Dissemination

From TinyOS Wiki

This lesson introduces the basic network dissemination primitive of Tinyos-2.

Contents

Introduction

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.

Implementations

There are three kinds of dissemination libraries for small data items in TinyOS 2.x: Drip, DIP, and DHV. They all provide an eventual consistency model and use Trickle timers underneath. Drip treats every data item as a separate entity for dissemination, and thus provides fine grain control of when and how quickly you want data items disseminated. DIP and DHV treat them as a group, meaning dissemination control and parameters apply to all data items collectively. Below is a summary.

	Radio Control	**Timing**	**Messaging**	**Data Size**
Drip	Application responsible for enabling/disabling radio for dissemination.	Trickle timers configured through **DisseminationTimerP** component. The tau values apply to all dissemination items.	Each data item is independently advertised and disseminated. Metadata is not shared among data items, meaning nodes do not need to agree on data sets a priori.	Data size controlled through typedef **t**. Must be smaller than the message payload size.
DIP	Application responsible for enabling/disabling radio for dissemination.	Single Trickle timer for disseminating all data items. Tau values configured in **Dip.h**.	Advertisement messages are used for a fixed data set meaning all nodes must agree on a fixed set of data item identifiers before dissemination. DIP advertisements can be adjusted based on message size through **Dip.h**	Data type is specified through typedef **t**, but the size is backed by an nx_struct **dip_data_t** in **Dip.h** (default: 16 bytes). The size of the nx_struct must be less than the message payload size.
DHV	DHV automatically starts the radio. Manually turning	Single Trickle timer for disseminating all data items. Tau values configured in **Dhv.h**.	Advertisement messages are used for a fixed data set meaning all nodes must agree on a fixed set of data item identifiers before dissemination. DHV	Data type is specified through typedef **t**, but the size is backed by an nx_struct **dhv_data_t** in **Dhv.h** (default: 16 bytes). The

	off the radio will prevent DHV from working though.		advertisements can be adjusted based on message size through **Dhv.h**	size of the nx_struct must be less than the message payload size.

To summarize, Drip should be used when you have few items and uncertainty of what items each node has. This flexibility requires much more messaging to occur throughout your network. DIP/DHV should be used when all nodes agree on a data set and high message efficiency is needed. In most cases, DHV uses fewer transmitted messages and converges the network twice faster than DIP.

Interfaces

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.

Setting up the application

Now let's build a simple dissemination application that updates two data item every 5 seconds. When nodes update their data item, they will toggle LEDs based on the parity of the payload. First create a new directory in the `apps` directory called `EasyDissemination`.

Configurations

Inside the `EasyDissemination` directory, create a file named `EasyDisseminationAppC.nc` with the following code (pick the correct column depending if you want to use Drip or DIP):

Drip/DIP

```
configuration EasyDisseminationAppC {}
implementation {
  components MainC;
  EasyDisseminationC.Boot -> MainC;
  components LedsC;
  EasyDisseminationC.Leds -> LedsC;
  components new TimerMilliC();
  EasyDisseminationC.Timer -> TimerMilliC;

  components EasyDisseminationC;
  components DisseminationC;
  EasyDisseminationC.DisseminationControl -> DisseminationC;

  components new DisseminatorC(uint16_t, 0x1234) as Diss16C;
  EasyDisseminationC.Value1 -> Diss16C;
  EasyDisseminationC.Update1 -> Diss16C;
  components new DisseminatorC(uint8_t, 0x5678) as Diss8C;
  EasyDisseminationC.Value2 -> Diss8C;
  EasyDisseminationC.Update2 -> Diss8C;
```

DHV

```
configuration EasyDisseminationAppC {}
implementation {
  components MainC;
  EasyDisseminationC.Boot -> MainC;
  components LedsC;
  EasyDisseminationC.Leds -> LedsC;
  components new TimerMilliC();
  EasyDisseminationC.Timer -> TimerMilliC;

  components EasyDisseminationC;
  components DisseminationC;
  EasyDisseminationC.DisseminationControl -> DisseminationC;

  components new DisseminatorC(uint16_t, 0x1234) as Diss16C;
  EasyDisseminationC.Value1 -> Diss16C;
  EasyDisseminationC.Update1 -> Diss16C;
  components new DisseminatorC(uint8_t, 0x5678) as Diss8C;
  EasyDisseminationC.Value2 -> Diss8C;
  EasyDisseminationC.Update2 -> Diss8C;
```

```
  components ActiveMessageC;
  EasyDisseminationC.RadioControl -> ActiveMessageC;

}
```

```
}
```

The first set of components are specific to our test application. It is used to boot nodes, start the 5-second timer, and control the LEDs.

The second set of components connects the application to the **DisseminationC** component, which provides the **StdControl** interface. Through this interface, the application can start and stop the dissemination component. Drip, DIP, and DHV all use this component.

The third set of components connects the application to a set of **DisseminatorC**s. Each **DisseminatorC** is a data item that the application wants disseminated. The two parameters to the generic component are: the typedef and the key. The typedef is a C type, which implicitly specifies the size of the data, and the key is a network wide unique identifier for the object.

The fourth set of components is required only by Drip. It enables and disables the radio. DIP and DHV do not require this because it enables the radio at bootup, and does not expect the user to manually control it.

Modules

Now let's look at how we write our actual application.

Drip/DIP

```
#include <Timer.h>

module EasyDisseminationC {
  uses interface Boot;
  uses interface Leds;
  uses interface Timer<TMilli>;

  uses interface StdControl as DisseminationControl;
  uses interface DisseminationValue<uint16_t> as Value1;
  uses interface DisseminationUpdate<uint16_t> as Update1;
  uses interface DisseminationValue<uint8_t> as Value2;
  uses interface DisseminationUpdate<uint8_t> as Update2;

  uses interface SplitControl as RadioControl;
}

implementation {
  uint16_t counter1;
  uint8_t counter2;

  task void showCounter() {
    if (counter1 & 0x1)
      call Leds.led0On();
    else
      call Leds.led0Off();

    if (counter2 & 0x1)
      call Leds.led2On();
    else
      call Leds.led2Off();
  }

  event void Timer.fired() {
    if ( TOS_NODE_ID  == 1 ) {
      counter1 = counter1 + 1;
      counter2 = counter2 + 1;
      call Update1.change(&counter1);
      call Update2.change(&counter2);
    }
  }

  event void Value1.changed() {
    const uint16_t* newVal = call Value1.get();
    if (TOS_NODE_ID != 1) {
      counter1 = *newVal;
    }
    post showCounter();
  }

  event void Value2.changed() {
    const uint8_t* newVal = call Value2.get();
```

DHV

```
#include <Timer.h>

module EasyDisseminationC {
  uses interface Boot;
  uses interface Leds;
  uses interface Timer<TMilli>;

  uses interface StdControl as DisseminationControl;
  uses interface DisseminationValue<uint16_t> as Value1;
  uses interface DisseminationUpdate<uint16_t> as Update1;
  uses interface DisseminationValue<uint8_t> as Value2;
  uses interface DisseminationUpdate<uint8_t> as Update2;


}

implementation {
  uint16_t counter1;
  uint8_t counter2;

  task void showCounter() {
    if (counter1 & 0x1)
      call Leds.led0On();
    else
      call Leds.led0Off();

    if (counter2 & 0x1)
      call Leds.led2On();
    else
      call Leds.led2Off();
  }

  event void Timer.fired() {
    if ( TOS_NODE_ID  == 1 ) {
      counter1 = counter1 + 1;
      counter2 = counter2 + 1;
      call Update1.change(&counter1);
      call Update2.change(&counter2);
    }
  }

  event void Value1.changed() {
    const uint16_t* newVal = call Value1.get();
    if (TOS_NODE_ID != 1) {
      counter1 = *newVal;
    }
    post showCounter();
  }

  event void Value2.changed() {
    const uint8_t* newVal = call Value2.get();
```

```
    if (TOS_NODE_ID != 1) {
      counter2 = *newVal;
    }
    post showCounter();
  }

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

  event void RadioControl.startDone(error_t err) {
    if (err != SUCCESS)
      call RadioControl.start();
    else {
      call DisseminationControl.start();
      counter1 = counter2 = 0;
      if ( TOS_NODE_ID  == 1 )
        call Timer.startPeriodic(2000);
    }
  }

  event void RadioControl.stopDone(error_t er) {}

}
```

```
    if (TOS_NODE_ID != 1) {
      counter2 = *newVal;
    }
    post showCounter();
  }

  event void Boot.booted() {
    call DisseminationControl.start();
    counter1 = counter2 = 0;
    if ( TOS_NODE_ID  == 1 )
      call Timer.startPeriodic(2000);
  }

}
```

In the module portion, the second set of interfaces controls dissemination, and uses the interfaces provided by the **DisseminatorC** components. Drip also has an extra interface for controlling the radio.

In the rest of the implementation, we first create two counters and a task to update the LEDs based on the counters. Then we create a timer event to update the the counters if the node ID is 1. The change function takes a pointer to the new data, which is then copied into the dissemination buffers.

Then for each value, we write an event handler for when the data has changed. This can either be from the network or from calling the change function. Because of this we choose to update the value only if we do not have node ID 1 (though the updated value should presumably be the same in that case). Once updated, we post a task to show the result on the LEDs.

The last piece of the code initializes our application. Because Drip requires the application to start the radio manually, it must start the radio before starting dissemination during bootup. DIP/DHV, in contrast, does this for the application, meaning it just needs to start dissemination when it boots.

Makefile

To compile the program, we use the following Makefile:

Drip

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

include $(MAKERULES)
```

DIP

```
COMPONENT=EasyDisseminationAppC
CFLAGS += -I$(TOSDIR)/lib/net
CFLAGS += -I$(TOSDIR)/lib/net/dip
CFLAGS += -I$(TOSDIR)/lib/net/dip/interfaces

include $(MAKERULES)
```

DHV

```
COMPONENT=EasyDisseminationAppC
CFLAGS += -I$(TOSDIR)/lib/net
CFLAGS += -I$(TOSDIR)/lib/net/dhv
CFLAGS += -I$(TOSDIR)/lib/net/dhv/interfaces

include $(MAKERULES)
```

Now you can install it on several nodes, with at least one having node ID 1. You should see LEDs change every 5 seconds as data is "disseminated" throughout 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].

Tunable Parameters in DIP/DHV

There are 3 sets of tunable parameters in DIP, which are all found in **Dip.h**. For DHV, there parameters are in **Dhv.h**. The first is the Trickle values.

```
#define DIP_TAU_LOW (1024L)
#define DIP_TAU_HIGH (65535L)
```

Trickle values control how frequent advertisements and data should be sent, and have millisecond granularity. **DIP_TAU_LOW** is the minimum period and set when data is actively changing. **DIP_TAU_HIGH** is the maximum period and set when no data is changing.

The second set of the tunable parameters is the data size, which is implicitly represented as a byte array. It must be smaller than TOSH_DATA_LENGTH - 8. The default value is 16 bytes.

```
typedef nx_struct dip_data {
  nx_uint8_t data[16];
} dip_data_t;
```

The last set are for the DIP advertisement sizes. Bigger messages means you can pack more information per advertisement. The default values are two pieces of information per message.

```
#define DIP_SUMMARY_VALUES_PER_PACKET 2
#define DIP_VECTOR_VALUES_PER_PACKET 2
```

You can compute the proper size with the following formulas:

```
DIP_SUMMARY_VALUES_PER_PACKET = (DTOSH_DATA_LENGTH - 5) / 3
DIP_VECTOR_VALUES_PER_PACKET = (DTOSH_DATA_LENGTH - 1) / 2
```

Data Injector in DIP

The last thing to know is the data injector. It allows you to disseminate data from a basestation rather than from a node directly. To do this, you first need to build the corresponding Java message files and injector file by modifying your Makefile to look like:

```
COMPONENT=EasyDisseminationAppC
BUILD_EXTRA_DEPS = DipMsg.class DipDataMsg.class DipData.class DipInject.class
CFLAGS += -I$(TOSDIR)/lib/net
CFLAGS += -I$(TOSDIR)/lib/net/dip
CFLAGS += -I$(TOSDIR)/lib/net/dip/interfaces

DipMsg.class: DipMsg.java
	javac -target 1.4 -source 1.4 DipMsg.java

DipDataMsg.class: DipDataMsg.java
	javac -target 1.4 -source 1.4 DipDataMsg.java

DipData.class: DipData.java
	javac -target 1.4 -source 1.4 DipData.java

DipMsg.java:
	mig java -target=$(PLATFORM) -java-classname=DipMsg $(CFLAGS) $(TOSDIR)/lib/net/dip/Dip.h dip_msg -o $@

DipDataMsg.java:
	mig java -target=$(PLATFORM) -java-classname=DipDataMsg -java-extends=DipMsg $(CFLAGS) $(TOSDIR)/lib/net/dip/Dip.h di

DipData.java:
	mig java -target=$(PLATFORM) -java-classname=DipData -java-extends=DipDataMsg $(CFLAGS) $(TOSDIR)/lib/net/dip/Dip.h d

DipInject.class:
	javac -target 1.4 -source 1.4 DipInject.java

include $(MAKERULES)
```

Before building, you will need to get the file `apps/tests/TestDIP/DipInject.java` and put it in your `EasyDissemination` directory. Once everything is built (e.g. make telosb), you are ready to run the data injector. The syntax is as follows:

```
$ java DipInject [key] [version] [data in quotes delimited by space]
```

key is the data key in hexadecimal.

version is the version number in decimal

data is the actual data in quotes delimited by space

For example, if you want to send key 10, version 2, and data "ab cd ef". You would type:

```
$ java DipInject 0a 2 "ab cd ef"
```

Related Documentation

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

Retrieved from "http://tinyos.stanford.edu/tinyos-wiki/index.php?title=Dissemination&oldid=3016"
Category: Tutorials

- This page was last modified on 8 March 2010, at 22:26.
- This page has been accessed 43,275 times.