Network Routing Application Programmer's Interface (API) and Walk Through 8.0

Dan Coffin, Dan Van Hook, Ramesh Govindan, John Heidemann, Fabio Silva
{dcoffin,dvanhook}@ll.mit.edu, {govindan,johnh,fabio}@isi.edu

March 6, 2001

1 Introduction

SCADDS data diffusion (at USC/ISI) and DRP (at MIT/LL) are both based on the core concept of subject-based routing. Although there are some fundamental differences between these approaches, we believe that both can be accommodated with the same Network Routing API.

An earlier version of this API was used by both network routing approaches. This document describes an updated version of the API, used by the SCADDS implementation. In addition to the Publish/Subscribe API, we introduce the Filter API in order to better support in-network processing (e.g. caching/aggregation) and mobile code.

1.1 Changes from draft 7.6

Since draft 7.6 (dated March 3rd, 2000), the following changes have been made:

- Attribution creation and manipulation has been simplified with attribute factories. The API now includes functions that create, find and match attributes. See section 2.1
- Send now accepts multiple attributes. (Section 3.6)
- A new section describing the new Filter API is included in section 2.4, along with a code example in section 8.
- Timers have also been added to the API and are described in section 4.
- The C++ header section was removed. Please consult the implementation for up-to-date headers. Walkthrough, and code examples have been updated to reflect these changes.

1.2 Other planned changes and implementation status

Although we have iterated on this API and used it in the October 2000 SensIT demo, we expect that the API will evolve as we gain more experience with how network routing is used.

We expect to continue to experiment with better APIs for sending large objects across the network. (We would like to include support for mobile code.)

As of December 2000, we do not expect to implement this revised API on the CE platform (the CE compiler does not support all of the required C++ constructs in libraries). An implementation exists for linux; this will be ported to the QNX nodes when they become available.

2 NR API Overview

This version of the API has been improved to better support attribute creation, manipulation and matching. We have introduced templates that facilitate the use of attributes. This API now uses STL vectors to group a set of attributes that describe interests and data. Following is an overview of the approach taken and a brief description of the use of these templates.

2.1 Attributes and Attribute Factories

Data requests and responses are composed of data attributes that describe the data. Each piece of the subscription (an Attribute) is described via a key-value-operator triplet, implemented with class Attribute.

- key indicates the semantics of the attribute (latitude, frequency, etc.). Keys are simply constants (integers) that are either defined in the network routing header or in the application header.

  Allocation of new key numbers will be done with an external procedure to be determined. Keys in the range 0-2999 are reserved and should not be used by an application.

- type indicates the primitive type that the key will be. This key will indicate what algorithms to run to match subscriptions. For example, checking to see if an INT32_TYPE is EQ is a different operation than checking to see if a STRING_TYPE is EQ. The available types are:

  ```
  INT32_TYPE     // 32-bit signed integer
  FLOAT32_TYPE   // 32-bit
  FLOAT64_TYPE   // 64-bit
  STRING_TYPE    // UTF-8 format
  BLOB_TYPE      // uninterpreted binary data
  ```

- op (the operator) describes how the attribute will match when two attributes are compared.

  The IS operator indicates that this attribute specifies a literal (known) value (the LATITUDE_KEY IS 30.456). Other operators (GE, LE, NE, etc.) mean that this value must match against an IS attribute. Matching rules are below.

  This version of the API supports all operators for all types. Note however that for blobs the API doesn't know how the information is encoded and will perform a bit wise comparison only.

In addition, attributes have values. Values have some type and contents. Some values also have a length (if it's not implicit from the type). Keys, operators, type and length can be extracted from an attribute via getKey(), getOp(), getType() and getLen() methods. (see below for details).

In order to ease attribute creation and manipulation, the API provides factories to create attributes. The attribute factories also include other functions that allow finding an attribute in a set of attributes. An example of how to define and create an attribute is shown below:

```
#define LATITUDE_KEY 5050 // Defines key value for the attribute

// Creates a factory for the LATITUDE_KEY attribute
NRSimpleAttributeFactory<float> LatitudeAttr(LATITUDE_KEY,
                                NRAttribute::FLOAT32_TYPE);
```

```
// Creates an attribute lat with the op IS and value 45.1
NRAttribute *lat = LatitudeAttr.make(NRAttribute::IS, 45.1));
```

These factories are available for all supported types and can be used to create INT32_TYPE (<int>), FLOAT32_TYPE (<float>), FLOAT64_TYPE (<double>), STRING_TYPE (<char *>) and BLOB_TYPE (<void *>).

Also, the method getVal() returns the value of the attribute. The value from the attribute created above can be accessed by:

```
float lat_val = lat->getVal();
```

Since several API functions require a set of attributes (to describe a subscription, publication, data, etc), the API defines the $NRAttrVec$ structure, which is a STL vector of pointers to attributes. As a consequence, this version of the Network Routing API does not require applications to explicitly pass the number of attributes in each API function interface. This information can easily be obtained using the STL vector's size() method.

Here's example code creating a set of attributes:

```
NRSimpleAttributeFactory<float> LatitudeAttr(LATITUDE_KEY,
                                NRAttribute::FLOAT32_TYPE);
NRSimpleAttributeFactory<float> LongitudeAttr(LONGITUDE_KEY,
                                NRAttribute::FLOAT32_TYPE);
NRSimpleAttributeFactory<char *> TaskNameAttr(TASK_NAME_KEY,
                                 NRAttribute::STRING_TYPE);
NRSimpleAttributeFactory<void *> TaskQueryDetailAttr(TASK_QUERY_DETAIL_KEY,
                                 NRAttribute::BLOB_TYPE);

main()
{
  NRAttrVec attrs;
 /*
  * Demonstrate making some attributes.
  * All duplicative information is in the factory.
  */

  // Push back is an STL vector operation
  attrs.push_back(LatitudeAttr.make(NRAttribute::IS, 45.1));
  attrs.push_back(LongitudeAttr.make(NRAttribute::IS, 104.1));
  attrs.push_back(TaskNameAttr.make(NRAttribute::IS, "Harry Potter"));
  char *query_detail = "QD";
  attrs.push_back(TaskQueryDetailAttr.make(NRAttribute::IS,
                                           (void *)query_detail,
                                           strlen(query_detail) + 1));
}
```

The following find interfaces are present in the attribute factories and can be used to find an attribute that has the same key as the factory in a set of attributes. The first form of the function searches the set of

attributes and returns the first one that matches the key. The second, more general, form allows application to specify where to start the search.

```
NRSimpleAttribute<T>* find(NRAttrVec *attrs,
                           NRAttrVec::iterator *place = NULL);
NRSimpleAttribute<T>* find_from(NRAttrVec *attrs,
                                NRAttrVec::iterator start,
                                NRAttrVec::iterator *place = NULL);
```

The following examples illustrate how these two functions can be used to find an attribute in a set of attributes:

```
// Find a latitude attribute in a set of attributes
NRSimpleAttribute<float> *lat = LatitudeAttr.find(&attrs);

if (!lat){
  // Attribute not present in the set
  ...
}

// Demonstrate extracting multiple attributes with the same key
NRAttrVec::iterator place = attrs.begin();
for (;;){
  NRSimpleAttribute<char *> *task = TaskNameAtt.find_from(&attrs, place,
                                                           &place);
  if (!task)
    break;
  // Process attribute
  cout << "Task = " << task->getVal() << endl;
  place++;
}
```

2.2 Matching rules

Data is exchanged when there are matching subscriptions and publications and the publisher sends data. Matches are determined by applying the following rules between the attributes associated with the publish (P) and subscribe (S):

```
For each attribute Pa in P, where the operator Pa.op is something other than IS
  Look for a matching attribute Sa in S where Pa.key == Sa.key and Sa.op == IS
    If none exists, exit (no match)
       else use Pa.op to compare Pa and Sa
If all are found, repeat the procedure comparing non-IS operators in S against IS operators in P.
  If neither exits with (no match), then there is a match.
```

For example, a sensor would publish this set of attributes (LAT IS 30.455, LON IS 104.1, TARGET IS tel,), while a user might look for TELs by subscribing with the attribute (TARGET EQ tel), or it might look for anything in a particular region with (TARGET EQ_ANY, LAT GE 30, LAT LE 31, LON GE 104, LON LE 104.5).

Filters, described later in Section 2.4, use one-way matching only. In this case, the matching procedure does not compare non-IS operators in the second set of attributes against IS operators in the first set.

2.3 Matching functions

This version of the API includes several functions that perform matching. Even though these functions are not typically needed by applications, we document them here. They are members of the NRAttribute class and their prototypes are as follows:

```
bool isEQ(NRAttribute *attr);
bool isGT(NRAttribute *attr);
bool isGE(NRAttribute *attr);
bool isNE(NRAttribute *attr);
bool isLT(NRAttribute *attr);
bool isLE(NRAttribute *attr);
```

All the above functions assume that both attributes have the same key and type. An example of the use of these functions is shown below:

```
NRAttribute *lat1 = LatitudeAttr.make(NRAttribute::IS, 45.1);
NRAttribute *lat2 = LatitudeAttr.make(NRAttribute::IS, 38.3);

bool flag = lat1->isGT(lat2);
// flag is true !
```

2.4 Filter API

Filters are application-specific code that runs in the network to accomplish application-specific processing like aggregation, caching, etc. With the Filter API, applications can specify a set of attributes along with a callback. This causes diffusion to send incoming messages matching those attributes to the application callback. Filters are a special case of the matching rules. When matching filters against incoming messages, the match is done one-way only. Only operators in the filter have to be satisfied in order to be a match. Note that if a filter is too generic, it will possibly receive messages from other applications too. For instance, if the only attribute specified is CLASS_KEY EQ INTEREST_CLASS, all interest messages from all applications will be forwarded to this callback.

When setting up a filter, the application has to specify its priority. This is used to define the order filters will be called when multiple filters match the same incoming message. Higher numbers are called first. This number should be discussed with other developers to avoid problems.

If there is a match, the incoming message will be forwarded to the callback in the form of a Message structure (described later in the Interfaces section). The message structure allows the callback to access information that would otherwise be hidden. This information include message last hop, which contains either a random id for each neighbor (that is picked upon start up and changed periodically) or the constant LOCALHOST_ADDR, indicating that the message originated from a local application.

The filter callback have control of this message, which can be forwarded, changed and then forwarded or discarded. Please refer to sections 5.3 and 5.4 for a detailed description on how to do this.

2.4.1 Message Flow with the Filter API

This section describes the message flow in Directed Diffusion when using the Filter API. The numbers in the text correspond to the numbers in figure 1, which illustrates the message flow.

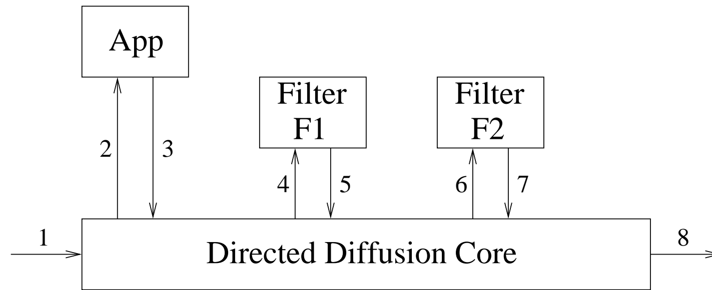

Figure 1: Message Flow in Directed Diffusion

Messages arriving at the Directed Diffusion Core module usually come from either the network (1) or from local applications (3), which use the publish/subscribe/send interface to send interest and data messages to the network.

Every incoming message is matched against all registered filters (see section 5.1 for a detailed description on how to add a filter). The result is a list ordered by filter priority containing all filters that resulted in a successful match.

Assuming that both filters F1 and F2 on figure 1 match the incoming message and P(F1) > P(F2), where P(X) is the priority of filter X, Diffusion will forward the incoming message to filter F1 (4) since it has the highest priority.

After processing the message, filter F1 can return the message to Directed Diffusion (5) by calling SendMessageToNext (see section 5.4). Note that the filter can return the same message it received from Diffusion, a modified message, a complete new message, multiple messages (by calling SendMessageToNext more than one time) or even no messages (by returning from the callback without calling SendMessageToNext). Directed Diffusion will match returned messages against all registered filters again and create a new list of filters (note that because filter F1 can change the message or send new messages, the list of filters matching returned messages can be different than the original list). If filter F1 is still present on this list, the message is sent to the filter coming immediately after F1 in the list (in our case, filter F2 will receive the message (6)). If the filter F1 is not present on this new list, the first filter on the list will receive the message. If there are no other filters after F1, the message will be simply discarded.

Filters can send messages directly to the network (e.g. (5), then (8)) or to a local application (e.g. (5), then (2)) by calling SendMessage (see section 5.3 for more information). In this case, other filters will not receive the message.

3 Interfaces

Following is a description of each of the methods that are part of the network routing API class.

3.1 Initialization

To initialize the NR class, there is a C++ factory called NR::createNR() that will create the NR class and return a pointer to it.

The prototype of the function is as follows:

```
static NR * NR::createNR();
```

createNR() is a method rather than a constructor so that it can actually create a specific subclass of class NR (one for MIT and one for ISI-W). It will create any threads needed to insure callbacks happen.

3.2 Subscribe

The application declares interest in via the NR::subscribe interface. This function accepts a list of interest attributes and uses this information to route data to the necessary nodes.

The prototype of the function is as follows:

```
handle NR::subscribe(NRAttrVec *subscribeAttrs, const NR::Callback * cb);
```

- subscribeAttrs is a pointer to a STL vector containing the elements that describe the subscription information (see notes above about the class type).
- cb indicates the class that contains the implementation of the method to be called when incoming data (or tasking information) matches that subscription. The class inherits from the following abstract base class:

  ```
  class Callback {
  public:
    virtual void recv(NRAttrVec *data, handle h) = 0;
  };
  ```

  After subscriptions are diffused throughout the network, data arrives at the subscribing node. The recv() method (implemented by the application) is called when incoming matching data arrives at the network routing level. This method is used to pass the incoming data and tasking information up to the caller. See section 3.7 for more detail about callbacks.

Subscribe returns a handle (which is an identifier of the interest declaration/subscription). This handle can be used for a later unsubscribe() call. If there is an error in the subscription call (based on local information, not the propagation of the interest), then a value of -1 will be returned as the handle.

Note that the condition that no data matches the attributes is not considered an error condition—if the application requires or expects data to be published, application-level procedures must determine conditions that might cause no data to appear. (As one example, if the goal is to contact a few nearby sensors, the application might start with a small region around it and expand or contract that region until the expected number of sensors reply.)

Subscribes are used to get both data and to find out about subscriptions from other nodes. To get data, (if you're a data sink) subscribe with CLASS_KEY IS INTEREST_CLASS. This subscription will propagate through the network and the callback will eventually trigger when matching data returns.

To find out about interests (if you're a sensor or data source), subscribe including CLASS_KEY EQ INTEREST_CLASS. The callback you provide will be called for each new subscription (with CLASS_KEY IS INTEREST_CLASS). In order to also find out whenever a subscription goes away due to unsubscribe or detection of node failure (timeout), subscribe including CLASS_KEY NE DATA_CLASS. This will result in callbacks with CLASS_KEY IS DISINTEREST_CLASS. These callbacks will be made at least once for each unique expression of interest. If multiple clients express interest in exactly the same thing, this callback will occur at least once.

3.3 Unsubscribe

The application indicates that it is no longer interested in a subscription via the NR::unsubscribe interface. This function accepts a subscription handle (originally returned by subscribe()) and removes the subscription associated with that passed handle.

The prototype of the function is as follows:

```
int NR::unsubscribe(handle subscription_handle);
```

- subscription_handle is a handle associated with the subscription that the application wishes to unsubscribe to. It was returned from the NR::subscribe interface.

The return value indicates success/failure. If there is a problem with the handle, then an error code of -1 is returned. Otherwise, 0 is returned for normal operation. If an unsubscribe is issued for a subscription that contains a task, then the each node that has received that task will be informed of the unsubscribe.

3.4 Publish

The application also indicates what type of data it has to offer. This is done via the publish function.

The prototype of the function is as follows:

```
handle NR::publish(NRAttrVec *publishAttrs);
```

- publishAttrs is an STL vector of publication declarations.

This function returns a handle (which is an identifier of the publication) that will be later used for unpublish(). If there is an error in the publication call, then a value of -1 will be returned as the handle. This handle is used later when unpublishing or sending data.

3.5 Unpublish

The publish interface has a matching unpublish interface (NR::unpublish).

The prototype of the function is as follows:

```
int NR::unpublish(handle publication_handle);
```

- publication_handle is the handle associated with the publication that the application wishes to unpublish. It was returned from the NR::publish interface.

The return value indicates success/failure. If there is a problem with the handle, then an error code of -1 is returned. Otherwise, 0 is return for normal operation.

3.6 Send

After the publications are set up the application can send data via the NR::send function. This function will accept a set of attributes to send associated with a publication handle (the handle used in the associated NR::publish function call). This method does not guarantee delivery, but the system will make reasonable efforts to get it to its destination.

The prototype of the function is as follows:

```
int NR::send(handle publication_handle, NRAttrVec *sendAttrs);
```

- publication_handle is the handle that is associated with the block of data that was sent.
- sendAttrs is a pointer to a STL vector containing the attributes to be associated with the send. (In addition to those attributes defined in the original publication.)

The return value indicates success/failure. If there is a problem with the arguments, then an error code of -1 is returned. Otherwise, 0 is returned for normal operation. If there is currently no one interested in the message (no matching subscription), then it will not consume network resources.

3.7 Recv

After the subscribe is issued from the application thread, the application thread can wait for the reception of information via the NR::Callback::recv() method.

The network routing system will provide a thread to make callbacks happen. Since there is only one such thread in the system, the callback function should return reasonably quickly. If the callback needs to do some compute-bound or IO-bound task, it should signal another thread.

The attributes received in this function should be treated as read-only. The API will delete them as soon as the callback returns. The application must copy the appropriate attribute(s) if they are needed after the recv function returns.

4 Timer API

In this version of the API we have included timers in order to support event-driven application. As described below, applications can setup timers to call an application provided callback. Timers can also be removed from the API's event queue.

4.1 Add Timer

The application can set up a timer via the NR::addTimer interface. The function calls the provided callback after the specified time has elapsed.

The prototype of the function is as follows:

```
handle addTimer(int timeout, void *param, TimerCallbacks *cb);
```

- timeout specifies the time to wait before calling the recv callback (in msecs)
- param is a pointer to any application structure that will be passed to the recv callback

- cb indicates the class that contains the implementation of the methods to be called when the timer expires or when the timer is being deleted. The class inherits from the following abstract base class:

```
class TimerCallbacks {
public:
  virtual int recv(handle hdl, void *p) = 0;
  virtual void del(void *p) = 0;
};
```

  The recv() method (implemented by the application) is called when the timer expires. The return value indicates if the timer should be rescheduled and the new timeout. A return value of 0 indicates that the application wants to reschedule the timer with the same timeout (provided previously in addTimer()). A negative return value indicates that the timer is to be deleted (the API will call the del() method described below). A positive return value causes the timer to be rescheduled with a new timeout.

  The del() method (also provided by the application) is called when a timer expires and is not rescheduled or when a timer is canceled by removeTimer() (see below). In this callback, the application should take care of freeing the memory pointed by p.

4.2 Remove Timer

The application indicates that it wants to cancel a pending timer via the NR::removeTimer interface. This function accepts a timer handle (originally returned by addTimer()) and removes the timer associated with that passed handle.

The prototype of the function is as follows:

```
int NR::removeTimer(handle timer_handle);
```

- timer_handle is a handle associated with the timer that the application wishes to cancel. It was returned from the NR::addTimer interface.

The return value indicates success/failure. If there is a problem with the handle, then an error code of -1 is returned. Otherwise, 0 is return for normal operation.

5 Filter API

The following sections describe the interfaces for the Filter API. It should be used to run application code in the network for application specific processing.

5.1 Add Filter

The application can set up a filter via the NR::addFilter interface. This function accepts a set of attributes that are matched against incoming messages.

The prototype of the function is as follows:

```
handle addFilter(NRAttrVec *filterAttrs, int16_t priority,
                 FilterCallback *cb);
```

- filterAttrs is a pointer to a STL vector containing the attributes of this filter. Note that for filters, matching is performed one-way (messages have to match the operators specified in the filter).
- priority is the filter priority (larger numbers are called before smaller ones). This number has to be agreed with other application developers.
- cb indicates the class that contains the implementation of the method to be called when an incoming message matches the filter. The class inherits from the following abstract base class:

```
class FilterCallback {
public:
  virtual void recv(Message *msg, handle h) = 0;
};
```

  The handle h is the same handle returned by addFilter() and the msg points to a message class containing the message that was matched against the filter's attributes. The message class is defined as follows:

```
class Message {
public:
  // Read directly from the packet header
  // (other internal fields omitted here)
  int32_t next_hop;   // Message next hop. Can be BROADCAST_ADDR if
                      // a neighbor node sent it to broadcast,
                      // LOCALHOST_ADDR if it came from a local application
                      // or it can be this node's ID if the message was sent
                      // by a neighbor to this node only.
  int32_t last_hop;   // Can be either a neighbor's ID or LOCALHOST_ADDR, if
                      // the message comes from a local application.

  // (raw packet is also available for internal use)

  // Other flags
  bool new_message;

  // Message attributes
  NRAttrVec *msg_attr_vec;
};
```

AddFilter returns a handle (which is an identifier of the filter). This handle can be used for a later removeFilter() call. If there is an error in the addFilter() call, then a value of -1 will be returned as the handle.

5.2 Remove Filter

The application indicates that it is no longer interested in a filter via the NR::removeFilter interface. This function accepts a filter handle (originally returned by addFilter()) and removes the filter associated with that passed handle.

The prototype of the function is as follows:

```
int NR::removeFilter(handle filter_handle);
```

- filter_handle is a handle associated with the filter that the application wishes to remove. It was returned from the NR::addFilter interface.

The return value indicates success/failure. If there is a problem with the handle, then an error code of -1 is returned. Otherwise, 0 is return for normal operation.

5.3 Send Message

The application can send any messages from within a filter directly to the network/other agents via the NR::sendMessage interface.

```
void sendMessage(Message *msg, handle h, int16_t agent_id = 0);
```

- msg is a pointer to the message the application wants to send.
- h is the handle received from addFilter.
- agent_id can specify the application (in the same node) that should receive the message (if no agent_id is specified, the message is sent to the node specified in msg->next_hop).

5.4 Send Message to Next

A filter callback can use the NR::sendMessageToNext interface to send a message (either the message passed with the callback or any other message) to the next filter (the filter whose priority comes right after the current filter's priority). An example of this is presented later in this document where a 'logging' filter with a high priority receives all messages before any other filter and then passes the same message (unmodified) to the next filter (thus being transparent to other filters).

```
void sendMessageToNext(Message *msg, handle h);
```

- msg is a pointer to the message being sent.
- h is the current filter handle.

This function should only be used from within a filter callback since it requires the current filter handle.

6 Publish/Subscribe API Walk Through

This walkthrough considers, at a high-level, what happens in the network when an user node is interested in a specific target.

A user node (say, Node A) wants information about TELs and expresses this interest by calling NR::subscribe with the following attributes:

```
CLASS_KEY IS INTEREST_CLASS
SCOPE_KEY IS GLOBAL_SCOPE
LONGITUDE_KEY GE 10
LONGITUDE_KEY LE 50
LATITUDE_KEY GE 20
LATITUDE_KEY LE 40
TASK_FREQUENCY_KEY IS 500
device_type EQ seismic
TARGET_KEY IS TEL
TARGET_RANGE_KEY LE 50
CONFIDENCE_KEY EQ_ANY *
TASK_NAME_KEY IS detect_track
TASK_QUERY_DETAIL_KEY IS [query_byte_code]
```

Note the distinction between IS which specifies a known value and EQ, which specifies a required match. All of the comparisons are currently ANDed together. The query parameters that interact with network routing (for example, lat/lon) must be expressed as attributes, but some other parameters may appear only in application-specific fields (such as the query_byte_code in this example).

NR::subscribe returns right away with a handle for the interest. Because this interest has a global scope, network routing forwards it to the neighboring nodes, that proceed using the pre-defined rules.

Nodes with sensor(s) tell network routing about the type of data they have available by publishing. An application on a node (say, Node B) would call NR::publish with the following attributes:

```
CLASS_KEY IS DATA_CLASS
SCOPE_KEY IS NODE_LOCAL_SCOPE
LONGITUDE_KEY IS 10
LATITUDE_KEY IS 20
TASK_NAME_KEY IS detectTrack
TARGET_RANGE_KEY IS 40
device_type IS seismic
```

After receiving the handle, the application can start sending data with the NR::send command. For example, each detection it might invoke send() with the attribute (CONFIDENCE_KEY IS .8) and then handle from the publish command. This attribute (CONFIDENCE_KEY) would be associated with the other attributes associated with publish handle and eventually delivered to anyone with a matching subscribe. Initially, since the node has no matching interest, the data will not propagate to other nodes. When interests arrive, data will begin to propagate.

In some cases, the application may wait to start sensing until it has been tasked, either to avoid doing unnecessary work, or to use the parameters in the interest to influence what it looks for. In this case, the sensor would get the task by subscribing to interests with NR::subscribe and the following attributes:

```
CLASS_KEY NE DATA_CLASS
SCOPE_KEY IS NODE_LOCAL_SCOPE
LONGITUDE_KEY IS 10
LATITUDE_KEY IS 20
TASK_NAME_KEY IS detectTrack
TARGET_RANGE_KEY IS 40
device_type IS seismic
```

(The only difference in these attributes with the publish call is in the CLASS key and operator.) The callback associated with this subscribe will then be called each time a new subscription arrives or goes away. Arrivals will have CLASS_KEY IS INTEREST_CLASS, unsubscribes will have CLASS_KEY IS DISINTEREST_CLASS.)

7 API Usage Examples

```
Step 1:  Initialization of Network Routing

{
    // This code is in the initialization routine of the
    // network routing client.

    .
    .
    .

    // Create an NR instance.  This will create the NR instance
    // and return a pointer to that instance. This
    // will only be created once per node.  Save this somewhere
    // where it can be used from wherever data needs to be sent.
    NR *dr;

    dr = NR::createNR();

    // Setup the publications and subscriptions for this NR client.
    // Note the details of the following calls are found in step 3
    setupPublicationsOnSensorNode(dr);
    setupSubscriptionsOnSensorNode(dr);

    .
    .
    .

    // Do any other initialization stuff here...
}

Step 2:  Create callbacks for incoming data/subscriptions

// In a header file somewhere in the client setup two callback
// classes.  One to handle incoming data and the other to handle
// incoming subscriptions/tasking.

class DataReceive : public NR::Callback {
public:
    void recv(NRAttrVec *data,
              NR::handle h);
```

```
};

class TaskingReceive : public NR::Callback {
public:
    void recv(NRAttrVec *data,
              NR::handle h);
};

// In the appropriate C++ file, define the following methods
void DataReceive::recv(NRAttrVec *data,
                       NR::handle h)
{
        // called every time there is new data.
        // handle it.
}

void TaskingReceive::recv(NRAttrVec *data,
                          NR::handle h)
{
    // Handle incoming tasking.
}
```

Step 3: Setup publications and subscriptions

```
void setupPublicationsOnSensorNode(NR *dr)
{
    // Setup a publication with 4 attributes.

    NRAttrVec attrs;

    attrs.push_back(NRClassAttr.make(NRAttribute::IS,
                                     NRAttribute::DATA_CLASS));
    attrs.push_back(DeviceAttr.make(NRAttribute::IS,
                                    "seismic");
    attrs.push_back(LatitudeAttr.make(NRAttribute::IS, 54.78));
    attrs.push_back(LongitudeAttr.make(NRAttribute::IS, 87.32));

    // publish these attributes save the pub_handle
    // somewhere like in the private data.
    pub_handle = dr->publish(&attrs);

    // Delete the attributes (they have been copied by publish)
    ClearAttrs(&attrs);

    if (pub_handle == -1)
    {
        // ERROR;
    }
```

```
}

// create two subscriptions (one for the sensor node and the
// other for the user node.

void setupSubscriptionsOnSensorNode(NR *dr)
{
    // (1) the first subscription indicates that I am interested in
    // receiving tasking subscriptions for this node.
    //
    // Each of the subscriptions will have its own callback
    // to separate incoming subscriptions and incoming data.

    // ****************
    // SUBSCRIPTION #1:  First setup the subscription for
    // tasking interests...

    NRAttrVec attrs;

    attrs.push_back(NRClassAttr.make(NRAttribute::EQ,
                                     NRAttribute::INTEREST_CLASS));
    attrs.push_back(DeviceAttr.make(NRAttribute::EQ_ANY, "");

    // Create the callback class that will be used to
    // handle subscriptions that match this subscription.
    TaskingReceive * tr = new TaskingReceive();

    // subscribe and save the handle somewhere
    // like in the private data.
    task_sub_handle = dr->subscribe(&attrs, tr);

    // Delete the attributes (they have been copied by publish)
    ClearAttrs(&attrs);

    if (task_sub_handle == -1)
    {
        // ERROR;
    }
}

void setupSubscriptionsOnUserNode(NR *dr)
{
    // (2) the second subscription indicates that I am interested in
    // nodes that have seismic sensors in a particular region
    // and have
    // them run a task call detectTel (with some specific query
    // byte code attached). This task instructs the nodes to send
    // data back.
```

```
    // ****************
    // SUBSCRIPTION #2:  First setup the subscription for
    // tasking subscriptions...

    NRAttrVec attrs;

    attrs.push_back(NRClassAttr.make(NRAttribute::IS,
                                     NRAttribute::INTEREST_CLASS));
    attrs.push_back(DeviceAttr.make(NRAttribute::IS, "seismic");
    attrs.push_back(LatitudeAttr.make(NRAttribute::GE, 44.0));
    attrs.push_back(LatitudeAttr.make(NRAttribute::LE, 46.0));
    attrs.push_back(LongitudeAttr.make(NRAttribute::GE, 103.0));
    attrs.push_back(LongitudeAttr.make(NRAttribute::LE, 104.0));
    attrs.push_back(TaskNameAttr.make(NRAttribute::IS, "detectTel"));
    attrs.push_back(TaskQueryDetailAttr.make(NRAttribute::IS,
                                            &query_byte_code.contents,
                                            query_byte_code.len));

    // Create the callback class that will be used to
    // handle subscriptions that match this subscription.
    DataReceive * drcv = new DataReceive();

    // subscribe and save the handle somewhere
    // like in the private data.
    data_sub_handle = dr->subscribe(&attrs, drcv);

    // Delete the attributes (they have been copied by publish)
    ClearAttrs(&attrs);

    if (data_sub_handle == -1)
    {
        // ERROR;
    }

Step 4:  Sending data
{
    .
    .
    // When one of the clients have data to send, then it will use
    // the NR::send method.  Assume that there is acoustic data to
    // send (matching the publish call above).  The data is found
    // in the variable adata with the size of adata_size.
    NRAttrVec attrs;

    attrs.push_back(AppBlobAttr.make(NRAttribute::IS,
                                     &adata,
                                     adata_size));
```

```
    if (dr->send(pub_handle, &attrs) == -1)
    {
       // ERROR;
    }

    delete attr;
    .
    .
}

Step 5:  Receiving data

(the ReceiveData callback is called every time data arrives)
```

8 Filter API example

In this section we describe a simple module that uses the Filter API to receive INTEREST messages received by data diffusion. The module sets up a filter with high priority (in order to get the packets before other modules). After logging their arrival (printing a packet has arrived along with where they came from), the Logging module returns these (unmodified) messages to data diffusion so other modules/applications can receive them.

```
// Step 1: Initialization

LoggingApp::LoggingApp()
{
   // Create Diffusion Routing class
   dr = NR::createNR();
   fcb = new MyFilterReceive;

   // Set up the filter
   filterHandle = setupFilter();
   printf("Logging App received handle %d\n", filterHandle);
   printf("Logging App initialized !\n");
}

// In a header file, the Filter API client setup a callback class.

// High priority to get message first
#define LOGGING_APP_PRIORITY 16

class MyFilterReceive : public FilterCallback {
public:
   void recv(Message *msg, handle h);
};
```

```
// setupFilter() creates an attribute that matches all
// INTEREST messages coming to diffusion routing

handle LoggingApp::setupFilter()
{
   NRAttrVec attrs;
   handle h;

   // This attribute matches all interest messages
   attrs.push_back(NRClassAttr.make(NRAttribute::EQ,
                                    NRAttribute::INTEREST_CLASS));

   h = ((DiffusionRouting *)dr)->addFilter(&attrs,
                                           LOGGING_APP_PRIORITY, fcb);
   ClearAttrs(&attrs);
   return h;
}

Step 2: Handle callbacks

// Implement the Filter API callback

void MyFilterReceive::recv(Message *msg, handle h)
{
   printf("Received a");

   if (msg->new_message)
      printf(" new ");
   else
      printf("n old ");

   if (msg->last_hop != LOCALHOST_ADDR)
      printf("INTEREST message from node %d\n", msg->last_hop);
   else
      printf("INTEREST message from agent %d\n", msg->source_port);

   // We shouldn't forget to send the message back to diffusion routing
   ((DiffusionRouting *)dr)->sendMessageToNext(msg, h);
}
```