Network Reprogramming

Jaein Jeong, Sukun Kim and Alan Broad
Aug 12, 2003

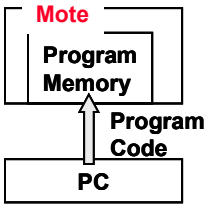
Overview

Concept

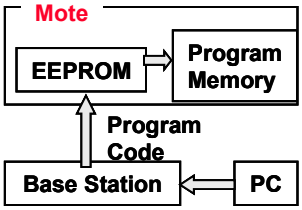
This document describes network reprogramming and complements the reference from Crossbow technology (tinyos-1.x/doc/Xnp.pdf). Network reprogramming is to load transfer the program code using wireless communication rather than direct connection to PC host (e.g. loading program through parallel port).

Network reprogramming works mainly in two stages.
First, the program code is stored outside the program memory through radio packets.

Second, the downloaded code is transferred to the program memory and the mote reboots with the new code.



Parallel programming.

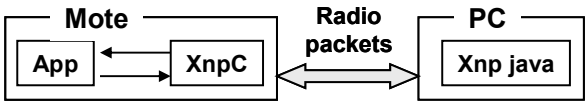


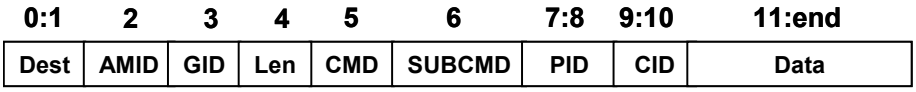
Network programming

Components

Network reprogramming consists of mote modules and a java program on PC host. On a mote side, XnpM module handles most of the functions like program download and query. And the main application needs to be wired to XnpC module.

On PC side, Xnp java program sends program code and commands through radio. Between a mote and PC, messages of reversed message ID (47) are transferred. The format of message is as follows:





- AMID: set to #47.
- CMD: type of command (e.g., download, query and etc).
- PID: Checksum for program code. Used for validation
- CID: Sequence number for capsule

Steps of network reprogramming

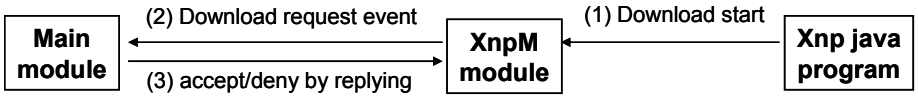
Network reprogramming is done in three steps: download phase, query phase and reprogram phase.

Download Phase

Network reprogramming starts with the Xnp java program telling the start of download:

- Start of download
  Network reprogramming starts with download start message:
  - Xnp java program sends a request and XnpM relays this request to the main module.

- Main module can reserve resources and acknowledges to XnpM.
- 'Download start' message is sent multiple times.



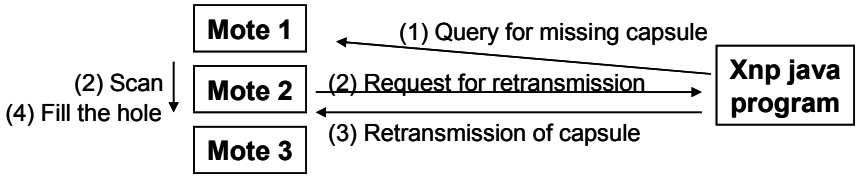
- Download
  After sending start of download message a couple of times, Xnp java program sends each line of program as a capsule. XnpM on the mote side, receives this capsule and stores in EEPROM.

Query Phase

Once Xnp java program finishes sending program capsules, it sends download terminate message to notify the end of download. Then, the mote searches any missing capsule in its EEPROM and asks the retransmission of it to PC side. This is done in following steps:

1. java program asks motes for missing capsules.
2. Each mote scans EEPROM and requests the retransmission of the next missing capsule.
3. In response, java program sends the missing capsule.
4. Other motes can also fill the hole as well as the requestor.

This Query loop ends when the java program doesn't get request for several times.



Reprogram Phase

In reprogram phase, the downloaded code is transferred to the program memory and the mote starts the new program. Reprogram phase works in the following steps:

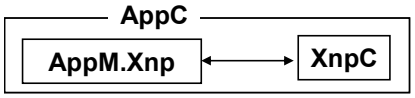
- First, java program sends a reprogram request.
- After checking the EEPROM for correctness, XnpM transfers control to the boot loader.
- The boot loader copies the code in EEPROM to program memory and reboots the system.

After reboot, mote ID and group ID needs to be set correctly.

How to wire apps for network programming

To make an existing application network reprogrammable, the following modifications are needed:

First, wire network reprogramming module in configuration file. The figure in the below shows a configuration file AppC which connects the configuration file of network reprogramming module (XnpC) to the application implementation (AppM).

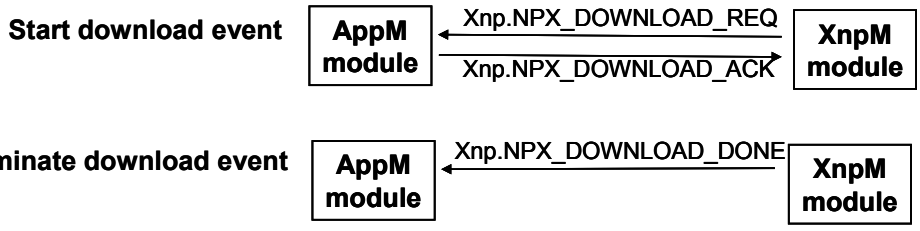


Second, implementation file (AppM) needs to be modified to handle events and set mote IDs.

Handling events:
Start download event (***Xnp.NPX_DOWNLOAD_REQ()***) is called when XnpM module receives start download command from the PC side. The event handler can accept or reject network reprogramming by sending corresponding acknowledgement.

Terminate download event (***Xnp.NPX_DOWNLOAD_DONE()***) is called when XnpM module receives terminate download command. The application can resume its task after receiving this event.

Setting mote IDs: The program code transferred in reprogram phase doesn't have mode ID and group ID. The mote ID and group ID are saved in a special location in EEPROM. After reboot, ***Xnp.NPX_SET_IDS()*** should be called to set the mote ID and group ID.



Finally, Makefile should be modified to enable network reprogramming.
If the identifier XNP is defined in Makefile, related files defined in XNP_DIR (this is defined in apps/Makerules) are added to the search path. If identifier AVRISP is defined, AVRISP serial programmer is assumed rather than parallel port.

Here is an example, XnpCount. This application is extended from CntToLedsAndRfm.

XnpCount.nc

```
configuration XnpCount {
}
implementation {
  components Main, Counter, IntToLeds, IntToRfm, TimerC,
XnpCountM, XnpC;

  Main.StdControl -> Counter.StdControl;
  Main.StdControl -> IntToLeds.StdControl;
  Main.StdControl -> IntToRfm.StdControl;
  Main.StdControl -> TimerC.StdControl;
  Counter.Timer -> TimerC.Timer[unique("Timer")];
  IntToLeds <- Counter.IntOutput;
  Counter.IntOutput -> IntToRfm;

  Main.StdControl -> XnpCountM.StdControl;
  XnpCountM.Xnp -> XnpC;
  XnpCountM.CntControl -> Counter.StdControl;
}
```

XnpCountM.nc

```
includes AM;

module XnpCountM {
  provides {
    interface StdControl;
  }
  uses {
    interface Xnp;
    interface StdControl as CntControl;
    interface TS;
  }
}
implementation {
#include "Xnp.h"
  uint16_t dest;
  uint8_t  cAck;

  command result_t StdControl.init() {
    call Xnp.NPX_SET_IDS();          //set mote_id and group_id
    return SUCCESS;
  }
  command result_t StdControl.start() {
    return SUCCESS;
  }
  command result_t StdControl.stop() {
    return SUCCESS;
}
```

```
  event result_t Xnp.NPX_DOWNLOAD_REQ(uint16_t wProgramID,
uint16_t wEEStartP, uint16_t wEENofP){
    call Xnp.NPX_DOWNLOAD_ACK(SUCCESS);
    call CntControl.stop();
    return SUCCESS;
  }

  event result_t Xnp.NPX_DOWNLOAD_DONE(uint16_t wProgramID,
uint8_t bRet,uint16_t wEENofP){
    if (bRet == TRUE)
       call CntControl.start();
    return SUCCESS;
  }
 }
```

Makefile

```
COMPONENT=XnpCount
SENSORBOARD=basicsb
XNP=yes
#AVRISP=yes

include ../Makerules
```

Location of Network reprogramming module

These are the directories related to network reprogramming:

- Tinyos-1.x/tos/lib/Xnp: main module for network programming.
- Tinyos-1.x/apps: sample applications
  - XnpCount, XnpRfmToLeds, XnpOscope
- Tinyos-1.x/tools/java/net/tinyos/xnp: java program

Installing network programmable code

Compile

Before an application is loaded to a mote, the application code should be compiled. Currently, network reprogramming only works for the platforms with ATmega128 microcontroller (mica2 and mica2dot). TinyOS release 1.1 contains all the latest tools to compile the application in ATmega128 native mode (avr-gcc, nesC and uisp). The following commands compile the application code.

`make mica2`	(for mica2 platform)
`make mica2dot`	(for mica2dot platform)

Loading application code

Until a mote has a network reprogrammable code running in its memory, it can load program code only through direct connection. Following commands load the application code into the flash memory:

```
make mica2 reinstall.<moteid>          (for mica2 platform)
make mica2dot reinstall.<moteid>       (for mica2dot platform)
```

These commands are expanded into the following commands in case parallel port is selected:

```
uisp -dprog=dapa -dpart=ATmega128 --wr_fuse_e=ff --erase
uisp -dprog=dapa -dpart=ATmega128 --wr_fuse_e=ff --upload if=main.srec
uisp -dprog=dapa -dpart=ATmega128 --wr_fuse_e=ff --verify if=main.srec
```

Loading boot loader

Next, a boot load should be loaded with the following command:

```
make mica2 inp         (for mica2 platform)
make mica2dot inp      (for mica2dot platform)
```

These commands are expanded into the following command:

```
uisp –dprog=dapa --upload if=<bootloader.srec>
```

<bootloader.srec> is the name of boot loader file for corresponding platform (inpispm2.srec for mica2 and inpispm2d.srec for mica2dot).

Example

This is an example to install XnpCount application to a mica2dot mote using AVRISP serial programmer.

```
Administrator@WILDCAT /cygdrive/c/tinyos-cvs/tos-1-1-0-candicate/tinyos-
1.x/apps/XnpCount
$ make mica2dot
    compiling XnpCount to a mica2dot binary
ncc -o build/mica2dot/main.exe -Os -board=basicsb -target=mica2dot -
I../../tos/lib/Xnp  -Wall -Wshadow -DDEF_TOS_AM_GROUP=0x7d -Wnesc-all -finline-
limit=100000 -fnesc-cfile=build/mica2dot/app.c -DIDENT_PROGRAM_NAME="XnpCount"
-DIDENT_INSTALL_ID=19657u -DIDENT_UNIX_TIME=1060836026L  XnpCount.nc -lm
c:/tinyos-cvs/tos-1-1-0-candicate/tinyos-1.x/tos/lib/Xnp/XnpM.nc:806: warning:
call via function pointer
    compiled XnpCount to build/mica2dot/main.exe
          15248 bytes in ROM
            569 bytes in RAM
avr-objcopy --output-target=srec build/mica2dot/main.exe
build/mica2dot/main.srec
```

```
Administrator@WILDCAT /cygdrive/c/tinyos-cvs/tos-1-1-0-candicate/tinyos-
1.x/apps/XnpCount
$ make mica2dot reinstall.43
    installing mica2dot binary
set-mote-id build/mica2dot/main.srec build/mica2dot/main.srec.out `echo
reinstall.43 |perl -pe 's/^reinstall.//; $_=hex if /^0x/i;'`
uisp -dprog=stk500 -dserial=/dev/ttyS1 -dpart=ATmega128 --wr_fuse_e=ff  --erase
--upload  if=build/mica2dot/main.srec.out
Firmware Version: 1.14
Atmel AVR ATmega128 is found.
Uploading: flash

Fuse Extended Byte set to 0xff

Administrator@WILDCAT /cygdrive/c/tinyos-cvs/tos-1-1-0-candicate/tinyos-
1.x/apps/XnpCount
$ make mica2dot inp
uisp -dprog=stk500 -dserial=/dev/ttyS1 -dpart=ATmega128 --upload
if=../../tos/lib/Xnp/inpispm2d.srec
Firmware Version: 1.14
Atmel AVR ATmega128 is found.
Uploading: flash
```

Using Java Application

Java application files for network reprogramming are stored in the following location: `$TOSROOT/tools/java/net/tinyos/xnp.`

Connecting to UART

Xnp java application can be connected to UART using the standard UART interface introduced at TinyOS 1.1 release.

By default, Xnp java application is connected to UART through SerialForwarder with hostname localhost and port 9001. When the java application needs to be connected to a different source, the environment variable MOTECOM can be set with the name of source.

Recommendation is to use TOSBase (a base program) and SerialForwarder. This allows us to use other UART monitoring program like Matlab while keeping the synchronization of UART packets.

Example

This is an example to run xnp java application.

This command runs xnp program for SerialForwarder with hostname localhost and port 9001 (sf@localhost:9001).

```
$ java net/tinyos/xnp/xnp &
[1] 3628
```

This command runs still xnp program for sf@localhost:9001, but SerialForwarder is not running. The error message shows that the source is not available.

```
Administrator@WILDCAT /cygdrive/c/tinyos-1.x/tools/java
```

```
$ java net/tinyos/xnp/xnp &
[2] 1816
[1]   Done                    java net/tinyos/xnp/xnp

Administrator@WILDCAT /cygdrive/c/tinyos-1.x/tools/java
$ java.net.ConnectException: Connection refused: connect sf@localhost:9001
Check the source sf@localhost:9001 or try a different source by setting MOTECOM
environment variable.

[2]+  Exit 1                  java net/tinyos/xnp/xnp
```

This command set the environment variable MOTECOM as serial@COM1:mica2 (direct connection to UART port 1 with mica2 baud rate).

```
Administrator@WILDCAT /cygdrive/c/tinyos-1.x/tools/java
$ export MOTECOM=serial@COM1:mica2

Administrator@WILDCAT /cygdrive/c/tinyos-1.x/tools/java
$ java net/tinyos/xnp/xnp &
[1] 1480

Administrator@WILDCAT /cygdrive/c/tinyos-1.x/tools/java
$ serial@COM1:57600: resynchronising
```

GUI Description

Although xnp java application send program code in unicast mode, broadcast mode is always recommended. In unicast mode, xnp java application checks delivery for each capsule and this makes the whole process very slow. Whereas in broadcast mode, xnp java application checks any missing packets after sending complete program capsules.

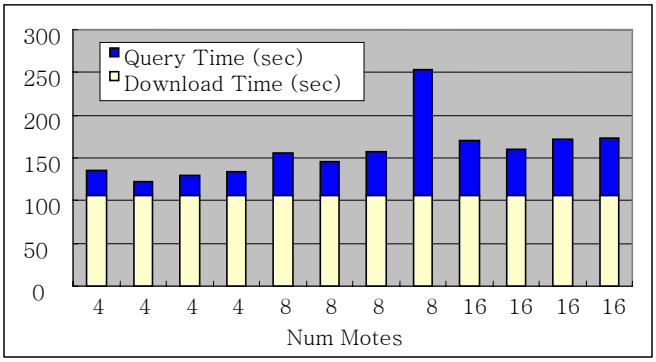
By default, Xnp java application displays the mote id in decimal. If '-hexid' option is specified, mote id is displayed in hexadecimal in the status message.

Due to the limitation of packet format, xnp java application displays only 8-bit as a mote id in the status message.

Test Results

We did experiments to measure the running time and the success rate of network reprogramming. As an application XnpCount was used. XnpCount is small in its size (37000 bytes, 841 capsules) and has minimum functionality. We could get reasonable success rate when motes are normal. For UART interface we used GenericBase (sync mode) and SerialForward.

Num Motes	Num Tries	Download Time (s)	Query Time (s)	Num Successes
4	4	106	24.25	4
8	4	106	72.25	7.75
16	4	106	63	16



At the time when we did the experiments, new UART interface was not released. We tested the network reprogramming informally with new UART interface (TOSBase and SerialForwarder) and we could see that the download time has increased a little, but the time for query time was got much smaller due to the reliable UART communication.

We found some cases when when network programming failed:

- Parameters in xnp java is not correct.
  - Interval between each write should be large enough for the base (TOSBase or GenericBase) to handle. 120 ms was found by experiment.
  - When xnp java application sends packets faster than the base can handle the base can get stuck and doesn't transfer any more packets (especially GenericBase, which is the old version of base).
- If the program code is not loaded correctly, the mote cannot download code.
- When a mote has low battery level, it cannot write into its EEPROM and doesn't proceed. In this case, the battery needs to be replaced.
  - If hardware modification (e.g. soldering loose battery connection) is made, the mote may not work properly. In this case we needed to load the code with parallel programming.