The MeshRouter Architecture

T. D. Gottschalk
Center for Advanced Computing Research
California Institute of Technology
Pasadena, CA 91125-7900

April, 2004

Abstract

The MeshRouter architecture supports interest-limited message exchange within large-scale distributed simulations. Broad characherizations of the intended types of underlying simulation applications are presented, and the concept of 'interest-limited communication' is carefully defined. The generality of the MeshRouter system relies on a strict factorization of application-specific components within an overall object-oriented (C++) framework. This factorization can be limited to two specific areas: underlying data primitives ('Message' and 'Interest') and the actual byte exchange mechanisms. This note emphasizes the application-independent components and structure of the overall MeshRouter architecture, and the isolation of application-specific details through required methods for a limited number of virtual objects. The JSAF/RTI-s distributed discrete event simulation is considered as a prototype application of interest, providing illustrations of both concrete object instances and initialization procedures.

Contents

List of Figures

1 Problem Definition

The MeshRouter system provides a framework for separate, possibly disparate processes interacting through a communications system, as suggested by the cartoon in Fig.(1).

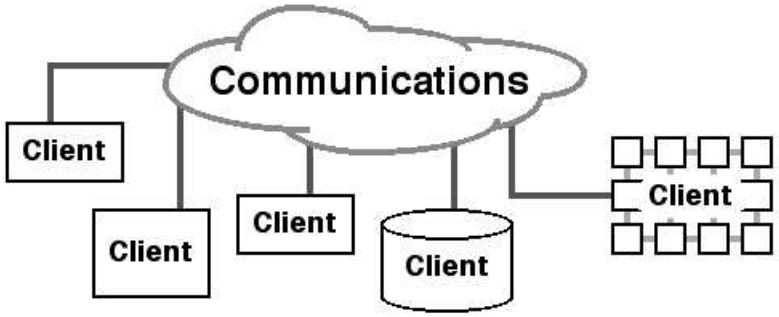

Figure 1: Schematic illustration of a communications-driven distributed simulation

The schematic in Fig.(1) is, of course, hopelessly general. Without some restrictions on the associated communications and processing, any proposed "framework" would be too unwieldy to be useful. Section 1.1 provides a set of additional assumptions and restrictions on the interpretation of Fig.(1), and Section 1.2 introduces the notion of "**Router-Directed, Interest-Limited Communications**" that provides the basis of the MeshRouter architecture defined in the rest of this work and in two companion documents (Refs.[1][2]).

1.1 Applications Of Interest

Within the wide class of applications that might fall under the general rubric of Fig.(1), attention is restricted to those satisfying a number of additional broad characterizations:

1. The overall computational problem is large, requiring large numbers of coordinated processors for its solution. These contributing processors define the 'Clients' of Fig.(1).
2. Individual clients cooperate through explicit message exchanges.
3. Nothing in the overall problem need be homogeneous, including
   (a) Individual client capabilities and/or responsibilities.
   (b) Message sizes, rates, etc.
   (c) Individual Source⇔Destination communications protocols (TCP/IP, proprietary SPP, etc.)
4. The overall computation/simulation is effectively time-stepped, with current actions by a client providing inputs for subsequent actions by other clients.

Added to this list is an essential overall requirement:

Scalability

Larger and larger underlying problems are modeled by (simply) attaching more and more clients to the the communications cloud in Fig.(1).

It should be noted that this scalability requirement itself can imply additional elements of heterogeneity. For example, high-fidelity modeling of some aspect of the overall problem might require a tightly-coupled parallel computational engine interacting as a single client with the rest of the more loosely distributed calculation. (This is the schematic intent of the 'checkerboard thingy' at the right in Fig.(1)).

1.2 Router-Based Communications

As the number of clients in Fig.(2) increases, the overall message traffic increases. In order to achieve a scalable system, there are, in fact, two additional requirements:

Fundamental Scalability Assumptions

1. Individual clients in Fig.(1) need to see only a limited subset of the full data traffic. That is, the "**Interest**" of an individual client is limited.
2. The communications network can be made aware of the interests of the individual clients. This is accomplished through explicit interest declaration messages generated by the client as part of the overall data streams for Fig.(1).

It should be noted that the first assumption is an essential restriction on the underlying problem itself, not merely a simplification for purposes of communications. If the data of interest to any one client are not limited, then the computational capabilities of the associated processor will be overwhelmed as the problem size increases. No clever communications network can achieve scalability if individual client processes need to see all the data, all the time.

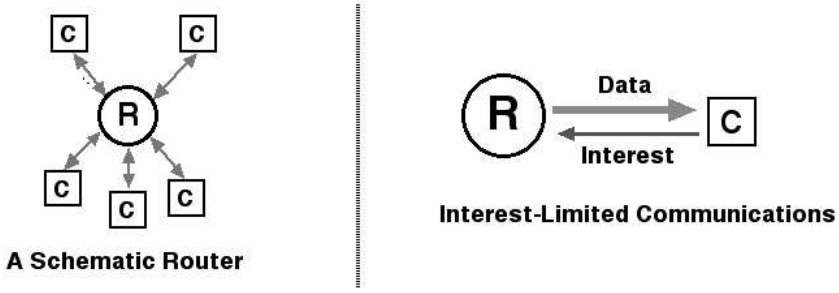

Figure 2: Schematic illustrations of a Router process and interest-limited data exchange between a Router and one of its clients.

Interest-limited communications can be implemented through a system of intermediate 'Routers' as illustrated by the schematic on the left in Fig.(2). There are a number of aspects for these Routers that can remain unspecified for the moment. For example, several routers might be implemented as separate threads on a single processor or the router associated with a specific client might change during an extended execution. The essential elements of the Router model for present purposes are illustrated in the right-hand part of Fig.(2) and can be stated as follows:

Requirements

1. At any point during overall program execution, a particular client process exchanges data with a specific router. (The 'Data' arrow on the right-hand side of Fig.(2) is actually bi-directional.)
2. The client can inform its router of the subset of interest states that are relevant, and the router delivers to the client only those data that match this interest declaration.

3. The interested-limited communications in Fig.(2) could, optionally, be bi-directional. That is, the routers should be capable of informing the client of the interest state of the "rest of the world", in order to suppress generation of irrelevant message traffic from the client. (This optional additional capability assumes, of course, some degree of 'interest awareness' on the part of the client processes.)

The simplest "one router handles everyone" configuration suggested on the left-hand side of Fig.(2) can service only a limited number of clients. As the overall problem size increases, scalability results from networks of interconnected routers. This can be done in a number of ways, with two particular models illustrated by the schematics in Fig.(3). The number of basic units that can be serviced by a single router is limited. In the Tree approach, the overall system grows both horizontally and vertically as the overall problem size increases. In contrast, the scaling model for the Mesh Topology involves only horizontal replication of the basic unit, with scaling achieved through multiple connectivity paths in the upper router layers.

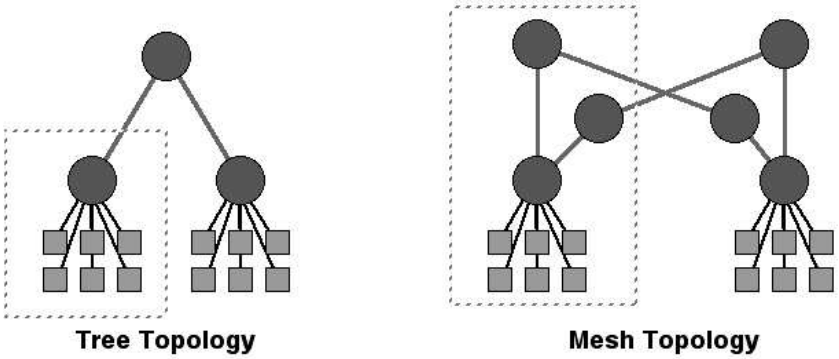

Figure 3: Examples of different communications topologies. The dashed boxes indicate the basic 'scaling units'.

The general MeshRouter architecture described in this note can implement either of the multi-router networks of Fig.(3) as an appropriate limiting case.

1.3 Historical Context: The SAF Simulations

The MeshRouter architecture is an extension of earlier work directed towards scalable implementations of Semi-Automated Forces (SAF)[3]-[5] simulations. Indeed, the Mesh Topology in Fig.(3) provided the framework for SAF implementations on Scalable Parallel Processors (SPPs) in the '*SF-Express*' project[6]. There is value in some comments of this earlier work.

The various SAF simulations (JSAF[5] , OneSAF[4], ModSAF[3], etc) are distributed, discrete event simulations in which most of the client processors for Fig.(1) are responsible for simulating the activities of some limited number of entities (trucks, tanks, etc.). Entities on different simulators interact through explicit message exchange. In the earliest ModSAF applications, with limited total entity count, all messages were simply dumped onto a broadcast network and examined in turn by all processors. As the underlying problem grew, a number of different techniques were explored to deal with the increasing communications burdens, such as the router-based system with explicit interest declarations introduced in the Ref.[6]. The initial, somewhat *ad hoc* implementations of interest-limited communications within the ModSAF software (e.g., multicast IP) have given way to more formalized systems under the High Level Architecture/ Run-Time Interfaces (HLA/RTI) [7]-[9] framework.

There are a number of different ways in which to view this work within the context of ongoing SAF research and development[10].

The SAF application itself is certainly the '*raison d'etre*' for this particular piece of research. Ongoing efforts to apply the SAF program to very large simulations with million-entity 'background' populations

require SPP and/or cluster computing together with scalable, interest-limited communications. Indeed, the MeshRouter software itself is being written and packaged as an optional additional communications component for future releases of the RTI-s software package.

However, the MeshRouter system is designed to implement efficient, interested-limited communications for general applications, not simply variants of the SAF family. The key to any claim of generality is the careful, object-oriented design of the system, with application-specific issues (e.g., "What is interest?") segregated into a daughter instances for a small number of base classes. The emphasis in this document and in its two companion documents[1][2] is on the **factorized general framework**, and the RTI-s application serves primarily to illustrate the model for implementing the required, application-specific specific daughter classes within the overall system.

Moreover, it would be misleading to characterize the MeshRouter system as merely a repackaging of existing or near-term RTI-s capabilities in better software wrappers. Ongoing applications of SAF tools to "urban warfare" environments with very large numbers of background entities are encountering a number of fundamental problems. The cleaner Framework⇔Application factorizations within the MeshRouter can, in fact, enable a number of significant improvements and extensions in JSAF applications of current interest. These include:

1. Wide-area router networks can be matched more closely to the underlying networks of physical communications links. (Performances of tree networks, for example, are clearly sensitive to the geographical location of the root node.)

2. The MeshRouter framework can support carefully tailored, possibly parallel I/O to communication-intensive components of the overall simulation.

An important example in this latter category is the "Simulation of the Location and Attack of Mobile Enemy Missiles" package (SLAMEM) of Toyon Research Corporation [11]. The essential problem here is the modeling of sensor systems that can see large fractions of the playing field. The techniques used in the SLAMEM software to distribute signal processing calculations among client simulators are only a partial solution, and truly scalable simulations of the sensor systems modeled in SLAMEM might well require significant extensions of the standard RTI-s message exchange procedures. It is suggested that the MeshRouter framework provides a good, cleanly-separated starting point for the needed communications generalizations.

2 Problem Factorization

There are two areas of the general, interested-limited communications model of Figs.(1,2) in which the details of the associated client processes are obviously essential:

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.

These are, in fact, the only application-specific constructs in the MeshRouter package. The overall system is implemented using an object-oriented (C++) framework. The application-specific data primitives and communication pipes are then implemented as specific instances of (abstract) base classes that fully specify object interactions.

The data primitives assumed within the MeshRouter are described in detail in Ref.[1], including specific realizations of these objects for the RTI-s application. Raw data-transfers are encapsulated in a small subset of (virtual) methods for the "Pipe" object defined in Ref.[2]. Brief summaries of the data primitives are presented in Section 2.1.

The basic router processes of Fig.(2) can be implemented in terms of a very limited number of objects, including the Pipe class. These building blocks of the MeshRouter are introduced in Section 2.2, with more detailed descriptions and interface specifications given in Section 3.

2.1 Low Level Primitives

The MeshRouter package uses three basic classes to encapsulate the application-specific nature of messages and the interest of client processes in the context of Fig.(1):

class Message: A simple container holding client messages as simple sets of bytes, with additional access to byte count and message type.

class MessageInterpreter: A virtual class with interfaces to extract the size in bytes and type (i.e., '*Interest Declaration*' or '*Other*') from a user message.

class Interest: A virtual class to maintain and manipulate the interest declaration messages associated with a client process.

The Message object is nominally a simple data container, augmented with a memory management system and a related MessageList object for managing collections of individual messages. The MessageInterpreter object is used to fill the Message objects from user data messages, including the particular task of identifying and tagging the interest declarations messages.

The Interest object (a virtual base class) specifies three sets of basic interfaces for interest manipulations:

Translation: Methods to *i*) initialize the Interest object from an interest declaration message and *ii*) create a valid user message describing the state of the Interest object.

Assessment: A simple interest assessment method:

```
bool Interest::HasInterest( const Message& message )
```

Interest-limited data exchange is implemented entirely in terms of this one method.

Manipulation: A limited set of methods to form collective interest states from interest objects associated with individual clients, such as the union of a set of interest objects.

The interface assumptions and requirements for these fundamental objects are defined in more detail in Ref.[1]. Specific implementations of the (virtual) MessageInterpreter and Interest objects for use within the RTI-s framework are also presented.

2.2 High Level Objects

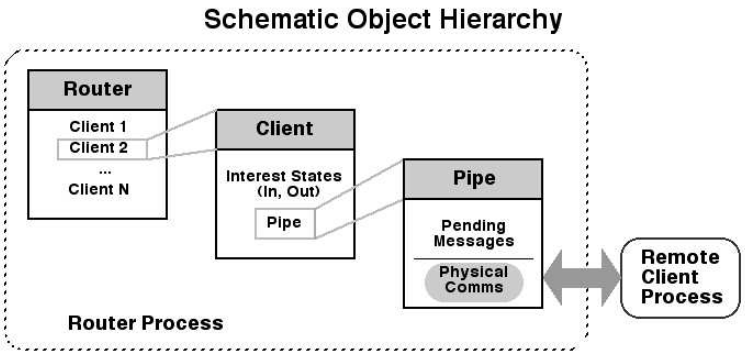

Figure 4: The fundamental MeshRouter objects. The dashed line encompasses the associated schematic router process in the sense for Fig.(2).

The MeshRouter software implements the router process of Fig.(2) using a limited number of software objects to manipulate the fundamental data objects of the previous section. A schematic of the primary objects within the system is shown in Fig.(4).

The **Router** object in Fig.(4) maintains the list of clients interacting with the router process. This list is, in general, dynamic, with clients attaching to and disassociating from the Router during the course of execution. The Router object also manages the overall 'time evolution' of the system, directing specific activities of the associated Client objects.

The **Client** object in Fig.(4) provides the Router's internal representation of the general client processes of Figs.(1,2) and is the primary point of interest-based message filtering. The Client object maintains the interest state of the (remote) client process, and forwards outgoing messages to the client process only if the basic 'HasInterest()' constraint is satisfied. The Client object also maintains a 'Rest of The World' interest state and can transmit this interest to the client process (thus suppressing superfluous message traffic from the remote client).

The **Pipe** object provides the interface between the internal world of Messages (MessageLists) within the MeshRouter and the real world "bits on a wire" communications of the full communications system of Fig.(1). The (virtual) Read(), Write() methods for Pipe objects complete the Framework⇔Application factorization within the MeshRouter implementation. These methods need to deal with a number of issues, such as packet sizing, check summing, etc., *depending on the expectations of the remote client process.* The hierarchical Pipe object model provides a simple, isolated point for dealing with this application-specific requirements without any impact on the rest of the MeshRouter framework.

The next section provides a more careful description of the data content and primary methods for the Router, Client, and Pipe objects. Before doing this, however, a warning/apology on the notation may be in order. In particular, the terms 'client' and 'router' now have dual meanings, referring to both general processes in regards to Fig.(1) and specific software objects. These dual meanings are not entirely compatible. For example, the router process is implemented using a Router object that itself contains Client objects. The client processes in Fig.(1) are simply message sources and sinks and are in a strict sense completely independent of the Client objects used within a router. The distinction between a client process and a Client object should be clear from both context and capitalization/punctuation. (Alternative object naming schemes, such as "AbstractedClientRepresentation" were considered but rejected as unnecessarily pedantic complications.)

3 Object Specifications And Responsibilities

The content and operations of the basic objects in Fig.(4) are most easily understood working from right to left in the schematic, moving from elementary user messages up to the full 'choreographer' responsibilities of the Router object.

The intent in this section is to provide a sufficient overview of the the objects and object interactions in the MeshRouter architecture that claims of "factorized generality" are at least plausibly demonstrated. This means that the following discussions are a bit detailed, focusing on the data content and essential methods of the (C++) objects of the MeshRouter software system.

3.1 Pipe Objects

Pipe objects are responsible for actual user message transmissions to and from the remote client processes of Fig.(1). This requires detailed knowledge (indeed, excruciatingly detailed knowledge) of all lowest level nits of the communications model. The Pipe object is ultimately required to speak precisely the language expected by the remote clients, including any optional dialects such as checksumming, Kerberos, and so on.

Communications generality is achieved through a simple object inheritance model, as shown very schematically in Fig.(5). The overall structure of the implied class hierarchy is as follows:

- All methods for Message exchange between a Pipe and other objects in the MeshRouter package (specifically, the Client objects) are implemented as methods in the Pipe base class.

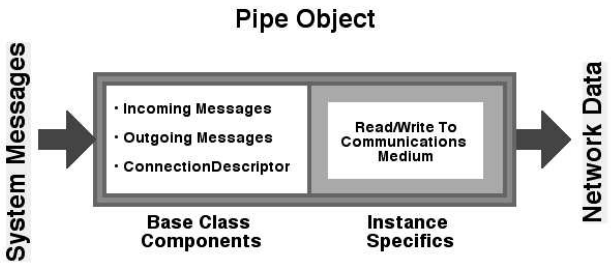

Figure 5: Schematic decomposition of the Pipe object, with base class manipulations of Messages and virtual, instance-specific methods for raw data communications.

- The actual data transmission (read/write) methods are purely virtual methods in the Pipe base class.
- The Client objects are responsible for periodically exercising the data transmission methods, so that data packets actually flows through the system.

3.1.1 Data Content and Fundamental Methods

```
// ----- 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 6: Object data for the Pipe base class.

The essential base class data for a Pipe object are shown in Fig.(6). The two MessageLists provide temporary input and output storage for messages exchanged over the communications link. The status flag is required to be true if and only if the actual communications link is fully functional.

The additional '`ConnectionDescriptor`' object is a small data container that specifies the nature of the communications link. After some experimentation, the object was implemented as a simple string holder with access methods:

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

These strings specify, respectively, the 'other end' of the communications link (typically as a network address) and the type of connection (e.g., "TCP/IP", "UDP", ...). The ConnectionDescriptor is used during aystem initializations and when an open Pipe (i.e., data member *d_status* is false) attempts to establish itself.

The primary methods of the Pipe class are listed in Fig.(7). The first two methods (implemented in the base class itself) simply transfer Messages to and from the temporary MessageList objects contained in the Pipe base class. The required actions for the three virtual methods are as follows.

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

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

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

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

Figure 7: Primary methods for the Pipe object.

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.[1]), 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.

Taken together with the (virtual) data primitives of Section 2.1, the three virtual methods in Fig.(7) complete the basic Framework⇔Application factorization within the overall MeshRouter framework.

3.1.2 Remarks

The generality of the virtual interfaces in Fig.(7) is not to be confused with the simplicity of the method implementations for specific daughter classes. As has been noted above, these methods must produce or process data streams in precisely the format expected by the client processes. Depending on the nature of the client processes in Fig.(1), the requirements on the virtual Pipe methods can be substantial.

The JSAF/RTI-s application provides some insights into the work involved in constructing concrete instances of the Pipe base class. The current RTI-s implementation provides object libraries implementing a number of steps (e.g., message bundling). The actual communications procedure are specified in a set of run-time configuration files (so-called "RID" files, short for "RTI Initialization Data" in the parlance of Ref.[6])

One reasonable approach is to build a concrete '`RtisPipe`' class using the available RTI-s objects and methods[2]. The initial applications of the MeshRouter framework to the JSAF simulation use this approach. There are some mild annoyances, such as working around the assumed discrete-event 'scheduler' that is buried deep within the RTI-s implementation, but the approach is fairly safe in the sense that the Pipe instances constructed in this manner can be kept in synch with ongoing embellishments in the RTI-s libraries.

For Pipes connected to external client processes, it is clearly a good idea to base the implementation upon the communications software of the underlying applications. The situation for Pipes connecting distinct router processes in full communications networks (e.g., Fig.(3)) is a bit less clear. For example, the standard RTI-s implementation absorbs and processes incoming interest declarations while the full Router and Client operations of Sections 3.2 and 3.3 require that these interest declarations flow all the way through the communications Pipe. Again, a short-term solution here is not difficult. A 'RouterToRouterRtisPipe'

could be built using the standard RTI-s objects, excluding the one object in the standard data chain that performs interest-screening and interest updates. At a minimum, efficient Router-to-Router Pipes can be built without buying into all the message memory management framework assumed/required for RTI-s client processes.

One can easily imagine a software world with a huge number of specific daughter classes for the Pipe base class. This is not, in fact, a bad situation. Consider again the implicit overall problem implied by Fig.(1): very large simulations using widely distributed computational assets. Large problems will require large router networks, possibly built up from sub-networks similar to the examples shown in Fig.(3). The communications demands on the individual links of a complicated network will be very different, and an optimal network would undoubtedly use different communications primitives (i.e., different daughter objects of class Pipe) for the different links.

The Pipe interface specification in Fig.(7) provides a clean, well-factorized procedure for implementing the overall MeshRouter system with communications methods that are tailored to the various levels in the overall network configuration.

3.2 Client Objects

The Client objects in Fig.(4) are essentially interest-aware traffic managers for the remote client process at the other end of the communications pipe. This object has three high-level responsibilities:

1. Maintain relevant Interest state declarations for the external client (both the current interest declaration for that client and the interest state describing what the rest of the world wants to hear).
2. Perform the actual interest screening of messages sent to the remote client.
3. Periodically flush the associated Pipe object (i.e., exercise the Read() and Write() methods), so that messages are actually exchanged over the communications links.

Implementation of the Client object is quite straightforward, with object data and basic methods as specified in Sections 3.2.1 and 3.2.2.

3.2.1 Data Content and ClientDescriptors

The basic data content of a Client object is shown in the header fragment in Fig.(8).

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

private:

  Pipe* d_pipe;                       // Actual Messaging Connection
  Interest *d_my_interest;            // Interest From Other End
  Interest *d_world_interest;         // ROW Interest To Other End
  ClientDescriptor d_descriptor;      // Client Characterization
```

Figure 8: Essential object data for class Client.

There is one Client object for each client process of a router, in the schematic sense of Fig.(2). The Client object holds the communications Pipe associated with its client process, and maintains the current interest states associated with that client. ('*d_my_interest*' is the interest of the remote client process, while '*d_world_interest*' contains the interest state of the 'Rest Of World' (ROW), from the perspective of the remote client.)

The schematic Router⇔Client connections in the left-hand part of Fig.(2) are not egalitarian, and the additional 'ClientDescriptor' object in Fig.(8) holds a variety of discrete characterizations of the associated client processes. These are accessed through a number of simple methods:

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

Client manipulations by the Router object of Section 3.3 vary according the values for these flags. In this regard, the overall segregation of a Router's Clients into "Upper" and "Lower" partitions is worth clarifying. Interconnected router networks, such as the Mesh Topology of Fig.(3), can have closed communications loops, and additional rules based on the Upper↔Lower classification are required to prevent infinite data cycling. For example:

> Messages received from a Lower Data Source are sent to all Data Sinks.
>
> Messages received from an Upper Data Source are sent only to Lower Data Sinks.

The '`IsPersistent()`' method determines the Router's actions on encountering a 'broken pipe' - meaning `Pipe::Status()` = *false*: Routers attempt to re-establish connections to persistent clients while simply discarding non-persistent clients with broken pipes.

The values associated with a ClientDescriptor object are all set during initialization, with simple defaults (non-router, non-persistent lower data source/sink) associated with the 'user' client processes of Fig.(1). The simple ClientDescriptor object provides a powerful, flexible generalization of the hard-wired router types ("Primary", "PopUp", "PullDown") in the original *SF-Express* work [6].

3.2.2 Client Methods

Ignoring again a host of object management and utility methods, the essential actions involving Client objects are managed through the limited set of methods listed in the header fragment in Fig.9.

```
// ----- Primary Operations Methods ---------------------------------

  void SendMessagesToRemote( const MessageList& out_messages );
  int GetMessagesFromRemote( MessageList& in_messages );
  bool RemoteInterestHasChanged() const;
  const Interest& GetClientInterest();
  void SetWorldInterest( const Interest& w_interest );
```

Figure 9: Fundamental methods for class Client.

`SendMessagesToRemote()` and `GetMessagesFromRemote()` are the primary drivers for interest-limited message exchange. The `GetMessagesFromRemote()` method retrieves incoming Messages from the Pipe and appends non-interest messages to the specified MessageList (`in_messages`). Interest declarations are extracted from the Pipe's message set and used to update the internal representation (*d_my_interest*) of the remote client's interest. The current state of *d_my_interest* is used to select the subset of the nominal outgoing message list in `SendMessagesToRemote( const MessageList& out_messages )` that actually go out the Pipe object.

Note that the Client object manages all bookkeeping related to the remote client's interest declarations. Remote interest declarations are identified and extracted from the MessageLists retrieved

from the Pipe and used to restrict outgoing data messages. The `RemoteInterestHasChanged()` and `GetClientInterest()` methods in Fig.(9) provide access to this interest state. The 'Rest Of World' (ROW) interest (*d_world_interest*) in Fig.(8) is a different matter. This interest state can determined only by the Router that manages the Client object, and the `SetWorldInterest()` provides the means for the Router to update ROW interest. The Client object compares the specified interest (*w_interest*) with its cached value (*d_world_interest*). If interest has changed, *d_world_interest* is updated and an outgoing interest declaration message is generated and sent to the Pipe.

3.3 Router Objects

The Router objects in Fig.(4) manage the overall operations of the MeshRouter system, with two high-level responsibilities:

1. Client Management
   (a) Listening for new connection requests.
   (b) Identifying and removing 'dead clients'.
2. Data Flow Management.

These tasks are actually done by the mid-level objects within the MeshRouter design, and the essential data content of the Router object is, accordingly, rather limited, as shown in Fig.(10).

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

private:
  Interest *d_my_lower_interest;      // Union, Lower Client Interest
  Interest *d_my_upper_interest;      // Union, Upper Client Interest

  ClientList d_clients;               // List Of Associated Clients

  std::vector<ConnectionManager*>
    d_listeners;                      // Connection Listeners
```

Figure 10: Essential object data for class Router.

The two Interest objects in Fig.(10) are storage areas for the collective interest states of the Router's upper and lower Clients, and the 'ClientList' is a straightforward list of all currently defined clients for the router. The basic Data Flow Management operations of the router are accomplished by straightforward manipulations of these objects, as is described in Section 3.3.2.

The 'ConnectionManager' in Fig.(10) is a new object/concept, and is involved in the Client Management responsibilities of the Router, as described in Section 3.3.1.

3.3.1 ConnectionManagers and Dynamic Clients

Robust operation of the schematic in Fig.(1) requires dynamic client management - meaning simply the ability to add and/or remove individual client processes during execution. The `Establish()` methods for Pipe and Client objects in Figs.(7,9) are part of this capability. System configuration files will typically specify the destination of a communications link (i.e., the `GetPeer()` content of the `ConnectionDescriptor` object in Fig.(6)), and Client objects are responsible for opening/establishing these communications links.

The Router objects in Fig.(4) need to listen for and respond to these Connection requests. This is done through a set of `ConnectionManager` objects, with individual ConnectionManagers listening to connections for individual Pipe classes (`TCPPipeConnectionManager, MemoryPipeConnectionManager,` ...).

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

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

Figure 11: Primary methods for the ConnectionManager class.

The `ConnectionManager` is implemented as a purely virtual base class, with interface specifications defined by the methods listed in Fig.(11). The primary operational method is `Listen()` in which the ConnectionManager responds to new connection attempts from remote processes and returns a (fully initialized) Pipe object whenever a new request has been detected and processed. The `Initialize()` and `Shutdown()` methods provide hooks for overall set up and termination of the (protocol-specific) manager.

The `ConnectionManager` class is a final piece in the overall Framework⇔Application factorization that has been stressed throughout this document. The steps needed to implement a new communications procedure into the overall MeshRouter framework are limited to:

1. Design and implement the appropriate derived `Pipe` class, including the essential `Read()`, `Write()` and `Establish()` methods for the virtual interfaces in Fig.(7).
2. Design and implement the associated `ConnectionManager` derived class, enabling Router objects to respond to `Establish()` requests.

3.3.2 Essential Router Methods

Leaving initialization and configuration issues aside for the moment, Router operations are driven by the three basic methods listed in Fig.(12).

```
// ----- Operational Components -----------------------------------

    int ListenForConnections();
    void DistributeData();
    void UpdateInterestStates();
```

Figure 12: Fundamental methods for class Router.

The `ListenForConnections()` method loops over the `ConnectionManagers` in *d_listeners*, invoking the `Listen()` method for the various types of connections. A non-null Pipe* return for the method in Fig.(11) indicates a successfully processed connection request. A new Client object is then created and added to the Router's list of active clients.

`DistributeData()` is little more than a double loop over the active clients, requesting new messages from each client using the `GetMessagesFromRemote()` method in Fig.(9) and then sending these messages to (nearly) all other clients using `SendMessagesToRemote()`. The only restrictions on these message retransmissions are those implied by the `ClientDescriptor` flags, as discussed in Section 3.2.1. Due to the efficient memory management layers for `Message` objects described in Ref.[1], there is very little overhead associated with transient 'Message duplications' during these nested data redistribution loops.

The Message objects retrieved and redistributed during `DistributeData()` exclude interest declarations. As was discussed in Section 3.2.2, the Client objects extract and process the incoming interest declarations from the client processes, returning only the non-interest messages. The Client processing during `DistributeData()` will generally result in modified client interest states, and the `UpdateInterestStates()`

method in Fig.(12) ensures that the total interest picture for all clients is complete and current. This method first recomputes the total upper and lower interest states (*d_my_lower_interest* and *d_my_upper_interest*) in Fig.(10) and then loops over all data source clients, recomputing the 'Rest Of World' interest state relevant for that client, and updating this interest declaration using the `SetWorldInterest()` method of Fig.(9).

4 Operations and Initializations

As with many object-oriented systems, a generic program '`main()`' for a router process involves not much more than two conceptually simple steps:

1. Create an appropriate Router object and do a bit of initialization stuff.
2. Repeat a standard operations cycle until directed to end overall execution.

In most cases, this standard structure is accompanied by a number of application-specific embellishments, such as interrupt handlers. This is the anticipated model for realizing the generic router process of Fig.(1), and simple main routines exist for the JSAF/RTI-s application.

This section explores some broad generalities for a generic 'program main()', as might be used to build a concrete router process for Figs.(2,3) from the fundamental objects available within the MeshRouter software. Ongoing operations are driven by the `Router::Tick()` method described in Section 4.1 and the Client management procedures in Section 4.2. Initialization of a Router process from configuration data is clearly application-specific and not, strictly speaking, a well-defined task within the MeshRouter framework. The discussion of Router initialization in Section 4.3 is based on the JSAF/RTI-s application, but is sufficiently high-level to illustrate the actions required for Router initializations for other applications.

4.1 Time Management and 'Tick()'

The most convenient picture for MeshRouter operations is that of a (loosely) time-stepped simulation, and the inner loop for ongoing execution with an instanced/initialized Router ('`my_router`') typically contains a code fragment like that shown in Fig.(13).

```
while(ContinueRunning)
{
  double timenow = MeshUtilities::WallClockTime();
  my_router.Tick(timenow);
}
```

Figure 13: Typical Router operations code fragment.

The `timenow` variable is assumed to be some measure of the current time in seconds. The reason for introducing a notion of time at all is efficiency:

1. Some of the object methods described above (in particular, `Pipe::Read()` and `Pipe::Write()` are likely to be slow operations.
2. The very simplistic timing control in Fig.(13) is adequate for controlling the frequency at which these I/O methods are invoked.

The `Client` objects from Section 3.2 contain additional data elements specifying minimum time delays between successive low-level Read() and Write() attempts over the Pipe, and time-stepping directed by the

Router is sufficient to ensure that the actual communications primitives are exercised often enough, but not excessively.

The actions taken by `Router::Tick( timenow )` are straightforward:

Router::Tick() Activities

1. Advance the 'operations' time of all the Clients to the specified value.
2. Possible management (additions or deletions) to the list of associated Client objects.
3. Message movement among the Clients.
4. Interest state updates.

The last two tasks are done using the `DistributeData()` and `UpdateInterestStates()` methods already described in Section 3.3.2. The client management in the second item is a bit more complex, and is described in the next subsection.

4.2 Client Management

The MeshRouter supports dynamic client management - meaning simply that the client processes in Fig.(1) are allowed to appear, disappear, reappear ... during the course of execution. To support dynamic management capabilities, the Clients associated with the Router object in Fig.(3) are divided into two categories:

Persistent Clients: Clients that are known and fully specified from the system configuration data. This means, in particular, that all aspects of the underlying connection (the specific `Pipe` class and the contents of the `ConnectionDescriptor` of Section 3.1.1) are fully specified.

Dynamic Clients: Clients that are not specified in the Router's configuration data but instead appear as explicit connection requests to Router during the course of execution.

Given this distinction between *Persistent* and *Dynamic* Clients, the actions taken in the second task of the **Router::Tick() Activities** list from the previous section are straightforward:

1. The Router first loops through its list of Clients, looking for Clients with a broken Pipe. If the Client associated with a broken Pipe is not Persistent, that Client is immediately deleted from the client list. Connection attempts for broken Persistent Clients are done, as needed, within the `DistributeData()` method of Fig.(12).

2. The Router then loops through its collection of `ConnectionManagers` (data element *d_listeners* in Fig.(10), invoking the `Listen()` method of Fig.(12). A non-null Pipe return indicates a successful new connection. A Dynamic Client is created for the Pipe and added to the client list.

The Router is responsible for initiating and maintaining operational connections to its persistent Clients. Dynamic clients are treated as transients, serviced while they are around but forgotten when/if they disappear.

The `Pipe::Establish()` and `ConnectionManager::Listen()` methods are generally slow operations involving network communications. For purposes of overall efficiency, Router object has a data element specifying minimum times between consecutive invocations of these methods.

4.3 Initialization and Configuration Files

From the perspective of the MeshRouter system itself, there are two essential tasks that must be done prior to starting the generic execution loop of Fig.(13).

1. The set of Persistent Clients must be created, including fully-specified Pipe objects for the underlying communications links and the discrete ConnectionDescriptor flag values for the Client objects.
2. The set of active ConnectionManagers in Fig.(10) needs to be populated.

For simple applications, this could be done with a limited number of hard-wired, specific main routines. The more general (and palatable) solution requires a single 'program main()' for all router processes, with initialization data retrieved from system configuration data files.

It is difficult to imagine a truly general solution to the initialization problem for general applications in the sense of Fig.(1). Instead, he approach to file-driven initialization in the current MeshRouter software is a perhaps best described as an experimental patchwork based in part on the initialization requirements of the JSAF/RTI-s application.

The current MeshRouter initialization system is built upon two additional objects:

class RouterConfig: A data container for holding initialization file data.

class RouterConnectivity: An interpreter object that can interrogate the RouterConfig data, and return values for a number of initialization-specific queries.

It should be stressed that these are not 'objects' in any nice, hierarchical C++ sense. Individual instances of these objects need to be designed, implemented, and linked for specific applications of the MeshRouter system.

The `RouterConfig` object is intended to be a container for initialization data. For the JSAF/RTI-s application, it simply holds a copy of the of the RID. Object initialization and access methods are all application specific. The RouterConfig object is made available to the ConnectionDescriptor and ConnectionManager objects within the MeshRouter so that application-specific aspects of the communications primitives are available for specific Pipe creations.

While all aspects of Pipe initialization can be safely hidden in an otherwise undefined object, the initialization of the Persistent Clients for a Router requires an interface for extracting specific information from the RouterConfig object. These required interfaces are provided by the RouterConnectivity object. The RouterConnectivity object has only an explicit constructor:

```
RouterConnectivity::RouterConnectivity( RouterConfig& conf )
```

and a number of query methods to extract information.

The 'appropriate queries' needed from the `RouterConnectivity` object are subject to ongoing refinements, but need to include items such as:

1. What kind of Router am I?
2. What kinds of ConnectionManagers do I need?
3. What are my Persistent clients?
4. What are my Persistent clients of a specific kind?

The concept of 'kind' in items 1 and 4 is a bit vague, but generically refers to selecting subsets of the clients for different values of the ClientDescriptor characterizations from Section 3.2.1 (e.g., `IsDataSource()`).

The RouterConfig and RouterConnectivity objects currently used for the JSAF/RTI-s application are adequate for initializing either of the two overall communications topologies shown in Fig.(3) (assuming a straightforward addition to the standard RID syntax to define mesh connectivity). For the tree case, this is

fairly straightforward, as all Routers behave the same and each Router has, at most, one Persistent Client - the Router above it in the network. Initializations within the mesh topology are clearly more complex, with Routers having several Persistent Clients of different types. The existing initialization scheme is flexible and reasonably simple. Moreover, the JSAF/RTI-s example provides a simple case study for the similar objects that would be required for other applications of the MeshRouter system.

References

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

[2] B. Barrett and T. D. Gottschalk, "Pipes and Connections", Caltech/CACR report, in preparation.

[3] A. Ceranowicz *et al.*, "Mododular Semi-Automated Forces (ModSAF)", Proceedings ot the 26th Winter Simulation Workshop (1994) 755-761.

[4] R. Wittman and C. Harrison, "OneSAF: A Product Line Approach to Simulation Development", MITRE Corporation document, 2001.

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

[6] 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.

[7] D. Van Hook and J. Calvin, "Data Distribution Management in RTI-1.3", Simulation Interoperability Workshop, 98S-SIW-206, March 1998.

[8] J. Dahman, R. Fujimoto and R. Weatherly, "The Department of Defense High Level Architecture", Proceedings of the 1997 Winter Simulation Conference.

[9] S. Rak, M. Salisbury, and R. MacDonald, "HLA/RTI Data Distribution Management in the Synthetic Theater of War", 97F-SIW-119, 1997 Fall Simulation Interoperability Workshop.

[10] 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.

[11] See, e.g., the Toyon WWW site, http://www.toyon.com.