Network Routing Application Programmer's Interface (API) and Walk Through 9.0.1

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

December 9, 2002

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, 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 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/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, dated May 31st, 2002, we changed our Timer API. Please refer to section 5 for more information. Also, since the release 8.0, dated March 4th, 2001, the following changes have been made:

- GEAR, the Geographic and Energy Aware Routing protocol, is now supported in diffusion. GEAR improves diffusion efficiency when geographic information is available. See section 3.3 for more information.
- As an option, the publish/subscribe API now also supports PUSH. PUSH improves diffusion efficiency in cases where there are many subscribers and few publishers. (By default diffusion is optimized for many publishers and fewer subscribers.) For more details, see section 3.2.
- The filter API has been improved, giving filter developers more freedom when exchanging messages.
- Several of the sample applications/filters provided in our distribution have been updated. They include several small features and fixes. This code can be used as a starting poing for application/filter developers.

- Attribute creation and manipulation has been improved with the addition of a few extra functions to help diffusion application/filter developers. More details, refer to section 2.4.

1.2 Other planned changes and implementation status

Although we have iterated on this API and used it in various demos and other experiments, 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. This should be included in a future version of this API.

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 diffusion attributes and naming, see [1]. For more detail about filter invocation, see [2].

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 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 (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 (i.e. IS can be used to specify a literal blob value that can only be matched with the EQ 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).

2.5 Filter API

Filters are application-specific code that run 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. As mentioned in section 2.2, filters are a special case of

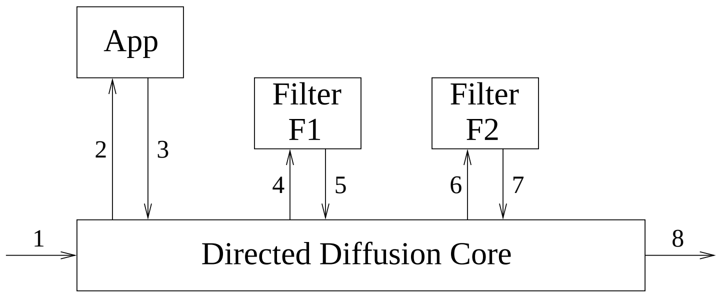

Figure 1: Message Flow in Directed Diffusion

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 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. Diffusion will not allow two filters to have the same priority, so 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 section 6.1). The message structure allows the callback 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 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 Directed Diffusion 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 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 6.1 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 SendMessage (see section 6.3). Note that the filter can return the same message it received from Diffusion, a modified message, a complete new message, multiple messages (by calling SendMessage more than one time) or even no messages (by returning from the callback without calling SendMessage). 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). What happens next depends if SendMessage is called with or without a new priority. Assuming that the priority remains the same, 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. If a new priority is specified in SendMessage, diffusion will start looking for filters whose priorities are below the specified priority. This allows filters to send messages to other filters directly, bypassing the default ordering imposed by diffusion.

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 SendMessage (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 diffusion works, see [3]. We now present a brief description of what happens when applications call Publish and Subscribe. For detailed API information, please refer to section 4.

3.1 Basic Diffusion

Each subscription (i.e. call to Subscribe) causes diffusion library to periodically send an *Interest* message (containing the subscription's attributes) to the network. These interests 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. 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).

Basic diffusion is therefore best suited to cases where there are many publishers and a few subscribers. If this situation is reversed, the PUSH option to diffusion (described next) 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 3.3.

3.2 Using PUSH

PUSH is an option to diffusion that reverses the relative costs of Publish and Subscribe. When applications use 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 PUSH are cheap since messages do not leave the Subscriber node. Publishers, on the other hand, have no way of telling if there Subscribers interested in their data therefore periodic exploratory data messages are flooded throughout the network.

PUSH offers more benefits when there are many Subscribers and/or when there are few data messages to be sent from Publishers to Subscribers.

3.3 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. Subscriptions that contain a reference to a *closed* region (i.e. have two latitude attributes and two longitude attributes, specifying a closed retangular region), are sent towards the specified region. This greatly improves diffusion's performance by avoid flooding *Interest* messages outside the subscription's region. 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.

Since *exploratory data* messages travel only through gradients, they are not going to be flooded outside the subscription's region.

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 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 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 through the network 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 PUSH semantics (please refer to section 3.2 for more information).

To find out about interests (if you're a sensor or data source), 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 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.

Note that some combinations of CLASS_KEY and SCOPE_KEY are not supported by this API. In these cases, subscribe() will return −1.

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. 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.

4.4 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 *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::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.

Note that if the application does not specify a SCOPE_KEY attribute, NR::publish will assume it to be NODE_LOCAL_SCOPE. If, however, the application wants to use PUSH semantics (please refer to section 3.2), both CLASS_KEY and SCOPE_KEY attributes need to be specified. For example:

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

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.

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 recv 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.
- 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).

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.

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

```
Subscription 1:

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
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 predefined 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:

```
Publication 1:

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:

```
Subscription 2:

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.)

7.1 Using PUSH and GEAR

There are a few things to consider when using PUSH or GEAR. Note that because Subscription 1 specifies a closed region, GEAR can be used to route this subscription towards nodes in the mentioned region. No changes are required to the application code. Note that GEAR is a separate program that needs to be started along with diffusion and gradient.

If the application developer wanted to use PUSH semantics, Subscription 1 and Publication 1 would have to be changed to:

```
Subscription 1: (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
device_type 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 1: (PUSH Semantics)

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

Note that the only difference is on the SCOPE_KEY attributes. Also, since PUSH causes interest messages to remain local to the Subscriber node, there is no reason to have Subscription 2, as there are no interest messages to match it.

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.

    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 INTEREST messages received by data diffusion. The filter calls 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 data diffusion so other filters/applications can receive them.

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 16 // This is higher than gradient/gear

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, 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.

[2] 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.

[3] 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.