Network Routing Application Programmer's Interface (API) and Walk Through 9.1

Fabio Silva, John Heidemann and Ramesh Govindan
{fabio,johnh,govindan}@isi.edu

November 1, 2006

1 Introduction

I-LENSE 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, and was co-authored by Dan Coffin and Dan Van Hook `{dcoffin,dvanhook}@ll.mit.edu`.

This version of this document describes the current API as used by the I-LENSE implementation[1]. In addition to the Publish/Subscribe API, we introduce the Filter API in order to better support in-network processing (e.g. caching/aggregation/mobile code) as well as the Timer API, which allows support for event-driven applications.

Also, this document does not provide information on how to compile, install, and run diffusion. Please refer to the distribution's README file. For information about how to run diffusion in ns-2, please refer to the directed diffusion chapter in the ns-2 documentation.

1.1 Recent changes to the API

Since the API's last version, release 9.0.1, dated December 9st, 2002, the following changes have been made:

- RMST, our new Reliable Multi-Segment Transport algorithm is now supported in the API (see section 4.8 for details on the new NR::sendRmst function). RMST can be used to provide guaranteed delivery for large objects. See section 3.5 for more details on RMST.
- GEAR, the Geographic and Energy Aware Routing algorithm, now supports forwarding messages to both geographic regions (cluster of nodes) or points (a single node). GEAR can greatly improve diffusion's efficiency when geographic information is available. Refer to section 3.4 for more information on how to use GEAR and which attributes need to be changed for using geographic points.
- In addition to *two-phase pull* and *one-phase push*, the publish/subscribe API now also supports *one-phase pull*. The *one-phase pull* algorithm is a good match for application scenarios

[1]This work was supported by DARPA under grant DABT63-99-1-0011 as part of the SCADDS project.

where a small number of sinks communicate with a large number of sources. In order to use *one-phase pull*, applications will need to use the new NRAlgorithm attribute (see sections 4.2 and 4.4 for more information). For a comparison of these various algorithms, including a description on when to use each of these, refer to section 3.

- The Filter API description has been revised, giving filter developers a better understanding on how filter iteract and how to select priorities.
- Several of the sample applications/filters provided in our distribution have been updated with several small features and fixes. In order to illustrate the use of GEAR, sample GEAR applications were added to the distribution.

1.2 Future changes

We have iterated on this API and used it in various simulations, demos, paper experiments. As the number of people using these API increase, and we experiment with new routing algorithms, we expect that this API will evolve as a result of the experience gained with how naming and network routing is used.

2 Network Routing 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.

For a more detailed description about attributes and naming, see [2]. For more detail about filter invocation, see [3].

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 NRAttribute.

- 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 with the same type and key are compared. Available operators are: IS, EQ, NE, GT, GE, LT, LE, EQ_ANY.

  The IS operator indicates that this attribute specifies a literal (known) value (e.g. a node publishing data can use the attribute LATITUDE_KEY IS 30.456 to specify its location). Other operators (GE, LE, NE, etc.) indicate a condition, which must be satisfied with an IS attribute for a positive match (e.g. a LATITUDE_KEY IS 30.456 attribute will match the LATITUDE_KEY GT 25.34 condition, resulting in a positive match). 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 (i.e. IS can be used to specify a literal blob value that can only be matched with the EQ, EQ_ANY, and NE operators).

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).

2.2 Matching Rules

Data is exchanged when there are matching subscriptions and publications and the publisher sends data. Filters receive messages that match a specified set of attributes. Since diffusion is based on the core concept of subject-based routing, it is very important to make sure attributes in publications, subscriptions and filters match.

For the Publish/Subscribe API, 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:

```
LATITUDE_KEY IS 30.455
LONGITUDE_KEY IS 104.1
TARGET_KEY IS tel
```

while a user might look for TELs by subscribing with the attribute:

```
TARGET_KEY EQ tel
```

or it might look for anything in a particular region with:

```
TARGET_KEY EQ_ANY
LATITUDE_KEY GE 30
LATITUDE_KEY LE 31
LONGITUDE_KEY GE 104
LONGITUDE_KEY LE 104.5
```

Filters, described later in Section 2.5, only use one-way matching. In this case, the matching procedure just compares non-IS operators in the first set of attributes against IS operators in the second set, calling it a match if this operation is successful. For instance, in order to receive all *interest* messages arriving at the node, a filter developer would add a filter with the following attribute:

```
CLASS_KEY EQ INTEREST_CLASS
```

2.3 Using Attributes

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 TEMPERATURE_KEY 5050 // Defines key value for the attribute

// Creates a factory for the TEMPERATURE_KEY attribute
NRSimpleAttributeFactory<float> TemperatureAttr(TEMPERATURE_KEY,
                                                NRAttribute::FLOAT32_TYPE);

// Creates a temperature attribute with the op IS and value 56.12
NRAttribute *temperature = TemperatureAttr.make(NRAttribute::IS, 56.12));
```

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 *>) attributes.

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

```
float room_temperature = temperature->getVal();
```

Since several API functions require a set of attributes (to describe a subscription, publication, filter, 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.

This is an example that creates a set of attributes:

```
NRSimpleAttributeFactory<float> TemperatureAttr(TEMPERATURE_KEY,
                                NRAttribute::FLOAT32_TYPE);
NRSimpleAttributeFactory<float> HumidityAttr(HUMIDITY_KEY,
                                NRAttribute::FLOAT32_TYPE);
NRSimpleAttributeFactory<char *> MovieNameAttr(MOVIE_NAME_KEY,
                                 NRAttribute::STRING_TYPE);
NRSimpleAttributeFactory<void *> LibraryCardAttr(LIBRARY_CARD_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(TemperatureAttr.make(NRAttribute::IS, 56.12));
  attrs.push_back(HumidityAttr.make(NRAttribute::IS, 0.78));
  attrs.push_back(MovieNameAttr.make(NRAttribute::IS, "Harry Potter"));
  char *card_id = "B34201S102";
  attrs.push_back(LibraryCardAttr.make(NRAttribute::IS,
                                       (void *) card_id,
                                       strlen(card_id) + 1));
}
```

Attribute factories also provide the following find interfaces, which 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 applications 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 temperature attribute in a set of attributes
NRSimpleAttribute<float> *temp_attr = TemperatureAttr.find(&attrs);

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

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

2.4 Additional Attribute Functions

In this section, we list additional functions that can be used to manipulate, compare and delete attributes.

2.4.1 Matching Functions

Our 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 !
```

2.4.2 Attribute Manipulation Functions

This version of the API includes the following additional functions that can be used to manipulate sets of attributes:

```
NRAttrVec * CopyAttrs(NRAttrVec *src_attrs);
```

- This function returns a pointer to a newly created attribute vector, containing a copy of all attributes from src_attrs.

```
void AddAttrs(NRAttrVec *attr_vec1, NRAttrVec *attr_vec2);
```

- This function adds a copy of all attributes from attr_vec2 to the existing attribute vector attr_vec1.
- After execution, attr_vec1 will contain its original attributes and a copy of all attributes present in attr_vec2 (which will remain unchanged).

```
void PrintAttrs(NRAttrVec *attr_vec);
```

- This function prints all attributes present in attr_vec using the function DiffPrint.

```
void ClearAttrs(NRAttrVec *attr_vec);
```

- This function will delete all attributes from attr_vec. After the execution of ClearAttrs, attr_vec will be an empty vector (which still has to be deleted).

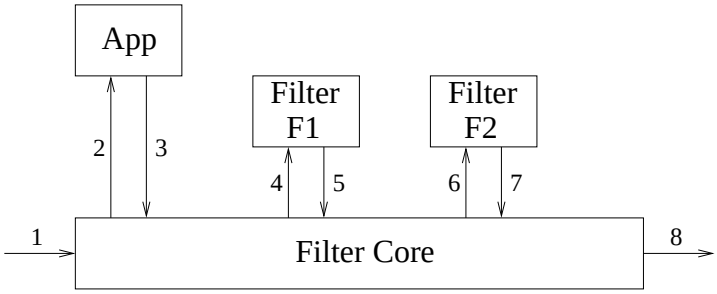

Figure 1: Message Flow in the Filter API

2.5 Filter API

Filters are application-provided software modules that allow applications to influence routing and data processing. Some uses of filters include routing, in-network aggregation, collaborative signal processing, caching, debugging, and monitoring. The filter architecture was designed to support a very high-level of user configurability. With the Filter API, filters can specify a set of attributes describing what messages they are interested along with a callback. This causes the filter core to send incoming messages matching those attributes to the filter callback. As mentioned in section 2.2, filters are a special case for 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 for a match to happen. 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 developer has also 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. Diffusion will not allow two filters to have the same priority, so this number should be discussed with other developers to avoid problems.

As a start point for developers, we suggest that filters doing message pre-processing (e.g. logging) have their priorities set above PRE_PROCESSING_COMPLETED_PRIORITY (currently configured as 200). Filters doing post-processing should have their priority below ROUTING_COMPLETED_PRIORITY (currently set to 50). Priorities between these two values should be used for filters implementing routing algorithms.

When the filter core matches an incoming message to a filter, it will send it to the callback provided in the form of a Message structure (described later in section 6.1). The message structure allows filter callbacks to access information that would otherwise be hidden from applications. This information include message last hop, which contains either a neighbor id (or a random id that is picked upon start up if node ids are not specified) or the constant LOCALHOST_ADDR, indicating that the message originated from a local application.

The filter callback has total control of this message, which can be forwarded, changed and then forwarded or discarded. Please refer to section 6.3 for a more detailed description.

2.5.1 Message Flow with the Filter API

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

Messages arriving at the filter 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.

Each incoming message is matched against all registered filters (see section 6.1 for a detailed description on how to add a filter). The result is a list ordered by filter priority containing all filters whose attributes 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, the filter core 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 the filter core (5) by calling NR::sendMessage (see section 6.3). Note that the filter can return the same message it received from the filter core, a modified message, a complete new message, multiple messages (by calling NR::sendMessage more than one time) or even no messages (by returning from the callback without calling NR::sendMessage). The filter core 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). What happens next depends if NR::sendMessage is called with or without a new priority. Assuming that the priority remains the same (the default behavior when the filter does not specify any priority in NR::sendMessage), 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 filter core will look at the message and see if it contains a a valid next hop address (or a valid application address). In that case, the filter core will forward the message to the appropriate node or application. Otherwise, the message will be simply discarded. If a new priority is specified in NR::sendMessage, the filter core will start looking for filters whose priorities are below the specified priority. This allows knowledgeable filters to send messages to other filters directly, bypassing the default ordering imposed by the filter core.

Filters can also send messages directly to the network (e.g. (5), then (8)) or to a local application (e.g. (5), then (2)) by calling NR::sendMessage with FILTER_MIN_PRIORITY (usually set to 1), see section 6.3 for more information). In this case, other filters will not receive the message.

3 Performance Evaluation

As of the 9.0 version of the API, diffusion supports several variants that can offer better efficiency for particular workloads. Developers therefore should know what performance/overhead to expect when using diffusion in order to better decide how to implement their algorithms using this Network Routing API. For a detailed description of how basic diffusion works, see [4]. Refer to [1] for a more detailed comparison of diffusion variants.

We now present a brief description of what happens when applications using *one-phase pull*, *two-phase pull* and/or *one-phase push* call publish and subscribe. We later describe how the use of geographic information can greatly reduce the cost of communication and how reliable transfers of large objects can be achieved using RMST, our Reliable Multi-Segment Transport algorithm. For detailed API information, please refer to section 4.

3.1 Two-phase-pull Diffusion

Each subscription (i.e. call to subscribe) causes the diffusion library to periodically send an *interest* message (containing the subscription's attributes) to the network. These interest messages are

flooded throughout the network. When they encounter publishers with matching data, simple (non-reinforced) gradients are set up from the publisher to the subscriber. Calls to publish, on the other hand, do not cause any messages to leave the diffusion library API. The attributes specified in the publish call are just stored so later they can be used. When an application calls send, the attributes provided are added to the attributes from a previous publish (using the handle provided). These attributes form a *data* message. Data messages are sent only through *reinforced* gradients. Periodically, in order to discover new paths or repair broken paths, diffusion marks a data message as *exploratory* and sends it to all nodes it has gradients (both reinforced and non-reinforced). When one of these messages reach a subscriber, it will cause a *positive reinforcement* to be sent to the publisher. That is what makes regular gradients become reinforced gradients.

In a brief summary, subscriptions cause periodic interest messages to flood the network and therefore are expensive. Publications (and then sends) incur no cost unless someone is interested in the data. Usually sends are cheap, sending data only on reinforced gradients, but periodically they are expensive (when marked exploratory).

Two-phase pull diffusion is therefore best suited to cases where there are many publishers and a few subscribers. If this situation is reversed, the *one-phase push* algorithm in diffusion, described in section 3.3, can provide better performance.

Also, basic diffusion must periodically flood information to all nodes. This cost can be reduced using geographic information and GEAR as described in Section 3.4.

3.2 One-phase pull Diffusion

One-phase pull is a subscriber-based system that avoids one of the two phases of flooding present in two-phase pull. As with two-phase pull, subscribers send *interest* messages that disseminate through the network, establishing gradients. Unlike two-phase pull, when an interest arrives at a source it does not mark its first data message as exploratory, but instead sends data only on the preferred gradient. The preferred gradient is determined by the neighbor who was the first to send the matching interest, thus suggesting the lowest latency path. Thus one-phase pull does not require reinforcement messages, the lowest latency path is implicitly reinforced.

One-phase pull has two disadvantages compared to two-phase pull. First, it assumes symmetric communication between nodes since the data path (source-to-sink) is determined by lowest latency in the interest path (sink-to-source). Two-phase pull reduces the penalty of asymmetric communication since choice of data path is determined by lowest-latency exploratory messages, both in the source-to-sink direction. However, two-phase pull still requires some level of symmetry since reinforcement messages travel reverse links. Although link asymmetry is a serious problem in wireless networks, many other protocols require link symmetry, including 802.11 and protocols that use link-level acknowledgments. We assume that the MAC layer will allow diffusion to identify asymmetric links.

Second, one-phase pull requires interest messages to carry a flow-id. Although flow-id generation is relatively easy (uniqueness can be provided by MAC-level addresses or probabilistically with random assignment and periodic reassignment), this requirement makes interest size grow with number of sinks. By comparison, though, with two-phase pull the number of interest messages grows with proportion to the number of sinks, so the cost here is lower. Second, the use of end-to-end flow-ids means that one-phase pull does not use only local information to make data dissemination decisions.

In [1], we show that one-phase pull is a much better algorithm for cases where a small number

of sinks pull data from a large number of sources. Also, as with two-phase pull, the cost of *interest* flooding in one-phase pull can be greatly reduced with GEAR if geographic information is available.

3.3 One-phase push Diffusion

One-phase push is an option to diffusion that reverses the relative costs of publish and subscribe. When applications use *one-phase push*, *interest* messages generated by subscriptions are not sent out to the network. They are kept local to the subscriber node. From time to time, *data* messages are turned into *exploratory data* messages, and are flooded throughout the network. When one of these messages reaches a subscriber node, a *positive reinforcement* message will be sent to the publisher in response, setting up reinforced gradients on its way. Non-exploratory data messages will travel only through these reinforced gradients. If there are no subscribers for this datatype, there will not be any reinforced gradients therefore non-exploratory messages will be suppressed at the publisher node.

In summary, subscriptions with *one-phase push* are cheap since messages do not leave the subscriber node. Publishers, on the other hand, have no way of telling if there are subscribers interested in their data therefore periodic exploratory data messages are flooded throughout the network.

One-phase push offers more benefits when there are many subscribers and/or when there are few data messages to be sent from publishers to subscribers.

3.4 Using GEAR

GEAR, our Geographical and Energy Aware Routing protocol, uses geographic information for making informed neighbor selection when routing packets towards a target region. In order to take advantage of GEAR, messages should contain a reference to either a *closed* region (i.e. have two latitude attributes and two longitude attributes, which specify a closed retangular region), or a point (one latitude and one longitude attribute, both with the EQ operator).

For the *one-phase and two-phase pull* algorithms, subscriptions (interests) should include the geographic attributes, containing the region of interest. Until a subscription gets inside the specified region, a node will forward interest messages only to a neighbor that is located closer to the region, setting up a single path from the subscriber to the region's border. This greatly improves diffusion's performance by avoid flooding *interest* messages outside the subscription's region. Nodes publishing data should include their location on the publications (latitude and longitude attributes with the IS operator), causing matching to occur. In *one-phase pull*, data messages will follow the interest path back to the subscriber. For *two-phase pull*, since *exploratory data* messages travel only through gradients, they are not going to be flooded outside the subscription's region.

When using GEAR with the *one-phase push* algorithm 3.3, geographic attributes containing a target region or point should be added to the publication, while subscribers should only include their node's location on their local subscription (in push, all subscriptions are local). In this case, *exploratory data* messages will travel through a single path until they reach the specified region, resulting in improved performance as there is no flooding.

3.5 Using RMST

RMST, Reliable Multi-Segment Transport [5], is a NACK-based reliability layer for diffusion that provides guaranteed delivery and fragmentation/reassembly for large data objects. RMST leverages

the reinforced routes established by diffusion to establish a "back-channel" from sink to sources on which repair requests and other control messages can be delivered. In order to use RMST, the rmst_filter must be loaded at each node in the sensor network. RMST, in its default mode, assumes that the MAC layer provides ARQ (e.g. S-MAC [6]). If a non-ARQ MAC layer is used, RMST should be put into "caching-mode" so that hop by hop repair is executed at the transport layer. Caching-mode is only feasible at nodes that have sufficient memory to cache all the fragments of the largest possible data object from the application. Sinks and sources automatically run in caching mode. RMST is fully integrated in the publish/subscribe API, and users can send a large object using RMST via the NR::sendRmst method added to the API (see section 4.8).

4 Publish/Subscribe API Interfaces

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

4.1 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.

4.2 Subscribe

The application declares interest in data 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 *subscribe_attrs, const NR::Callback * cb);
```

- subscribe_attrs 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 4.7 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 NR::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
SCOPE_KEY IS GLOBAL_SCOPE
```

If no SCOPE_KEY attribute is specified, the API will assume GLOBAL_SCOPE and this subscription will propagate throughout the network (*two-phase pull* semantics) and the callback will eventually trigger when matching data returns. If the SCOPE_KEY attribute is set to NODE_LOCAL_SCOPE, this subscription will follow the *one-phase push* semantics (please refer to section 3.3 for more information). In order to use *one-phase pull* semantics, subscriptions will have to include the NRAlgorithm attribute (set to NR::ONE_PHASE_PULL_ALGORITHM) [2]. A *one-phase pull* subscription will then look like:

```
CLASS_KEY IS INTEREST_CLASS
SCOPE_KEY IS GLOBAL_SCOPE
ALGORITHM_KEY IS ONE_PHASE_PULL_ALGORITHM
```

To find out about interests (if you're a sensor or data source and would like to know if there are subscribers in the network interested in a particular data type), subscribe with:

```
CLASS_KEY EQ INTEREST_CLASS
SCOPE_KEY IS NODE_LOCAL_SCOPE
```

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 NR::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 with the exact same set of attributes (identical subscriptions), this callback will occur at least once.

Note that some combinations of CLASS_KEY and SCOPE_KEY are not supported by this API. In these cases, NR::subscribe will return -1.

[2] In our next major release we plan to use the NRAlgorithm attribute for also selecting the *two-phase pull* and *one-phase push* algorithms.

4.3 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 NR::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.

4.4 Publish

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

The prototype of the function is as follows:

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

- publish_attrs 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 NR::send and NR::unpublish (when sending and unpublishing data). If there is an error in the publication call, then a value of -1 will be returned as the handle.

Note that if the application does not specify a SCOPE_KEY attribute, NR::publish will assume it to be NODE_LOCAL_SCOPE (*two-phase pull* semantics). If, however, the application wants to use the *one-phase push* semantics (please refer to section 3.3), both CLASS_KEY and SCOPE_KEY attributes need to be specified. The following example illustrates a publication using the *one-phase push* algorithm:

```
CLASS_KEY IS DATA_CLASS
SCOPE_KEY IS GLOBAL_SCOPE
```

In order to use the *one-phase pull* algorithm, publications will also have to include the NRAlgorithm attribute, setting the algorithm to NR::ONE_PHASE_PULL_ALGORITHM.

4.5 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.

4.6 Send

After 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 *send_attrs);
```

- publication_handle is the handle that is associated with the block of data that was sent.
- send_attrs 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.

4.7 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.

4.8 SendRmst

When a user wishes to send data with guaranteed delivery or fragmentation/reassembly for large data objects, they do so via the NR::sendRmst function. It is similar to the NR::send function with the addition of guaranteed delivery. The function's prototype is as follows:

```
int NR::sendRmst(handle publication_handle, NRAttrVec *send_attrs, int fragment_size);
```

- publication_handle is the handle that is associated with the block of data being sent. The associated publication must contain an RmstTargetAttr attribute.
- send_attrs is a pointer to a STL vector containing attributes to be sent. The send_attrs must include an RmstDataAttr containing the blob to be delivered.
- fragment_size is the fragment size that the data blob pointed to by the RmstDataAttr will be fragmented into. The fragment size + packet overhead + the cumulative size of the other attributes must be less than or equal to the sensor network mtu.

Similarly to NR::send, this function will return -1 if there is a problem with the arguments or 0 if the call was successful (meaning the blob was sent to the rmst filter in the node; delivery will occur asynchronously). An example of how to use RMST is provided in sample source and sink files included in the diffusion distribution.

5 Timer API

In our Network Routing API, we have included support for timers in order to support event-driven applications. As described below, applications and filters can setup timers to call an application provided callback. Timers can also be removed from the API's event queue.

In diffusion 3.1.3, our Timer API has changed. This section describes this updated version. The old API, described in earlier versions of this document, is still supported but as we plan to drop it in our next major release, developers are urged to use the API described here.

5.1 Add Timer

Applications 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 NR::addTimer(int timeout, TimerCallback *callback);
```

- timeout specifies the time to wait before calling the expire() callback (in msecs)
- callback 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 TimerCallback {
public:
  TimerCallback() {};
  virtual ~TimerCallback() {};
  virtual int expire() = 0;
};
```

Note that there is no need for any callback-specific parameters in NR::addTimer as application and filter developers can add instance variables and task-specific functions to the derived class. For instance, a timer used to send a message after a specific timeout, will probably include the actual message to be sent.

The expire() 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 the application wants to reschedule the timer with the same timeout (provided previously in NR::addTimer). A positive return value causes the timer to be rescheduled with a new timeout (in msecs). A negative return value indicates the timer is to be deleted from the event queue. Note that it is the user's responsibility to delete the timer callback. For example:

```
int SendDetectionTimer::expire()
{
  // Send detection
  dr_->send(handle_, detection_attrs_);

  // Delete this timer
  delete this;

  return -1;
}
```

5.2 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 NR::addTimer) and removes associated timer from the event queue.

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.

6 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.

6.1 Add Filter

Application can set up filters via the NR::addFilter interface. This function accepts a set of attributes that are matched against incoming messages, as described in section 2.2.

The prototype of the function is as follows:

```
handle addFilter(NRAttrVec *filter_attrs, u_int16_t priority,
                 FilterCallback *cb);
```

- filter_attrs 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. Valid filter priorities are in the 2 to 253 range. Please refer to section 2.5 for directions on how to assign priorities.

- 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 NR::addFilter and the message 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.

  // Other flags
  bool new_message_;

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

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

6.2 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 NR::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.

6.3 Send Message

A filter callback can use the NR::sendMessage interface to send a message to the next filter, to an arbitrary filter, or to the the network/other applications. An example of how to use this function is presented later, in section 8.2, where the 'log' filter, using a high priority, receives all messages before any other filter and then passes it to other filters, after logging some information on the terminal (since messages are not changed, this is totally transparent to other filters downstream).

```
void NR::sendMessage(Message *msg, handle h,
                     u_int16_t priority = FILTER_KEEP_PRIORITY);
```

- msg is a pointer to the message the application wants to send.
- h is the handle received from addFilter.
- priority can be used to tell diffusion to start looking for matching filters which have priorities lower than the one specified here. If this parameter is ommited, diffusion will follow its regular rules for determining which filter should get this message next. For further details, please refer to section 2.5.

If diffusion cannot find any filters matching a message sent by NR::sendMessage, it will try to send the message to the network (node specified in msg->next_hop_ if msg->next_port_ is set to 0) or to a local application (if msg->next_port_ is different than 0 and msg->next_hop_ is LOCALHOST_ADDR).

6.4 Blacklisting

In this version of the API document, we introduce two functions to the Filter API, NR::addToBlacklist and NR::clearBlacklist. Please note that *blacklisting* is still experimental and these APIs may change in the future.

6.4.1 Add to Blacklist

Filters can use the NR::addToBlacklist function to *blacklist* neighbors. Messages from blacklisted neighbors will be ignored by the filter core and will not be matched against any filters. This interface can be used by routing algorithms that need to ignore messages from unreliable neighbors (RMST uses it to remove links whose reliability falls under a certain number). The prototype for this function is as follows:

```
int NR::addToBlacklist(int32_t node);
```

- node is a neighbor that will be blacklisted.

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

6.4.2 Clear Blacklist

Filters can empty the blacklist by calling the NR::clearBlacklist function. This call will cause the filter core to clear its blacklist, allowing communication from all neighbors to reach the filter stack. This function can be used by filters, who previously added nodes with non-optimal links to the blacklist, and now want to try these links again (e.g. nodes may have moved or a better link may not be available anymore). The NR::clearBlacklist function does not take any parameters and its prototype is as follows:

```
int NR::clearBlacklist();
```

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

7 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.

7.1 Using one-phase and two-phase pull

As both *one-phase* and *two-phase pull* algorithms are very simular from the API point of view, both are described together in this section and any differences are pointed out in the text. An user node (sink), node A, wants information about TELs and expresses this interest by calling NR::subscribe with the following attributes:

```
Subscription 1: (two-phase pull)

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
SENSOR_TYPE_KEY EQ seismic
TARGET_KEY IS TEL
TARGET_RANGE_KEY LE 50
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 (condition). In this example, the conditions are seismic sensors detecting targets closer than 50 in the region with longitude between 10 and 50 and latitude between 20 and 40. All of the comparisons are currently ANDed together. The query parameters that interact with network routing (for example, latitude and longitude) 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 predefined rules.

If the application prefers to use the *one-phase pull* algorithm (instead of *two-phase pull*), subscription 1 should include the ALGORITHM_KEY IS ONE_PHASE_PULL_ALGORITHM attribute.

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

```
Publication 1: (two-phase pull)

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

After receiving the handle, the application can start sending data with the NR::send function. For example, for each detection it might invoke NR::send with the confidence attribute (CONFIDENCE_KEY), including confidence information for each individual detection (0.8, 0.9, etc). This attribute (CONFIDENCE_KEY) would be merged with the corresponding attributes associated with the publish handle (e.g. from publication 1) 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. Again, as in the subscription, in order to use the *one-phase pull* algorithm, the publication should include the an ALGORITHM_KEY attribute (ALGORITHM_KEY IS ONE_PHASE_PULL_ALGORITHM). Note in publication 1 the TASK_NAME_KEY attribute with an EQ operator (the same attribute appear with an IS operator in subscription 1). If it was done the other way around (EQ in the subscription and IS in the publication), the publication could match other types of interests with less restrictive conditions, possibly from a different application running concurrently in the sensor network. This is particularly important in subscription 2 below.

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 node would get the task by subscribing to interests with NR::subscribe and the following attributes:

```
Subscription 2: (two-phase pull)

CLASS_KEY NE DATA_CLASS
SCOPE_KEY IS NODE_LOCAL_SCOPE
LONGITUDE_KEY IS 10
LATITUDE_KEY IS 20
TASK_NAME_KEY EQ 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.)

7.2 Using one-phase push and GEAR

There are a few things to consider when using *one-phase push* or GEAR. As subscription 1 already specifies a valid closed region, GEAR can be used to route this subscription towards nodes in the mentioned region. A valid closed region includes two latitude and two longitude attributes, specifying a closed retangle with the LT, LE, GT, and GE operators. If users wanted to specify a geographic point of interest, subscription 1 would be changed to subscription 1*A* below, which includes latitude and longitude attributes with EQ operators.

```
Subscription 1A: (two-phase pull with a geographic point of interest)
CLASS_KEY IS INTEREST_CLASS
SCOPE_KEY IS GLOBAL_SCOPE
LONGITUDE_KEY EQ 10
LATITUDE_KEY EQ 20
TASK_FREQUENCY_KEY IS 500
SENSOR_TYPE_KEY EQ seismic
TARGET_KEY IS TEL
TARGET_RANGE_KEY LE 50
TASK_NAME_KEY IS detect_track
TASK_QUERY_DETAIL_KEY IS [query_byte_code]
```

If no sensor nodes with a matching publication are located in geographic coordinates specified in subscription 1*A*, no matching will occur. Note that because GEAR is a separate program, it will need to be started along with the filter core and two_phase_pull.

If the application developer wanted to use *one-phase push* semantics, subscription 1 and publication 1 would have to be changed to subcripttion 1*B* and publication 1*B* below (the only difference is in the SCOPE_KEY attributes).

```
Subscription 1B: (one-phase push semantics)

CLASS_KEY IS INTEREST_CLASS
SCOPE_KEY IS NODE_LOCAL_SCOPE
LONGITUDE_KEY GE 10
LONGITUDE_KEY LE 50
LATITUDE_KEY GE 20
LATITUDE_KEY LE 40
TASK_FREQUENCY_KEY IS 500
SENSOR_TYPE_KEY EQ seismic
TARGET_KEY IS TEL
TARGET_RANGE_KEY LE 50
TASK_NAME_KEY IS detect_track
TASK_QUERY_DETAIL_KEY IS [query_byte_code]

Publication 1B: (one-phase push semantics)

CLASS_KEY IS DATA_CLASS
SCOPE_KEY IS GLOBAL_SCOPE
LONGITUDE_KEY IS 10
```

```
LATITUDE_KEY IS 20
TASK_NAME_KEY EQ detectTrack
TARGET_RANGE_KEY IS 40
SENSOR_TYPE_KEY IS seismic
```

Because *interest* messages remain local to the subscriber node (sink) in *one-phase push*, there is no reason for having a subscription 2 (subscription for interests) as these will not reach the sensor (soruce) node.

For the same reason, including geographic coordinates in the subscription will limit the matching data to reading from those coordinates, but will not allow GEAR to improve the algorithm performance in this case (as because *interest* messages never leave the node, there is nothing to improve). However, when using *one-phase push*, we can benefit from geographic routing in a different way. GEAR can limit the flooding of *exploratory data* messages containing either a closed geographic region or a geographic point just as it does for *interest* messages. Look at subscription $1C$ and publication $1C$ below.

```
Subscription 1C: (one-phase push semantics with GEAR)

CLASS_KEY IS INTEREST_CLASS
SCOPE_KEY IS NODE_LOCAL_SCOPE
LONGITUDE_KEY IS 45
LATITUDE_KEY IS 19
TASK_FREQUENCY_KEY IS 500
SENSOR_TYPE_KEY EQ seismic
TARGET_KEY IS TEL
TARGET_RANGE_KEY LE 50
TASK_NAME_KEY IS detect_track
TASK_QUERY_DETAIL_KEY IS [query_byte_code]

Publication 1C: (one-phase push semantics with GEAR)

CLASS_KEY IS DATA_CLASS
SCOPE_KEY IS GLOBAL_SCOPE
LONGITUDE_KEY GE 30
LONGITUDE_KEY LE 50
LATITUDE_KEY GE 15
LATITUDE_KEY LE 20
TASK_NAME_KEY EQ detectTrack
TARGET_RANGE_KEY IS 40
SENSOR_TYPE_KEY IS seismic
```

In this case, the publisher (source) node is pushing its local detections to subscribers (sinks) located in a particular region. GEAR can improve the performance in this case by limiting the flooding of the *exploratory data* messages outside this region. The use of a geographic point (instead of a closed region) is also supported.

8 API Usage Examples

This section provides examples on how to use the Network Routing Publish/Subscribe and Filter APIs.

8.1 Publish/Subscribe Example

```
Step 0: Includes and Definitions

#include "diffapp.hh" // Everything from diffusion you will need
                      // You should implement your application in
                      // a class derived from DiffApp, something like:

class MyApplication : public DiffApp
{
public:

  MyApplication(int argc, char **argv);

  // Your stuff goes here
};

Step 1: Initialization of Network Routing

{
    // This code is in the initialization routine of the
    // network routing client. It should be somewhere in
    // your class

    .
    .
    .

    // Parse diffusion related command line arguments (this is not
    // necessary, but won't hurt either). It's defined in the DiffApp
    // class
    parseCommandLine(argc, argv);

    // Create an NR instance.  This will create the NR instance
    // and return a pointer to that instance. You can use the dr_
    // pointer (defined in the DiffApp class) to store it...
    // Specifying diffusion_port_ (also defined in the DiffApp class),
    // is optional, and is used to pass the port specified in the -p
    // command line option

    dr_ = NR::createNR(diffusion_port_);

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

    .
    .
    .

    // 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()
{
    // Setup a publication with 4 attributes, using two-phase pull

    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()
{
    // (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(NRScopeAttr.make(NRAttribute::IS,
                                     NRAttribute::NODE_LOCAL_SCOPE));
    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()
{
    // (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 the attributes (they have been copied by send)
    ClearAttrs(&attrs);
    .
    .
}

Step 5:  Receiving data

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

8.2 Filter API Example

In this section we describe a simple module that uses the Filter API to receive all INTEREST messages received by the filter core. The filter calls NR::addFilter with a high priority (in order to get the packets before other filters). After logging an arrival (printing a packet has arrived along with where it came from), the Log filter returns this (unmodified) message to the filter core so other filters/applications can receive it.

Please note that the following code is a modified version of the log filter (supplied with the current diffusion release).

```
// Step 0: Includes

#include "diffapp.hh" // Includes everything it is needed from diffusion

#define LOG_FILTER_PRIORITY 210 // This is a pre-processing filter

class LogFilter : public DiffApp
{
  LogFilter(int argc, char **argv);

  // Your stuff here. It's recommended that all applications and filters
  // derive from the DiffApp class
}

// Step 1: Initialization

LogFilter::LogFilter(int argc, char **argv)
{
  // Create Diffusion Routing class
  parseCommandLine(argc, argv); // Not necessary, but recommended.
                                // Takes care of setting up debug level,
                                // diffusion port

  dr_ = NR::createNR(diffusion_port_); // Both dr_ and diffusion_port_ are
                                       // part of the DiffApp class

  filter_callback_ = new LogFilterReceiver();

  // Set up the filter
  filter_handle_ = setupFilter();
```

```
  DiffPrint(DEBUG_ALWAYS, "Log Filter received handle %d\n", filter_handle_);
  DiffPrint(DEBUG_ALWAYS, "Logging App initialized !\n");
}

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

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

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

handle LogFilter::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_FILTER_PRIORITY,
                                           filter_callback_);
  ClearAttrs(&attrs);
  return h;
}

Step 2: Handle callbacks

// Implement the Filter API callback

void LogFilterReceive::recv(Message *msg, handle h)
{
  DiffPrint(DEBUG_ALWAYS, "Received a");

  if (msg->new_message_)
    DiffPrint(DEBUG_ALWAYS, " new ");
  else
    DiffPrint(DEBUG_ALWAYS, "n old ");

  if (msg->last_hop_ != LOCALHOST_ADDR)
    DiffPrint(DEBUG_ALWAYS, "INTEREST message from node %d\n", msg->last_hop_);
   else
    DiffPrint(DEBUG_ALWAYS, "INTEREST message from agent %d\n", msg->source_port_);

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

References

[1] John Heidemann, Fabio Silva, and Deborah Estrin. Matching data dissemination algorithms to application requirements. Technical Report ISI-TR-571, USC/Information Sciences Institute, April 2003.

[2] John Heidemann, Fabio Silva, Chalermek Intanagonwiwat, Ramesh Govindan, Deborah Estrin, and Deepak Ganesan. Building efficient wireless sensor networks with low-level naming. In *Proceedings of the Symposium on Operating Systems Principles*, pages 146–159, Chateau Lake Louise, Banff, Alberta, Canada, October 2001. ACM.

[3] John Heidemann, Fabio Silva, Yan Yu, Deborah Estrin, and Padmaparma Haldar. Diffusion filters as a flexible architecture for event notification in wireless sensor networks. Technical Report ISI-TR-556, USC/Information Sciences Institute, April 2002.

[4] Chalermek Intanagonwiwat, Ramesh Govindan, and Deborah Estrin. Directed diffusion: A scalable and robust communication paradigm for sensor networks. In *Proceedings of the ACM/IEEE International Conference on Mobile Computing and Networking*, pages 56–67, Boston, MA, USA, August 2000. ACM.

[5] Fred Stann and John Heidemann. Rmst: Reliable data transport in sensor networks. In *Proceedings of the First International Workshop on Sensor Net Protocols and Applications*, page to appear, Anchorage, Alaska, USA, April 2003. USC/Information Sciences Institute, IEEE.

[6] Wei Ye, John Heidemann, and Deborah Estrin. An energy-efficient mac protocol for wireless sensor networks. In *Proceedings of the IEEE Infocom*, pages 1567–1576, New York, NY, USA, June 2002. USC/Information Sciences Institute, IEEE.