Pipes and Connections

Brian R. Barrett
Information Sciences Institute
University of Southern California
Marina del Rey, CA

Thomas D. Gottschalk
Center for Advanced Computing Research
California Institute of Technology
Pasadena, CA 91125

August, 2004

Abstract

This document describes the low-level **Pipe** and **ConnectionManager** objects of the MeshRouter system. The overall MeshRouter framework provides a general scheme for interest-limited communications among a number of client processes. This generality is achieved by a carefully factorized, object-oriented software implementation. Within this framework, the Pipe and ConnectionManager (base) classes defined in this note specify the interfaces for *i)* actual 'bits on the wire' communications and *ii)* dynamic client insertions during overall system execution. Two specific implementations of the Pipe class are described in detail: a '**MemoryPipe**' linking objects instanced on a single processor and a more general '**rtisPipe**' providing inter-processor communications built entirely from the standard RTI-s library used in current JSAF applications. Initialization procedures within the overall MeshRouter system are discussed, with particular attention given to dynamic management of inter-processor connections. Prototype RTI-s router processes are discussed, and simple extensions of the standard system configuration data files are presented.

Contents

List of Figures

1 Context

The general application of interest for the MeshRouter communications architecture is a number of distinct processes exchanging messages by way of explicit router processes. This is illustrated in the schematics of Fig.(1). In the simplest model shown on the left, a single router manages message exchanges among some number of external clients. This single router model performs poorly for large numbers of clients, and the more general configuration requires an interconnected system of router processes, as suggested by the right hand side of the figure.

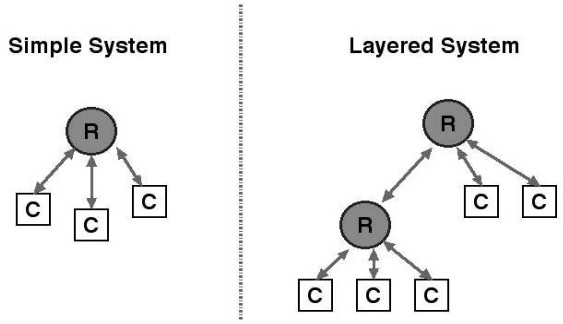

Figure 1: Schematic illustration of the basic Router system. External clients (C) exchange messages through a network of intermediate routers (R).

A high-level description of the overall MeshRouter system is provided in Ref.[1]. The applicability of this approach to general problems of (interest-limited) communications results from a careful factorization of two aspects of the communications problem:

Data Primitives: The nature of basic, low-level constructs such as "Message" and "Interest".

Raw Communications: The formats, protocols, etc. by which raw sets of bits are sent from one processor to another.

Provided that 'Raw Communications' is understood to include the associated task of recognizing and responding to connection requests from external processes, these two categories are the only components of the MeshRouter system that depend on specifics of the client processes.

The MeshRouter architecture effects the required Framework⇔Application factorization through a straightforward hierarchical object design (C++). The overall system is built using a number of (abstract) base classes to encapsulate the general 'Data Primitive' and 'Raw Communications' components, with daughter classes implementing the application specific details. The objects used for the data primitives are described in Ref.[2], with the specific daughter class used for the RTI-s application [3] are presented there in some detail. This note provides the corresponding implementation details for the general 'Raw Communications' tasks, again using RTI-s as a prototype for a specific implementation.

Section 2 describes the fundamental **Pipe** and **ConnectionManager** classes, defining the data content and interfaces assumed within the overall MeshRouter architecture. Details for two specific implementations, the **MemoryPipe** and **rtisPipe**, are contained in Section 3.

The actual router processes for the schematic in Fig.(1) are ultimately built from the basic Router objects from Ref.[1]. This requires additional schemes for initializing the overall system, presumably from some sort of data file. The associated procedures and processes now in place for the JSAF/RTI-s application are described in Section 4.

2 Fundamental Objects

A high-level overview of the objects involved in the schematic router processes of Fig.(1) is shown in Fig.(2). These fundamental building blocks are described in detail in Ref.[1].

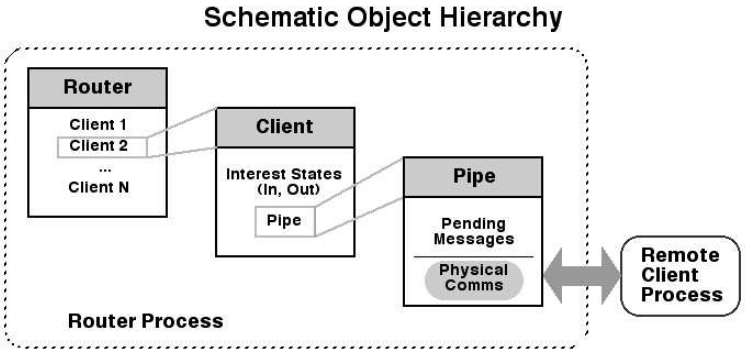

Figure 2: The fundamental MeshRouter objects. The dashed line encompasses the associated schematic router process in the sense of Fig.(1).

The **Router** objects are the overall managers for the interest-limited data flow within the MeshRouter scheme, and the **Client** objects are internal representations of the remote client processes (the 'C' components of Fig.(1)). As is described in Ref.[1], the implementations of the Router and Client objects are independent of application-specific details, such as message formats and physical communications mechanisms.

Each Client object in Fig.(2) contains an additional **Pipe** object that manages the actual data exchanges with the (remote) client processes. Pipe objects are described in detail in Section 2.1.

The schematic in Fig.(2) is incomplete in the sense that it suggests only the *ongoing* operations of the system, ignoring all questions of additions and deletions of remote client processes in the high level schematic of Fig.(1). Dynamic client management is accomplished through an additional class of **ConnectionManager** objects, as described in Sec.2.2.

The Pipe and ConnectionManager objects described below are (virtual) base classes, defining the general interfaces used by the higher level objects in the overall MeshRouter system. Specific instances of these objects (including instances appropriate to the JSAF/RTI-s application) are described later in Section 3.

There is one final object involved in data distribution and dynamic client management, the **ConnectionDescriptor** described in Section 2.2.3. This is actually a simple data container and as such is more properly part of the overall MeshRouter description of Ref.[1]. The brief discussion in Section 2.2.3 is included to clarify the overall interactions between Routers and ConnectionManagers during dynamic client management, as summarized n Section 2.3.

2.1 Pipes

As is suggested in Fig.(2), Pipe objects provide the sole interface between the fixed internal message formats and manipulations of the MeshRouter system *per se* and the application-specific requirements of the external clients in Fig.(1). This is accomplished using a simple, hierarchical object model, as illustrated in Fig.(3).

The left part of Fig.(3) indicates the base class content of Pipe objects, as described in Section 2.1.1, while the right side indicates the instance-specific (virtual) methods described in Section 2.1.2.

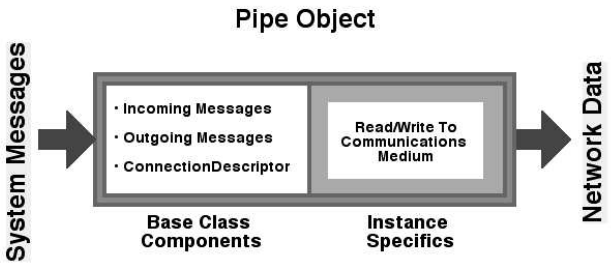

Figure 3: Schematic decomposition of the Pipe object, with base class manipulations of Messages and virtual, instance-specific methods for raw data communications.

2.1.1 Base Class Methods and Data Content

```
// ----- Base Class Object Data ------------------------------------

protected:

  bool d_status;                      // Health and Status Flag
  MessageList d_outgoing_queue;       // Messages To Be Sent Out
  MessageList d_incoming_queue;       // Temporary Storage, New Messages
  ConnectionDescriptor d_descriptor;  // Connection Descriptive Data
```

Figure 4: Object data for the Pipe base class.

The essential data content of the Pipe base class is shown in Fig.(4). The two MessageList objects provide temporary storage for message exchanges with the remote client process. These lists are maintained in the application-independent form described in Ref.[2]. The status flag distinguishes fully operational Pipes (i.e., fully established communications 'arrows' in the sense of Fig.(2)) from unestablished 'open Pipes'.

The **ConnectionDescriptor** is a simple data container that specifies the nature of the underlying communications link, as required to initialize communications within the general framework of Fig.(1). As might be expected, the appropriate data content for this low-level 'nit' was the subject of considerable experimentation during design of the overall MeshRouter framework. In the end, it was found best to implement this object as a simple data container with two primary access methods:

```
const std::string ConnectionDescriptor::GetPeer()
const std::string ConnectionDescriptor::GetType()
```

The '**GetPeer()**' method returns an identifier for the remote end of the communications link in Fig.(2). In typical applications, this is a host name or IP address, although other interpretations are plausible (e.g., a simple node name in an MPI-based implementation). The '**GetType()**' method returns a string defining the type of underlying connection (e.g., "TCP/IP", "UDP", ...). The ConnectionDescriptor class also contains an additional data entry and appropriate set/query methods to hold global connection specification data, as maintained within the **RouterConfig** object defined below in Section 2.2.2.

All interactions between Client and Pipe objects in Fig.(2) are done through base class Pipe methods. The essential, non-virtual methods are listed in Fig.(5).

```
// ----- Message Queue Management ----------------------------------

  void AddMessageList( const MessageList& mlist );
  void AddMessage( const Message& message );
  int RetrieveMessages( MessageList& messages );

// ----- Status Interrogation --------------------------------------

  bool Status() const;

// ----- ConnectionDescriptor Management ---------------------------

  void SetConnectionDescriptor( const ConnectionDescriptor& desc );
  ConnectionDescriptor& GetConnectionDescriptor();
  const ConnectionDescriptor& GetConstConnectionDescriptor() const;
```

Figure 5: Base class (real) methods for the Pipe object.

The **AddMessageList** method simply appends the indicated list of messages to the outgoing queue of Fig.(4) while **RetrieveMessages()** retrieves all messages from the incoming queue (clearing the queue). The **Status()** method provides external access to the health/status state of the Pipe while the remaining methods of Fig.(5) provide access to the ConnectionDescriptor component of Fig.(4).

There are, in fact, a host of additional base class methods in the current implementation of the Pipe object. These include debug/print methods and methods to interrogate, prune and clear the outgoing message queue. These additional methods are all extremely straightforward with one exception: methods to deal with 'Priority Messages'. This extension is discussed in Section 2.1.3.

2.1.2 Virtual Methods

The instance-specific virtual interfaces for the Pipe object are limited to the three methods listed in Fig.(6):

```
// ----- Virtual Interfaces ----------------------------------------

  virtual bool Read() = 0;
  virtual bool Write() = 0;
  virtual bool Establish() = 0;
```

Figure 6: Virtual/Interface methods for the Pipe object.

From the perspective of the Client⇔Pipe interactions of Fig.(2), these three methods completely specify the instance-specific actions for the right hand side of Fig.(3). The assumed/required actions for these three methods are as follows:

Read(): Look for new incoming user messages from the remote client. Reformat these messages as MeshRouter message objects (using basic Message methods, as described in Ref.[2]), and add them to *d_incoming_queue*.

Write(): If the outgoing MessageList is non-empty, attempt to send some number of (reformatted) messages to the remote client process. Successfully transmitted messages are to be removed from *d_outgoing_queue*.

Establish(): If the current Pipe::Status() is false, attempt to create the associated communications link to the remote client process, using the specifications contained in the ConnectionDescriptor. The *d_status* data element must be set to 'true' when a successful connection is established.

The message reformatting associated with the Read() and Write() methods is as described in Ref.[2], effecting the translation between internal MeshRouter Message representations and that expected by the remote client processes.

The Client objects in Fig.(2) are responsible for invoking the methods of Fig.(6). For example, the **Establish()** method is executed if the Pipe::Status() flag is false (subject to additional '*persistence*' constraints, as discussed in Section 2.3). The **Read()** and **Write()** methods are invoked regularly for established Pipes. The amount of actual data transfers accomplished per invocation depend on specifics of the derived Pipe class implementation. In particular, the interface specifications do not mandate that *d_outgoing_queue* be completely drained for each invocation of Write().

2.1.3 Extension: Priority Messages

The MessageList objects in Fig.(4) are effectively operated as simple FIFOs. Messages are added to one end of the list, and messages are retrieved and processed in reverse order from the other end of the list. This maintains a sense of (approximate) time ordering for message processing within the system.

This procedure could easily be generalized to accommodate tagged 'high priority' messages, with the Pipe base class required to insert recognized high priority messages into the 'next out' end of the internal message list. Such a method (**AddPriorityMessage()**) already exists. The shortcomings in the current implementation have to deal with the *recognition* of high priority messages as they flow through the system.

The required extensions to manage message priorities are straightforward. Since the entire concept of 'message priority' is application dependent, the task of assigning message priorities is properly done in the **MessageInterpreter** object of Ref.[2], with an additional **Message::GetMessagePriority()** method added to the MeshRouter's basic message framework. The subsequent list manipulations within the Client and Pipe objects would simply interrogate this priority tag and insert incoming messages onto the appropriate ends of the internal lists.

Implementation of priority tags for individual messages is extremely straightforward. (For the specific JSAF/RTI-s application, for example, priority tags are already part of the standard message headers.) Once individual Message priority tags are in place, a variety of application-specific schemes for priority management are plausible within the general methods of Figs(5,6).

2.2 ConnectionManagers and Related Objects

Dynamic client management is an essential requirement of the MeshRouter framework. In this regard, a generic start-up procedure for the system schematic of Fig.(1) might be as follows:

1. The individual processes associated the routers and clients in Fig.(1) are all started independently.
2. Certain processes within Fig.(1) initiate communications links to appropriate router processes.
3. The routers must respond to this requests, thus establishing the communications links.

The **ConnectionManager** base class defined in Section 2.2.1 provides the mechanisms for router responses to connection requests. For real applications such as JSAF/RTI-s, the simple GetPeer() and GetType() information of the ConnectionDescriptor object of Section 2.1.1 is generally not adequate for complete specification of the underlying physical communications. This shortcoming is resolved through the (essentially singleton) **RouterConfig** data container described in Section 2.2.2. Robust operation of the MeshRouter framework requires a number of additional characterizations of the inter-processor links in Fig.(1). This information is contained in the **ClientDescriptor** object defined in Section 2.2.3.

2.2.1 ConnectionManagers

The ConnectionManager class defines the mechanism by which router processes listen for and respond to connection requests from remote clients. This is implemented through a (virtual) base class with primary methods as listed in Fig.(7).

```
// ----- Primary Interfaces ----------------------------------------

  virtual bool Initialize() = 0;
  virtual Pipe* Listen(std::string UniqueID) = 0;
  virtual void Shutdown() = 0;

// ----- Set RouterConfig ------------------------------------------

  void SetConfig(std::string flavor,  RouterConfig& rc );
```

Figure 7: Interface methods for the ConnectionManager base class.

The essential method in this class is **Listen()**, in which the ConnectionManager is required to detect incoming connection requests, returning a fully initialized Pipe object whenever such a request is detected. This is clearly an application-specific task. The associated **rtisPipeConnectionManager** for the JSAF/RTI-s application is described in Section 3.

The types of connections (UDP, TCP, MPI, ...) needed in an actual execution associated with Fig.(1) is ideally a run-time decision determined by configuration file information. This is supported through a straightforward factory procedure, with the essential method

```
virtual ConnectionManager* create(std::string type, RouterConfig *conf);
```

creating and returning a concrete ConnectionManager instance for the daughter class indicated by the 'type' string. During initialization procedures, the Router objects of Fig.(2) will typically create and store a list of active ConfigurationManager instances based on configuration file specifications. During subsequent operations, the Listen() methods of all instanced ConnectionManagers are executed to detect and process incoming connection requests.

The remaining base class methods in Fig.(7) provide additional generic interfaces for initializing and terminating instance-specific Connection managers. Of particular interest here is the **SetConfig()** method at the end of Fig.(7). This is the general interface for specifying the lowest-level details of the actual communications protocols, as is discussed in the following subsection.

2.2.2 The RouterConfig Data Container

The need for an additional Pipe specification mechanism is easily appreciated by considering the host of run-time communications options available within the JSAF/RTI-s application: encryption, checksumming, and so on. This information must be made available to the Establish() and Listen() methods of Figs.(5,6).

The nature of connection specification data is entirely dependent on details of the user applications in Fig.(1). It is not feasible to anticipate all the query methods needed in specific Listen() and Establish() implementations. As such, it is not possible to design a standard 'base:daughter' object hierarchy to maintain the required system configuration data. Instead, the MeshRouter framework uses a simple, user-defined data container: the **RouterConfig object**.

Aside from constructors and destructors, the MeshRouter framework makes no assumptions on the nature of this object. It is a compile/link option within the software, with users responsible for providing

the appropriate implementation source code. The MeshRouter itself simply passes a pointer to the RouterConfig object around as needed. The **SetConfig()** method for the ConnectionManager base class listed in Fig.(7) is one example. Similar specification methods exist for the ConnectionDescriptor object of Section 2.1.1.

The RouterConfig implementation for the JSAF/RTI-s application described below in Section 3.2.2 provides an instructive illustration of this general object/interface.

2.2.3 ClientDescriptors

In order to support general interconnection schemes among routers and clients, it is necessary to introduce one additional object describing the 'character' of individual communications links in Fig.(1). This is the role of the **ClientDescriptor** object defined in Ref.[1]. The ClientDescriptor is a simple data container with straightforward access methods such as:

```
bool ClientDescriptor::IsDataSource() const
bool ClientDescriptor::IsDataSink() const
bool ClientDescriptor::IsUpper() const
bool ClientDescriptor::IsPersistent() const
```

As was noted above, the ClientDescriptor is not, strictly speaking, part of the the Pipe/ConnectionManager framework itself. Instead, ClientDescriptors are contained within the Client objects of Fig.(2) and are used to manage message flows during general Router⇔Client interactions.

The 'IsPersistent()' flag is relevant for purposes of this note, as it ultimately controls invocations of the Pipe::Establish() method of Fig.(6), as described in the following section.

2.3 Operational Details: Lost and Found Clients

In order to indicate how the various objects and methods of the previous sections are all tied together, it is useful to consider a brief, high-level overview of initializations and operations of the schematic router process within the dashed boundary of Fig.(2). The focus in this section is on dynamic client management within the MeshRouter framework. Normal communications in the sense of the Read() and Write() methods of Fig.(6) are automatically invoked by the higher level Router and Client objects of Ref.[1],

Router Initializations

The router process initializes the appropriate RouterConfig data object of Section 2.2.2 and then initializes two internal lists from the configuration data:

1. A set of applicable ConnectionManagers (Section 2.2.1).
2. An initial set of known client processes, specified by 'type' and 'peer' values in the sense of the ConnectionDescriptor object of Section 2.2.1

The initial clients from the configuration file are all tagged as 'persistent' in the sense of Section 2.2.3. ConnectionDescriptors and (unestablished) Pipe objects are created. A Client object with this ConnectionDescriptor and Pipe is created and added to the Router object's internal client list.

In practice, router initialization typically involves the creation of specific instances of ConnectionManager and Pipe objects from simple (i.e., character string) specifications within the configuration data. This is accomplished through a standard (C++) factory formalism, with **PipeFactory** and **ConnectionFactory** base classes and appropriate specific derived instances.

Ongoing Operations: Existing Clients

The Router periodically examines its list of associated Clients, looking for clients with false values of Pipe::Status() in the sense of Fig.(5). The action taken on encountering such a 'broken pipe' depends on the IsPersistent() flag associated with the Client object:

Persistent Client: An attempt is made to establish the indicated connection through the Pipe::Establish() method of Fig.(6).

Non-Persistent Client: The associated Client object is removed from the router's internal list and destroyed.

Ongoing Operations: New Clients

The Router periodically steps through its list of associated ConnectionManager objects, exercising the Listen() method of Fig.(7) for each manager. If a non-null Pipe is returned, a new Client object is created for that Pipe and added to the router's client list. All Client objects created in this manner are marked as Non-Persistent, in the sense of Section 2.2.3.

3 Specific Implementations

This section describes the two specific instances of the general Pipe and ConnectionManager formalism now available within the MeshRouter software: the **MemoryPipe** and the **rtisPipe**.

3.1 MemoryPipes

The MemoryPipe class is used to support communications between distinct Router objects instanced within a single executable process, as in the high-level '**MeshTriad**' object described below in Section 4. The intent is simply to tie together two Pipe objects in the sense of Fig.(4) with, e.g., the *d_incoming_queue* list of one MemoryPipe associated with the *d_outgoing_queue* MessageList of a specific MemoryPipe at the other end of the communications link (noting that MemoryPipes always exist in tightly-coupled pairs).

```
// ----- Object Data ---------------------------------------------

private:

  MemoryPipe *d_partner;              // Other "End" Of Pipe
  MessageList *d_read_list;           // My Sent Messages
  MessageList *d_write_list;          // My Incoming Messages
```

Figure 8: Additional object data for the MemoryPipe class.

The additional object data needed for the MemoryPipe object is almost trivial, as shown in Fig.(8). These pointers are initially all null and are set (pairwise) by the additional method

```
bool Bind( MemoryPipe& other);
```

The Bind() method is invoked once per pair. Two additional MessageLists are created for the read/write lists of one object. The read/write lists of the other, partner object are simply set to these same lists in reversed order.

The actions of the fundamental Read() and Write() methods of Fig.(6) are extremely simple:

Read(): All queued messages in *d_write_list* are transferred to the base-class *d_incoming_queue* MessageList of Fig.(4)

Write(): All queued messages in *d_outgoing_queue* are transferred to *d_read_list*.

(These actions perhaps clarify the peculiar naming conventions in Fig.(8) with, e.g., *d_read_list* associated with Read() actions in the partner MemoryPipe.)

In practice, the full ConnectionManager apparatus of Section 2.2 is not used within typical router processes involving MemoryPipes. For example, the MemoryPipes within the MeshTriad object of Section 4 are simply created in pairs, bound, and inserted into the appropriate (persistent) Client objects.

The existing, incomplete MemoryPipeConnectionManager implementation is based on a simple 'pending queue' formalism:

1. Each end of a MemoryPipe⇔MemoryPipe connection is regarded as 'persistent' in the sense of Section 2.2.3. This means that both ends will eventually invoke the Establish() method of Fig.(6).'
2. When the first side of the MemoryPipe pair executes the Establish() method, an appropriate 'MemPipeRequest' object is created, containing descriptors to both ends of the associated link and a pointer to the requesting MemoryPipe.
3. When the second side of the link eventually executes Establish(), the associated queued request is found, the two pipes are Bound(), and the status flag for each pipe is set to true.

In practice, the shortcoming in the current implementation has to deal with the way in which the 'GetPeer()' strings from the two MemoryPipes are paired together. The correct solution would involve lists of associated pair names in some sort of configuration file. Since dynamic MemoryPipes are not used in existing MeshRouter applications, this extension remains undeveloped.

3.2 rtisPipes

Unlike the MemoryPipe of the preceding section, the rtisPipe is a full implementation of all the general capabilities of Section 2. This implementation is is important for several reasons:

1. The underlying communications primitives are regarded as 'cast in stone'. That is, the MeshRouter formalism is required to adapt to RTI-s constraints, not the other way around.
2. Given the careful, Framework⇔Application factorization within the overall MeshRouter framework, it is, in fact, fairly straightforward to construct the required Pipe and ConnectionManager instances entirely from objects within the standard RTI-s library.
3. The resulting "RTI-s/Meshrouter" system is compatible with ongoing, independent developments within RTI-s itself and is thus appropriate for use in ongoing JSAF/RTI-s applications, such as the Joint Urban Operation (JUO) activities described in, e.g., Refs.[3][4].

In view of these characterizations (and claims), the descriptions below are a bit detailed. Relevant aspects of the RTI-s framework are given in Section 3.2.1. The next thee subsections describe the implementations of the RouterConfig, Pipe, and ConnectionManager objects within the RTI-s framework. Some plausible refinements on this initial, largely vanilla implementation are mentioned in Section 3.2.5

3.2.1 Basic Considerations

The RTI-s uses a flexible '*data_path*' framework for message exchanges among clients, as indicated in the schematic flow diagram in Fig.(9). User messages enter the RTI-s services, are processed by a number of intermediate processing steps ('*dataflow_nodes*'), and eventually go out onto the actual communications network. The process is repeated (reverse order of the *dataflow_nodes*) on the receiving end.

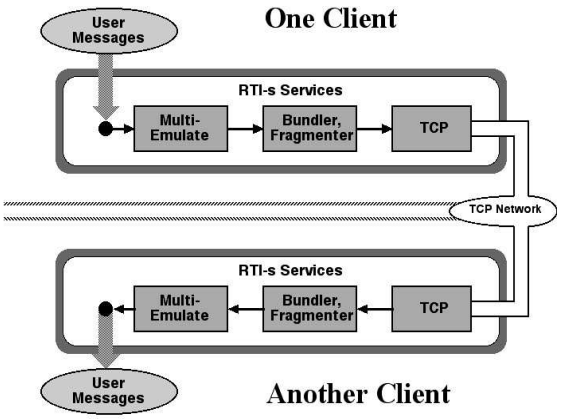

Figure 9: Schematic illustration of data flow between clients in RTI-s.

The overall communications flow for Fig.(9) is implemented through a linked list of *dataflow_node* objects. The type/order of these nodes, as well as node-specific parameters (e.g., Kerberos authentication for the TCP node) are all specified in the standard RTI-s 'RID' initialization data file.

This suggests a straightforward implementation model for the needed **rtisPipe** object:

1. The **rtisPipe** class inherits from both the MeshRouter **Pipe** base class and the RTI-s **datapath_node** class.
2. The **RouterConfig** object for RTI-s must provide suitable access to the RID file.

There is, in fact, only one significant complication, and that has to do with enumerated interest management.

The '**MultiEmulate**' data flow node in Fig.(9) is the standard RTI-s module for enumerated interest management. It is necessary to make the interest-state information processed and maintained in the MultiEmulate node available to the broader MeshRouter framework. (Put differently, a standard RTI-s dataflow absorbs/uses incoming interest declarations while the MeshRouter framework need access to these declarations). A very simple solution to this 'disappearing interest' problem is presented in Section 3.2.3

3.2.2 The RouterConfig Object and Pipe Creation

Conceptually, the RouterConfig object for RTI-s MeshRouter implementations is little more than a wrapper around the standard RTI-s RID object. This means that all the standard librid and liburlreader functionality of RTI-s is transparently available, including run-time identification of the appropriate RID file from environment variables.

The actual RTI-s RouterConfig implementation is essentially that of a singleton. Only one actual instance is created during an execution, and all subsequent instances simply point to the initially instanced object.

Fragments from the **rtisPipe** object definition are shown in Fig.(10), illustrating the multiple inheritance mentioned in the previous section. The interest here is on the constructor. All rtisPipe instances are created through an **rtisPipeFactory** instance of the general factory formalisms noted in Section 2. The essential creation method for any PipeFactory has the form:

```
Pipe* PipeFactory::create(
    std::string type,       // Identifier (TCP, UDP, ...)
    std::string hostname,   // Peer Identifier
```

```
class rtisPipe : public Pipe, public dataflow_node
{
 public:
  rtisPipe(data_path *);
  virtual void receive(ldm_buf *buf);
  virtual void remote_subs_changed();
  ... ...
 private:
  data_path *path;
};
```

Figure 10: Essential structure and data of the rtisPipe class.

```
    RouterConfig *conf );   // Additional Configuration Information
```

When an rtisPipe is created by an rtisPipeFactory, the RID specifications are used to create the entire data path specification (i.e., the nature and ordering of the appropriate component 'processing boxes' in Fig.(9)), and this path is stored in the created rtisPipe object through the explicit constructor.

3.2.3 Pipe Read() and Write() Implementations

There is no distinction between 'data' and 'interest' messages in the overall MeshRouter Pipe::Read() and Pipe::Write() methods of Section 2.1. The Client objects in Fig.(2) identify and process the interest declaration messages and manage overall interest-limited message exchange.

The implementation within RTI-s is a bit different in that interest management is done by one of the *dataflow_nodes* in the overall *data_path* (specifically, the 'MultiEmulate' box in Fig.(9)). This requires different processing procedures for interest and non-interest messages within the Read() and Write() implementations.

Non-Interest Messages

The RTI-s packages user messages (i.e., 'byte strings') using a general '*ldm_buf*' data container. Write() and Read() for these messages involves little more than translating:

MeshRouter Message $\Leftrightarrow$ RTI-s ldm_buf

and then invoking basic dataflow_node I/O methods inherited through Fig.(10):

Write(): Create an *ldm_buf* and fill it from the Message contents, using standard methods of the *data_path *path* element of Fig.(10). Send it off into the full data path. Release the buffer and delete the sent message from *d_outgoing_queue* component of Fig.(4).

Read(): Accept an incoming *ldm_buf* object from the RTI-s processing of Fig.(9). Create and fill a Message object to hold this data and push it onto the *d_incoming_queue* list of the Pipe object.

The description of Read() is actually only notional. The RTI-s processes incoming messages through a callback mechanism (method 'receive()'). The processing described above is actually done in the **rtisPipe::receive(***ldm_buf** **)** method. The rtisPipe::Read() method simply exercises the standard RTI-s scheduler to ensure that receive() is invoked.

Interest Messages

As noted previously, the MultiEmulate node in the standard RTI-s data flow chain enforces interest-limited communications. This means two things:

1. The declared interest state of the remote end is maintained within the MultiEmulate data flow nodes.
2. Remote interest declarations are completely processed/absorbed in the MultiEmulate node.

Some additional work is needed to make the remote interest declarations visible to the overall MeshRouter system. The simplest solution is to leave standard MultiEmulate interest processing untouched and then generate additional 'incoming' messages to MeshRouter layers to reflect interest changes.

The required additional message generation is accomplished through the 'remote_subs_changed()' method in Fig.(10). This is a standard RTI-s callback invoked when the MultiEmulate node in Fig.(9) detects a change in the remote interest declaration. The rtisPipe version of the callback simply creates an appropriate interest Message from this state and adds it to the *d_incoming_queue* list of Fig.(4).

A similar, 'redundant' reliance upon MultiEmulate is also used for outgoing interest declarations during rtisPipe::Write(). When an incoming interest message is processed by Write(), it is converted to an RTI-s interest declaration object and passed to the MultiEmulate data flow node through the standard *dataflow_node::set_subs()* method. The MultiEmulate node in Fig.(9) will eventually regenerate and send the outgoing interest declaration message to the remote process.

3.2.4 Connection Management

The RTI-s already implements procedures for data-driven specification of 'listeners' for all manners of connection request processing, as needed for an rtisConnectionManager. This leads to a straightforward, multiple-inheritance implementation, as shown by the header fragment in Fig.(11).

```
class rtisPipeConnectionManager : public ConnectionManager, public new_path_client
{
 public:
  virtual void new_path(data_path *path);
  ... ...
 private:
  void SetServer(server *s);
  std::list<Pipe*> new_connections;
  server *d_server;
};
```

Figure 11: Essential structure and data of the rtisPipe class.

The RTI-s detects and processes new connection requests through an asynchronous callback procedure, with the relevant method:

```
void new_path_client::new_path(data_path *path);
```

The callback instance for rtisPipeConnectionManager creates a new rtisPipe for the specified path, sets the peer and type flags, and then pushes the created pipe onto a temporary list (the '*new_connections*' data element in Fig.(11). The rtisPipe::Listen() method (Fig.(7) tickles the connection path and, if the temporary list is non-empty, returns (and pops) the front entry.

The RTI-s server object (*d_server*) in Fig.(11) does all the real work associated with recognizing and processing external connection requests. This component of Fig.(11) is created within the standard

```
ConnectionManager* create( ... )
```

method for the rtisConnectionManagerFactor, as discussed in Section 2.2.1.

3.2.5 Plausible Extensions

Implementation of RTI-s versions of the Pipe and ConnectionManager classes was ultimately straightforward, given the multiple-inheritance object designs in Figs.(10,11) and availability of the full RID file information through the RouterConfig implementation of Section 3.2.2. The RTI-s⇔MeshRouter integration is clean and essentially complete in the initial implementation, with the exception of a few hard-wired values (e.g., the *libmcast* version number).

One needed extension of the current formalism has already been mentioned in Section 2.1.3. Once the **MessageInterpreter** object Ref.[2] has been generalized to support user-defined message priorities, the pending queue manipulations of Section 3.2.2 will require minor extensions to support priority-modified message flows.

A somewhat awkward low-level problem has to deal with the data transfer

$$[\text{ RTI-s } \textit{ldm\_buf} \text{ }] \Rightarrow [\text{ MeshRouter } \textbf{MessageObj} \text{ }]$$

needed in rtisPipe::Read(). There are two problems:

1. The efficient *ldm_buf* methods for data extraction are private.
2. The methods in Ref.[2] for initializing a MessageObj assume that data are presented in a contiguous byte array.

The first problem was solved by extending the set of friends for *class ldm_buf* to include the needed MeshRouter objects. This appears to be a standard procedure within th RTI-s code, as the friend list for *ldm_buf* is quite lengthy. It should be noted that the additional friends for *ldm_buf* were the only modifications to the RTI-s source code needed for the rtisPipe implementation.

A short-term solution of the second problem was implemented in terms of an explicit MessageObj initialization method given an *ldm_buf* object. This works, but stretches the overall Framework⇔Application factorization of the MeshRouter architecture. A cleaner implementation, using a hierarchy of '**Message-Helper**' classes, is under development.

A final refinement worth pursuing has to deal with the multiple processing of interest declarations noted in Section 3.2.3. The interest declaration reformatting and interest state maintenance tasks done by the MultiEmulate node in Fig.(9) are already done by the Client objects in Fig.(2). It may be possible to specify simpler data flow chains for MeshRouter processors without the MultiEmulate node.

4 Prototype Router Processes and Initializations

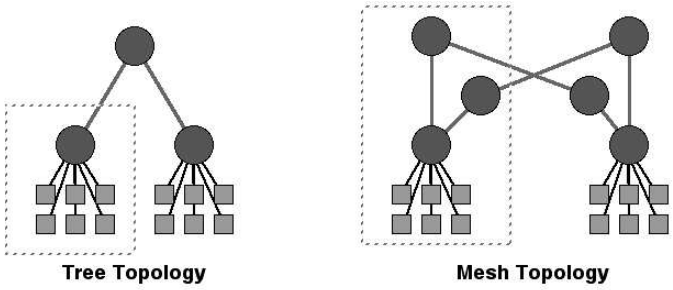

Figure 12: Examples of different communications topologies. The dashed boxes indicate the basic 'scaling units' in the context discussed in Ref.[1].

The Meshrouter architecture supports a variety of topologies for the communications network linking the client processes. Two particular examples are illustrated in Fig.(12). The solid circles in Fig.(12)

represent full router processes in the sense of Fig.(1). The concern in this section is the construction of these processes from basic MeshRouter objects and the run-time initialization of the processes for the RTI-s application.

The Tree Topology in Fig.(12) is, in fact, already supported and used within the JSAF/RTI-s application. The run-time configuration is based on a simple 'vertical hierarchy', with the RID file specifying the 'upper' communications partner for each element (simulator or router) in the network.

The Mesh Topology in Fig.(12) represents a more complex communication network derived from the scalable SFExpress communications model introduced in Ref.[5]. The basic building block in this network is a set of simulators associated with three routers (called 'Pop-Up', 'Pull-Down', and 'Primary', top to bottom within the dashed box). While all data flows within the tree topology are essentially egalitarian, those associated with the communications links for the mesh differ depending on the router type. (This is, in fact, the rationale for the ClientDescriptor object defined in Section 2.2.3.)

Section 4.1 presents a set of plausible extensions to the standard RID file connectivity specifications capable of supporting both topologies of Fig.(12). An associated **MeshRouter** object suitable for implementing any of the schematic routers (solid circles) in Fig.(12) is presented in Section 4.2.

However, implementation of the full Mesh Topology in Fig.(12) with separate processors dedicated to each router is not particularly efficient. (It was found in Ref.[5], for example, that the loads on Pop-Up and Pull-Down routers are small, even for large mesh configurations with a few dozen basic triads.) A more efficient procedure involves three Router objects implemented within a single processor. This **MeshTriad** model is described in Section 4.3, providing an effective reimplementation of the scalable SFExpress communications framework for JSAF/RTI-s applications.

Finally, Section 4.4 describes a single executable process for the RTI-s/JSAF application that implements either the MeshTriad or MeshRouter execution model (run time decision) as well as a variety of standard, interactive console features from RTI-s.

4.1 RID File Extensions

The standard RTI-s RID formalism specifies communications links within a '**connectivity_map**' component of the larger '**stream_manager**' segment within the RID file. The assumed syntax within this segment is summarized in Fig.(13).

```
(stream_ manager
  ...
  (connectivity_map
    (node "host1" "type" "host2")
    (node "host1" "type" "host2")
    ...
  )
)
```

Figure 13: RID file stream_manager and connectivity_map components.

The intent/consequences of an individual "(node ...)" entry within the connectivity_map is as follows:

> The process on host1 is to initiate a connection of the specified type (TCP, UDP, ...) to the (upper) process on host 2, where "upper" refers to the standard vertical hierarchy implicit in the Tree network of Fig.(12).

In order to generalize the stream_manager to support the richer communications topology of the mesh network, a new '**mesh_map**' component has been developed, with basic syntax/content as shown in Fig.(14).

```
(mesh_map
  (mesh
    (primary "host1" "type" popup "host2")
    (primary "host1" "type" pulldown "host2")
    (pulldown "host1" "type" popup "host2")
    ...
  )
)
```

Figure 14: Illustration of the proposed '**mesh_map**' extension to the stream_manager RID file segment. Multiple *mesh* segments can exist within a single *mesh_map*.

Before addressing specifics of Fig.(14), it seems fair to characterize these extensions as a real "work in progress". The syntax and nomenclature in Fig.(14) is clearly derived from the structure of the old SFExpress architecture and, indeed, the generalized RID specifications in Fig.(14) were implemented with the specific intent of seamlessly accommodating SFExpress-like router networks within standard RTI-s procedures. The model in Fig.(14) is adequate for this limited task. However, it does not support a number of obvious extensions within the overall MeshRouter framework, such as additional 'shadow routers' in the schematic of Fig.(12) to provide some level of fault tolerance with respect to router failures. In this regard, the additional RID semantics in Fig.(14) are best regarded as provisional.

Returning to specifics of Fig.(14), the three different entry types for "**Host1 To Host2 communications of specified Type**" in the the map fragment distinguish the three classes of router to router connections (and ClientDescriptors) needed in the Mesh Topology model of Fig.(12). In particular,

Primary To PopUp: The upper router is a data sink but not an interest sink for the lower router.

Primary To Pulldown: The upper router is a data source and an interest sink.

PullDown To PopUp: The upper router is a data source and an interest sink, with the additional provision that the two routers must not be part of the same triad in the sense of Fig.(12)

Extended RID file semantics and parsing are done in an auxiliary **RouterConnectivity** object within the MeshRouter. In addition to the most basic entries listed in Fig.(14), this object supports a number of shorthand procedures for specifying entire router triads. Of particular interest is the specification of an entire MeshTriad process (Section 4.2) with a single entry

(triad "hostname" "type")

within a '(mesh ...)' component of Fig.(14).

Finally, it is worth noting that the "type" entries in Fig.(14) are a bit more extensive than those of the standard RID connectivity_map of Fig.(13). In particular, the 'MemoryPipe' connectivity of Section 3.1 is supported, provided that the two host entries in the connectivity entry are the same.

4.2 The MeshRouter Object

The essential content and tasks of high-level objects associated with either a single router or a triad for Fig.(12) are straightforward:

Data Initialization: Create and bind the required '**MessageInterpreter**' object from Ref.[2].

Router Initialization: Create and initialize the appropriate Router object, according to RID file specifications.

Operations: Repeatedly exercise the router's '**Tick()**' method until the overall process is terminated.

Data initialization involves a few basic methods from Ref.[2]. Ongoing operations are described in Section 4.3. The concern in this section is initialization of a single (generic) Router object in the sense of any of the solid circles in Fig.(12).

The router initialization task is straightforward. An instance of the RouterConfig object from Section 2.2.3 is created and two lists are extracted:

1. The set of all 'listener' entries from the RID file.
2. The set of all connectivity entries (Figs.(13,14)) whose first hostname matches the current process.

The first list defines the set of appropriate **ConnectionManagers** from Section 2.2.1. These are created using an appropriate factory mechanism and passed to the Router object (using the **AddConnectionManager()** method described in Ref.[1]).

The second list defines the router's set of 'Persistent Clients', in the sense defined in Section 2.3. For each entry in this list, an appropriate (open) Pipe is created, and a ClientDescriptor is created and set according to details of the associated connectivity_map or mesh_map entry. A Client object is created for the (Pipe,ClientDescriptor) pair and inserted into the Router.

4.3 The MeshTriad Object

The MeshTriad Object contains three Router objects: The Primary, PopUp, and PullDown routers for a single triad in the Mesh Topology configuration of Fig.(12). The Primary⇔PopUp and Primary⇔PullDown connections within a triad are done with MemoryPipes. These pipes are created and fully configured explicitly during MeshTriad::Initialize().

ConnectionManager and (persistent) Client initialization is complicated by the fact that there are two distinct types of connection requests that must be managed during MeshRouter::Tick():

1. Connection requests form PullDown routers on different MeshTriad processes.
2. Connection requests from other (unknown) processes.

The first category defines clients for the PopUp router while those in the second category must be added instead to the Primary router.

The solution to this client segregation problem is straightforward:

1. The entire '(mesh ...)' segment of Fig.(14) associated with the current process/host is processed and a list of all remote triad hosts is established.
2. Persistent Clients connecting the PullDown router to PopUps on all other triads are created and added to the (local) PullDown router.
3. ConnectionManagers are created from RID specifications and stored in a list maintained within the MeshTriad object itself.

Subsequent connection requests are managed explicitly during MeshTriad::Tick(). The origin of the connection request is compared with the stored list of triad host names. Connections from known triad hosts are attached to the PopUp router. All others are sent to the Primary.

4.4 The MeshRouter/RTI-s Process

The generic '`program main()`' for incorporation of the MeshRouter framework within JSAF/RTI-s applications is extremely simple:

1. A few global objects (e.g., RouterConfig) are initialized.
2. The appropriate high level object (MeshRouter or MeshTriad) is created, according to command line specifications and initialized from RID specifications.
3. A main execution loop is started, continually invoking the Tick() method for the instanced object from step 2.

The actual implementation within the existing MeshRouter source code incorporates one significant extension.

An additional, tickable 'Console' object has been created, with an appropriate instance (**RtisParserConsole**) for the JSAF/RTI-s application. The Console object is initialized ate the start of execution, and the notional

```
for(;;) Tick();
```

processing in step 3 above executes the Tick() methods for both the router/triad and the console. This procedure supports easy implementation/access of many of the standard, run-time monitoring capabilities available within RTI-s.

References

[1] T. D. Gottschalk, "The MeshRouter Architecture", Caltech/CACR report, in preparation.

[2] T. D. Gottschalk, "MeshRouter Router Primitives: Messages, Interest, and Interpreters", Caltech/CACR report, in preparation.

[3] A. Ceranowicz, *et al.*, "Reflections on Building the Joint Experimental Federation" (JSAF), Proceedings of the 2002 Interservice/Industry Training, Simulation and Education Conference.

[4] A. Ceranowicz, R. Dehncke and T. Cerri, "Moving Toward a Distributed Continuous Experimentation Environment", Proceedings ot the 2003 Interservice/Industry Training, Simulation and Education Conference.

[5] S. Brunett and T. Gottschalk, "Scalable ModSAF Simulations with More Than 50,000 Vehicles Using Multiple Scalable Parallel Processors", 98S-SIW-181, 1998 Spring Simulation Interoperability Workshop.