have been sent/received.

TM-Kit users must do the following to an existing RTI to use these services:

1) Make calls to initiate LBTS computations and define handlers that are called when a LBTS computation initiated by another processor has been detected, or when an LBTS computation has completed.

2) Add calls to TM-Kit on message sends and receives so that TM-Kit can properly account for transient messages. In particular, the current version of TM-Kit requires the caller to specify the number of destinations receiving each message. “Destinations” is defined below. This is a non-trivial issue if multicast communications are used. MCAST provides this information on each message send.

3) The type for time values must be defined, as well as functions operating on time values. These definitions are compiled into TM-Kit when the library is built.

4) TM-Kit requires that certain information be piggybacked onto time stamped messages. This information must be added each time a message is sent, and extracted and passed to TM-Kit when messages are received.

In the current specification, TM-Kit does not directly handle time stamped messages. Thus, message queueing and determining when it is “safe” to deliver time stamped messages is performed outside of TM-Kit. TM-Kit should essentially be viewed as a calculator that computes LBTS values.

The underlying computation model for the simulation is viewed as a collection of processors that communicate through message channels. A parallel computer could be designated as a single processor if it provides its own internal time management mechanism (e.g., a Time Warp implementation), or each processor within the parallel computer can be viewed as a separate processor, in which case TM-Kit provides time management among processors in the parallel machine. Channels are used only for the purpose of determining which processors communicate.
with which others. Each channel can have multiple senders and multiple receivers. A point-to-
point connection is a special case of a channel.

IMPLEMENTATION

INITIALIZATION

Each processor must call the following procedures when it begins executing in order to initialize
the library. These procedures must be called before RTIKIT_Init() is called.

```c
void TM_UsingTM (void)
```
Sets a flag in RTI-Kit to indicate the TM-Kit library will be used. It must be called before
calling RTIKIT_Init().

```c
void TM_Parameter (parameter, value)
```
This procedure sets a parameter to be used by TM-Kit. It must be called before calling
RTIKIT_Init(). Parameter values are defined below. The name indicates what
should be specified in the parameter field above, and the type specified in parentheses
indicates the type of the parameter itself:

TIME VALUES

Different applications may use different representations of the logical time type. In particular, the
time field may contain multiple fields (e.g., priorities, tie breakers, etc.). The LBTS algorithm must
be able to do comparisons of time values. Thus, the software using the TM services must specify
the format of the time type, and provide functions for performing certain operations on time
values. These functions can be implemented by macro definitions to maximize performance.

The application using TM-Kit must define the following that are compiled with TM-Kit when the
library is built, if the default time representation (type double) is not used:

```c
TM_Time
```
Type used for time values. Normally, this will be defined as a primitive data type (e.g.,
double) or a structure.

```c
TM_Time TM_Min(TM_Time A, TM_Time B)
```
This function compares two time values A and B, and returns the smaller of the two.

```c
TM_Time TM_Add(TM_Time A, TM_Time B)
```
This function adds two time values A and B, and returns their sum.

The following functions define standard logical comparison operators.
TM_LT (TM_Time A, TM_Time B)
TM_LE (TM_Time A, TM_Time B)
TM_GT (TM_Time A, TM_Time B)
TM_GE (TM_Time A, TM_Time B)
TM_EQ (TM_Time A, TM_Time B)

TM_LT returns TRUE if A < B, else it returns FALSE. Similarly, TM_LE, TM_GT, TM_GE, and TM_EQ return TRUE if A<=B, A>B, A>=B, or A==B, respectively.

This is a constant defining the identity operator for the TM_Min operation. In other words, TM_Min(TM_IDENT, X) returns X for any value of X. TM_IDENT is normally defined as the largest representable time value.

In the current implementation, these functions and constants are defined as macros in the file “timedefs.h”. TM_Time is defined as a double precision floating point number (type double). To change the time definition, a new version of this file should be created, and RTI-Kit recompiled to use the new definitions. TM_IDENT is often an architecture dependent value. If this is the case, it is recommended TM_IDENT refer to a value defined in the “arch.h” file where architecture dependent definitions are declared.

NOTIFICATION OF MESSAGE SENDS AND RECEIVES

The time management software must be notified of message sends and receives so that transient messages can be taken into account by the LBTS computation. In addition, the TM software may need to piggyback information onto messages. The following are defined for this purpose:

TM_TagType

This is the type of information piggybacked onto each time stamp ordered message. The caller need not be concerned with the definition of this type, but must provide a field of this type in each time stamp ordered message.

TM_PutTag (TM_TagType *Tag)

This procedure must be called prior to sending a time stamped message so TM-Kit can place information into the message. Tag is a pointer into the outgoing message buffer indicating where TM-Kit’s information is to be written. The size of information placed into the message is fixed, and may be determined by sizeof(TM_TagType).

TM_Out (long Count)

This procedure must be called after sending a time stamped message so TM-Kit can account for transient message. Count indicates the number of destinations that will receive the message (see definition of “destination” below).

TM_In (TM_Time TimeStamp, TM_TagType *Tag)
The caller indicates it has just received a message with time stamp `TimeStamp`. `Tag` is a pointer to the information placed in the message by TM-Kit through the `TM_PutTag` procedure when the message was sent. Each destination must call this procedure exactly once.

A “destination” may be a physical destination, e.g., a processor, or a logical destination, e.g., an entity (there could be several entity destinations per processor). Usually, it will be most efficient to define a destination as a processor receiving a message. The main criteria in defining a destination is that the total number of message sends (as specified in the `Count` parameter of `TM_Out`) matches the total number of calls to `TM_In` when all messages have been received.

When using the MCAST library, each call to `MCAST_Send` returns the number of processors that will receive the message. Thus, each processor receiving the message should call `TM_In` exactly once when the message is received. One way to accomplish this is to attach a handler to each group on which time stamp ordered messages can be received. This handler can be used to call `TM_In` once when a message arrives on that group.

**LBTS Computation**

An LBTS computation is initiated by one or more processors calling the `TM_StartLBTS` procedure, at which time a local time value must be specified. The LBTS computation will compute a global minimum among (1) the local minimum value provided by each processor, and (2) the time stamp of all transient messages in the network while the LBTS computation is being performed. TM-Kit is made aware of the latter by calls to `TM_Out` and `TM_In`. The local value provided by each processor will typically be the processor’s lookahead plus the minimum among all unprocessed or partially processed messages within that processor, i.e., lookahead plus the minimum among all messages stored in the RTI’s local queues, and the current logical time of the simulator at that processor if it is not blocked waiting for a time advance. Note that TM-Kit does not automatically add lookahead to time values in computing LBTS.

Each global LBTS computation is assigned a unique transaction number; transactions are numbered sequentially starting with zero. If two processors simultaneously initiate a new LBTS computation, only one new LBTS computation will actually be initiated and both initiators will be given the same transaction number. A new LBTS computation can be started while one or more other LBTS computations are in progress, so there can be multiple LBTS computations in progress at one time. The LBTS computations may not necessarily complete in the order in which they were initiated.

After an LBTS computation is initiated by one processor, it is propagated by TM-Kit to other processors. If an LBTS computation spreads to a processor that has not already initiated an LBTS computation via a `TM_StartLBTS` call, an application defined “LBTS-Started” handler is called to notify the processor an LBTS computation has been initiated, and to request that processor’s local minimum. The LBTS-Started handler must either:

1) **accept** the initiation of the LBTS computation and return a local minimum value to TM-Kit; this is equivalent to calling `TM_StartLBTS`, or

2) **defer** responding to the LBTS computation; in this case, the processor must eventually respond to the LBTS computation by invoking `TM_StartLBTS` or else the LBTS computation will never complete.

When the LBTS computation has completed, an “LBTS-Done” handler is called in each processor to notify that processor of the new LBTS value. Each processor will be called exactly once for each LBTS computation that completes.
At most one LBTS-Started handler can be defined at one time for a processor, however this handler can be changed at any time during the execution. A different LBTS-Done handler can be used for each LBTS computation.

Different approaches may be taken to start LBTS computations. For example, a central controller can be designated that is responsible for starting all LBTS computations. Alternatively, a distributed asynchronous approach may be taken where each processor initiates an LBTS computation when it feels it needs an updated value.

The following procedures are defined:

```c
long TM_StartLBTS(TM_Time MinTime, TM_LBTSDoneProc LBTSDone(),
                   long *TransactionID)
```

Initiate an LBTS computation. `MinTime` is the value used in computing LBTS. It is typically the processor's lookahead plus either the minimum logical time of any entity within the processor if the processor is not blocked, waiting for LBTS to advance. If the processor is blocked, `MinTime` is usually the minimum time of the next local event plus that processor's lookahead. The `LBTSDone()` parameter is a procedure that is called when the LBTS computation has completed. If this parameter is NULL, the processor is not notified when the computation has completed. `TransactionID` returns the transaction number assigned to the new LBTS computation. This procedure returns:

- **TM_Success** if the LBTS computation was successfully initiated.
- **TM_Failed** if the computation could not be initiated. This could happen if the maximum number of pending LBTS computations are already in progress.

```c
TM_SetLBTSStartProc (TM_LBTSStartedProc StartHandler)
```

This procedure sets the processor's LBTS-Started handler to `StartHandler`. An execution error occurs if no handler has been defined when TM-Kit discovers another processor has initiated a new LBTS computation.

The application using TM-Kit must define the following procedures.

```c
long StartHandler (long TransactionID, TM_Time *MinTime,
                   TM_LBTSDoneProc *DoneHandler())
```

This procedure is called when the processor detects an LBTS computation was initiated by another processor, and this processor has not already initiated its participation in the computation by calling `TM_StartLBTS`. `TransactionID` indicates the number assigned to the transaction. The procedure must return:

- **TM_ACCEPT** if it is accepting the new computation. In this case, `MinTime` returns this processor's local minimum value, and `DoneHandler` returns a pointer to the handler that should be called when the computation has completed. If `DoneHandler` is NULL, the processor is not notified when the computation is completed.
- **TM_DEFER** indicates the processor wishes to defer providing its local minimum value. The local minimum must be provided on a subsequent `TM_StartLBTS` call, or the
LBTS computation will never complete. In this case, the values returned in MinTime and DoneHandler are ignored by TM-Kit.

```c
void DoneHandler (TM_Time Result, long TransactionID)
```

This procedure is called when an LBTS computation has completed. Result is the LBTS value that was computed. TransactionID indicates the transaction number of the LBTS computation that just completed.

Typically, `TM_StartLBTS` will initiate an LBTS computation that is completed at some later time. However, it is possible the LBTS computation will be completed within the `TM_StartLBTS` procedure itself. In other words, it is possible the applications DoneHandler procedure will be called before `TM_StartLBTS` has returned. This would happen if the LBTS computation is essentially finished and is waiting for this processor to initiate, in which case only a local computation is required.

---

## RTI-Core

### Rationale

RTI Core is intended to pull together the resources of the other libraries within FDK to present a unified interface to the RTI developer. RTI Core’s main focus is to provide basic time management functionality that is useful to a wide range of RTIs. RTI Core manages starting LBTS computation, responding to LBTS computation, and the delivery of messages freeing the RTI developer from these tasks. RTI Core also provides Retraction Handle Management (RHM). RHM maintains retraction handle information on all messages sent and received and provides a mechanism for determining if a sent message may be retracted, and a mechanism to support the annihilation of messages within the RTI.

Below is a description of the Time Management and Retraction Handle Management capabilities of RTI Core. Each section will begin with a brief higher-level overview followed by a detailed description of the interface and its intended use. Each section will conclude with a detailed description of all the internal design.

### Implementation

#### Time Management

There are six functions that the RTI must register with the RTI Core. These functions should be registered before `Core_InitRTI` is called. `Core_InitRTI` will check if each function is registered and if not it will return with an error code indicating which function is not registered.

```c
typedef fdkErrorCode (*DeliverOneMessageProc)(TM_Time, char *,
long, long);
```

Delivers messages to the Federates. Messages that are expected to be delivered are requestRetraction, reflectAttributeValue, and receiveInteraction.
typedef fdkErrorCode (*DeliverROEventsProc)(void);

Delivers messages contained in the RTI's receive order queue to the federate.

typedef fdkErrorCode (*DeliverTSOEventsProc)(TM_Time);

Delivers messages contained in the RTI's time stamp order queue to the federate.

typedef void (*TimeAdvanceGrantProc)(TM_Time);

Invokes the federates time Advance Grant function. The RTI should implement a function that calls the federate's function as opposed to allowing the Core to directly call the federates function.

 typedef TM_Time (*TSOMinProc)(void);

Returns the time stamp of the message at the head of the TSO queue.

typedef char * (*TSOPopProc)(TM_Time *, long *, long *);

Pops the message at the head of the TSO queue.

fdkErrorCode Core_InitRTI(void);

Initialized the RTI Core state vector and verifies that all necessary RTI functions have been registered. Will return on of the following fdkErrorCodes:

<table>
<thead>
<tr>
<th>fdkSUCCEEDED</th>
<th>Initialization succeeded.</th>
</tr>
</thead>
<tbody>
<tr>
<td>fdkFAILED</td>
<td>Initialization Failed.</td>
</tr>
<tr>
<td>fdkOUTOFMEMORY</td>
<td>Could not allocate memory for Retraction Handle Manager.</td>
</tr>
<tr>
<td>fdkDOMNR</td>
<td>DeliverOneMessage not registered.</td>
</tr>
<tr>
<td>fdkDTSOENR</td>
<td>DeliverTSOEvents not registered.</td>
</tr>
<tr>
<td>fdkDROENR</td>
<td>DeliverROEvents not registered.</td>
</tr>
<tr>
<td>fdkTAGNR</td>
<td>TimeAdvanceGrant not registered.</td>
</tr>
<tr>
<td>fdkTSOMNR</td>
<td>TSOMin not registered.</td>
</tr>
<tr>
<td>fdkTSOPNR</td>
<td>TSOPop not registered.</td>
</tr>
</tbody>
</table>

fdkErrorCode Core_RegisterDeliverOneMessage(DeliverOneMessageProc p);

Registers DeliverOneMessage with RTICore. Will return one of the following fdkErrorCodes:

<table>
<thead>
<tr>
<th>fdkSUCCEEDED</th>
<th>Registration succeeded.</th>
</tr>
</thead>
<tbody>
<tr>
<td>fdkFAILED</td>
<td>Registration failed.</td>
</tr>
</tbody>
</table>

fdkErrorCode Core_RegisterDeliverROEvents(DeliverROEventsProc p);

Registers RegisterDeliverROEvents with RTICore. Will return one of the following fdkErrorCodes:
**fdkSUCCEEDED**  Registration succeeded.

**fdkFAILED**  Registration failed.

**fdkErrorCode Core_RegisterDeliverTSOEvents(DeliverTSOEventsProc p);**

Registers **RegisterDeliverTSOEvents** with RTICore. Will return one of the following fdkErrorCodes:

<table>
<thead>
<tr>
<th>fdkSUCCEEDED</th>
<th>Registration succeeded.</th>
</tr>
</thead>
<tbody>
<tr>
<td>fdkFAILED</td>
<td>Registration failed.</td>
</tr>
</tbody>
</table>

**fdkErrorCode Core_RegisterTimeAdvanceGrant(TimeAdvanceGrantProc p);**

 Registers **Time Advance Grant** with RTICore. Will return one of the following fdkErrorCodes:

<table>
<thead>
<tr>
<th>fdkSUCCEEDED</th>
<th>Registration succeeded.</th>
</tr>
</thead>
<tbody>
<tr>
<td>fdkFAILED</td>
<td>Registration failed.</td>
</tr>
</tbody>
</table>

**fdkErrorCode Core_RegisterTSOMin(TSOMinProc p);**

 Registers **TSOMin** with RTICore. Will return one of the following fdkErrorCodes:

<table>
<thead>
<tr>
<th>fdkSUCCEEDED</th>
<th>Registration succeeded.</th>
</tr>
</thead>
<tbody>
<tr>
<td>fdkFAILED</td>
<td>Registration failed.</td>
</tr>
</tbody>
</table>

**fdkErrorCode Core_RegisterTSOPop(TSOPopProc p);**

 Registers **TSOPop** with RTICore. Will return one of the following fdkErrorCodes:

<table>
<thead>
<tr>
<th>fdkSUCCEEDED</th>
<th>Registration succeeded.</th>
</tr>
</thead>
<tbody>
<tr>
<td>fdkFAILED</td>
<td>Registration failed.</td>
</tr>
</tbody>
</table>

**TM_Time Core_GetCurrentTime(void);**

Returns the current time of RTICore.

**TM_Time Core_GetLookAhead(void);**

Returns the lookahead value being used by RTICore.

**fdkErrorCode Core_SetLookAhead(TM_Time newLA);**

Sets the lookahead value RTICore will use. Will return one of the following fdkErrorCodes:

<table>
<thead>
<tr>
<th>fdkSUCCEEDED</th>
<th>Successfully.</th>
</tr>
</thead>
<tbody>
<tr>
<td>fdkFAILED</td>
<td>Registration failed.</td>
</tr>
</tbody>
</table>

**TM_Time Core_GetExtraLA(void);**

Returns the extra lookahead value being used by RTICore.
Sets the extra lookahead value RTICore will use. Will return one of the following fdkErrorCodes:

<table>
<thead>
<tr>
<th>fdkSUCCEEDED</th>
<th>Registration succeeded.</th>
</tr>
</thead>
<tbody>
<tr>
<td>fdkFAILED</td>
<td>Registration failed.</td>
</tr>
</tbody>
</table>

Disables the current LBTS computation and prevents other LBTS computations from starting. Will return one of the following fdkErrorCodes:

<table>
<thead>
<tr>
<th>fdkSUCCEEDED</th>
<th>Successfully disabled LBTS computations.</th>
</tr>
</thead>
<tbody>
<tr>
<td>fdkFAILED</td>
<td>Failed.</td>
</tr>
</tbody>
</table>

Enables LBTS computations. Disabled LBTS computations will be allowed to execute and new LBTS computations may be started. Will return one of the following fdkErrorCodes:

<table>
<thead>
<tr>
<th>fdkSUCCEEDED</th>
<th>Successfully enabled LBTS computations.</th>
</tr>
</thead>
<tbody>
<tr>
<td>fdkFAILED</td>
<td>Failed.</td>
</tr>
</tbody>
</table>

Implements the Next Event Request service. Will return one of the following fdkErrorCodes:

<table>
<thead>
<tr>
<th>fdkSUCCEEDED</th>
<th>NER successfully initiated.</th>
</tr>
</thead>
<tbody>
<tr>
<td>fdkTMPENDING</td>
<td>A Next Event Request, Time Advance Request, or Flush Queue Request is already pending.</td>
</tr>
<tr>
<td>fdkTIMEPASSED</td>
<td>The request time has already passed.</td>
</tr>
</tbody>
</table>

Implements the Time Advance Request service. Will return one of the following fdkErrorCodes:

<table>
<thead>
<tr>
<th>fdkSUCCEEDED</th>
<th>TAR successfully initiated.</th>
</tr>
</thead>
<tbody>
<tr>
<td>fdkTMPENDING</td>
<td>A Next Event Request, Time Advance Request, or Flush Queue Request is already pending.</td>
</tr>
<tr>
<td>fdkTIMEPASSED</td>
<td>The request time has already passed.</td>
</tr>
<tr>
<td>fdkTMESMALL</td>
<td>TM_EPSILON too small.</td>
</tr>
</tbody>
</table>

Implements the Flush Queue Request service. Will return one of the following fdkErrorCodes:

<table>
<thead>
<tr>
<th>fdkSUCCEEDED</th>
<th>FQR successfully initiated.</th>
</tr>
</thead>
<tbody>
<tr>
<td>fdkTMPENDING</td>
<td>A Next Event Request, Time Advance Request, or Flush Queue Request is already pending.</td>
</tr>
</tbody>
</table>

fdkErrorCode Core_Tick(void);
Gives time to the RTI to perform internal services and to permit delivery of messages to the federate. Messages are delivered only during tick calls. Will return one of the following fdkErrorCodes:

<table>
<thead>
<tr>
<th>fdkErrorCode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fdkSUCCEEDED</td>
<td>Tick call returned with no errors.</td>
</tr>
<tr>
<td>fdkTMINCON</td>
<td>State of RTICore in an inconstant state. Either 1) TSO min &lt; LBTS, 2) Retract Min &lt; LBTS, 3) Pending Time &lt; LBTS.</td>
</tr>
<tr>
<td>fdkLMLTLBTS</td>
<td>The last attempted LBTS computation failed because Local Min &lt; LBTS.</td>
</tr>
<tr>
<td>fdkCNSLBTS</td>
<td>The last attempted LBTS computation failed because of internal TM-Kit error.</td>
</tr>
</tbody>
</table>

fdkErrorCode Core_PrintRTIState (FILE *out);

Prints the RTI state to file descriptor out. Will return one of the following fdkErrorCodes:

<table>
<thead>
<tr>
<th>fdkErrorCode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fdkSUCCEEDED</td>
<td>Succeeded</td>
</tr>
<tr>
<td>fdkBADFD</td>
<td>Bad File Descriptor.</td>
</tr>
</tbody>
</table>

fdkErrorCode Core_InitDebug(FILE *fout);

Sets the debug output file descriptor to file descriptor fout. If fout is NULL the debug output file descriptor is set to stderr. Will return one of the following fdkErrorCodes:

<table>
<thead>
<tr>
<th>fdkErrorCode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fdkSUCCEEDED</td>
<td>Succeeded.</td>
</tr>
<tr>
<td>fdkBADFD</td>
<td>Bad File Descriptor.</td>
</tr>
</tbody>
</table>

INTERNAL DESIGN

static fdkErrorCode core_DoEventDelivery(void);

Initiates delivery of messages to the federate, and starts an LBTS computation is needed. Will return one of the following fdkErrorCodes:

<table>
<thead>
<tr>
<th>fdkErrorCode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fdkSUCCEEDED</td>
<td>Succeeded.</td>
</tr>
<tr>
<td>fdkTMINCON</td>
<td>State of RTICore in an inconstant state. Either 1) TSO min &lt; LBTS, 2) Retract Min &lt; LBTS, 3) Pending Time &lt; LBTS.</td>
</tr>
<tr>
<td>fdkLMLTLBTS</td>
<td>The last attempted LBTS computation failed because Local Min &lt; LBTS.</td>
</tr>
<tr>
<td>fdkCNSLBTS</td>
<td>The last attempted LBTS computation failed because of internal TM-Kit error.</td>
</tr>
</tbody>
</table>

static void core_FlushQueueRequestDelivery(void);

Delivers all messages in the TSO queue, the RO queue, and all retractions not yet delivered to the federate.

static int core_TryToRelease(TM_Time NERTime, TM_Time *ReleaseTime);

Delivers messages to the Federate if time has advanced. Returns 1 if time has advanced or 0 if time has not advanced.
static fdkErrorCode core_StartLBTS(TM_Time AdvanceTime);

Will return one of the following fdkErrorCodes:

<table>
<thead>
<tr>
<th>fdkErrorCode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fdkSUCCEEDED</td>
<td>Succeeded.</td>
</tr>
<tr>
<td>fdkLMLTLBTS</td>
<td>The last attempted LBTS computation failed because Local Min &lt; LBTS.</td>
</tr>
<tr>
<td>fdkCNSLBTS</td>
<td>The last attempted LBTS computation failed because of internal TM-Kit error.</td>
</tr>
</tbody>
</table>

static void core_Compute_Local_LBTS(TM_Time *MyTime);

Computes the local LBTS value and returns this value via MyTime. The value returned is reported to the global LBTS computation.

static long core_MyLBTSStarted(long Trans, TM_Time *MyTime, TM_LBTSDoneProc *MyProc);

When TM-Kit detects an LBTS computation has been started by another federate core_MyLBTSStarted is called to obtain the local minimum time of the federate.

static void core_MyLBTSDone(TM_Time LBTSValue, long Trans);

When TM-Kit completes an LBTS computation MyLBTSDone is called to notify the RTI that a new LBTS value has been computed and is LBTSValue.

**RETRACTION HANDLE MANAGEMENT (RHM)**

Retraction Handle Management provides basic functionality to implement the retraction of messages. The functionality provided include generating federation unique retraction handles, maintaining tables to store the retraction handles of messages sent and received, determining when a message should be delivered to the federate or annihilated.

There are three data structures that an RTI developer must be familiar with that are used in the RHM interface. The RHM interface can be partitioned into three categories Initialization, Sender Side RHM, and Receiver Side RHM. Initialization functions initialize the RHM module and debug code. Sender Side RHM functions are used when sending messages. Receiver Side RHM functions are used when receiving messages.

**DATA STRUCTURES**

typedef unsigned long CoreRetractionNumber;
typedef struct
{
    CoreRetractionNumber rn;
    unsigned long        fedID;
}CoreRetractionHandle;
typedef struct
{
    CoreRetractionHandle rh;
    TM_Time              ts;
    MCAST_Handle         mh;
}CoreRetractInfo;
The **CoreRetractionHandle** uniquely identifies a message in the federation. The handle consists of a **CoreRetractionNumber** that is generated by the RHM and the federate identifier. This structure contains the **MCAST_Handle** and time stamp of the message with **CoreRetractionHandle rh**.

**INITIALIZATION**

```c
fdkErrorCode Core_InitRHM(void);
```

Initializes the Retraction Handle Manager. Before any RHM function can be called **Core_InitRHM** must be called. Will return one of the following fdkErrorCodes:

<table>
<thead>
<tr>
<th>fdkErrorCode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FdkSUCCEEDED</td>
<td>Succeeded.</td>
</tr>
<tr>
<td>fdkOUTOFMEMORY</td>
<td>Could not allocate memory for RHM data structures.</td>
</tr>
</tbody>
</table>

```c
fdkErrorCode Core_InitRHMDebug(FILE *fout);
```

Print all debug statements generated by the RHM module to **fout**. If **fout** is NULL all messages will be printed to **stderr**. The RHM debug messages are turned on and turned off by the environment variable **RHMDEBUG**. If **RHMDEBUG** is defined then the debug messages will be printed to **fout**, if **RHMDEBUG** is not defined then debug messages will not be printed. Will return one of the following fdkErrorCodes:

<table>
<thead>
<tr>
<th>fdkErrorCode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FdkSUCCEEDED</td>
<td>Succeeded.</td>
</tr>
<tr>
<td>FdkBADFD</td>
<td>Bad File Descriptor.</td>
</tr>
</tbody>
</table>

**SENDER SIDE RHM**

```c
fdkErrorCode RHM_PutTag(TM_Time ts, MCAST_Handle mh,
                         CoreRetractionHandle *rid);
```

Returns the next valid **RetractionID**, contained in the **CoreRetractionHandle rid** for this federate. The **CoreRetractionHandle** pointed to by **rid** should reside on the message about to be sent this ensures that information needed by the RHM is on all messages that require this information. The Successful calls will result in a federation unique **RetractionID** that can be used to uniquely identify a message in the federation. The **RetractionID** will be stored in the Sender Retraction Table with the specified time stamp, **ts**, and MCAST Handle, **mh**. Should the message be retracted the MCAST Handle is used to identify the recipients of the message. **rid** is the new **RetractionID**. The new **RetractionID** is generated by adding **RTIKIT_numnodes** to the previous **RetractionID** returned. Will return one of the following fdkErrorCodes:

<table>
<thead>
<tr>
<th>fdkErrorCode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FdkSUCCEEDED</td>
<td>Succeeded.</td>
</tr>
<tr>
<td>FdkFAILED</td>
<td></td>
</tr>
<tr>
<td>fdkOUTOFMEMORY</td>
<td></td>
</tr>
</tbody>
</table>

```c
int RHM_IsRetractable(CoreRetractionHandle rID, CoreRetractInfo *ri);
```
Will determine if the message with the specified RetractionID can be retracted. The CoreRetractInfo pointed to by ri contains the MCAST_Handle, and time stamp of the message in question. The MCAST_Handle is needed to free memory used by the message. Will return one of the following fdkErrorCodes:

<table>
<thead>
<tr>
<th>NOTRETRACTABLE</th>
<th>Unable to retract this message</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOTRETRACTABLENE</td>
<td>RHM has no information of this message, cannot retract</td>
</tr>
<tr>
<td>RETRACTABLE</td>
<td>This message can be retracted</td>
</tr>
</tbody>
</table>

**NOTE:** Core_IsRetractable destroys the entry if found in the table. So once called with rID subsequent calls with rID will return fdkNOTRETRACTABLE. Should this behavior be changed? Maybe a destructive and nondestructive version of this call? It assumes that if you are checking if a message can be retracted then you must be retracting the message. This also prevents the case where RTICore can accidentally allow the RTI to send more than one retract for a message.

```c
fdkErrorCode Core_PrintSenderRetractionTable(FILE *out);
```

Prints out the entire contents of the Sender Retraction Table to the specified file descriptor. Will return one of the following fdkErrorCodes:

<table>
<thead>
<tr>
<th>FdkSUCCEEDED</th>
<th>Succeeded.</th>
</tr>
</thead>
<tbody>
<tr>
<td>FdkBADFD</td>
<td>Bad File Descriptor.</td>
</tr>
</tbody>
</table>

**RECEIVER SIDE RHM**

```c
fdkErrorCode RHM_In(CoreRetractionHandle rID, TM_Time ts, MCAST_Addr Msg, int MsgSize, long MsgType);
```

Saves the RetractionID, time stamp, message pointer (MCAST_Addr), message size, and message type of received retraction messages. When other user messages are about to be delivered by the RTI to the federate the RTI will call Core_IsDeliverable which will return information needed by the RTI to perform memory management. Will return one of the following fdkErrorCodes:

<table>
<thead>
<tr>
<th>FdkSUCCEEDED</th>
<th>Succeeded.</th>
</tr>
</thead>
<tbody>
<tr>
<td>FdkOUTOFMEMORY</td>
<td></td>
</tr>
</tbody>
</table>

```c
int RHM_IsDeliverable(CoreRetractionHandle rID, MCAST_Addr *Msg, int *MsgSize);
```

Determines if a retraction for the message identified by rID is stored in the Receiver Retraction Table. If a retraction for a message has been received then the message must not be delivered to the federate or the retraction has to be forwarded to the federate in the case the message has already been delivered. Msg is a pointer to the retraction message and MsgSize is the size of the retraction message. This function destroys the retraction's entry in the Receiver Retraction Table. Will return one of the following fdkErrorCodes:

<table>
<thead>
<tr>
<th>ISDELIVERABLE</th>
<th>The message should be delivered to the federate</th>
</tr>
</thead>
</table>
The message has been retracted and should not be delivered to the federate.

RHM uses three data structures to manage retraction handles, one on the sender side and two on the receiver side. On the sender side a hash table is used to store retraction handles. The retraction handle, time stamp, and the destination's MCAST_Handle are all stored in the hash table every time a TSO message is sent. When a previously sent message is to be retracted the hash table is searched to determine if the message can be legally retracted. If the time stamp of the message is less than LBTS + lookahead the message cannot be retracted.

When a retraction is received it is treated differently than other user messages. A pointer to the retraction message is managed by RHM. When a user message is about to be delivered the receiver side RHM is consulted to determine if a retraction for the message has been received. If a retraction has been received then the user message is not delivered to the federate. Otherwise, the user message is delivered to the federate. The receiver side RHM consists of two data structures. The receiver side hash table supports fast lookups given a retraction handle, and a receiver side TSO queue that is used to quickly determine the time stamp of the earliest retraction. This time stamp information is used when calculating the local minimum time stamp.

```c
int Core_RetractionHandleCompare(const CoreRetractionHandle *h1, const CoreRetractionHandle *h2)

If h1 is equivalent to h2 EQUAL is returned else NOTEQUAL is returned.

static fdkErrorCode core_InitSenderRetraction(int initial_table_size, int increment_size);

Initializes data structures used to keep track of retraction handles of sent messages. Will return one of the following fdkErrorCodes:

<table>
<thead>
<tr>
<th>fdkSUCCEEDED</th>
<th>Succeeded.</th>
</tr>
</thead>
<tbody>
<tr>
<td>fdkOUTOFMEMORY</td>
<td>Unable to allocate memory for Sender Retraction data structures.</td>
</tr>
</tbody>
</table>

static coreSenderRHHashEntry *core_makeRetractionHash(int size);

Allocates memory for the sent retraction data structures. Returns a pointer to the allocated memory or NULL if memory could not be allocated.

static fdkErrorCode core_ResizeSenderRHHash(void);
Resizes the sender retraction handle hash table if it is not currently large enough to accommodate all retraction handles that must be stored. Will return one of the following fdkErrorCodes:

<table>
<thead>
<tr>
<th>fdkSUCCEEDED</th>
<th>Succeeded.</th>
</tr>
</thead>
<tbody>
<tr>
<td>fdkOUTOFMEMORY</td>
<td>Unable to allocate memory for sender retraction hash table.</td>
</tr>
</tbody>
</table>

**static int core_SenderHashFunction(CoreRetractionHandle *rh);**

Given a Retraction Handle returns the position the Retraction Handle is expected to be in the sender retraction hash table.

**static fdkErrorCode core_GetNextRetractionHashEntry(TM_Time ts, MCAST_Handle mh, coreSenderRHHashEntry *rh);**

Prepares a sender retraction hash entry by filling in the time stamp of the message, the destination MCAST_Handle, and retraction handle number in rh. Will return one of the following fdkErrorCodes:

| fdkSUCCEEDED | Succeeded.                  |

**static fdkErrorCode core_SaveSenderRHHashEntry(coreSenderRHHashEntry rh);**

Saves the entry created by core_GetNextRetractionHashEntry in the sender retraction hash table. Will return one of the following fdkErrorCodes:

| fdkSUCCEEDED | Succeeded.                  |

**static int core_VerifyRetractionHashEntry(coreSenderRHHashEntry rh);**

Used by RHM_PrintSenderRetractionHandles to print generate a report of the retractions handles currently stored.

**static fdkErrorCode core_InitReceiverRetraction(int size, int incr);**

Initializes the receiver retraction data structures.

**static int core_ReceiverHashFunction(CoreRetractionHandle *rh);**

Given a Retraction Handle returns the position the Retraction Handle is expected to be in the received retraction hash table. Will return one of the following fdkErrorCodes:

<table>
<thead>
<tr>
<th>fdkSUCCEEDED</th>
<th>Succeeded.</th>
</tr>
</thead>
<tbody>
<tr>
<td>fdkOUTOFMEMORY</td>
<td>Unable to allocate memory for received retraction table.</td>
</tr>
</tbody>
</table>

**static fdkErrorCode core_ResizeReceiverRetraction(void);**

Not implemented. When implemented will return one of the following fdkErrorCodes:
Succeeded.

Unable to allocate memory to resize.

Not implemented.

static fdkErrorCode core_SaveERHQue(CoreRetractionHandle RH, TM_Time TS, MAST_Addr Msg, int MsgSize, long MsgType, int *loc);

Saves the retraction handle in the receiver retraction queue. Will return one of the following fdkErrorCodes:

- **fdkSUCCEEDED**: Succeeded.
- **fdkOUTOFMEMORY**: Received retraction queue is full.

static fdkErrorCode core_SaveERHHash(CoreRetractionHandle RH, TM_Time TS, int QLoc, long MsgType);

Saves the retraction handle in the receiver retraction hash table. Will return one of the following fdkErrorCodes:

- **fdkSUCCEEDED**: Succeeded.
- **fdkOUTOFMEMORY**: Received retraction hash table is full.

static fdkErrorCode core_SendRemainingRetracts(void);

Delivers all retractions to the federate. Will return one of the following fdkErrorCodes:

- **fdkSUCCEEDED**: Succeeded.

TM_Time core_MinRetract(void);

Returns the time stamp of the retract with the smallest time stamp in received retraction handle table.

SESSION MANAGER

RATIONALE

The session manager (SM) is intended to shield the federates from variations in the environmental setup for RTI-Kit modules on different computing and communication platforms. Further, in the case of remotely executed simulations using the Jane framework (Perumalla and Fujimoto 1999; Perumalla and Fujimoto 2001), it mediates communication between the simulation and the Jane client/server. Additionally, it provides a console interface to the simulation, and also supports application-specific instrumentation information exchange between the simulation and the Jane client.

IMPLEMENTATION

The SM expects information on machine configuration in the form of an environment variable named NODEINFO, whose value describes the federation configuration. The value should be of the form npe:nodename,...,nodename:index where npe is the total number of federates in the federation, nodename,...,nodename is the list of nodes on which the federates are executing, and
index is the index of this federate into this node list (0..npe-1). Note that the order of the node list must be specified to be exactly the same for all federates. Note also that the node corresponding to the index must be the same as the node on which the index'th federate is executed.

The SM also requires an environment variable called SESSIONNAME whose value is unique across multiple federation sessions. A simple way of assigning the session name to make it unique across sessions is to use the process ID of one of the federates (say, the first federate)\(^1\).

The SM provides optional console interface that checks for user commands from standard input and executes them. This optional functionality is turned off by default. It can be turned on using the environment variable POLLSTDIN. The SM checks to see if an environment variable called POLLSTDIN is defined and is set to the string "TRUE". If so, it enables console interface processing; else it turns it off.

Note that if the console interface is enabled, the federate should have valid standard input and output descriptors. Thus, if the federates are spawned using the rsh remote shell command, it should be ensured that the standard input and output of the rsh-spawned process are not closed on the remote processor (this can be done, for example, using the "-n" argument to rsh).

**Execution Architectures**

The SM supports three different architectures under which the federation is executed. In the first architecture shown in Figure 6, each federate is manually spawned individually on the command line by the user (for example, using a telnet or rlogin session on each processor). In this mode, the console interface, if enabled, is provided in the same terminal as the rlogin or telnet session.

---

\(^1\) In the current TCP FM implementation, the federates exchange certain information (port numbers) using a file named after the session name in the home directory of the user. If the session name is not unique across executions, old information may be exchanged in later sessions, leading to undesirable behavior.
In the second execution architecture as shown in Figure, the federates are run using the spawner tool provided with the FDK. In this case, the spawner redirects the standard input and output of the SM to that of the spawner. As a result, the console interface of the SM gets exported to that of the spawner.

![Figure 10: Execution architecture when federates are spawned via the FDK spawner](image)

In the last execution architecture shown in Figure 11, the federates are run using the Jane server/client tool provided with the FDK. The Jane tool in turn uses the spawner, with the appropriate command line settings for the spawner. In this case, the spawner redirects the standard input and output of the SM to that of the Jane client. As a result, the console interface of the SM gets exported to a graphical window within the Jane Java client.

![Figure 11: Execution architecture when federates are spawned using the Jane framework](image)

When the Jane client/server is used for federation execution, the SM turns on additional features that are not available in the other execution modes. First, it supports the Jane protocol for exporting and importing data between the federates and the Jane client. Secondly, it periodically reports LBTS advances to the client, so that the client can graphically display the time advances to the user. It also provides support for exporting application-specific data to the client. The client can be augmented with additional Java classes using the Java client API to receive the application-specific data and process it (for example, provide graphical display of the simulation entity activity).

FUNCTIONS

The following are the SM interface functions.
int SESSION_Init( void );

This is invoked by the RTIKIT_Init() to initialize the SM on that processor. Using the environment variables NODEINFO and SESSIONNAME, the SM gets the information about the federation (number of processors, hostnames of the processors, and the local processor ID), and sets up the environment expected by the FM communication module of the RTI-Kit.

void SESSION_Tick( void );

This is invoked by the RTIKIT_Tick() to give cycles to the SM for processing. If the console interface is turned on (using the POLLSTDIN environment variable, as described previously), it polls the standard input for user commands, and processes the commands, if any. As a performance optimization, the SM avoids polling for every SESSION_Tick() call. Instead, it polls every \( n \) calls to SESSION_Tick(), where \( n \) is a tuning parameter.

In addition, if the federate is running under the Jane client/server framework, the SM periodically reports the current LBTS value to the Jane client using the Jane communication protocol.

void SESSION_instrument( const char *lbl, const char *fmt, ... );

This is invoked by the federate to send any application-specific (instrumentation) data to the Jane client. If the federate is not running under Jane, the data is discarded. If not, the data is forwarded to the client. The first argument is an application-specific string label/tag identifying the type of data, so that the client can use it to distinguish among multiple types of application data. The second argument fmt is a format argument roughly similar to the format argument of printf(), except that “%” signs should be omitted. The variable arguments that follow fmt should match the number and type of arguments specified by fmt. In the format, the code for integers is “d”, for doubles is “f”, for characters is “c”, and for strings is “s”. The format string can be NULL or empty to represent zero arguments.

### OTHER UTILITIES

**KIT**

The Kit module presently contains a simple buffer management module (buffers.c), a simple random number generator library (gtw-rand.h, gtw-rand.c and gtw-seeds file), and a wrapper module (kit.h and kit.c) to initialize and tick the different RTI-Kit modules.

The wrapper module is expected to be modified or deleted completely in future a version of the FDK, and the other portions of this module (buffer management and random number generation) are expected to be moved into some other RTI-Kit modules as appropriate. The current random number generator library based on gtw-rand.h and gtw-rand.c is expected to be replaced by a better and updated library.
BUFFER MANAGEMENT

A simple buffer manager is implemented according to the interface given in RTIKIT/COMMON/SRC/buffers.h. A pool of buffers can be pre-allocated and initialized using MB_MakePool(). A free buffer can be obtained using MB_GetBuffer(), and later returned to the pool using MB_FreeBuffer(). All buffers are of a fixed maximum size, as specified with a parameter to MB_MakePool(). The following type is defined:

```
MB_BufferPool
```

Type for a pool of buffers.

The following procedures are defined:

```
MB_BufferPool MB_MakePool (long NBuffers, long BufferSize)
```

Create a new buffer pool with NBuffers, each of size BufferSize bytes. A handle to the buffer pool is returned. 0 is returned if the operation failed and the buffer pool could not be created.

```
ADDR_TYPE MB_GetBuffer (MB_BufferPool Pool)
```

Allocate a single buffer from buffer pool Pool. Return a pointer to this buffer, or the value 0 if the buffer could not be allocated because the pool is empty.

```
void MB_FreeBuffer (MB_BufferPool Pool, ADDR_TYPE Buffer)
```

Return the memory buffer pointed to by Buffer to the pool Pool.

RANDOM NUMBER GENERATOR

A simple random number generation library is provided in gtw-rand.h and gtw-rand.c. Multiple distributions are supported, including exponential and poisson. The library must be initialized using TW_RandInit() before using the generators.

WRAPPER

The RTIKIT_Init() is a convenience wrapper function that invokes the initialization procedures of the RTI-Kit modules, namely, the Session Manager, the FM library, the Log Manager, the MCAST library, and the Time Management library. The RTIKIT_Tick() is a convenience wrapper function that invokes the tick procedures of the RTI-Kit modules, namely, SESSION_Tick() to let the session manager perform console interface actions, FM_extract() to process incoming network data, MCAST_Tick() to perform multicast functions, and TM_Tick() to perform LBTS processing.
The random number module (gtw-rand.c and gtw-rand.h) implements the random number generator described in Communications of the ACM, June 1988, by Pierre L'ecuyer. The generator has a reported period of greater than $10^{18}$.

To initialize the random number generator, use the \texttt{TWRandInit} function:

\begin{verbatim}
void TWRandInit( TWSSeed* S, int r);
\end{verbatim}

This function initializes the seeds specified by \texttt{S}. The second parameter allows for a differing set of seeds should more than one thread of execution call this initialization routine, or to allow for multiple generators to be run concurrently.

There are several routines for retrieving the next value from the generator:

\begin{verbatim}
double TWRandUnif( TWSSeed* S);
long TWRandInteger( TWSSeed* S, long low, long high);
long TWRandBinomial( TWSSeed* S, long N, double P);
double TWRandExponential( TWSSeed* S, double Lambda);
double TWRandGamma( TWSSeed* S, double shape, double scale);
long TWRandGeometric( TWSSeed* S, double P);
double TWRandNormal( TWSSeed* S);
double TWRandNormalSD( TWSSeed* S, double Mu, double Sd);
long TWRandPoisson( TWSSeed* S, double Lambda);
\end{verbatim}

- \texttt{TWRandUnif} returns a uniform value in the range $[0.0, 1.0)$
- \texttt{TWRandInteger} returns an integer in the range $[\text{low}, \text{high}]$
- \texttt{TWRandBinomial} returns the number of trials until \texttt{N} successes give probability \texttt{P}.
- \texttt{TWRandExponential} returns an exponential random variable with mean \texttt{Lambda}.
- \texttt{TWRandGamma} returns a gamma distribution random variable with shape and scale.
- \texttt{TWRandGeometric} returns the count of the first success with probability \texttt{P}
- \texttt{TWRandNormal} returns a normal random variable in the range $[0.0, 1.0)$
- \texttt{TWRandNormalSD} returns a normal random variable with mean \texttt{Mu} and standard deviation \texttt{Sd}.
- \texttt{TWRandPoisson} returns a Poisson random variable with mean \texttt{Lambda}.

The heap module (heap.c and heap.h) implements an in-place min-heap priority queue, with $O(\log N)$ complexity for both insertions and deletions. In addition to removing the top element as in traditional priority queue, this implementation supports the deletion of any element in the heap. This is useful in simulations to implement event retraction.
The type definitions used in the heap interface are as follows.

```c
#define HEAP_DEFAULT_TYPE -1
typedef struct HEAP_Struct *HEAP_PQ;
typedef struct HEAP_NodeS *HEAP_Node;
typedef double KEY_TYPE;
```

The heap priority queue type is defined as `HEAP_PQ`. A heap element is defined as `HEAP_Node`, which uses a forward declaration to a user-defined type `HEAP_NodeS`. The type of key for comparing elements is defined by default as a double, aliased to `KEY_TYPE`.

The functions included in the interface are as follows:

```c
HEAP_PQ HEAP_Create( int, int );
HEAP_Node HEAP_Insert( HEAP_PQ, KEY_TYPE, void * );
void *HEAP_InsertWithType( HEAP_PQ, KEY_TYPE, void *, long, long );
void *HEAP_Delete( HEAP_PQ, KEY_TYPE * );
void *HEAP_DeleteWithType( HEAP_PQ, KEY_TYPE *, long *, long * );
void *HEAP_DeleteArbitrary( HEAP_PQ, HEAP_Node, KEY_TYPE * );
void *HEAP_DeleteArbitraryWithType( HEAP_PQ, HEAP_Node, KEY_TYPE *, long *, long * );
void HEAP_Dump( FILE *, HEAP_PQ );
KEY_TYPE HEAP_Min(HEAP_PQ);
int HEAP_Count( HEAP_PQ );
```

`HEAP_Create()` is used to create a heap. It receives two arguments: the first specifies the initial size (number of elements), and the second specifies the size of the increments by which the heap is expanded when necessary. The returned handle should be used to refer to the newly created heap in all subsequent operations on that heap. Multiple heaps can be created, and each heap receives a unique handle.

`HEAP_Insert()` and `HEAP_InsertWithType()` are used to insert data into the heap, with a given key (e.g. timestamp) and user-defined element type tag. A handle to the heap element in which the data is stored is returned. `HEAP_Delete()` and `HEAP_DeleteWithType()` are used to delete an element from the top of the heap. `HEAP_DeleteArbitrary()` and `HEAP_DeleteArbitraryWithType()` are useful to delete a specific element from the heap, even if that element is not necessarily at the top of the heap. The handle of the element to be deleted must be passed for arbitrary deletion. The data associated with the deleted element is returned. The heap is readjusted after the deletion to retain O(logN) complexity. `HEAP_Min()` returns the key value of the top (minimum) element in the heap. It returns MAXDOUBLE if the heap is empty. `HEAP_First()` returns the top (minimum) element in the heap, and returns NULL if the heap is empty. `HEAP_Count()` returns the number of elements in the heap.
PART IV: APPENDICES

PERFORMANCE

TIME ADVANCE GRANT (TAR) BENCHMARK

The TAR benchmark measures the performance of the time management services, and in particular, the time required to perform LBTS computations. This benchmark contains N federates, each repeatedly performing Time Advance Request calls with the same time parameter, as would occur in a time stepped execution. The number of time advance grants observed by each federate, per second of wallclock time is measured.

EXPERIMENTAL SETUP

JEDI CONFIGURATION

<table>
<thead>
<tr>
<th>Processor</th>
<th>Seventeen 8-processor machines with Pentium III 550 MHz</th>
</tr>
</thead>
<tbody>
<tr>
<td>Communication Interconnect</td>
<td>Gigabit-Ethernet</td>
</tr>
<tr>
<td>Operating System</td>
<td>Redhat Linux version 6.2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Number of Processors</th>
<th>DRTI (TAGs/second)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>7141</td>
</tr>
<tr>
<td>4</td>
<td>1093</td>
</tr>
<tr>
<td>8</td>
<td>260</td>
</tr>
<tr>
<td>16</td>
<td>137</td>
</tr>
<tr>
<td>32</td>
<td>69</td>
</tr>
<tr>
<td>64</td>
<td>32</td>
</tr>
<tr>
<td>120</td>
<td>5</td>
</tr>
</tbody>
</table>

BEETLE CONFIGURATION:

<table>
<thead>
<tr>
<th>Processor</th>
<th>48 Dual processor 300MHz Pentium II</th>
</tr>
</thead>
<tbody>
<tr>
<td>Communication Interconnect</td>
<td>100 Mbps Fast Ethernet</td>
</tr>
<tr>
<td>Operating System</td>
<td>Solaris 7 for x86</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Number of Processors</th>
<th>DRTI (TAGs/second)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>8819</td>
</tr>
<tr>
<td>4</td>
<td>4537</td>
</tr>
<tr>
<td>8</td>
<td>2527</td>
</tr>
<tr>
<td>16</td>
<td>1117</td>
</tr>
</tbody>
</table>
**Nighthawk Configuration:**

<table>
<thead>
<tr>
<th>Processor</th>
</tr>
</thead>
<tbody>
<tr>
<td>SGI Origin 2000 with sixteen 195MHz R10000 processors</td>
</tr>
<tr>
<td>Communication Interconnect</td>
</tr>
<tr>
<td>Shared Memory</td>
</tr>
<tr>
<td>Operating System</td>
</tr>
<tr>
<td>Irix 6.5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Number of Processors</th>
<th>DRTI (TAGs/second)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>30149</td>
</tr>
<tr>
<td>4</td>
<td>12747</td>
</tr>
<tr>
<td>8</td>
<td>7114</td>
</tr>
<tr>
<td>16</td>
<td>2866</td>
</tr>
</tbody>
</table>

**One-Way Latency (OWL) Benchmark**

The latency benchmark measures the latency for best effort receive-order communications. This program uses two federates, and round-trip latency was measured. Specifically, latency times report federate-to-federate delay from when the Update Attribute Values service is invoked to send a message containing a wallclock time value and a payload of N bytes until the same federate receives an acknowledgement message from the second federate via a Reflect Attribute Values callback. The acknowledgement message has a payload of one byte. The one-way latency time is reported as the round trip time divided by two.

**Experimental Setup**

**Jedi Configuration (Same as Above)**

<table>
<thead>
<tr>
<th>Attribute Size</th>
<th>DRTI (OWL microseconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>99</td>
</tr>
<tr>
<td>16</td>
<td>93</td>
</tr>
<tr>
<td>128</td>
<td>100</td>
</tr>
<tr>
<td>512</td>
<td>95</td>
</tr>
<tr>
<td>1024</td>
<td>87</td>
</tr>
</tbody>
</table>

**Nighthawk Configuration (Same as Above)**

<table>
<thead>
<tr>
<th>Attribute Size</th>
<th>DRTI (OWL microseconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>26</td>
</tr>
<tr>
<td>16</td>
<td>26</td>
</tr>
<tr>
<td>128</td>
<td>27</td>
</tr>
<tr>
<td>512</td>
<td>32</td>
</tr>
<tr>
<td>1024</td>
<td>34</td>
</tr>
</tbody>
</table>
**INSTALLATION**

**HOW TO BUILD THE FDK**

1. Extract the package where you want it. There is not currently a separate installation phase.

   $ gzip -c -d fdk-3.0b3.tar.gz | tar xvf -

2. Build the RTIKit and associated RTIs.

   $ cd fdk-3.0b3/FEDSIM
   $ make

   This will build for a specific platform and a specific message transport mechanism. The platform name is calculated by the `aimk` script during compilation, and is generally kept in the `ARCH` environment variable. The transport mechanism is stored in the `FMTYPE` environment variable. `FMTYPE` is taken from the environment, but defaults to TCP.

**HOW TO RUN THE DRTI TM_PING APPLICATION**

Test the kit by running the tm_ping application.

For reference, `${instdir}` is the installation directory, likely `$HOME/fdk-3.0b3`, or something similar. `${host}` is the name of another machine to which you have access.

1. First, run the application on a single host. Some things to watch out for at this stage:

   * The script that runs tm_ping uses rsh. You must have rsh enabled to any host which you specify to the runon script. Namely, you should see listings of your home directory when you execute

     $ rsh `hostname` ls -la
     $ rsh ${host} ls -la

     where `${host}` is the name of a computer to which you have access.

   * The name used for any host must be the name that it calls itself, returned by running "hostname" on that machine. I get this name by executing

     $ host=`rsh ${aliasname} hostname`

   * `${arch}` is the name of the architecture of the current host. This name is generated by the script

     `${instdir}/FEDSIM/MISC/aimk`

     While it's building, note the value assigned to `ARCH`, and save it.
*  \( \text{fmtype} \) is the name of the transport method used by your architecture. It defaults to "TCP". Other options are visible in \( \text{instdir}/FEDSIM/RTIKIT/FM/SRC \)

Again, this is defined at compile-time, and should be noted.

Change to the application directory.

\$ cd \{instdir\}/FEDSIM/RTIS/DRTI/APP/PING

Run the application, making the proper substitutions.

*  In sh-based shells (sh, ksh, and bash):

\$ ARCH=\{arch\} FMTYPE=\{fmtype\} ./runon `hostname` `hostname`

*  In csh-based shells (csh, tcsh):

\$ setenv ARCH \{arch\}
\$ setenv FMTYPE \{fmtype\}
\$ ./runon `hostname` `hostname`

2.  Run on two hosts, with a shared file system, such as nfs.

For this run, the machines involved must share the users home directory. This means that a \$HOME/file on the first host is the same file as on the second host. If your machines do not feature a shared file system, see below for an alternative run method.

Here are additional things to keep in mind:

- All hosts must be of the same architectural type as the current (local) host.
- All hosts must use the same transport type as the current (local) host.
- The "runon" script takes a list of hostnames as parameters. The first of these must be the name of the local host.

Change to the application directory.

\$ cd \{instdir\}/FEDSIM/RTIS/DRTI/APP/PING

Run the application, making the proper substitutions.

*  In sh-based shells (sh, ksh, and bash):

\$ ARCH=\{arch\} FMTYPE=\{fmtype\} ./runon `hostname` \{host\}

*  In csh-based shells (csh, tcsh):

\$ setenv ARCH \{arch\}
\$ setenv FMTYPE \{fmtype\}
\$ ./runon `hostname` \{host\}

3.  Run on two hosts, without a shared file system.
This requires a different strategy. We'll set all environment variables manually. We'll run the binary on the local host first, then start other hosts manually. You'll need several separate login windows for this method.

A. Start on the local host. Do the following in a window on the local host (${local_host_name}).

The environment variable SESSIONNAME contains the name of a file from which tm_ping will gain information about a well-known port. The file should be located in the home directory of the user running tm_ping. The file will be created by the first host (host "0").

The environment variable NODEINFO contains a list of hosts involved in the federation. The format is:

<n>:<host 0>,<host 1>,...,<host n - 1>:i

Where:

<n> = the number of nodes in the federation

<host #> = the name of a specific host

<i> = the number of this node, from 0 to n - 1.

* In sh-based shells (sh, ksh, and bash):

$ SESSIONNAME=fdk-run.session_file
$ NODEINFO=2:${local_host_name},${foreign_host_name}:0
$ export SESSIONNAME NODEINFO
$ ${instdir}/FEDSIM/RTIS/DRTI/APP/PING/OBJ/${arch}/${fmtype}/tm_ping

* In csh-based shells (csh, tcsh):

$ setenv SESSIONNAME fdk-run.session_file
$ setenv NODEINFO 2:${local_host_name},${foreign_host_name}:0
$ ${instdir}/fedsim/rtis/drti/app/ping/obj/${arch}/${fmtype}/tm_ping

B. Examine the file created on the local host. The filename was specified by the SESSIONNAME environment variable. The file will be created in the home directory of the user who ran tm_ping.

C. Copy the file to your home directory on the foreign host (${foreign_host_name}). You can do this by creating it manually, such as with an editor, or by using FTP. The file should have the same name and contents as on ${local_host_name}, and should be in the home directory of the user who will run tm_ping from ${foreign_host_name}.

D. Run tm_ping on the foreign host:

* In sh-based shells (sh, ksh, and bash):

$ SESSIONNAME=fdk-run.session_file
$ NODEINFO=2:${local_host_name},${foreign_host_name}:1
$ export SESSIONNAME NODEINFO
$ {instdir}/FEDSIM/RTIS/DRTI/APP/PING/OBJ/${arch}/${fmtype}/tm_ping

* In csh-based shells (csh, tcsh):

$ setenv SESSIONNAME fdk-run.session_file
$ setenv NODEINFO 2:${local_host_name},${foreign_host_name}:1
$ {instdir}/fedsim/rtis/drti/app/ping/obj/${arch}/${fmtype}/tm_ping

4. This distribution also features an application called the spawner, which is good for invoking simulations in a cluster environment. An example of using the spawner may be found at ${instdir}/FEDSIM/RTIS/DRTI/APP/PING/run-linux-tcp.

RELEASE NOTES

ABOUT RELEASE 3.0

The FDK 3.0 release is a significant change from the past versions. Major changes from the 2.x releases include two new major RTI-Kit modules (RTICore and DDM-Kit) and a significant rework of the DRTI. The DRTI now uses the 1.3 I/F specification precisely. In 2.x versions, the DRTI used a slightly modified version. RTICore is a significant addition to the RTI-Kit. It represents a major simplification in the process of constructing new RTIs. The DDM-Kit is experimental, and should be treated as “alpha” code. Please DO NOT yet rely on the DDM-Kit interfaces; they are subject to change without warning.

Although the DRTI has been greatly improved, it is in a state of flux. There are a number of pending additions to the code base which were not quite completed before 3.0 began testing. Count on continual additions to DRTI functionality through the 3.x releases.

The directory structure and install process is being modified. The interfaces to RTI-Kit and the RTIs are stable, but the location of the libraries, includes, etc., will be refined and simplified. (The code base grew faster than we could keep up with!)

KNOWN BUGS AND LIMITATIONS

RTI-Kit. The RTI-Kit compiles with lots of warnings. These do not cause runtime errors, as far as we know, and will be cleaned up during the process of straightening out the install, and build scripts.

DRTI. There are many limitations to the DRTI. Please read the code to understand them. Notable are the following

- The FDK does not support dynamic joins and resigns
- A federate cannot create more than one federation. (Actually, creating a federation is not supported. All the create/init logic is in the joinFederation call!)
- Tags aren’t supported for any of the calls
- All messages are always sent via reliable transport.
If you are interested in helping to improve the FDK and DRTI, we welcome feedback and contributions. Report bugs, feedback or suggestions to fdk@cc.gatech.edu, and keep up with FDK progress by visiting the FDK homepage frequently (http://www.cc.gatech.edu/computing/pads/fdk.html).
PART V: REFERENCES


GENERAL HLA REFERENCES

http://www.dmsomil/


<table>
<thead>
<tr>
<th>Reference</th>
<th>Title</th>
</tr>
</thead>
</table>