

TinyOS

Hands-on Session



Hi. I'm Răzvan Musăloiu-E. and I'm going to be your host for this hands-on session.

Goals

1. Install TinyOS

2. Layout of **tinyos-2.x**

3. Write two applications

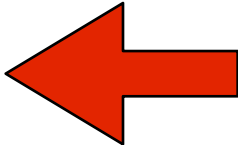
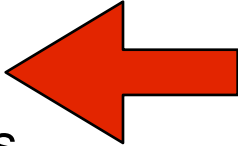
(A) DisseminationDemoClient

(B) CollectionsDemoClient



Today we are going to try to achieve 3 goals. First we'll look at the options available for installing TinyOS, then we will take a quick tour of the layout of the tinyos-2.x sources and then we'll discuss and try to write two application: one based on dissemination and one based on collection.

Options

- LiveCD
 - XubunTOS
 - Customized Ubuntu 8.10 LiveCD
 - Native
 - Linux
 - .rpm packages
 - .deb packages
 - Windows: Cygwin + .rpm packages
 - MacOS X
 - stow
 - macports
-  **Today**
-  **Recommended**



3

The quickest way to get TinyOS is using a LiveCD. Right now we have two of these: one is XubunTOS and the other one is a customized Ubuntu 8.10 LiveCD.

XubunTOS is based on an older version of Xubuntu and beside tinyos-2.x it also contains tinyos-1.x and TOSKI, the ArchRock TinyOS Kernel. The customized Ubuntu 8.10 LiveCD comes only with tinyos-2.x. When 9.04 will be out we will start using that.

The other direction to get TinyOS is to install it on your current OS. For Linux we have a set of .rpm packages and Stanford is maintaining an Ubuntu/Debian repository with the .deb packages.

If you want to run TinyOS on Windows you'll have to install cygwin.

For MacOS X there are also two options: one based on stow with prebuilt binaries and also a one based on macports.

Other Options

- VMware
 - Jetos
 - based on JeOS (Ubuntu Server 8.04)
 - optimized for ssh access
 - very small: 190MB compressed
 - Lenny
 - based on Debian 5.0 “Lenny”
 - graphical interface using XFCE
 - bigger: 300MB compressed
 - XubunTOS



Other popular way to run TinyOS is using VMware. For this we have several images: Jetos is very small and optimized for ssh access; Lenny is an image based on the the latest Debian stable release. Some people also like to run XubunTOS.

Components

- NesC: nesc_*.deb
- Cross compiler
 - binutils: msp430-binutils-tinyos_*.deb
 - gcc: msp430-gcc-tinyos_*.deb
 - libc: msp430-libc-tinyos_*.deb
 - gdb (optional)
- Deputy: deputy-tinyos_*.deb



Here is the way things are split into packages. NesC compiler has its own package and so is deputy, a tool that is used by Safe TinyOS.

Each architecture has its own set of packages. The binutils is the one containing the assembler, linker (ld) and several other tools for dealing with object files (objdump, readelf, etc). Then there is the gcc which contains the C compiler and finally a libc package that contains the precompiled C library. The gdb debugger also has its own packet.

Environment

```
export TOSROOT=$HOME/local/src/tinyos-2.x
export TOSDIR=$TOSROOT/tos

export MAKERULES=$TOSROOT/support/make/Makerules

export CLASSPATH=$TOSROOT/support/sdk/java/tinyos.jar:.
export PYTHONPATH=$TOSROOT/support/sdk/python
```



This is the way the environment needs to be setup. The most important one is the MAKERULES because that we are going to include in the Makefiles of any applications.

Architectures

- AVR
 - mica2, mica2dot
 - micaz
 - btnode
 - IRIS
- ARM
 - imote2
- MSP430
 - telosb, sky
 - shimmer
 - eyesIFX
 - tinynode
 - epic
- 8051
 - CC2430
 - CC1110/CC1111



The AVR and MSP430 are the best supported platforms. The ARM is also supported for the Imote2 platform. We are also working on making 8051 a first class citizen. This is used in some several System-on-a-Chip like CC2430/CC1110/C1111.

Layout

- + `tinynos-2.x`
 - + `apps`
 - + `docs`
 - + `support`
 - + `tools`
 - + `tos`



This is the high level layout of the source code. In the following slides we will briefly go over each of these directories.

Layout

- + **apps**
 - + **Blink**
 - + **Null**
 - + **RadioCountToLeds**
 - + **MultihopOscilloscope**
- + **tests**
 - + ...
 - + ...
- + **docs**
- + **support**
- + **tools**
- + **tos**



- As the name indicates, the apps/ directory contains the applications. Some popular applications are:
- **Blink**: toggles the leds with various frequencies. This is the equivalent of “Hello World!” for TinyOS.
 - **Null**: it really doesn’t do anything. Can be used to check the current draw in sleep mode.
 - **RadioCountToLeds**: 2 motes are needed for this application. It can be used to see if the radio works.
 - **MultihopOscilloscope**: this application implements network collection using CTP.

In apps/tests/ there are a many tests that are used to check if various parts of TinyOS are working properly. Some of the code is also useful as examples.

Layout

- + **apps**
- + **docs**
 - + **html**
 - + **pdf**
 - + **txt**
 - + **...**
- + **support**
- + **tools**
- + **tos**



The docs/ contains the TEPs (TinyOS Enhancement Proposals) in various formats. These documents describe in great detail various parts of TinyOS.

Layout

- + apps
- + docs
- + support
 - + make
 - Makerules
 - + avr/
 - + msp/
 - + ...
 - + sdk
- + tools
- + tos



Layout

- + apps
- + docs
- + support
 - + make
 - + sdk
 - + c
 - + cpp
 - + java
 - + python
- + tools
- + tos



In support/sdk/ there is one folder for several languages. Each of them contains various tools and libraries that implements the communication between PC and motes.

Layout

- + **support**
 - + **sdk**
 - + **c**
 - + **blip**
 - + **sf**
 - + **cpp**
 - + **sf**
 - + **java**
 - **tinyos.jar**
 - + **python**
 - + **tinyos**
 - **tos.py**



The support/sdk/c/blib contains the code necessary for the IPv6 stack contributed by Berkeley.

The support/sdk/c/sf contains a few programs (sf, sflisten, prettylisten, etc) and also the libmote.a C library.

The support/sdk/cpp/sf contains a more complex and versatile version of sf.

The tinyos.jar from support/sdk/java contains all the classes necessary to access the motes from Java.

The support/sdk/python/tos.py is used by Deluge T2 and the tos-dump program.

Layout

- + **apps**
- + **docs**
- + **support**
- + **tools**
- + **tos**
 - + **chips**
 - + **interfaces**
 - + **lib**
 - + **platforms**
 - + **sensorboards**
 - + **systems**
 - + **types**



Layout

- + **tos**
 - + **chips**
 - + **atm128**
 - + **msp430**
 - + **pxa27x**
 - + **cc2420**
 - + **cc1000**
 - + **at45db**
 - + **stm25p**
 - + **sht11**
 - + **...**



The `tos/chips` contains one directory for each supported chip. Some are MCUs (`atm128`, `msp430`), radios (`cc2420`, `cc1000`), flash chips (`at45db`, `stm25p`), sensors (`sht11`), etc.

Layout

- + **tos**
 - + **chips**
 - + **interfaces**
 - **Boot.nc**
 - **SplitControl.nc**
 - **StdControl.nc**
 - **...**
 - + **lib**
 - + **platforms**
 - + **sensorboards**
 - + **systems**
 - + **types**



Layout

```
+ tos
  + lib
    + net
    + printf
    + timer
    + tosthreads
    + serial
      - SerialActiveMessageC.nc
      - SerialAMSenderC.nc
      - SerialAMReceiverC.nc
      - ...
    + ...
```



The tos/lib contains the more complicated platform independent code. The printf library, the serial stack, the timer support are a few examples.

Layout

```
+ tos
  + lib
    + net
      + ctp
      + 4bitle
      + drip
      + Deluge
      + dip
      + blip
      + ...
```



The tos/lib/net contains the network protocols. 4bitle and le are link estimators, ctp and lqi are collection protocols, drip and dip are dissemination protocols, Deluge is a bulk dissemination service use to implement network reprogramming.

Layout

- + **tos**
 - + **systems**
 - **AMReceiverC.nc**
 - **AMSenderC.nc**
 - **MainC.nc**
 - **LedsC.nc**
 - **TimerMilliC.nc**
 - **...**



In tos/systems are a few general components like:

- MainC: provides the Booted event which is signaled after a mote comes up.
- LedsC: access to the Leds interface to control the leds.
- TimerMilliC: used to get timers with millisecond accuracy.
- AMSenderC and AMReceiveC: used to send and receive packets over the radio.

Layout

- + **tos**
 - + **chips**
 - + **interfaces**
 - + **lib**
 - + **platforms**
 - + **sensorboards**
 - + **systems**
 - + **types**
 - **TinyError.h**
 - **messsssage.h**
 - ...



Finally, in `tos/types` there are a few important `.h` files: `TinyError.h` contains the definitions for all the error codes and `message.h` contains the definition of the `message_t` structure which is used by radio and serial packets.

Applications

DisseminationDemo
CollectionDemo



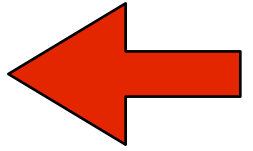
Let's now move to the the two applications.

Dissemination Demo



DisseminationDemo

- DisseminationDemoClient
 - start the radio
 - start Drip
 - when a new value is received print its contents
- DisseminationDemoServer
 - start the radio
 - start Drip
 - start a periodic timer
 - on each firing of the timer increment a counter and disseminate it



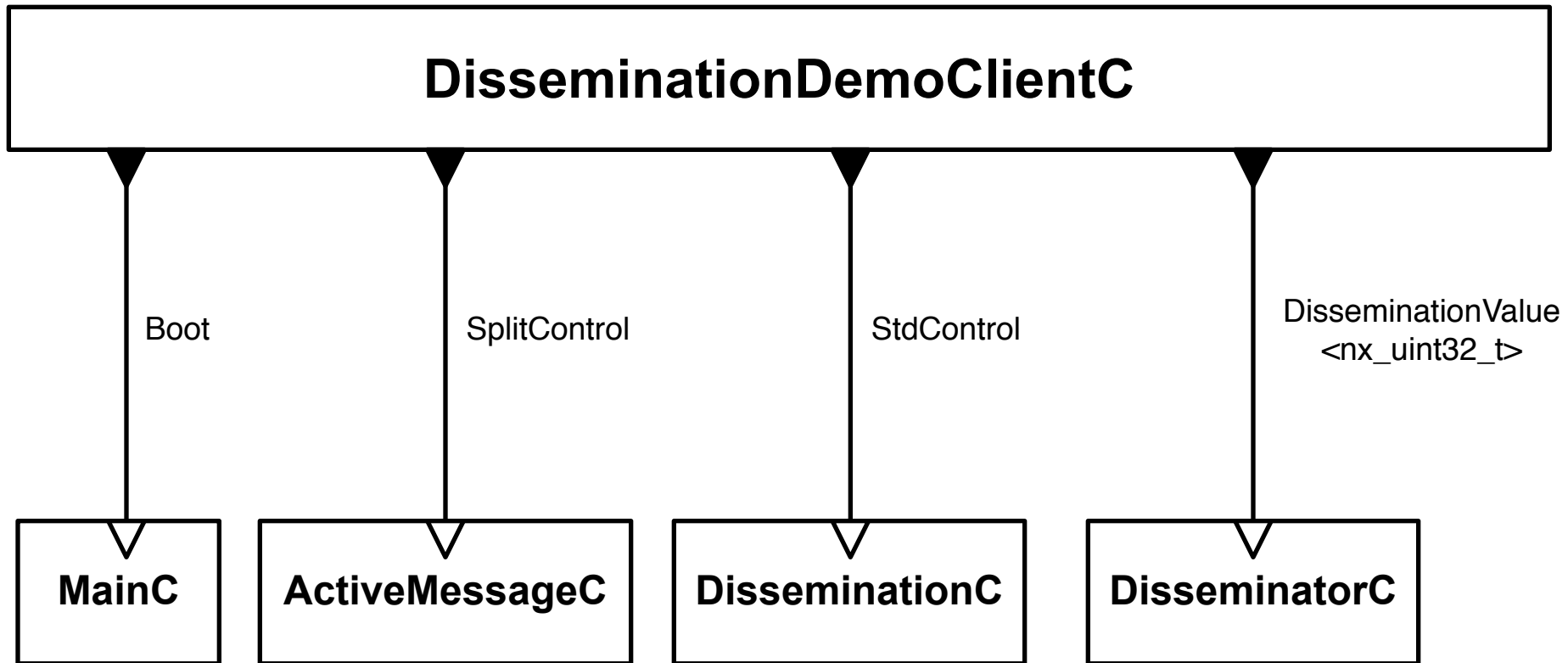
The DisseminationDemo is made of two applications: a client and a server.

The DisseminationDemoClient will need to perform the following tasks: turn on the radio, start the dissemination service and when a new value is received it needs to print it on the serial port.

The DisseminationDemoServer will also start the radio and the dissemination service but after that it will start a timer and each time the timer fires it will increment a counter and disseminate it.

I'm going to run the server so what we are going to do next is to try to write and run the client part.

DisseminationDemoClient



These are the components we going to use and the way they are interconnected with the application.

The Boot from MainC is used in order to receive a signal that the mote is up, the SplitControl from ActiveMessageC is used to turn the radio on, the StdControl from DisseminationC is used to start the dissemination service and the DisseminatorC is used to subscribe to the updates of a certain dissemination key. This key plays in the dissemination the same role as ports play in UDP and TCP.

DisseminationDemoClient

- Interfaces

- Boot
- StdControl
- SplitControl
- DisseminationValue<t>

- Components

- MainC
- ActiveMessageC
- DisseminationC
- DisseminatorC



25

Here are a summary of the interfaces and components we are going to use. In the following slides we are going to briefly look at each of them.

tos/interfaces/Boot.nc

```
interface Boot {  
    event void booted();  
}
```



The Boot interface is very simple and straightforward. It contains only one signal which is called after the mote receives power.

tos/interfaces/StdControl.nc

```
interface StdControl
{
    command error_t start();
    command error_t stop();
}
```



Only two commands are in StdControl: start and stop. Their return code indicates if the start or stop succeeded or not.

tos/interfaces/SplitControl.nc

```
interface SplitControl
{
    command error_t start();
    event void startDone(error_t error);

    command error_t stop();
    event void stopDone(error_t error);
}
```



This is similar with StdControl but it's split phase so the startDone and stopDone are the one that indicates when and if the operation was successful. The most important components that provide this interface are the ActiveMessageC and SerialActiveMessageC.

tos/lib/net/DisseminationValue.nc

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



This is the interface provided by DisseminatorC. The changed event is signaled when an update to the key is received. The get can be used to obtain a pointer to the current value. The set command can be used to set an initial value. The t is the type of the key.

tos/system/MainC.nc

```
configuration MainC {  
    provides interface Boot;  
    uses interface Init as SoftwareInit;  
}  
  
implementation {  
    ...  
}
```



tos/platforms/telosa/ActiveMessageC.nc

```
configuration ActiveMessageC {  
    provides {  
        interface SplitControl;  
        ...  
    }  
}  
  
implementation {  
    ...  
}
```



tos/lib/net/drip/DisseminationC.nc

```
configuration DisseminationC {  
    provides interface StdControl;  
}  
  
implementation {  
    ...  
}
```


tos/lib/net/drip/DisseminatorC.nc

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



DisseminatorC is a generic component and the two parameters we'll have to give are the type of disseminated value and the key. The DisseminationUpdate which we are not going to go into is used by the server to inject updates.

Makefile

```
COMPONENT=DisseminationDemoClientAppC
```

```
CFLAGS += -I%T/lib/net
```

```
CFLAGS += -I%T/lib/net/drip
```

```
CFLAGS += -I%T/lib/printf
```

```
include $(MAKERULES)
```



Commands

```
$ make telosb
```

```
$ make telosb install,42
```

```
$ tos-dump.py serial@/dev/ttyUSB0:115200
```



And these are the commands to compile, install and read the results.

Summary

`tos/interfaces/Boot.nc`

`tos/interfaces/StdControl.nc`

`tos/interfaces/SplitControl.nc`

`tos/system/MainC.nc`

`tos/platforms/telosa/ActiveMessageC.nc`

`tos/lib/net/drip/DisseminationC.nc`

`tos/lib/net/drip/DisseminatorC.nc`



DisseminationDemoClientAppC.nc

```
configuration DisseminationDemoClientAppC { }

implementation
{
  components MainC;
  components DisseminationC;
  components new DisseminatorC(nx_uint32_t, 2009);
  components DisseminationDemoClientC;
  components ActiveMessageC;

  DisseminationDemoClientC.Boot -> MainC;
  DisseminationDemoClientC.DisseminationStdControl -> DisseminationC;
  DisseminationDemoClientC.DisseminationValue -> DisseminatorC;
  DisseminationDemoClientC.RadioSplitControl -> ActiveMessageC;
}
```



The type of the key is a `nx_uint32_t` and the key value is 2009.

DisseminationDemoClientC.nc

```
module DisseminationDemoClientC
{
  uses {
    interface Boot;
    interface DisseminationValue<nx_uint32_t>;
    interface StdControl as DisseminationStdControl;
    interface SplitControl as RadioSplitControl;
  }
}

implementation
{
  nx_uint32_t counter;

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

  ...
}
```



DisseminationDemoClientC.nc

```
module DisseminationDemoClientC
{
    ...
}

implementation
{
    ...

    event void RadioSplitControl.startDone(error_t error)
    {
        call DisseminationStdControl.start();
    }

    event void DisseminationValue.changed()
    {
        printf("R: %lu\n", *(call DisseminationValue.get()));
        printfflush();
    }

    event void RadioSplitControl.stopDone(error_t error) { }
}
}
```



The printfflush is necessary to force the printf library to send the contents of its buffer immediately.

<http://docs.tinyos.net/index.php/lpsn2009-tutorial>



The code for the server is available on the TinyOS wiki at this address.

CollectionDemo

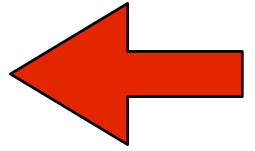


41

Let's now move to the second application.

CollectionDemo

- CollectionDemoClient
 - start the radio
 - start CTP
 - start a periodic timer
 - on each firing of the timer increment a counter and sent it over CTP
- CollectionDemoServer
 - start the radio
 - start CTP
 - when a new value is received print its contents

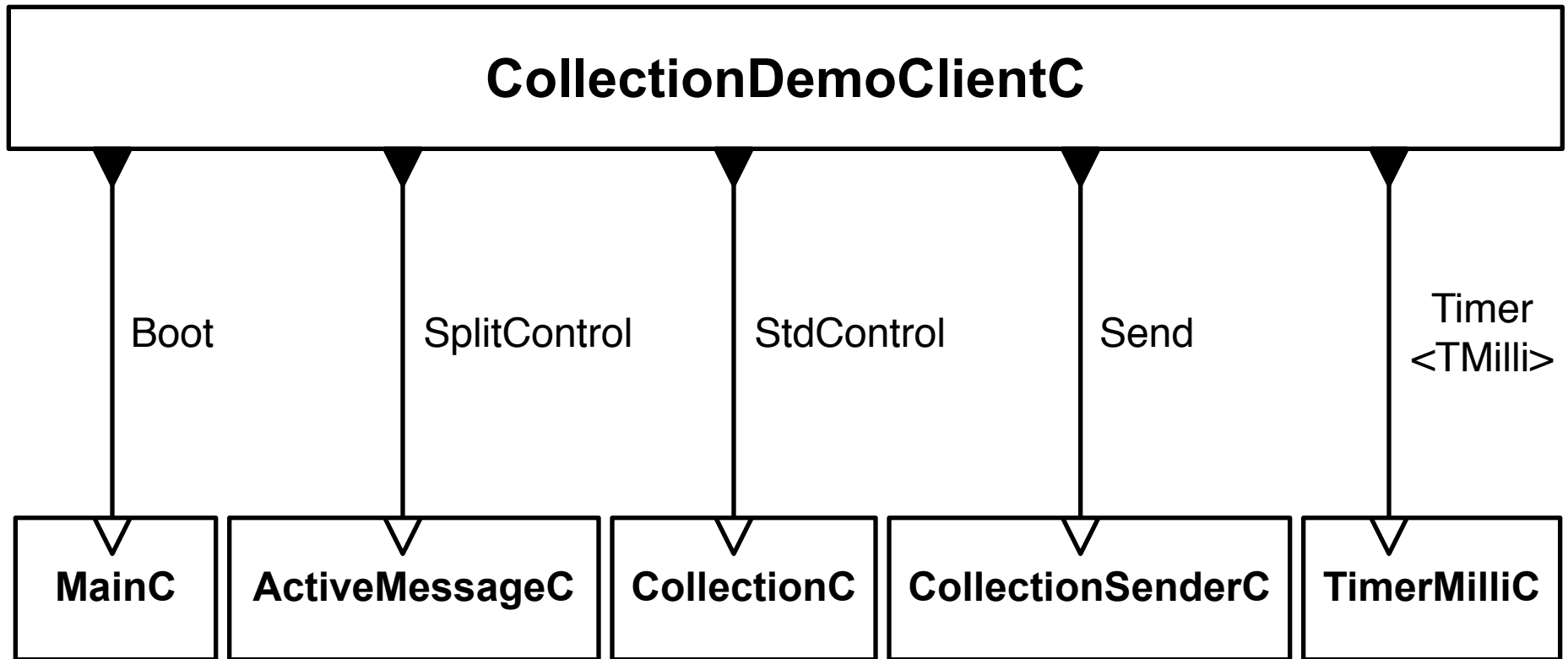


Again we have a client and server. The client will have to start the radio, start the collection service and the periodically increment a counter and then the value.

The server will be the root of the collection tree and will receive all the values and print them on the screen. This is the code I'm going to run.

Now let's try to write and run the code for the client.

CollectionDemoClient



This is the components and interfaces we need. The CollectionC is component that offers the StdControl interface for collection. The CollectionSenderC is the offering a Send interface to send the value up on the tree.

CollectionDemoClient

- Interfaces

- Boot
- StdControl
- SplitControl
- Send
- Timer<TMilli>

- Components

- MainC
- ActiveMessageC
- CollectionC
- CollectionSenderC
- TimerMilliC



CollectionDemoClient

- Interfaces

- Boot
- StdControl
- SplitControl
- **Send**
- **Timer<TMilli>**

- Components

- MainC
- ActiveMessageC
- **CollectionC**
- **CollectionSenderC**
- **TimerMilliC**



tos/interfaces/Send.nc

```
interface Send {  
    command error_t send(message_t* msg, uint8_t len);  
    event void sendDone(message_t* msg, error_t error);  
    command uint8_t maxPayloadLength();  
    command void* getPayload(message_t* msg, uint8_t len);  
  
    command error_t cancel(message_t* msg);  
}
```



tos/lib/net/ctp/CollectionC.nc

```
configuration CollectionC {  
  provides {  
    interface StdControl;  
    ...  
  }  
}  
  
implementation {  
  ...  
}
```

tos/lib/net/ctp/CollectionSenderC.nc

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

implementation {
  ...
}
```



tos/system/TimerMilliC.nc

```
generic configuration TimerMilliC() {  
    provides interface Timer<TMilli>;  
}  
  
implementation {  
    ...  
}
```

Makefile

```
COMPONENT=CollectionDemoClientAppC
```

```
CFLAGS += -I%T/lib/net
```

```
CFLAGS += -I%T/lib/net/ctp
```

```
CFLAGS += -I%T/lib/net/4bitle
```

```
CFLAGS += -I%T/lib/printf
```

```
include $(MAKERULES)
```



This is the Makefile we are going to use. Note that I picked the 4bitle for link estimation.

Summary

`tos/interfaces/Boot.nc`

`tos/interfaces/StdControl.nc`

`tos/interfaces/SplitControl.nc`

`tos/interfaces/Send.nc`

`tos/lib/timer/Timer.nc`

`tos/system/MainC.nc`

`tos/system/TimerMilliC.nc`

`tos/platforms/telosa/ActiveMessageC.nc`

`tos/lib/net/ctp/CollectionC.nc`

`tos/lib/net/ctp/CollectionSenderC.nc`



CollectionDemoClientAppC.nc

```
configuration CollectionDemoClientAppC { }

implementation
{
  components MainC;
  components ActiveMessageC;
  components CollectionC;
  components new CollectionSenderC(16);
  components new TimerMilliC() as Timer;
  components CollectionDemoClientC;

  CollectionDemoClientC.Boot -> MainC;
  CollectionDemoClientC.RadioSplitControl -> ActiveMessageC;
  CollectionDemoClientC.CollectionStdControl -> CollectionC;
  CollectionDemoClientC.Send -> CollectionSenderC;
  CollectionDemoClientC.Timer -> Timer;
}
```



Note that the collection id I picked is 16.

CollectionDemoClientC.nc

```
module CollectionDemoClientC
{
  uses {
    interface Boot;
    interface SplitControl as RadioSplitControl;
    interface StdControl as CollectionStdControl;
    interface Send;
    interface Timer<TMilli>;
  }
}

implementation
{
  message_t msg;

  typedef nx_struct {
    nx_uint8_t string[8];
    nx_uint16_t counter;
  } name_t;
  name_t *name;

  ...
}
```



Note that the collection id I picked is 16.

CollectionDemoClientC.nc

```
module CollectionDemoClientC
{
    ...
}

implementation
{
    ...

    event void Boot.booted()
    {
        name = call Send.getPayload(&smsg, sizeof(name_t));
        strcpy((char*)name->string, "name");
        name->counter = 0;
        call RadioSplitControl.start();
    }

    ...
}
}
```



Please replace "name" with your name.

CollectionDemoClientC.nc

```
module CollectionDemoClientC
{
    ...
}

implementation
{
    ...

    event void RadioSplitControl.startDone(error_t error)
    {
        call CollectionStdControl.start();
        call Timer.startPeriodic(1024);
    }

    ...
}
```



CollectionDemoClientC.nc

```
module CollectionDemoClientC
{
    ...
}

implementation
{
    ...

    event void Timer.fired()
    {
        error_t error;
        name->counter++;
        error = call Send.send(&smsg, sizeof(name_t));
        printf("S: %d %d\n", name->counter, error);
        printfflush();
    }

    event void Send.sendDone(message_t* msg, error_t error) { }
    event void RadioSplitControl.stopDone(error_t error) { }
}
}
```



<http://docs.tinyos.net/index.php/lpsn2009-tutorial>



The code for the server is available on the TinyOS wiki at this address.

The End.

