Nucleus Network Management

Gilman Tolle

August 14, 2005

Contents

1 Introduction

The purpose of a network management system for you, the developer, is to make it easy to expose information about the functioning of your networked applications. The purpose of a network management system for you, the administrator, is to make it easy to gather information about the functioning of the networked application you are monitoring.

In your previous applications, you have probably written your own message types, generated `mig` classes, and written client-side Java applications just to get data out of your motes. The Nucleus system is a matched set of TinyOS components and Java tools that make exposing and collecting network information easier.

You can use Nucleus to expose information in three different ways:

- Attribute Queries: Let's say that you are writing a multihop routing component, and want to get a snapshot of the current state of the routing graph. You can expose the address and cost of each node's next hop and the list of potential next hops as Nucleus Attributes. You can then inject a Nucleus Query into the network to gather the current contents of these attributes.

  More generally, Nucleus allows the developer to associate a human-readable name with an item of data. The item of data must have a known constant length. An attribute can also be a list of data items each with known constant length, and individual items can be accessed by index. Nucleus then assigns a small integer identifier to the attribute name, so that a compact query can be constructed. The query can be submitted over the network or the serial connection, and the results can be directed to the network or the serial connection.

- RAM Queries: Without modifying your program at all, Nucleus allows you to remotely retrieve the contents of symbols in RAM. If you want to expose an attribute without much extra work, you can just create a variable to store it, and use the Nucleus RAM Query to retrieve its contents.

  For Nucleus, the name of a RAM symbol is the name of the variable, prefixed by the component name if it is a component-local variable. The identifier of a RAM symbol is the actual location in RAM. The length of the RAM symbol is the actual size of the memory referenced by the symbol. The query processor just reads the data directly from RAM when the symbol is requested.

- Log Events: Now let's say that your routing protocol is adaptive, and you want to study its adaptation pattern. You can expose a Nucleus Event that contains a message like "Node %d changed its parent to Node %d, with cost %d", and the data necessary to make this event meaningful. When the component changes parents you can signal the event, which can be stored locally or sent off the node immediately.

The Nucleus framework can save you the time and effort you would have spent on writing your own custom messages for collecting this data. The RAM and code overhead of Nucleus is small enough that you can leave the data collection framework in your program even after you have finished debugging it. Then, when your application begins to perform strangely in the field, you will be able to gather more data about what's going on.

2 Installation

This section shows how to prepare your development machine to begin using Nucleus.

2.1 Obtaining the Software

Software to get:

1. A working TinyOS development environment. You should already have this, but if you don't, go to `http://www.tinyos.net` and follow the documentation there.
2. Cygwin, if you are using Windows.
3. The `ncftp` package if you are running Cygwin. The installation of `XML::Simple` depends on it.
4. **nesC 1.2**: Nucleus makes heavy use of the nesC 1.2 attribute tags and generic components.
   (a) Download `nesc-1.2alpha??.tar.gz` from `http://sourceforge.net/projects/nescc`
   (b) Unpack it into `/opt` and change to the `nesc-1.2alpha??` directory.
   (c) Run `./configure`
   (d) Run `make`
   (e) Run `make install`
5. **Perl XML::Simple**: Nucleus also depends on the `XML::Simple` Perl module.
   (a) Run `cpan install XML::Simple`
   (b) Press the enter key through all the prompts
   (c) Unblock the `perl` program if Windows asks
   (d) When you can't press enter anymore, you have to select your continent, country, and a few servers to access. This part is only necessary if this is the first time you have run `cpan`.
   (e) Keep pressing enter until the rest is done.

Check out the `tinyos-1.x` tree from CVS.

If you already have the tree, then run `cvs update -d` in these directories:

- `tinyos-1.x/tools/make/`, which is needed by the Nucleus build process.
- `tinyos-1.x/beta/Drip/`, which contains the TinyOS components and Java tools for the Drip dissemination layer.
- `tinyos-1.x/beta/Drain/`, which contains the TinyOS components and Java tools for the Drain collection layer.
- `tinyos-1.x/contrib/nucleus/`, which contains the TinyOS components, Java tools, and build scripts for the Nucleus system.

If you don't have these directories, run `cvs update -d` in the `tinyos-1.x/beta` directory and in the `tinyos-1.x/contrib` directory.

Enable the new Make system by adding this line to your `/.bashrc`:

```
export MAKERULES=$TOSDIR/../tools/make/Makerules
```

2.2 Compiling the Java Tools

Once you have updated your CVS tree, you can compile the Nucleus Java tools.

First, you need to add the Java directories to your `$CLASSPATH` environment variable. Here is how to do this if you are running the `bash` shell. Your mileage may vary.

Insert these lines into your `$(HOME)/.bashrc`.

```
export CLASSPATH="$CLASSPATH;<full path>/tinyos-1.x/beta/Drip/tools/java/"
export CLASSPATH="$CLASSPATH;<full path>/tinyos-1.x/beta/Drain/tools/java/"
export CLASSPATH="$CLASSPATH;<full path>/tinyos-1.x/contrib/nucleus/tools/java/"
```

NOTE: You must replace the `<full path>` in those lines with the full path to your TinyOS tree. If you are running in Cygwin, you MUST include the Windows drive specifier, like this: `C:/Program Files/UCB/cygwin/opt`. Forward slashes are preferred.

Once you have edited `.bashrc`, type `source $HOME/.bashrc`, or close and re-open your terminal window.

Then, you can compile the Java tools. Run `make` in each of the following directories:

1. `tinyos-1.x/beta/Drip/tools/java/net/tinyos/drip`
2. `tinyos-1.x/beta/Drain/tools/java/net/tinyos/drain`
3. `tinyos-1.x/contrib/nucleus/tools/java/net/tinyos/nucleus`

2.3 Compiling a TinyOS Program with Nucleus

Now, we are going to modify an TinyOS application to compile with Nucleus. If you are doing this for the first time, try this on the `CntToLeds` application in `tinyos-1.x/apps`. You will perform these steps for every application that you want to make Nucleus-enabled.

Add the following lines to your application's `Makefile`:

```
TOSMAKE_PATH += $(TOSDIR)/../contrib/nucleus/scripts

CFLAGS += -I$(TOSDIR)/../beta/Drip
CFLAGS += -I$(TOSDIR)/../beta/Drain
CFLAGS += -I$(TOSDIR)/../contrib/nucleus/tos/lib/Nucleus
```

Make sure to place them before the `include ... Makerules` line. Also, make sure you use CFLAGS, not PFLAGS. The Nucleus build process needs to place an include before the includes specified here, and it depends on setting PFLAGS for this.

Now, to compile your application with Nucleus, use this command:

```
make <platform> nucleus
```

The application should compile normally, with some extra messages in the build process.

This procedure makes it possible to compile an application with Nucleus. The next section will show you how to actually add the Nucleus components to your application, and how to gather data from the application.

3 RAM Queries

The simplest use of Nucleus is to retrieve RAM symbols from your running application.

3.1 Preparing your TinyOS Application

Enable the Nucleus query components by adding the following lines to your application's top-level configuration, usually located in `<your application name>.nc`:

```
components MgmtQueryC;
Main.StdControl -> MgmtQueryC;
```

Including `MgmtQueryC` will also include the rest of the components needed by Nucleus.

Then, compile and install your application on a mote:

```
make <platform> nucleus install,<nodeid>
```

3.2 Using the Java Query Tools

Now that your application has become Nucleus-enabled and has been installed, you can retrieve the values of RAM symbols.

First, attach a SerialForwarder to your mote. Use the following command:

```
java net.tinyos.sf.SerialForwarder -comm <your MOTECOM> &
```

For details on the `MOTECOM`, see the TinyOS Tutorial.

Nucleus provides a command-line query tool that you can use to retrieve RAM sybols (and attributes). For now, we will be running this tool directly, but you could also consider running it from another script as part of a larger data-gathering system.

The Nucleus Query tool is called `net.tinyos.nucleus.NucleusQuery`. To test that the query tool is correctly set up, run the following command:

```
java net.tinyos.nucleus.NucleusQuery
```

Now, make an alias for it by adding the following line to your `.bashrc`:

```
alias nquery='java net.tinyos.nucleus.NucleusQuery'
```

Run `source $HOME/.bashrc` or re-open your terminal window.

The tool needs to access a file called `nucleusSchema.xml` that has been generated by the build process and placed in your TinyOS application's `build/<your platform>/` directory.

For now, make sure to run the Nucleus Query tool from the application's `build/<your platform>/` directory. If you would prefer to run the query tools from the application's main directory, then you can add this option to the `nquery` command line: `-f build/<your platform>/nucleusSchema.xml`.

To get a list of the available RAM symbols, you can open the `nucleusSchema.xml` in a text editor. This file also contains lists of attributes and events. For the list of RAM symbols, find the `<symbols>` section. Global symbols like `TOS_LOCAL_ADDRESS` keep their original names and symbols within components are prefixed with the name of the component, like `Counter.state`.

Choose the name of the RAM symbol you want to retrieve.

We are going to start by querying the mote that you just installed. This mote is directly attached to the serial port. By default, the Nucleus Query tool disseminates queries to the entire network using Drip, and retrieves responses over the Drain collection tree. Because we want to query a directly-attached mote, you run `nquery` with a few extra flags, like this:

```
nquery -s link -d serial <RAM Symbol Name>
```

The `-s link` flag specifies that you are injecting the query to the mote attached to the local link. The `-d serial` flag indicates that the query response should also be sent over the serial port.

One second later, which is the default delay, you should see results looking like this:

```
<nodeid>: <RAM Symbol Name> = <RAM Symbol Value>
```

You can also submit a query for multiple RAM symbols at the same time:

```
nquery -s link -d serial <RAM Symbol A> <RAM Symbol B> ...

<nodeid>: <RAM Symbol A> = <RAM Symbol A Value>
<nodeid>: <RAM Symbol B> = <RAM Symbol B Value>
```

The lack of types for RAM symbols results in some important limitations on Nucleus RAM Query:

- Every RAM symbol will be interpreted as an unsigned integer.
- RAM symbols representing structures and arrays will be returned in their entirety, but will not be further parsed by Nucleus Query.
- To display a RAM symbol as a sequence of hex bytes instead of as an integer, add the `-b` flag to the `nquery` command line. You can then use other tools to reconstruct the data.

4 Attribute Queries

In addition to implicitly exposing your data through RAM symbols, you can explicitly expose a piece of data as a Nucleus Attribute. Exposing your data as a Nucleus Attribute gives you some additional advantages over exposing it as a RAM symbol:

- Your data will be exposed with a type, which can be used by the tools to parse the bytes into a meaningful structure.
- You can expose a list of items as a single attribute, and retrieve individual elements of the list. This allows you to collect things like a neighbor table or a received messages counter for each different Active Message type.

Nucleus attributes are not tied to a specific compilation of an application. If you are using RAM symbols and you change your application, the location of every RAM symbol may change. Then, if some nodes are running version 1 of your application and other nodes are running version 2 of the application, different data items will be returned to the same query.

Nucleus attributes are more abstract than RAM symbols. They may be more appropriate when you want to separate the name of the attribute from the name of the variable, or when you want different programs to export the same attribute.

4.1 Exposing an Attribute in TinyOS

This section explains what you need to change in your application in order to expose a Nucleus attribute. Nucleus Attributes are always exported by nesC modules. The easiest way to export an attribute is to add it to the module that contains the data. As an illustrative example, consider a routing component that looks like this:

```
module RoutingM {
  [uses some communication components, etc]
}
implementation {
  uint16_t currentParent;

  [has some routing logic to change that parent]
}
```

We'll expose the current parent as a Nucleus Attribute. These are the basic steps:

1. Decide on a name for the attribute (i.e. `RoutingParent`).
2. Expose the attribute.
3. Write code to give it out when it is asked for.

There are three changes you have to make to your module `.nc` file:

1. Add this line to the top of the file:

   ```
   includes Attrs;
   ```

2. Add a line to the module definition, like this:

   ```
   module RoutingM {
     provides interface Attr<uint16_t> as RoutingParent @nucleusAttr("RoutingParent");
   }
   ```

   This line does four things:

   - It provides a standard interface that can be used to access the attribute. (`Attr`)
   - It associates a type with the attribute. (`uint16_t`) The `< >` brackets are the exact syntax you must use, because this type is being used as an argument to a nesC 1.2 generic interface. In this example, we are creating an interface `Attr` of type `uint16_t`.
   - It sets a local name for the `Attr` interface by using the `as` command.
   - It marks the provided interface as an exposed attribute with the `@nucleusAttr()` tag. Remove this tag, and the Nucleus system will not expose the attribute. The argument to this tag becomes the human-readable name of the attribute. (`RoutingParent`)

3. Then, add code to respond with the attribute when it is requested.

   ```
   implementation {
     uint16_t currentParent;

     [has some routing logic to change that parent]

     command result_t RoutingParent.get(uint16_t* buf) {
       memcpy(buf, &currentParent, sizeof(uint16_t));
       signal RoutingParent.getDone(buf);
       return SUCCESS;
     }
   }
   ```

   This piece of code does four things:

   - It implements a command that will be called by the query system when the attribute is requested.
   - It copies the attribute value from a module variable into the given buffer pointer. Note that you must use `memcpy` to copy the attribute. Using `=`, like `*buf = currentParent;` will not work due to pointer alignment restrictions.
   - It sends a signal to indicate that the attribute has been copied.
   - It returns `SUCCESS` to the querier to indicate that the attribute has been successfully retrieved.

And that's it. During compilation, the Nucleus scripts will:

1. find every exposed attribute in your application by looking for the `@nucleusAttr()` tags
2. assign each attribute a unique numeric identifier
3. generate a nesC configuration that wires each of them into the Attribute dispatching component.

If your attribute is not wired to anything, the code will be eliminated by the nesC compiler. When you compile without `make <platform> nucleus` the attributes will not be included. Future releases will include the ability to select which attributes will be included in the application.

Now, reinstall your application with `make <platform> nucleus install,<nodeid>`.

4.2 Using the Java Query Tools

Querying for Attributes is very similar to querying for RAM symbols. After building your application, open the `nucleusSchema.xml` file in a text editor. The `<attributes>` section contains the list of available attributes, with names and types as specified in the program.

As described in section 3.2, you must first run `SerialForwarder` and set up the `nquery` alias.

4.2.1 Retrieving Attributes from One Mote

You can retrieve the value of an Attribute from your attached node with this command:

```
nquery -s link -d serial <Attribute Name>
```

You should see output like this:

```
<nodeid>: <Attribute Name> = <Attribute Value>
```

You can submit queries for multiple Attributes at the same time:

```
nquery -s link -d serial <Attribute Name A> <Attribute Name B> ...
```

You should see output like this:

```
<nodeid>: <Attribute Name A> = <Attribute Value A>
<nodeid>: <Attribute Name B> = <Attribute Value B>
```

To keep Nucleus Query as simple as possible, the size of a Nucleus Query response is limited to a single message. This results in the following limitations:

- If the value of a single Attribute or RAM symbol is too large to fit in a single message, it will not be returned.
- If the combined size of the Attribute or RAM symbol values is too large to fit in a single message, values at the end of the list will not be returned.

4.2.2 Retrieving Attributes from a Whole Network

If you want to retrieve Attributes or RAM symbols from the entire network, you must take the following steps:

1. Attach a mote running the `TOSBase` application to your PC. To query a whole network with Nucleus, you have to inject the queries and retrieve the responses through a `TOSBase`.
2. Build a Drain collection tree by running the following Java command:

   ```
   java net.tinyos.drain.Drain
   ```

Once this command has completed, the tree is built. It will take a few seconds to run. This tree will then be used to collect values from remote nodes. It will persist until you build another tree. If you start to have trouble retrieving data from the network, the environment may have shifted enough to justify building a new tree. You may also want to rebuild the tree before each query, if you query infrequently enough.

3. Run the Nucleus Query command without the -s and -d flags:

   ```
   nquery <Attribute Name> ...
   ```

The `nquery` tool waits for a specified amount of time to collect responses before returning the results. Set this response delay with the `-t <delay in 10ths of a second>` flag. The default value is `10`, or one second.

This flag sets two different things: how long each node will wait before responding, and how long the tool will wait before assuming that each node has responded. Nodes respond at a random time less than or equal to the delay. The tool waits an additional second te account for routing delays. If your network has many nodes, response traffic sent with the default delay may overwhelm the network. You may want to increase the delay in these situations.

4.3 More Ways to Expose Attributes

4.3.1 Exposing List Attributes

To expose an attribute that represents a list of data items, such as a neighbor table, you must use a slightly different interface in your TinyOS component. This code shows you how to do it:

```
module RoutingM {
  <uses some communication components, etc>

  provides interface AttrList<uint16_t> as RoutingNeighbors @nucleusAttr("RoutingNeighbors");
}
```

Note that you are now providing an `AttrList` interface instead of an `Attr` interface. Every element in the list must be of the specified type. The rest is unchanged.

```
implementation {
  uint16_t neighbors[ROUTING_MAX_NEIGHBORS];

  <has some logic to fill that table>

  command result_t RoutingNeighbors.get(uint16_t *buf, uint8_t pos) {
    if (pos >= ROUTING_MAX_NEIGHBORS) {
      return FAIL;
    }
    memcpy(buf, &neighbors[pos], sizeof(uint16_t));
    signal RoutingNeighbors.getDone(buf);
    return SUCCESS;
  }
}
```

Note that the `AttrList.get()` command provides a buffer pointer as well as an index. This is the position that is being requested. Check whether the position is valid, then copy the element at that position into the buffer. The internal representation of your list is your choice.

To retrieve an element of a list-structured attribute, use the following `nquery` command:

```
nquery <Attribute Name>.<list index integer>
```

4.3.2 Exposing Split-Phase Attributes

If your attribute's value must be obtained through a split-phase operation, like a sensor reading or the result of a complex calculation, you can fill the attribute buffer in a split-phase fashion. Here is some example code:

```
module TempSensorM {
  provides interface Attr<uint16_t> as Temperature @nucleusAttr("Temperature");
  uses interface ADC;
}
```

Exposing an split-phase attribute uses the same interface as a non-split-phase attribute.

```
implementation {

  uint16_t *savedBuf;
  uint16_t savedReading;

  command result_t Temperature.get(uint16_t *buf) {
    // make sure that only one request is outstanding
    if (savedBuf != NULL)
      return FAIL;

    savedBuf = buf;
    call ADC.getData();
    return SUCCESS;
  }
}
```

You will have to save the buffer pointer given to the `get()` command, so that you can fill it when the data is ready.

```
  async event result_t ADC.dataReady(uint16_t data) {
    savedReading = data;
    post TemperatureTask();
    return SUCCESS;
  }

  task void TemperatureTask() {
    memcpy(savedBuf, &savedReading, sizeof(uint16_t));
    signal Temperature.getDone(savedBuf);
    savedBuf = NULL;
  }
}
```

Once the data is ready, copy it into the buffer and signal the `sendDone` event for the attribute.

4.3.3 Exposing Globally-Unique Attributes

Normally, the Nucleus scripts automatically assign a number to each attribute. This number is only unique within the application, meaning that in a heterogeneous network, a different attribute may be given the same number as your attribute. Then, your query will retrieve strange data.

To solve this problem, you can assign your own number to an attribute. This is very similar to the current TinyOS practice of assigning `AM` types. Here is some example code:

```
[... in some .h file ...]

enum {
  ATTR_ROUTINGPARENT = <some 16-bit integer>
};
```

Like `AM`, you should make an enumerated type to represent that number. The prefix doesn't really matter, but it may be useful to mark this value as an attribute identifier.

```
[... in a module .nc file ...]

includes <the .h file that defines the number>

module RoutingM {
  <uses some communication components, etc>

  provides interface Attr<uint16_t> as RoutingParent
    @nucleusAttr("RoutingParent", ATTR_ROUTINGPARENT);
}
```

The only difference here is that the identifier has been passed as an argument to the `@nucleusAttr` tag. This enables the build scripts to use your number instead of the default number.

By placing the number in a `.h` file, you make it possible for any component that wants to expose the `ATTR_ROUTINGPARENT` attribute to give your number to the `@nucleusAttr()` tag.

5 Event Logging

This section will be filled in once the Event Logging system is complete. But, the Query system works now!