MeshRouter Primitives:
Messages, Interest, and Interpreters

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

April, 2004

Abstract

The MeshRouter architecture provides a general framework for interest-limited message exchanges among client processes. There are two general areas in which the MeshRouter implementation depends on specifics of the associated clients: *i*) the lowest level inter-processor communications model and *ii*) the nature/content of exchanged messages and "interest". This note describes the hierarchical object design (C++ sense) used to implement the basic Message and Interest objects of the MeshRouter. Interfaces are defined in terms of abstract base classes, and specific inherited objects appropriate for the RTI-s/JSAF application are presented as concrete examples. The MeshRouter system includes a substantial memory management component for efficient use of the basic Message objects. This system is summarized and plausible near-term extensions are noted.

Contents

1 Context **1**

2 Interest-Managed Communications **2**
2.1 MessageInterpreter Objects 3
2.1.1 A Plausible Type Extension 3
2.2 Message Objects 4
2.3 Interest Objects 4
2.3.1 Interest and Message Interactions 4
2.3.2 Interest Manipulations 5
2.4 Summary: Application Isolation 5

3 Specific Illustration: RTI-s/JSAF **6**
3.1 The RtisMessageInterpreter Object 6
3.2 The RtisInterest Object 6

4 Message Object Management **7**
4.1 Message Management Objects 7
4.1.1 MessageObj: The 'Real' Message Object 7

4.1.2 The BasicMemoryManager Object 8
4.1.3 The Message Object: A Surrogate 8
4.2 Plausible Extensions 9

List of Figures

1 Schematic illustration of the basic Router system. External clients (C) exchange messages through a network of intermediate routers (R). 1
2 Interface specifications for (virtual) class **MessageInterpreter**. 3
3 Message manipulation interfaces for (virtual) class **Interest**. 4
4 Interest manipulation interfaces for (virtual) class **Interest**. 5
5 The standard message header within RTI-s 6
6 Essential contents of class **MessageObj** 7
7 Essential methods of class **BasicMemoryManager** 8

References

[1] T. D. Gottschalk, ”The MeshRouter Architecture”, 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.*, “Reflections on Building. the Joint Experimental Federation” (JSAF), Proceedings of the 2002 Interservice/Industry Training, Simulation and Education Conference.

[4] 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.

[5] See, for example, *The C++ Standard Template Library*, P. Plauger *et al.*, Prentice Hall, 2001.

1 Context

The MeshRouter system provides efficient, interest-limited delivery of messages among a number of client processes. Message delivery is accomplished by way of a number of intermediate 'Router' objects. Details of the client processes and the exchanged messages are largely irrelevant (although the qualifying term "interest-limited" is critical and needs to be defined with some care). The simplest router system involves a number of clients exchanging messages among themselves by way of a single intermediate router, as shown in the left hand side of Fig.(1). More complex (and typical) systems will involve networks of interconnected routers, as suggested by the right hand side of Fig.(1).

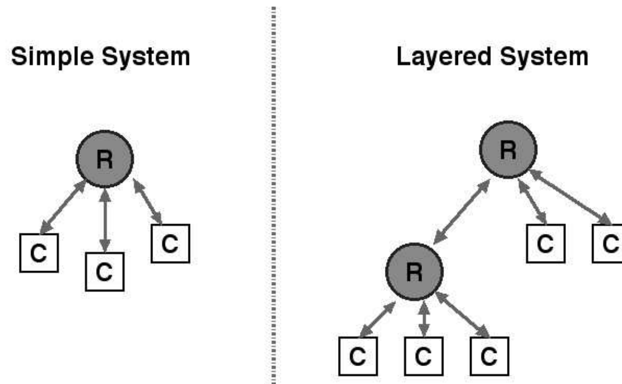

Figure 1: Schematic illustration of the basic Router system. External clients (C) exchange messages through a network of intermediate routers (R).

The 'application indifference' of the MeshRouter system is achieved through a carefully controlled, object-oriented (C++) design. The important, high-level objects within this architecture include:

Routers: Overall managers for interested-restricted data flow. These objects (indicated by the circles in Fig.(1) also maintain and manage lists of attached clients, supporting dynamic additions and deletions.

Clients: Abstracted internal representations of the remote client processes (the squares in Fig.(1) represent the remote clients, not the internal Client objects). To a large extent, Router objects are simply managed collections of associated Client objects.

Pipes: Objects to manage actual data flow between the internal Client objects and the external client processes. These are represented by the shaded arrows in Fig.(1).

These fundamental objects of the MeshRouter system are described in a companion document [1]. For present purposes, it is sufficient to note that the Router and Client implementations are completely independent of specifics of the underlying problem. Pipe objects encapsulate the application-specific nature of the communications, enabling, as it were, the MeshRouter network to speak in the language understood by the client processes. This is done through a simple C++ object hierarchy, as described in Ref.[2]. The Pipe base class implements all interactions with Client objects. Instance-specific Read() and Write() methods for daughter classes (e.g., *TcpPipe, MPIPipe, CarrierPigeonPipe*, etc.) perform the actual raw data exchanges indicated by the arrows in Fig.(1).

The viability of the general data-exchange paradigm of Fig.(1) ultimately rests on the establishment of a general framework for the two data primitives:

‘***Message***’ and ‘***Interest***’.

The rest of this note describes the assumptions and implementations for these basic objects within the MeshRouter architecture.

Section 2 contains a more precise definition of ‘*interest managed communications*’ and introduces the three associated fundamental objects:

Message: Representation of a generic user datum within the MeshRouter framework.

MessageInterpreter: An object providing access to minimal essential parameters (e.g., message size in bytes) from an individual user datum.

Interest: An object quantifying the subset of all possible data that should be delivered to a particular client.

The Message class is a simple, ‘minimally smart’ data container. The MessageInterpreter and Interest classes, on the other hand, are virtual, defining only the interfaces required/assumed by all other objects in the MeshRouter architecture. Specific instances of these MessageInterpreter and Interest objects for use with the RTI-s/JSAF application [3] are presented in Section 3. This provides a particular illustration of the (only) work required in adapting the general MeshRouter data primitive to a specific data-exchange application.

In typical applications, an individual message entering a particular Router in Fig.(1) will be of interest to many of the attached clients. In order to avoid unnecessary duplications of actual user data, the Message objects within the MeshRouter are implemented as surrogates (“smart pointers”) so that an individual user message can exist in multiple internal queues with minimal overhead. This object/memory management system is described briefly in Section 4.

2 Interest-Managed Communications

Simple message-exchange applications are prone to a serious communications problem. As the number of processors increases, the network message traffic increases. This model will eventually fail unless ‘unimportant’ messages are kept away from the incoming queues of the individual processors.

The standard high-level solution to this problem (as adapted, e.g., in HLA/RTI[4]) is fairly straightforward:

1. Individual messages within the distributed system can be characterized by auxiliary tags (‘interest states’).
2. The client processes (e.g., as in Fig.(1)) can determine subsets of all possible interest states that are relevant for its (local) part of the full simulation and/or calculation.
3. Client processes can announce their collective interest by means of explicit ‘interest declaration’ messages.
4. The communications system (e.g., the full MeshRouter system) is responsible for delivering to a client only those messages that match the client’s interest declaration.

Perhaps the simplest example of interest-limited communication is a discrete event simulation evolving on a large playing field. A client that is managing entities in some restricted portion of the playing field has no use for information about external objects that are on different, sufficiently distant portions of the field.

The MessageInterpreter, Message, and Interest objects defined in the following subsections provide a simple but general framework for interest-managed communications with minimal assumptions on the nature of the underlying user data and/or client processes.

2.1 MessageInterpreter Objects

From the overall perspective of Fig.(1) a message is little more than a collection of bytes. In order to manage these data, two characteristics are needed from each message:

1. **Size:** The number of bytes in the message.
2. **Type:** An indicator differentiating interest declarations from generic data.

The **MessageInterpreter** class defines the interface for extracting this basic information from a user message. There are two essential aspects to this object, as shown in the header fragment in Fig.(2).

```
enum Type {                         // Basic Type Enumeration
  EMPTY,                            // An Empty/Null Message
  DATA,                             // Non-Interest User Data
  SUBSCRIBE,                        // Interest Subscription
  INVALID };                        // Invalid/Uninterpretable

// ----- Interpret Interface Specification -------------------------

virtual bool Interpret(
  const char* user_data,            // I: A User Message
  MessageInterpreter::Type& type,   // O: Message Type
  uint32& bytes ) const = 0;        // O: Message Size (In Bytes)
```

Figure 2: Interface specifications for (virtual) class **MessageInterpreter**.

The fundamental method MessageInterpreter::Interpret() clearly requires knowledge of the content of user messages. Such knowledge is application-specific, and a concrete MessageInterpreter instance is needed in any actual execution of the MeshRouter software. The RtisMessageInterpreter object used for the RTI-s/JSAF application provides a simple prototype for a real MessageInterpreter instance, as is described in Section 3.

2.1.1 A Plausible Type Extension

It should be noted that the interface specification in Fig.(2) is based upon a fundamental assumption on the overall data flow within the communications framework of Fig.(1):

Implicit Assumption

> Interest declarations from the client processes are buried within the otherwise homogeneous stream of messages flowing into a router from a client.

In order to perform its tasks, the MeshRouter needs to extract and act upon the interest declaration messages while essentially ignoring the content of all other messages. The MessageInterpreter object is the sole point of message segregation within the MeshRouter architecture, with the essential task of identifying the interest declaration messages that change the internal states of Router and Client objects.

It is easy to imagine another class of messages that might warrant non-passive action by the MeshRouter, for example, some sort of 'MessageInterpreter::CONTROL' messages. These might be useful in directing the MeshRouter system itself to perform management tasks, such as checkpointing or data statistics output. The MessageInterpreter interface in Fig.(2) could easily be extended to recognize these options with, e.g., some set of callback methods in place to process a recognized control messages.

The problem with this extension comes with the generation of the control messages. All messages within the MeshRouter architecture are generated by the client processes in Fig.(1), and implementation of any 'control layer' messages would impose additional assumptions and requirements on these clients. The implementation procedure would be straightforward (e.g., an abstract 'ControlObject' base class, an 'RtisControlObject' specific instance, ...). For simplicity, however, this additional message class is not implemented in the initial MeshRouter system.

2.2 Message Objects

A simple **Message** class is used to hold individual user messages within the MeshRouter architecture. The essential methods for this object are very straightforward, consisting of:

1. Initialization from a string of user data and an explicit or implicit MessageInterpreter.
2. Access to the byte count and type (i.e., MessageInterpreter::Type) of the associated user message.
3. Read-only access to the actual user data.

Unlike the (virtual) MessageInterpreter of the previous section or the Interest object described in Section 2.3, the Message object is a concrete class - a single fixed form for all user messages within the system.

While the definitions and interfaces for the Message object are simple, the actual implementation is somewhat sophisticated. For example, surrogate objects and "smart pointers" are used to allow multiple instances of a single user message without duplicating the actual content of the user data. These details are described in Section 4.

2.3 Interest Objects

The final, and most complex fundamental object in the MeshRouter framework is the **Interest** class. Since the notion of "interesting message" is clearly application-specific, the actual Interest object defines only an abstract base class, specifying the required methods and interfaces for actual implementations (such as the RtisInterest class described in Section 3).

It is useful to divide the basic methods of the Interest class into two broad categories, dealing separately with *i*) interactions between 'Interest' and 'Message' objects, and *ii*) basic manipulations involving multiple Interest objects.

2.3.1 Interest and Message Interactions

There are three essential interactions between "Interest" and "Message" objects, as specified by the header fragment for class Interest shown in Fig.(3).

```
virtual bool SetFromMessage(
  const Message& message,           // I: Interest Declaration Message
  bool& changed ) = 0;              // O: Interest Changed Flag

virtual Message WriteToSubscription() const = 0;

virtual bool HasInterest( const Message& message ) const = 0;
```

Figure 3: Message manipulation interfaces for (virtual) class **Interest**.

The first method sets the internal data content of the Interest object according to the contents of the specified Message. This method should fail, returning false, if the specified Message is not of type MessageType::SUBSCRIBE. The 'changed' flag indicates whether the newly set interest state differs from that prior to invoking SetFromMessage().

The second method creates a Message describing the current contents of the Interest object. In practice, this means translating the internal interest state into an appropriate set of bytes (i.e., the 'user data' representation) and then simply creating a standard Message object from this string of bytes.

The final method is the essential decision method for interest-managed communication, returning true if and only if the message in question satisfies the "Is it interesting?" criteria of the Interest object. In terms of the communications schematic of Fig.(1), the only Messages flowing along the shaded arrows are those satisfying the HasInterest() test of the Interest objects associated with the links.

2.3.2 Interest Manipulations

The router objects in the Fig.(1) schematic maintain a number of collective interest states associated with the individual interests of the attached clients. For example, the lower router in the right-hand part of Fig.(1) has a collective Interest defined by the union of the individual Interest states of the three lower clients.

```
virtual bool IsSameInterest( const Interest& other ) const = 0;
virtual void OrWithOther( const Interest& other ) = 0;
virtual void SetToOther( const Interest& other ) = 0;
virtual void Clear() = 0;
```

Figure 4: Interest manipulation interfaces for (virtual) class **Interest**.

The required collective interest manipulations within the full MeshRouter system are done using only the four basic methods listed in Fig.(4). The requirements for these four methods are straightforward:

1. Test for equality of the interest state of this Interest object and another specified object.
2. Increment the interest state through an inclusive logical 'or' with the interest state of another object.
3. Replace the interest state of this object with that of another object.
4. Clear the current interest state ('interested in nothing').

2.4 Summary: Application Isolation

The method specifications and requirements listed in Figs.(2-4) provide a complete 'factorization' of interest management within the MeshRouter architecture. Given the appropriate, application-specific implementations of the MessageInterpreter and Interest base classes (e.g., the RtisMessageInterpreter and RtisInterest objects described in the next section), the various message segregation and routing activities for the fundamental Router and Client objects of Fig.(1) are completely independent of application-specific details.

The sole remaining application-specific parts of the full MeshRouter system deal with raw data exchanges along the shaded arrows of Fig.(1) (The 'bits on a wire' part, as it were.) This Framework⇔Application factorization is accomplished in the (low level) Read() and Write() methods of the basic Pipe objects, as is discussed in Ref.[2].

3 Specific Illustration: RTI-s/JSAF

The RTI-s/JSAF application of Ref.[3] provided both the impetus and first concrete application of the MeshRouter system. Implementations of the appropriate instances of the basic MessageInterpreter and Interest objects for this application are straightforward, using standard objects within the current RTI-s software.

3.1 The RtisMessageInterpreter Object

Messages within the RTI-s framework are simply buckets of bytes, each beginning with a standard header:

```
struct message_header
{
    uint8  node_id;          // Which datanode sent/receives this message
    uint8  client_id;        // Which client sent/receives this message
    uint8  priority;         // The QOS priority of this message
    uint8  unused;           // Available for future use
    uint32 stream;           // The stream number
    uint32 size;             // The size in bytes of this message
    uint32 federation;       // The id of this federation (0 == all feds)
};
```

Figure 5: The standard message header within RTI-s

(The data types 'uint8' and 'uint32' are typedefs for unsigned 8-bit and 32-bit integers). Given this standard header, the actions of RtisMessageInterpreter::Interpret() are straightforward:

1. Cast the start of the user data area to a message_header.
2. Deal with byte ordering issues (all messages within RTI-s are passed in network byte ordered form).
3. Extract the message size and message type from the reordered header.

The 'size' entry in Fig.(5) is the required message size for the Fig.(2) interface, and the MessageInterpreter::Type value is determined by the 'node-id' value. Interest declarations have

$$\text{node_id} = \text{SUBSCR_NODE_ID},$$

where the (constant) SUBSCR_NODE_ID is defined in a standard RTI-s header file.

3.2 The RtisInterest Object

The overall interest management scheme in RTI-s is based on a segregation of all possible messages into a (large) number of discrete categories called 'streams' (this is, in fact, the 'stream' component of the standard message header in Fig.(5)). Some streams might indicate a particular geographic subregion of the full playing field, others might indicate radio messages of some frequency, etc.

The interest state of a client is conceptually a single bit vector, with the ith bit on if and only if the client is interested in the ith stream. In practice, this is implemented using a more sophisticated sparse bit vector object (class **sparse_bit_vector**), as most clients have interest in only a small portion of the full interest space.

The RtisInterest object is easily implemented using the standard RTI-s sparse_bit_vector object. The various interest manipulation methods of Section2.3.2 are all done using available sparse_bit_vector methods. The "HasInterest()" method of Fig.(3) requires nothing more than examining the indicated bit of the underlying sparse_bit_vector.

The remaining SetFromMessage() and WriteToSubscription() methods of Fig.(3) are done using Read() and Write() methods of the sparse_bit_vector class. This required some care. In particular, these methods must deal with message-headers and padding, as expected and used by the remote clients. Getting all this right did take a bit of time (in particular, the 'extra' message size part was evident only on working through the bowels of the relevant RTI-s object implementation).

Any significant changes in the RTI-s interest management framework will, of course, require modifications of the associated RtisInterest and/or RtisMessageHeader objects. For example, the simple grid scheme for position-based interest is not optimal for geographically dispersed messages, such as descriptions of large smoke clouds. Generalizing RTI-s interest representation to deal better with geographically distributed entities might take some work. Generalizing the RtisInterest object to accommodate these extensions would be straightforward. Most importantly, these would be the only modifications to the software needed to keep the MeshRouter package 'in synch' with the application.

4 Message Object Management

The MeshRouter system is intended to move large numbers of messages among large numbers of clients. In order to minimize costly creations and duplications of the underlying data objects, the MeshRouter uses a system of memory-managed surrogates. The essential components of this system are described in Section 4.1 and some plausible extensions of the current system are noted in Section 4.2.

4.1 Message Management Objects

The current Message memory management system for the MeshRouter is built from a number of fairly standard components, as defined and described in the following subsections.

4.1.1 MessageObj: The 'Real' Message Object

```
    bool Create(
      const char* user_data,            // I: User Message Contents
      const MessageInterpreter& mint ); // I: Message Interpreter

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

  private:
    uint32 d_bytes;                     // Message Size In Bytes
    MessageInterpreter::Type d_type;    // Message Type
    uint32 d_allocated_bytes;           // Allocation Block Size
    char *d_data;                       // Message Contents
```

Figure 6: Essential contents of class **MessageObj**

An individual user message is represented by a **MessageObj** object. The essential methods and data content for this class are shown in Fig.(6). Note that creation of a MessageObj involves both the raw user data and the appropriate MessageInterpreter object from Section 2.1. The Size and Type values from the MessageInterpreter are stored within the MessageObj. The full MessageObj specification provides a

number of creation variants (e.g., constructors for specified user data) as well as const access methods to the object's data.

The actual user message contents (the '*char *d_data*' part of Fig.(6)) is an independent copy of the user data passed to the Create() method in Fig.(6). This copying is necessary in order to ensure that Message manipulations within the MeshRouter are completely divorced from message retrieval/transmittal implementations deep within the specific Pipe implementations of Ref.([2]).

Constant creation and destruction of the message buffers for Fig.(6) would be extremely inefficient. The next component of the Message Management system addresses this problem.

4.1.2 The BasicMemoryManager Object

The MeshRouter software uses a number of simple stacks to maintain/recycle blocks of memory for a number of specified block sizes. The number of stacks and the block sizes for the stacks are effectively run-time parameters of the system. Access to this managed collection of memory is through two methods of the BasicMemoryManager class, as indicated in Fig.(7).

```
char* GetMemory(
  int bytes_needed,               // I: Requested MemorySize
  int& bytes_received );          // O: Actual Memory Block Size

void ReturnMemory(
  char* user_memory,              // I: Memory From GetMemory()
  int bytes_received );           // I: Bytes Size From GetMemory()
```

Figure 7: Essential methods of class **BasicMemoryManager**

These methods are used to create and destroy the actual data areas (char* d_data) contained within the MemoryObj object of Section 4.1.1. The 'bytes_received' output from GetMemory() is stored in the 'd_allocated_bytes' field of Fig.(6) and then passed back through ReturnMemory() when the MessageOb is eventually destroyed.

In addition to its collection of fixed-size memory stacks, the BasicMemoryManager maintains a list of 'overflow' blocks, created as needed when the bytes_needed input to GetMemory() exceeds the largest managed block size.

4.1.3 The Message Object: A Surrogate

The Message objects used within the MeshRouter are standard 'smart pointers' to an associated MessageObj. The actual data content of a Message object is almost trivial:

1. A (naked) pointer to an associated MessageObj object.
2. A 'Reference Counter', indicating the total number of active references to the MessageObj within the system.

Standard Message manipulations (assignment and copy constructors) simply copy the pointer and increment the reference counter. Message destruction decrements the counter, and the associated MessageObj itself is deleted when the reference count falls to zero.

The Message object is seen to be light-weight and small, meaning that duplications are not costly, and Messages can efficiently be held in Standard Template Library [5] containers (e.g, std::list⟨Message⟩).

It should be stressed that the Message surrogate provides the only access to data within the full MeshRouter system. That is, the various higher-level objects in the system (Routers, Clients, Pipes, etc)

are all implemented in terms of Message and MessageList objects. The underlying MessageObj containers are inaccessible - as is necessary for the system to be both robust and efficient.

4.2 Plausible Extensions

As just described, the existing Message management system is quite effective in that:

1. A given user message entering a router is copied once and only once.
2. Copies are made onto existing, reusable memory blocks.
3. Manipulations within the overall MeshRouter system are done using the lightweight surrogate Message objects.

While the existing implementation is quite reasonable, there are some straightforward extensions that might be noted.

The processing associated with Fig.(1) will typically produce and consume enormous numbers of messages. Depending on the 'smarts' (or lack thereof) of the compiler, the constant creation and deletion of even the simple, surrogate Message objects of Section 4.1.3 could be a resource drain. This could be solved with a simple 'reusable object' stack in a fairly straightforward fashion.

The second plausible extension is a bit more subtle, though possibly more important. In the current implementation, both the BasicMemoryManager and MessageInterpreter are accessed through static class methods. That is, these objects are treated as global entities. This simplification aided initial development and testing of the system. However, reliance on these global objects means that the current implementation is not thread-safe.

The notion of a thread-safe MeshRouter architecture is attractive. In particular, one could well imagine several router systems (each "system" contains a single Router object together with its associated set of Client objects) running as separate threads on a single CPU. The existing 'MemoryPipe' class described in Ref.[2] could easily be generalized to transfer Messages between different MessageLists in a thread-safe, efficient manner. The real roadblock to the thread-safe implementation would be 'interference' associated with a single MessageObj environment associated with all the threads.

A solution here is fairly obvious. A new class ('DataManager', or some other equally innocuous name) would be developed containing a BasicMemoryManager, an independent copy of the appropriate MessageInterpreter, and any other relevant global constructs. Independent DataManager objects would be created for each instanced thread and passed through to the high-level Router and Client objects associated with the thread. This would, in fact, remove almost all impediments to thread-safely, with the remaining trouble points being the low-level Read and Write methods in the Pipes.