Network Protocols

From TinyOS Wiki

This lesson introduces the two basic network primitives of Tinyos-2: Dissemination and Collection.

Contents

Dissemination

The goal of a dissemination protocol is to reliably deliver a piece of data to every node in the network. It allows administrators to reconfigure, query, and reprogram a network. Reliability is important because it makes the operation robust to temporary disconnections or high packet loss. Dissemination is fully explained in TEP 118.

In TinyOS 2.x, dissemination provides two interfaces: DisseminationValue and DisseminationUpdate. Let's take a look at these two interfaces:
tos/lib/net/DisseminationUpdate.nc:

```
interface DisseminationUpdate<t> {
  command void change(t* newVal);
}
```

tos/lib/net/DisseminationValue.nc:

```
interface DisseminationValue<t> {
  command const t* get();
  event void changed();
}
```

DisseminationUpdate is used by producers. The command ***DisseminationUpdate.change()*** should be called each time the producer wants to disseminate a new value, passing this new value as a parameter. DisseminationValue is for consumers. The event ***DisseminationValue.changed()*** is signalled each time the disseminated value is changed (the producer has called ***change***), and the command ***get*** allows to obtain this new value.

We build now a simple application (EasyDissemination) where one node (the producer) periodically disseminates the value of a counter to rest of the nodes in the network (consumers). As a first step, create a new directory in `apps` named `EasyDissemination`:

```
$ cd tinyos-2.x/apps
$ mkdir EasyDissemination
```

Inside this directory, create a file `EasyDisseminationC.nc`, which has this code:

```
#include <Timer.h>

module EasyDisseminationC {
  uses interface Boot;
  uses interface SplitControl as RadioControl;
  uses interface StdControl as DisseminationControl;
  uses interface DisseminationValue<uint16_t> as Value;
  uses interface DisseminationUpdate<uint16_t> as Update;
  uses interface Leds;
  uses interface Timer<TMilli>;
}

implementation {

  uint16_t counter;

  task void ShowCounter() {
    if (counter & 0x1)
      call Leds.led0On();
    else
      call Leds.led0Off();
    if (counter & 0x2)
      call Leds.led1On();
    else
      call Leds.led1Off();
    if (counter & 0x4)
      call Leds.led2On();
    else
      call Leds.led2Off();
  }

  event void Boot.booted() {
    call RadioControl.start();
  }

  event void RadioControl.startDone(error_t err) {
    if (err != SUCCESS)
      call RadioControl.start();
    else {
      call DisseminationControl.start();
      counter = 0;
      if ( TOS_NODE_ID  == 1 )
        call Timer.startPeriodic(2000);
    }
  }

  event void RadioControl.stopDone(error_t err) {}

  event void Timer.fired() {
    counter = counter + 1;
    // show counter in leds
    post ShowCounter();
    // disseminate counter value
    call Update.change(&counter);
  }

  event void Value.changed() {
    const uint16_t* newVal = call Value.get();
    // show new counter in leds
    counter = *newVal;
    post ShowCounter();
  }
}
```

We assume that the base station is the node with ID = 1. First note that the base station will periodically (every 2 seconds) increment a 3-bit counter, display the counter using its three leds, and disseminate it through the network. This is done using the change command provided in the DisseminationUpdate interface:

```
call Update.change(&counter);
```

Second, note that when a node receives a change notification, it updates its counter value and shows it on the leds:

```
event void Value.changed() {
    const uint16_t* newVal = call Value.get();
    // show new counter in leds
    counter = *newVal;
    post ShowCounter();
  }
```

The `EasyDisseminationAppC.nc` provides the needed wiring:

```
configuration EasyDisseminationAppC {}
implementation {
  components EasyDisseminationC;

  components MainC;
  EasyDisseminationC.Boot -> MainC;

  components ActiveMessageC;
  EasyDisseminationC.RadioControl -> ActiveMessageC;

  components DisseminationC;
  EasyDisseminationC.DisseminationControl -> DisseminationC;

  components new DisseminatorC(uint16_t, 0x1234) as Diss16C;
  EasyDisseminationC.Value -> Diss16C;
  EasyDisseminationC.Update -> Diss16C;

  components LedsC;
  EasyDisseminationC.Leds -> LedsC;

  components new TimerMilliC();
  EasyDisseminationC.Timer -> TimerMilliC;
}
```

Note that both Dissemination interfaces we use are provided by the module DisseminatorC.
This module provides the Dissemination service:
tos/lib/net/Dissemination/DisseminationC.nc:

```
 generic configuration DisseminatorC(typedef t, uint16_t key) {
  provides interface DisseminationValue<t>;
  provides interface DisseminationUpdate<t>;
}
```

Note that we need to specify to the Disseminartor module a type t and a key. In our case, the value we want to disseminate is just an unsigned two-byte counter. The key allows to have different instances of DisseminatorC.

To compile this program we use the following Makefile:

```
COMPONENT=EasyDisseminationAppC
CFLAGS += -I$(TOSDIR)/lib/net \
          -I$(TOSDIR)/lib/net/drip

include $(MAKERULES)
```

Now install this program into several nodes (make sure you have one base station, that is, one node whose ID is 1) and see how the counter displayed in the base station is "disseminated" to all the nodes belonging to the network. You will also notice that dissemination works across resets, i.e., if you reset a node it will rapidly re-'synchronize' and display the correct value after it reboots.

For more information, read TEP118 [Dissemination].

Collection

Collection is the complementary operation to disseminating and it consists in "collecting" the data generated in the network into a base station. The general approach used is to build one or more collection *trees*, each of which is rooted at a base station. When a node has data which needs to be collected, it sends the data up the tree, and it forwards collection data that other nodes send to it.

We build now a simple application (EasyCollection) where nodes periodically send information to a base station which collects all the data.
As a first step, create a new directory in `apps` named `EasyCollection`:

```
$ cd tinyos-2.x/apps
$ mkdir EasyCollection
```

Inside this directory, create a file `EasyCollectionC.nc`, which has the following code:

```
#include <Timer.h>

module EasyCollectionC {
  uses interface Boot;
  uses interface SplitControl as RadioControl;
  uses interface StdControl as RoutingControl;
  uses interface Send;
  uses interface Leds;
  uses interface Timer<TMilli>;
  uses interface RootControl;
  uses interface Receive;
}
implementation {
  message_t packet;
  bool sendBusy = FALSE;

  typedef nx_struct EasyCollectionMsg {
    nx_uint16_t data;
  } EasyCollectionMsg;

  event void Boot.booted() {
    call RadioControl.start();
  }

  event void RadioControl.startDone(error_t err) {
    if (err != SUCCESS)
      call RadioControl.start();
    else {
      call RoutingControl.start();
      if (TOS_NODE_ID == 1)
        call RootControl.setRoot();
      else
        call Timer.startPeriodic(2000);
    }
  }

  event void RadioControl.stopDone(error_t err) {}

  void sendMessage() {
    EasyCollectionMsg* msg =
      (EasyCollectionMsg*)call Send.getPayload(&packet, sizeof(EasyCollectionMsg));
    msg->data = 0xAAAA;

    if (call Send.send(&packet, sizeof(EasyCollectionMsg)) != SUCCESS)
      call Leds.led0On();
    else
      sendBusy = TRUE;
  }
  event void Timer.fired() {
    call Leds.led2Toggle();
    if (!sendBusy)
      sendMessage();
  }
```

```
  event void Send.sendDone(message_t* m, error_t err) {
    if (err != SUCCESS)
      call Leds.led0On();
    sendBusy = FALSE;
  }

  event message_t*
  Receive.receive(message_t* msg, void* payload, uint8_t len) {
    call Leds.led1Toggle();
    return msg;
  }
}
```

Lets take a look at this program. First note that all nodes turn on the radio into the Boot sequence:

```
  event void Boot.booted() {
    call RadioControl.start();
  }
```

Once we are sure that the radio is on, we start the routing sub-system (that is, to generate the collection *tree*):

```
call RoutingControl.start();
```

Next we need to specify the root of the collection tree, that is, the node that will receive all the data packets. For this, we use the interface RootControl:
tos/lib/net/RootControl.nc

```
interface RootControl {
    command error_t setRoot();
    command error_t unsetRoot();
    command bool isRoot();
}
```

This interface controls whether the current node is a root of the tree. Using the setRoot() command and assuming that the base station ID is 1, we select the root of the collection *tree* as follows:

```
if (TOS_NODE_ID == 1)
    call RootControl.setRoot();
else
    call Timer.startPeriodic(2000);
```

The remaining nodes in the network periodically generate some data and send it to the base station. To send and receive data we use two interfaces that will be wired to the collection tree. That is, when we call the send command, the data packet will be sent through the collection tree. Similarly, the receive event will be only called in the root of the tree, that is, in the base station. When the base station receives a "collected" packet it just toggle a led. Now we will see how to wire these interfaces .
The `EasyCollectionAppC.nc` provides the needed wiring:

```
configuration EasyCollectionAppC {}
implementation {
  components EasyCollectionC, MainC, LedsC, ActiveMessageC;
  components CollectionC as Collector;
  components new CollectionSenderC(0xee);
  components new TimerMilliC();

  EasyCollectionC.Boot -> MainC;
  EasyCollectionC.RadioControl -> ActiveMessageC;
  EasyCollectionC.RoutingControl -> Collector;
  EasyCollectionC.Leds -> LedsC;
```

```
  EasyCollectionC.Timer -> TimerMilliC;
  EasyCollectionC.Send -> CollectionSenderC;
  EasyCollectionC.RootControl -> Collector;
  EasyCollectionC.Receive -> Collector.Receive[0xee];
}
```

Most of the collection interfaces (RoutingControl, RootControl and Receive) are provided by the CollectionC module. The send interface is provided by CollectionSenderC which is a virtualized collection sender abstraction module.
This is an extract of the signature of the CollectionC module and CollectionSenderC:
tos/lib/net/ctp/CollectionC.nc

```
configuration CollectionC {
  provides {
    interface StdControl;
    interface Send[uint8_t client];
    interface Receive[collection_id_t id];
    interface Receive as Snoop[collection_id_t];
    interface Intercept[collection_id_t id];

    interface Packet;
    interface CollectionPacket;
    interface CtpPacket;

    interface CtpInfo;
    interface CtpCongestion;
    interface RootControl;
  }
```

tos/lib/net/ctp/CollectionSenderC:

```
  generic configuration CollectionSenderC(collection_id_t collectid) {
  provides {
    interface Send;
    interface Packet;
  }
```

Note that the sender and receive interfaces requires a collection_id_t to differentiate different possible collections trees.
Note also that the CollectionC module provides some other interfaces in addition to the ones used in this example. As we explained previously, the CollectionC module generates a collection tree that will be using for the routing. These interfaces can be used get information or modify this routing tree. For instance, if we want to obtain information about this tree we use the CtpInfo interface (see tos/lib/net/ctp/CtpInfo.nc) and if we want to indicate/query if any node/sink is congested we use the CtpCongestion interface (see tos/lib/net/ctp/CtpCongestion.nc)

Finally, to compile this program we create the following Makefile:

```
COMPONENT=EasyCollectionAppC
CFLAGS += -I$(TOSDIR)/lib/net \
          -I$(TOSDIR)/lib/net/le \
          -I$(TOSDIR)/lib/net/ctp
include $(MAKERULES)
```

Now install this program into several nodes (make sure you have one base station, that is, one node whose ID is 1) and see how all the packets generated in the nodes are collected in the base station.

For more information, read TEP119 [Collection].

To experiment further

If you want to experiment with a more complex application take a look at apps/tests/TestNetwork/ which combines dissemination and collection into a single application.

For information about how to specify Device ID, please refer to Mote-mote radio communication or Lesson 4: Component Composition and Radio Communication[1] (http://www.tinyos.net/tinyos-1.x/doc/tutorial/lesson4.html)

Related Documentation

- TEP 118: Dissemination (http://www.tinyos.net/tinyos-2.x/doc/html/tep118.html)
- TEP 119: Collection (http://www.tinyos.net/tinyos-2.x/doc/html/tep119.html)

< **Previous Lesson** | **Top** | **Next Lesson** >

Retrieved from "http://tinyos.stanford.edu/tinyos-wiki/index.php?title=Network_Protocols&oldid=4461"

- This page was last modified on 9 September 2010, at 06:18.
- This page has been accessed 66,530 times.